САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА АСТРОФИЗИКИ

А.Б. Шнейвайс

ОСНОВЫ ПРОГРАММИРОВАНИЯ

(ФОРТРАН, СИ)

Второй семестр

для студентов специалитета «АСТРОНОМИЯ»

САНКТ-ПЕТЕРБУРГ
2020

Рецензенты: кандидат техн. наук, доцент В.Б. Синильщиков
(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова)
кандидат физ.-мат. наук, доцент В.Б. Титов
(СПбГУ)

Печатается по постановлению
Учебно-методической комиссии по укрупнённой группе
направлений и специальностей 03.00.00"Физика и астрономия"

Шнейвайс А.Б.

Основы программирования (ФОРТРАН, СИ). Второй семестр
для студентов специалитета «АСТРОНОМИЯ»: Учебное пособие
–СПб,2020.– 430 с.

Учебное пособие «Основы программирования (ФОРТРАНе, СИ) Второй семестр» содержит информацию по дисциплине «Программирование», излагаемую для студентов по специальности «АСТРОНОМИЯ» астрономического отделения математико-механического факультета СПбГУ во втором семестре.
Рассматриваются темы: операции над данными в языках ФОРТРАН и СИ, использование рекурсии, краткий обзор основных структур данных (массивы, структуры, указатели в СИ и ФОРТРАНе), использование массивов в качестве формальных и фактических аргументов процедур, динамические массивы, операции и функции современного ФОРТРАНа по работе с массивами, форматирование данных ввода/вывода и др.
Почти каждая тема завершается соответствующим домашним заданием. Пособие содержит несколько приложений: в частности, по усовершенствованию make-файла, замеру времени работы отдельных фрагментов программы

Содержание

ПРЕДИСЛОВИЕ

Учебное пособие «Основы программирования на ФОРТРАНе и СИ» (второй семестр) является продолжением изложения материала, который осваивался студентами первого курса астрономического отделения математико-механического факультета в первом семестре на основе пособия «Основы программирования на ФОРТРАНе и СИ» (первый семестр) (www.astro.spbu.ru: образование — учебные материалы — программирование — 1-ый семестр; (см. также http://hdl.handle.net/11701/15400 — репозиторий СПбГУ).

Первый раздел настоящего пособия представляет собой обзор элементарных операций, многими из которых студенты активно пользовались во время практических занятий. Раздел насыщен короткими программами, соответствующими теме. Некоторые операции языка СИ реализованы в ФОРТРАНе встроенными функциями, которые так же рассмотрены в первом разделе.

Содержание второго раздела «Рекурсия в программировании». Первоначальные версии ФОРТРАНа не допускали рекурсии. В данном пособии на простых примерах поясняется как описываются рекурсивные процедуры в современном ФОРТРАНе. Отмечаются достоинства и недостатки рекурсивного описания. Рассмотрена ФОРТРАН-версия решения известной задачи о Ханойской башне. На элементарном уравне проводится сравнение затрат времени при решении одной и той же задачи рекурсивной и нерекурсивной процедурами. (в частности, задачи о расчёте N-го числа Фибоначчи и задачи о поиске изолированного корня уравнения методом деления отрезка пополам).

Третий раздел — обзор тем «Массивы», «Структуры», «Указатели». Поясняется, что в программировании, как правило, часто бывают востребованы не только простые переменные, но и более сложные составные структуры данных: массивы, как именованный набор данных одинакового типа; структуры, как именнованный набор данных, который может состоять из элементов разных типов. В СИ имя массива трактуется как указатель на его начальный элемент. Поэтому в разделе затронута тема «Указатели». Как показывает опыт эта тема труднее для восприятия первокурсникам нежели темы «Массивы» и «Структуры». Поэтому соответствующие примеры и пояснения в данном разделе довольно объёмны. Рассмотрены задачи построения стека и кольцевого списка

Четвёртый раздел — введение в тему «Массивы» (описания массивов, расположение в оперативной памяти, индексация, массивов в качестве аргументов процедур, знакомство с одной из встроенных ФОРТРАН-функцией SUM, суммирующей элементы массива).

Пятый раздел — «Одномерный динамический массив». В современном ФОРТРАНе предоставлена возможность выделения памяти под массивы в процессе работы исполнимого файла, чего не позволяли старые ФОРТАН-компиляторы. В данном разделе излагается организация работы с операторами allocatable, allocate и deallocate, реализующими возможность описания, размещения и освобождения памяти (отведённой под массивы оператором allocate), что в значительной мере упростило (по сравнению с ФОРТРАНом-77) работу с многомерными массивам (в частности, с матрицами).

Шестой раздел — «Операции ФОРТРАНа над массивами» Рассмотрены понятия инициализации, секции массива, виды индексов секции (скалярный, индексный триплет, векторный индекс). Приводятся примеры использования секций массива, заполнение матриц. Поясняется использование функции RESHAPE, реализующей размещение содержимого массива одной формы нужным образом по элементам массива другой формы. Демонстрируется присваивание значений элементам массива по маске, а также операторы и констукции where и forall.

Седьмой раздел.— Функции ФОРТРАНа-95 для работы с массивами Рассматриваются примеры и обсуждаются результаты работы многих встроенных функций современного ФОРТРАНа, нацеленных на работу с массивами.

Восьмой раздел (обзорный). — Форматирование данных ввода вывода. В процессе обучения в первом и втором семестрах студентам неоднократно приходилось вводить и выводить данные. В случае Фортрана при решении задач домашних заданий студент обычно использует самую непритязательную форму ввода/вывода — форму под управлением списка ввода/вывода. Однако, при проверке работ студентам приводятся и поясняются примеры форматного ввода/вывода (его достоинства и недостатки), так что к моменту изложения содержания данного раздела, у студента уже выработан некоторый навык работы по форматному вводу/выводу. Другими словами, обучающийся готов к восприятию подобной информации.

Девятый раздел. Контрольная работа. Моделирование работы разностной машины Чарльза Бэббиджа. Задача простая. Однако, темы программирования, изученные в течение второго семестра, которые можно включить в условия разных вариантов, достаточно многобразны: одномерный и двумерный массивы; форматные дескрипторы, для организации наглядного вывода, внутренний файл (использование переменной типа CHARACTER для организации динамического повторителя; явное указание интерфейса процедур; модульное программирование; быстрый и простой переход на иную разновидность типа REAL.

Наконец, анализ зависимости точности результатов расчёта (на основе сравнения последних с результатами расчёта по схеме Горнера) от машинной погрешности округления входных данных.

В пособии имеется несколько приложений (разделы 10-16):

- Приложение I. Функция передачи типа **transfer**.
- Приложение II. Азы GNU-утилиты **make** (часть вторая).
- Приложение III. О подсчёте времени.
- Приложение IV. Операторы ввода/вывода.
- Приложение V. Простые поэлементные функции.
- Приложение VI. Числовые характеристики представления данных.
- Приложение VII. Некоторые опции **gfortran**а.

При решении задач четвёртого и пятого домашних заданий, нацеленных на использование статических и размещаемых одномерных массивов для численного вычисления интералов, обучающиеся могут воспользоваться пособием

«ПРАКТИКА ПРОГРАММИРОВАНИЯ (вычисление интегралов)»,

помещённым на сайте НИАИ СПбГУ в разделе WWW-ресурсы.

Во втором семестре обучающимся достаточно ознакомиться с материалом, изложенным в первых двух разделах упомянутого пособия (остальные разделы могут быть востребованы в третьем семестре).

1 Выражения и операции.

1.1 Выражение.

Выражение – формула для получения значения.

Пример выражения	Тип аргументов	Тип результата
((i+j)+abs(i-j))/2	**integer**, **int**	**integer**, **int**
sqrt(x)	**real** **double**	ФОРТРАН: **real** СИ: **double**
'ЩИ'//' да '//'КАША'	**character(2)** **character(4)**	ФОРТРАН: **character(10)**
a .and. b **a && b**	**logical** **bool**	ФОРТРАН: **logical** C++ : **bool**

Выражение состоит из

- **операндов**,
- знаков **операций** и
- **символов-разделителей**.

Операнд – это константа, переменная, либо другое выражение (в частности, вызов функции).
Порядок интерпретации выражения компилятором может быть изменен посредством заключения частей выражения в круглые скобки.
Разделители в Си [ ] ( ) { } , ; : ⋯ * = #;

Из них ФОРТРАН использует круглые скобки, двоеточие и точку с запятой.
Элементарная операция над данными задается знаком операции.
Есть две группы операций: **одноместные** или унарные (один операнд) и **двуместные** или бинарные (два операнда).

В СИ элементарных операций больше чем в ФОРТРАНе. ФОРТРАН вместо отсутствующих операций, имеющихся в СИ, как правило, использует встроенные функции. Аналогично и СИ при отсутствии операции, имеющейся в ФОРТРАНе (например, операции конкатенации или возведения в степень), использует свои функции.

По типу выполняемой операции различают арифметические, поразрядные, логические, операции отношения и другие.

Изложение темы дается по [1], [2], [8], [6], [4], [3], [5].

1.2 Основные операции ФОРТРАНа и СИ.

1. Арифметические операции.
2. Логические операции и операции отношения.
3. Тернарная (условная) операция.
4. Операция присваивания.
5. Операция sizeof.
6. Операции сдвига.
7. Поразрядные логические операции.
8. Неявное преобразование типов в выражениях.
9. Операции явного преобразования типа.
10. Операция последовательного вычисления
11. Приоритет операций.

1.2.1 Арифметические операции

Операция	C, C++	Примечание	ФОРТРАН	Примечание
+ - * /	+ - * /	5/3=1 5.0/3=1.6(6)	+ - * /	то же, что и для СИ
Расчет остатка от деления двух целых	%	Только к целым операндам 5%3=2	Вместо операции использует функцию **mod** mod(5,3)= 2 mod(5.,3.)= 2.0 mod(5d0,3d0)=2d0	Имя **mod родовое:** обеспечивает перегрузку функций

Замечания.

1. В C есть операции ++ и −− **инкремента** и **декремента** (приращения и уменьшения). **Префиксная** и **постфиксная** формы обеих применимы только к переменным. **Префиксная** сначала изменяет операнд на значение, соответствующее типу операнда, а затем использует его найденное значение. **Постфиксная** сначала использует значение операнда до его изменения, а уж затем изменяет. В составном выражении переход с префиксной формы записи на постфиксную изменяет значение выражения. Проанализируем работу программы:

```
#include <stdio.h>
int main(void)
{ int a=1, b=1,c; float f=1.7; double  w=3.3;
 printf("a=%d b=%d c=%d  f=%e   w=%e\n",a,b,c,f,w);
 c=a++;     printf(" c=%d  a=%d\n",c,a);  //  c=1;   a=2
 c=++a;     printf(" c=%d  a=%d\n",c,a);  //  c=3;   a=3
 c=++b;     printf(" c=%d  b=%d\n",c,b);  //  c=2;   b=2
 c=(a++)+5; printf(" c=%d  a=%d\n",c,a);  //  c=a+5=8; a=4
 c=(++a)+5; printf(" c=%d  a=%d\n",c,a);  //  c=(a+1)+5=4+1+5=10; a=5
 f++;       printf(" f=%g\n", f);         //  f=1.7+1=2.7
 printf("(++f)+5=%g f=%g\n",(++f)+5,f);   // (++f)+5=(2.7+1)+5=8.7  f=2.7
 printf(" f=%g\n",f);                     //  f=3.7
 printf("(f++)+5=%g f=%g\n",(f++)+5,f);   // (f++)+5=(3.7+1)+5=8.7  f=3.7
 printf(" f=%g\n",f);                     //  f=4.7
 printf(" w=%lg\n",w);                    //  w=3.3
 printf("(--w)+5=%lg w=%lg\n",(--w)+5,w); //  (--w)+5=(3.3-1)+5=7.3  w=3.3
 printf(" w=%lg\n",w);                    //  w=2.3
 printf("(w--)+5=%lg  w=%lg\n",(w--)+5,w);//  (w--)+5=  2.3+5  =7.3  w=2.3
 printf(" w=%lg\n",w);                    //   w=1.1
 return 0;
}
```

При передаче двух аргументов функции **printf**, первый из которых составное выражение со значением префиксной формы (++**f**), сначала передается значение второго аргумента, то есть **f=2.7**, которое и видно на печати. Расчет составного выражения происходит уже после этого так, что изменение ++**f** не зафиксируется в ранее выведенном значении **f**. Однако, изменение **f** по инкременту в составном выражении происходит, что видно из печати в следующей строке.

2. В современном ФОРТРАНе есть понятие **родового имени функции**, означающее, что имя функции обслуживает целый ряд близких по смысловой нагрузке алгоритмов, различающихся, например, типом аргумента. В старых версиях ФОРТРАНа вызов функции **mod** требовал двух аргументов только целого типа, вызов **amod** – исключительно типа **real*4**, **dmod** – **real*8**, заставляя программиста помнить и знать все три имени. Функция **mod** современного ФОРТРАНа способна вызвать нужную из указанных выше функций автоматически в соответствии с типом аргумента за счет механизма **перегрузки функций**, которого не существовало ранее.

3. Напомним, что и в ФОРТРАНе и в СИ тип результата операции деления определяется типом операндов. Так что **5/3=1**.

4. Операция % работает только с операндами целого типа.

5. Письменно обоснуйте результаты работы программы:

```
program tstmod; implicit none           ! Файл tstmod.f90
real a1, a2, a3, b; integer i
write(*,'(a,i1,a,e12.6,a,e19.13)') '  mod(5,3)=', mod(5,3),&
&'  mod(5.0,3.0)=', mod(5.0,3.0),  '  mod(5d0,3d0)=', mod(5d0,3d0)
a1=5.00;  a2=5.12;  a3=5.123456789; b=3.0; write(*,1000) b
do i=1,12
  write(*,'(6e12.4)') a1, amod(a1,b), a2, amod(a2,b), a3, amod(a3,b)
  a1=a1*10; a2=a2*10; a3=a3*10
enddo
1000 format(1x,'b=',e15.7/&
&   6x,'a1',5x,'amod(a1,3.0)',5x,'a2',5x,'amod(a2,3.0)',&
&   5x,'a3',5x,'amod(a3,3.0)')
end
```

```
  mod(5,3)=2  mod(5.0,3.0)=0.200000E+01  mod(5d0,3d0)=0.2000000000000E+01
 b=  0.3000000E+01
      a1     amod(a1,3.0)     a2     amod(a2,3.0)     a3     amod(a3,3.0)
  0.5000E+01  0.2000E+01  0.5120E+01  0.2120E+01  0.5123E+01  0.2123E+01
  0.5000E+02  0.2000E+01  0.5120E+02  0.2000E+00  0.5123E+02  0.2346E+00
  0.5000E+03  0.2000E+01  0.5120E+03  0.2000E+01  0.5123E+03  0.2346E+01
  0.5000E+04  0.2000E+01  0.5120E+04  0.2000E+01  0.5123E+04  0.2457E+01
  0.5000E+05  0.2000E+01  0.5120E+05  0.1996E+01  0.5123E+05  0.5703E+00
  0.5000E+06  0.2000E+01  0.5120E+06  0.1969E+01  0.5123E+06  0.2688E+01
  0.5000E+07  0.2000E+01  0.5120E+07  0.1500E+01  0.5123E+07  0.0000E+00
  0.5000E+08  0.2000E+01  0.5120E+08  0.1000E+01  0.5123E+08  0.1000E+01
  0.5000E+09  0.2000E+01  0.5120E+09  0.0000E+00  0.5123E+09  0.0000E+00
  0.5000E+10  0.2000E+01  0.5120E+10  0.0000E+00  0.5123E+10  0.1000E+01
  0.5000E+11  0.1000E+01  0.5120E+11  0.1000E+01  0.5123E+11  0.2000E+01
  0.5000E+12  0.0000E+00  0.5120E+12  0.0000E+00  0.5123E+12  0.1000E+01
```

6. В СИ аналог функции **dmod** ФОРТРАНа имеет имя **fmod**

```
#include <iostream>
#include <cmath>
using namespace std;
int main(void)
{double a=5.123456789012345;
 cout.setf(ios::right); cout.setf(ios::scientific);
 cout.width(25); cout.precision(16);
cout<<"         a="<<a<<endl; cout<<"fmod(a,3.0)="<<fmod(a,3.0)<<endl;
a=a*pow10(14);
cout<<"         a="<<a<<endl; cout<<"fmod(a,3.0)="<<fmod(a,3.0)<<endl;
return 0;
}
```

1.2.2 Логические операции и операции отношения.

Название операции	Типичное обозначение	ФОРТРАН	С или С+
Отрицание	$\neg$	**.not.**	**!**
Конъюнкция	$\wedge$	**.and.**	**&&**
Дизъюнкция	$\vee$	**.or.**	**||**
Исключающее или		**.xor.**	
Эквивалентность	$\equiv$	**.eqv.**	
Неэквивалентность		**.neqv.**	

В СИ значению **истина** соответствует любое значение отличное от нуля. Поэтому **!777** дает **0** (то есть **ложь**), а **!0** дает **1**.

Название операции	Типичное обозначение	ФОРТРАН	СИ и СИ++
Строго меньше	$<$	**.lt.** или $<$	$<$
Меньше или равно	$\leq$	**.le.** или $<=$	$<=$
Равно	$=$	**.eq.** или $==$	$==$
Не равно	$\neq$	**.ne.** или $/=$	**!=**
Больше или равно	$\geq$	**.ge.** или $>=$	$>=$
Строго больше	$>$	**.gt.** или $>$	$>$

Полезно проанализировать результаты простой программы (см., например, [8]):

```
#include <stdio.h>
int main(void)
{ float v1, v2;   printf("Введите v1:"); scanf("%f",&v1);
                  printf("Введите v2:"); scanf("%f",&v2);
 printf("%g > %g  дает %d\n",v1, v2, v1>v2);
 printf("%g < %g  дает %d\n",v1, v2, v1<v2);
 printf("%g == %g дает %d\n",v1, v2, v1==v2);
 printf("%g >= %g дает %d\n",v1, v2, v1>=v2);
 printf("%g <= %g дает %d\n",v1, v2, v1<=v2);
 printf("  !%g   дает %d\n",v1, !v1);
 printf("  !%g     дает %d\n",v2, !v2);
 printf("%g || %g дает %d\n",v1, v2, v1||v2);
 printf("%g && %g дает %d\n",v1, v2, v1&&v2);     return 0;  }
```

1.2.3 Условная (тернарная) операция СИ ? :

Часто используется для получения более выразительной записи. Иногда называется **тернарной** (ternary – три, тройка, триада, тройной, т.е. состоящей из трех составных частей).

(Операнд1)	**?**	**Операнд2**	**:**	**Операнд3**
(либо целого,либо вещественного типа, либо типа указатель)		вычисляется, если операнд1 не равен нулю		вычисляется если операнд1 равен нулю

Сначала вычисляется **(Операнд1)**, вырабатывающий булево значение: **истину** или **ложь**. Если оно **истина**, то выполняется **(Операнд2)**, иначе **(Операнд3)**.
Примеры, использования тернарной операции:

```
#include <stdio.h>
int main(void)
{ int a, b, c, i;    float u, v, w, p;
 printf("Введите a:"); scanf("%d",&a); printf(" a=%d\n",a);
 printf("Введите b:"); scanf("%d",&b); printf(" b=%d\n",b);
 printf("min(%d,%d)=%d\n",a,b, a<b ? a:b);// Печать минимального из двух чисел.
 printf("max(%d,%d)=%d\n",a,b, a>b ? a:b);// Печать максимального из них.
 printf("a : %s\n", a%2==0 ? " четное  " : " нечетное"); // Четно ли a?
 printf("b : %s\n", b%2==1 ? " нечетное" : "  четное "); // Нечетно ли b?
 u=a; v=b; w=2*u; p=(u+v+w)/2;            // Можно ли построить треугольник
 printf("u=%g  v=%g  w=%g  p=%g  %s\n",   // с длинами сторон u, v и w=2*u
 u, v, w, p, (u<p) && (v<p) && (w<p) ? "Можно" : "Нельзя");
 return 0;
}
```

Результаты двух пропусков этой программы:

```
Введите a:3
 a=3
Введите b:4
 b=4
min(3,4)=3
max(3,4)=4
a :  нечетное
b :   четное
u=3  v=4  w=6  p=6.5  Можно
```

```
Введите a:4
 a=4
Введите b:3
 b=3
min(4,3)=3
max(4,3)=4
a :  четное
b :  нечетное
u=4  v=3  w=8  p=7.5  Нельзя
```

Тернарная операция позволяет модифицировать программу из пункта **1.2.2** так, чтобы получить результат в более удобочитаемом (как считается в [8]) виде:

```
#include <stdio.h>
int main(void)
{ char *ttrue="ИСТИНА", *ffalse="ЛОЖЬ";
 float v1, v2;  printf("Введите v1:"); scanf("%f",&v1);
                printf("Введите v2:"); scanf("%f",&v2);
 printf("%g > %g  это %s\n",v1, v2, v1>v2  ? ttrue : ffalse);
 printf("%g < %g  это %s\n",v1, v2, v1<v2  ? ttrue : ffalse);
 printf("%g == %g это %s\n",v1, v2, v1==v2 ? ttrue : ffalse);
 printf("%g >= %g это %s\n",v1, v2, v1>=v2 ? ttrue : ffalse);
 printf("%g <= %g это %s\n",v1, v2, v1<=v2 ? ttrue : ffalse);
 printf("  !%g   это %s\n", v1,       !v1  ? ttrue : ffalse);
 printf("  !%g   это %s\n", v2,       !v2  ? ttrue : ffalse);
 printf("%g || %g это %s\n",v1, v2, v1||v2 ? ttrue : ffalse);
 printf("%g && %g это %s\n",v1, v2, v1&&v2 ? ttrue : ffalse);
return 0;
}
```

```
Введите v1:1.2          Введите v1:0.0          Введите v1:1.2
Введите v2:0            Введите v2:1.2          Введите v2:1.0
1.2 > 0  это ИСТИНА     0 > 1.2  это ЛОЖЬ       1.2 > 1  это ИСТИНА
1.2 < 0  это ЛОЖЬ       0 < 1.2  это ИСТИНА     1.2 < 1  это ЛОЖЬ
1.2 == 0 это ЛОЖЬ       0 == 1.2 это ЛОЖЬ       1.2 == 1 это ЛОЖЬ
1.2 >= 0 это ИСТИНА     0 >= 1.2 это ЛОЖЬ       1.2 >= 1 это ИСТИНА
1.2 <= 0 это ЛОЖЬ       0 <= 1.2 это ИСТИНА     1.2 <= 1 это ЛОЖЬ
  !1.2   это ЛОЖЬ         !0   это ИСТИНА         !1.2   это ЛОЖЬ
  !0   это ИСТИНА         !1.2   это ЛОЖЬ         !1   это ЛОЖЬ
1.2 || 0 это ИСТИНА     0 || 1.2 это ИСТИНА     1.2 || 1 это ИСТИНА
1.2 && 0 это ЛОЖЬ       0 && 1.2 это ЛОЖЬ       1.2 && 1 это ИСТИНА
```

1.2.4 Операция присваивания

Оператор присваивания ФОРТРАНа нацелен только на пересылку значения, выработанного выражением справа от значка оператора присваивания = , в переменную, имя которой помещено слева от =.

Операция присваивания СИ (обозначается тем же значком =) не только пересылает значение, но и *полагает* это переданное значение своим результатом. Поэтому в СИ, если выгодно, в одном предложении допустима запись сразу нескольких присваиваний:

```
#include <stdio.h>
int main(void)
 { double u,v,w,x,y,z;
   u=v=w=x=y=z=1.3l;
   printf("u=%g  v=%g  w=%g  x=%g  y=%g  z=%g\n",u,v,w,x,y,z);
   return 0; }
```

Кроме того, в СИ есть **комбинированная операция присваивания**, в которой запись операции присваивания комбинируется с записью любой из операций

```
 *  /  +  -  %    <<     >>          &           ^           |
              операции сдвига    поразрядная  поразрядное   поразрядная
                                  конъюнкция  исключающее   дизъюнкция
                                     and          xor          or
```

Правило записи комбинированной операции присваивания:

v op= выражение;.

Здесь **v** – имя переменной, а **op** – одна из указанных операций.

Смысловой эквивалент комбинированной операции :

v=v op выражение;.

Например,

```
#include <stdio.h>                      //   Результат работы программы:
int main(void)                          //
{double u,v,w,x,y,z;                    //
  int i=5;                              //
  u=v=w=x=y=z=1.3;                      //
  u*=2;    printf("u*=2   =%g\n",u);    //           u=u*2   =2.6
  v/=1.3;  printf("v/=1.3 =%g\n",v);    //           v=v/1.3 =1
  w+=5;    printf("w+=5   =%g\n",w);    //           w=w+5   =6.3
  x-=1.05; printf("x-=1.05=%g\n",x);    //           x=x-1.05=0.25
  i%=3;    printf("i%=3   =%d\n",i);    //           i=i%3   =2
  return 0;
}
```

1.2.5 Операция sizeof

Результат операции **sizeof** – количество байт нужное для размещения значения того или иного типа. Операндом операции **sizeof** может быть имя типа, имя переменной или выражение.

Если переменная является массивом, то **sizeof** возвращает число байт, необходимое для размещения всех элементов массива. Например,

```
#include <iostream>
using namespace std;
int main(void)
{ int i,r;  char c;  long int k;  double f;  long double ff;
  char cc[8]; double ar[10]; long double dr[100];
 cout<<"       sizeof(    char  )="<<sizeof(char)<<endl;
 cout<<"       sizeof(   short  )="<<sizeof(short)<<endl;
 cout<<"       sizeof(    int   )="<<sizeof( int)<<endl;
 cout<<"       sizeof( long int )="<<sizeof(long int)<<endl;
 cout<<"       sizeof(   float  )="<<sizeof(  float )<<endl;
 cout<<"       sizeof(  double  )="<<sizeof( double )<<endl;
 cout<<"      sizeof(long double)="<<sizeof(long double)<<endl;
 cout<<"  sizeof( i)="<<sizeof(i)<<endl;
 cout<<"  sizeof( c)="<<sizeof(c)<<endl;
 cout<<"  sizeof( k)="<<sizeof(k)<<endl;
 cout<<"  sizeof( f)="<<sizeof(f)<<endl;
 cout<<"  sizeof(ff)="<<sizeof(f)<<endl;
 cout<<"         sizeof( cc)="<<sizeof(cc)<<endl;
 cout<<"         sizeof( ar)="<<sizeof(ar)<<endl;
 cout<<"         sizeof( dr)="<<sizeof(dr)<<endl;
 cout<<"         sizeof(i+5)="<<sizeof(i+5)<<endl;
 cout<<"         sizeof(f*f)="<<sizeof(f*f)<<endl;
 return 0;
}
```

Результат работы приведенной программы:

```
       sizeof(    char  )=1
       sizeof(   short  )=2
       sizeof(    int   )=4
       sizeof( long int )=4
       sizeof(   float  )=4
       sizeof(  double  )=8
      sizeof(long double)=12
  sizeof( i)=4
  sizeof( c)=1
  sizeof( k)=4
  sizeof( f)=8
  sizeof(ff)=8
         sizeof( cc)=8
         sizeof( ar)=80
         sizeof( dr)=1200
         sizeof(i+5)=4
         sizeof(f*f)=8
```

В современном ФОРТРАНе имеется несколько встроенных функций, родственных СИ-операции **sizeof**. В ФОРТРАНе-95 это — функции **kind** и **size**.

Встроенная ФОРТРАН-функция **kind(x)** возвращает параметр разновидности объекта (константы, переменной и т.д) того или иного типа, то есть количество байт нужное для размещения одного его значения . Например,

```
program testkind; implicit none
integer(1)    i; integer(2) j; integer(4) k; integer(8) m
character(33) c
complex  q; complex*8 r; complex(8) rr; complex*16 r2; real a, aa(5)
real*8 b
write(*,*) kind(i),' ', kind(j),' ', kind(k ),' ', kind( m )
write(*,*) kind(a),' ', kind(b),' ', kind(aa),' ', kind(3.4)
write(*,*) kind(c),' ', kind(1+r2),' ',kind(sin(7.0)),' ', kind(8.3d0)
write(*,*) kind(q),' ', kind(r),' ', kind(r2),' ',kind(rr)
end
```

Результат ее работы

```
1           2           4           8
4           8           4           4
1           8           4           8
4           4           8           8
```

1. Аргумент у ФОРТРАН-функции **kind** не может быть именем типа (как могло быть в случае СИ-операции **sizeof**).

2. Кроме того, когда аргумент **kind** – имя массива, то результат работы – размер в байтах одного элемента, а не всего массива.

3. В старых версиях ФОРТРАНа количество байт, отводимое под значение типа **real**, указывалось при её описании после слова **real** через символ ***** (*звёздочка*) — **real*8**. Такой способ возможен и в современном ФОРТРАНе, хотя рекомендуется использовать круглые скобки — **real(8)**. Однако, если при описании объекта типа **complex*8** восьмёрка указывает полную длину значения типа **complex** (т.е. 4 байта на вещественную часть и четыре байта на мнимую), то описание **complex(8)** указывает количество байт отводимое под каждый компонент (т.е. восемь байт под вещественную и восемь под мнимую). Именно поэтому программе **testkind** значение **kind(r)=4**, но значение **kind(rr)=8**.

Встроенная ФОРТРАН-функция **size(array)** определяет размер массива, т.е.полное число элементов массива **array**.

Полный синтаксис вызова

result=size(array[,dim[,kind]])

(здесь квадратные скобки — не элемент синтаксиса ФОРТРАНа, а указание, что помещённый внутри их фрагмент необязателен). Если при вызове **size** помимо **array** указан ещё и аргумент **dim**, то в качестве результата будет возвращено число элементов массива **array** по указанному в **dim** номеру измерения массива, т.е. длина экстента массива по измерению **dim**. При наличии **kind** третьего аргумента результат, найденный **size**, будет приведён к **integer**-типу, указанному в **kind**. Например,

```
program testsize; implicit none; real a(2,3,4,5)
write(*,*) 'size( a )=',size( a )
write(*,*) 'size(a,1)=',size(a,1); write(*,*) 'size(a,2)=',size(a,2)
write(*,*) 'size(a,3)=',size(a,3); write(*,*) 'size(a,4)=',size(a,4)
write(*,*) 'size( a(2,:,:,:) )=',size( a(2,:,:,:) )
write(*,*) 'size( a(:,3,:,:) )=',size( a(:,3,:,:) )
write(*,*) 'size( a(:,:,4,:) )=',size( a(:,:,4,:) )
write(*,*) 'size( a(:,:,:,5) )=',size( a(:,:,:,5) )
write(*,*) 'size( a(2,3,:,:) )=',size( a(2,3,:,:) )
write(*,*) 'size( a(2,3,4,:) )=',size( a(2,3,4,:) )
write(*,*) 'size( a(2,3,:,5) )=',size( a(2,3,:,5) )
end
```

Результат ее работы

```
 size( a )=         120
 size(a,1)=           2
 size(a,2)=           3
 size(a,3)=           4
 size(a,4)=           5
 size( a(2,:,:,:) )=          60
 size( a(:,3,:,:) )=          40
 size( a(:,:,4,:) )=          30
 size( a(:,:,:,5) )=          24
 size( a(2,3,:,:) )=          20
 size( a(2,3,4,:) )=           5
 size( a(2,3,:,5) )=           4
```

В GNU-расширении ФОРТРАНа 2003 имеется встроенная функция **sizeof(x)**, которая, как и СИ-шная **sizeof**, вычисляет число байт памяти, необходимой для хранения значения аргумента **x**.

```
program test_sizeof; implicit none
real  f4; real(8) f; real(16) ff; integer i
complex*8 c4; complex(8) c8
character cc(8); real(8) ar(10); real(16) dr(100)
write(*,*) 'sizeof( f4)=',sizeof(f4); write(*,*) 'sizeof(  f)=',sizeof( f)
write(*,*) 'sizeof( ff)=',sizeof(ff); write(*,*) 'sizeof( cc)=',sizeof(cc)
write(*,*) 'sizeof( ar)=',sizeof(ar); write(*,*) 'sizeof(i+5)=',sizeof(i+5)
write(*,*) 'sizeof(f*f)=',sizeof(f*f)
write(*,*) 'sizeof( c4)=',sizeof(c4);  write(*,*) 'sizeof( c8)=',sizeof(c8)
end program test_sizeof
```

```
 sizeof( f4)=                   4    !  Результат работы test_sizeof
 sizeof(  f)=                   8    ! ==============================
 sizeof( ff)=                  16
 sizeof( cc)=                   8
 sizeof( ar)=                  80
 sizeof(i+5)=                   4
 sizeof(f*f)=                   8
 sizeof( c4)=                   8
 sizeof( c8)=                  16
```

Результат работы **sizeof(x)**, зависит от разновидности типа аргумента. В ФОРТРАНе и СИ описатели разновидностей типа и обозначаются по разному, и, более того, количество этих разновидностей различно.

Поэтому, если в **gfortran**-сборке исполнимого файла участвуют объектные файлы функций, написанных на СИ (т.е. полученных **gcc**/**g++**-компилятором), то в исходном ФОРТРАН-тексте важно правильно указать требуемый тип разновидности аргумента СИ-функции.

Современный ФОРТРАН использует для этого встроенный модуль **ISO C BINDING** (*International Standard Organization* — привязка ISO C), в котором определены некоторые функции (в частности, и **C_sizeof**) и установлено соответствие ФОРТРАН- и СИ-типов посредством соответствующих именованных констант.

Подробнее об **ISO C BINDING**, о совмещённом (или *интероперабельном*) программировании мы немного узнаем в третьем семестре. Более строгое изложение вопроса будет дано в четвёртом.

1.2.6 Операции сдвига

СИ:

$\ll$	**Сдвиг влево** операнда, указанного слева от знака операции, на число двоичных разрядов,указанных справа.
$\gg$	**Сдвиг вправо** операнда, указанного слева от знака операции, на число двоичных разрядов, указанных справа.

```
#include <iostream>
using namespace std;
int main(void)
{unsigned short int j, k, p, q; signed short int n, r, s;
 cout.fill('.'); p=32768; r=32768;
 cout<<"unsigned short      signed short"<<endl;
 cout<<"       int               int"<<endl;
 for (k=1; k<=16; k++)
   { j=1 << k; n=1 << k; q=p>>k; s=r>>k; cout << 1 << "<<" ;
     cout.width(2); cout << k << " = ";
     cout.width(6); cout << j << "   " << 1 << "<<";
     cout.width(2); cout << k << " = ";
     cout.width(6); cout << n << "   ";  cout << p << ">>" ;
     cout.width(2); cout << k << " = ";
     cout.width(6); cout << q << "   " << r << ">>";
     cout.width(2); cout << k << " = ";
     cout.width(6); cout << q <<endl;
 }
return 0;
}
```

```
unsigned short      signed short
       int               int
1<<.1 = .....2   1<<.1 = .....2   32768>>.1 = .16384   -32768>>.1 = .16384
1<<.2 = .....4   1<<.2 = .....4   32768>>.2 = ..8192   -32768>>.2 = ..8192
1<<.3 = .....8   1<<.3 = .....8   32768>>.3 = ..4096   -32768>>.3 = ..4096
1<<.4 = ....16   1<<.4 = ....16   32768>>.4 = ..2048   -32768>>.4 = ..2048
1<<.5 = ....32   1<<.5 = ....32   32768>>.5 = ..1024   -32768>>.5 = ..1024
1<<.6 = ....64   1<<.6 = ....64   32768>>.6 = ...512   -32768>>.6 = ...512
1<<.7 = ...128   1<<.7 = ...128   32768>>.7 = ...256   -32768>>.7 = ...256
1<<.8 = ...256   1<<.8 = ...256   32768>>.8 = ...128   -32768>>.8 = ...128
1<<.9 = ...512   1<<.9 = ...512   32768>>.9 = ....64   -32768>>.9 = ....64
1<<10 = ..1024   1<<10 = ..1024   32768>>10 = ....32   -32768>>10 = ....32
1<<11 = ..2048   1<<11 = ..2048   32768>>11 = ....16   -32768>>11 = ....16
1<<12 = ..4096   1<<12 = ..4096   32768>>12 = .....8   -32768>>12 = .....8
1<<13 = ..8192   1<<13 = ..8192   32768>>13 = .....4   -32768>>13 = .....4
1<<14 = .16384   1<<14 = .16384   32768>>14 = .....2   -32768>>14 = .....2
1<<15 = .32768   1<<15 = -32768   32768>>15 = .....1   -32768>>15 = .....1
1<<16 = .....0   1<<16 = .....0   32768>>16 = .....0   -32768>>16 = .....0
```

ФОРТРАН:

ISHFT (i,shift)	Возвращает результат сдвига содержимого первого параметра **i** на **shift** битов влево при **shift** $\geq$ **0** (или вправо при **shift** $<$ **0**). Тип результата совпадает с типом первого параметра **i**. Биты, пересекающие левую или правую границы ячейки, исчезают; а биты, вдвигающиеся в ячейку с противоположной стороны, оказываются нулевыми.

```
program testconst; implicit none; integer*2 i,ia, ib, ix, iy, k
ix=1; iy=32767; write(*,1000)
do k=1,16; ia=ishft(ix,k); ib=ishft(iy,-k)
  write(*,1001) k, ia, ia, ia, ia,  ib, ib, ib, ib
enddo
write(*,'(a,i4 )') ' ishft(3, 2)=', ishft(3, 2)
write(*,'(a,i4 )') ' ishft(2,-1)=', ishft(2,-1)
write(*,'(a,3i4)') ' ishft( (/5,5,5/) , (/2,-1,0/) ) =',&
&                  ishft( (/5,5,5/) , (/2,-1,0/) )
 1000 format(1x,'Сдвиг : Числа :   Результат сдвига налево :',&
&                     ' Числа :   Результат сдвига направо ',/&
&            1x,'на  К :       :',27('-'),':',7x,':',27('-')/&
&            1x,'бит   : ОДИН  :',1x,'  двоичной  :  8-й : 16-й :',&
&               ' 32767 :',1x,'  двоичной  :  8-й : 16-й ')
 1001 format(1x,i5,i7,b16, o7, z6, i8,b16,o7,z5)
      end
```

```
 Сдвиг : Числа :   Результат сдвига налево : Числа :   Результат сдвига направо
 на  К :       :---------------------------:       :---------------------------
 бит   : ОДИН  :   двоичной  :  8-й : 16-й : 32767 :   двоичной  :  8-й : 16-й
     1      2              10      2     2   16383  11111111111111  37777 3FFF
     2      4             100      4     4    8191   1111111111111  17777 1FFF
     3      8            1000     10     8    4095    111111111111   7777  FFF
     4     16           10000     20    10    2047     11111111111   3777  7FF
     5     32          100000     40    20    1023      1111111111   1777  3FF
     6     64         1000000    100    40     511       111111111    777  1FF
     7    128        10000000    200    80     255        11111111    377   FF
     8    256       100000000    400   100     127         1111111    177   7F
     9    512      1000000000   1000   200      63          111111     77   3F
    10   1024     10000000000   2000   400      31           11111     37   1F
    11   2048    100000000000   4000   800      15            1111     17    F
    12   4096   1000000000000  10000  1000       7             111      7    7
    13   8192  10000000000000  20000  2000       3              11      3    3
    14  16384 100000000000000  40000  4000       1               1      1    1
    15 -327681000000000000000 100000  8000       0               0      0    0
    16      0              0      0     0       0               0      0    0
 ishft(3, 2)=  12
 ishft(2,-1)=   1
 ishft( (/5,5,5/) , (/2,-1,0/) ) =  20   2   5
```

1.2.7 Поразрядные (побитовые) логические операции.

СИ:		
	&	Поразрядное логическое **И** (and)
	$\wedge$	Поразрядное исключающее **ИЛИ** (xor)
	|	Поразрядное логическое **ИЛИ** (or)
	$\sim$	Поразрядная инверсия (побитовое отрицание)

Поразрядная инверсия — унарная операция (проводится над всеми битами единственного операнда, указанного справа от нее). Ее назначение: инвертирование содержимого операнда (каждый единичный бит заменяется нулевым, а очередной нулевой – единичным). Операция $\sim$ позволяет изменить знак у содержимого переменной целого типа посредством: $\sim \mathbf{k} + \mathbf{1}$, хотя, вероятно, привычнее **-k**. Например,

```
#include <iostream>
using namespace std;
int main()          // Смена знака числа из m.          Результат:
{int m=5;                                              //
 cout<<"m="<<m<<"  -m="<<-m<<"  ~m+1="<<~m+1<<endl; // m=5  -m=-5  ~m+1=-5
 return 0; }
```

Остальные операции – **бинарные**. Результат действия поразрядной операции однозначно определяется соответствующей таблицей истинности:

Операнд	Инверсия
1	**0**
0	**1**

Операнд 1	Операнд 2	AND	OR	XOR
1	**1**	**1**	**1**	**0**
1	**0**	**0**	**1**	**1**
0	**1**	**0**	**1**	**1**
0	**0**	**0**	**0**	**0**

Побитовые операции действуют как обычные **конъюнкция**, **дизъюнкция**, **исключающее ИЛИ**, но по отношению к каждому биту. Так

- **поразрядная дизъюнкция** помещает **1** в те и только те двоичные разряды, в которых эта **1** была хотя бы у одного из операндов.
- **поразрядная конъюнкция** – в те и только те, в которых **1** была обязательно у обоих операндов.
- **Поразрядное исключающее ИЛИ** запишет **1** в те и только те разряды, по которым содержимое операндов различалось.

Обычные (непоразрядные) логические операции просто вырабатывали одно из значений (**1** или **0**), которое служило в дальнейшем ключем повтора или выбора. **Поразрядные операции выгодны**, когда важен не один ответ типа "ДА/НЕТ", а **формировка значения**, представляемого желательным двоичным кодом. Например, пусть необходимо выяснить:

"Есть ли единица в третьем бите (начиная с младшего) двоичного представления числа, хранящегося в переменной целого типа?"

Реализуем в одной программе три варианта решения:

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{int  n, f, k, w;   cout<<"введи целое:"; cin>>n; cout<<" n="<<n<<endl;
 w=n;             // сохранение n;
 k=0;             // 1-й вариант: a) обнуление номера двоичного разряда.
 do  { f = w % 2; //============: b) нашли текущую младщшую двоичную цифру,
       w /=2;                 //: c) подготовились к выделению очередной
       k++;                   //: d) нашли номер ее разряда.
     }
 while ((w!=0) && (k<=2));    //: e) пока он не равен 2 на повтор.
             cout<< ( f ? "Есть!"   : "Нет!  ")<<endl;
 f= n & 4;   cout<< ( f ? "Есть!!"  : "Нет!! ")<<endl;  // <--= 2-й вариант
 f= n & 0x04; cout<< ( f ? "Есть!!!" : "Нет!!!")<<endl; // <--= 3-й вариант.
 k=  17; printf("k=%d %o %0x\n",k, k, k);
 w= 021; printf("w=%d %o %0x\n",w, w, w);
 n=0x11; printf("n=%d %o %0x\n",n, n, n);
 return 0;
}
```

Иногда удобно в тексте программы целые константы записать в восьмеричной или шестнадцатеричной системах счисления. В СИ константа, начинающаяся с **0** рассматривается как восьмеричное целое; а код, начинающийся двух символов **0x** (или **0X**) – как шестнадцатеричное. Так константы $(17)_{10} = (21)_8 = (11)_{16}$ численно равны.

```
введи целое:6   // Результат работы приведенной выше программы.
 n=6            // Вряд ли теперь нужно кого-то убеждать в удобстве
 Есть!          // побитовой конъюнкции.
 Есть!!
 Есть!!!
k=17 21 11      // Демонстрация равеннства (17)_10=(021)_8=(0x11)_16
w=17 21 11
n=17 21 11
```

Аналогичным образом **поразрядная конъюнкция** позволяет быстро находить и остаток от деления целого на любую степень двойки:

```
#include <iostream>
using namespace std;
int main()
{ cout<<"38 & 1="<< (38 & 1)<<endl; // то есть  38 - четное число.
                                    // т.к у него в разряде единиц НУЛЬ.
  cout<<"38 % 2="<< (38 % 2)<<endl; // Конечно, четность можно выяснить и так.
  cout<<"(37 &  1)= "<<(37 &  1)<<" = (37 %  2) = "<< (37 %  2)<<endl;
  cout<<"(38 &  3)= "<<(38 &  3)<<" = (38 %  4) = "<< (38 %  4)<<endl;
  cout<<"(38 &  7)= "<<(38 &  7)<<" = (38 %  8) = "<< (38 %  8)<<endl;
  cout<<"(38 & 15)= "<<(38 & 15)<<" = (38 % 16) = "<< (38 % 16)<<endl;
  return 0; }

38 & 1=0   // Результат работы.  Если число делится нацело на 2**n, то
38 % 2=0                      // все младшие цифры его двоичного пред-
(37 &  1)= 1 = (37 %  2) = 1  // ставления от 0-й до (n-1)-ой нулевые.
(38 &  3)= 2 = (38 %  4) = 2  // Если же не делится, то именно они и
(38 &  7)= 6 = (38 %  8) = 6  // есть величина остатка.
(38 & 15)= 6 = (38 % 16) = 6
```

Поразрядное исключающее ИЛИ получает код отличия содержимого одной переменной от содержимого другой переменной. Поэтому, изменив содержимое одной из них на этот код, всегда можно повторным применением этой же операции к тем же переменным восстановить замененное, то есть $(\mathbf{a}\hat{}\mathbf{b})\hat{}\mathbf{a} \equiv \mathbf{a}$.
Подобный прием очень полезен на практике. Например, пусть надо обменять содержимым две переменные **a** и **b** целого типа, не используя дополнительной рабочей переменной. Реализуем решение двояко:

на базе арифметических операций + и -	**a=17 b=43**	на базе поразрядного исключающего ИЛИ ^	**a=010001 b=101011**
a=a+b	a=60	$a = a\hat{}b$	**a=111010**
b=a-b	b=17	$b = a\hat{}b$	**b=010001**
a=a-b	a=43	$a = a\hat{}b$	**a=101011**

в одной и той же программе (для сравнения результатов).

```
#include <iostream>
using namespace std;
int main()
{ short int a, b, c, d;
  cout<<"Введите два целых значения"<<endl;
  cin>>a>>b;
  c=a; d=b; cout<<" a="<< a<<" b="<< b <<"     c="<< c<<" d="<< d <<endl;
  a+=b;   c^=d; cout<<"  a+=b = "<<a<<"    c^=d = "<<c<<endl;
  b=a-b; d=c^d; cout<<" b=a-b = "<<b<<"   d=c^d = "<<d<<endl;
  a-=b;   c^=d; cout<<"  a-=b = "<<a<<"    c^=d = "<<c<<endl;
            cout<<" a="<< a<<" b="<< b <<"     c="<< c<<" d="<< d <<endl;
  return 0;
}

Введите два целых значения
  a=43 b=17      c=43 d=17    // Интересно посмотреть на работу
  a+=b = 60      c^=d = 58    // этой программы, когда сумма чисел
 b=a-b = 43     d=c^d = 43    // выходит за границы допустимого
  a-=b = 17      c^=d = 17    // для выбранного типа данных диапазона
 a=17 b=43      c=17 d=43    // например, a=25000 и b=15000.

Введите два целых значения
 a=25000 b=15000     c=25000 d=15000
  a+=b = -25536     c^=d = 23344
 b=a-b = 25000    d=c^d = 25000
  a-=b = 15000     c^=d = 15000
 a=15000 b=25000     c=15000 d=25000
```

ФОРТРАН вместо упомянутых операций СИ содержит соответствующие **встроенные функции** с параметрами **целого типа**.

iand(i, j)	– поразрядная конъюнкция.
ieor(i, j)	– действие поразрядного исключающего ИЛИ.
ior(i, j)	– поразрядная дизъюнкция (неисключающее ИЛИ)
not(i)	– поразрядная инверсия (побитовое отрицание)

Современный ФОРТРАН позволяет записывать целые числа и в восьмеричной, и в шестнадцатеричной, и в **двоичной** системах счисления:

```
program tstbit1; implicit none ! Задание констант в 2-, 8-, 16-й системах
integer a, b,c, d,e, f,g       !                                счисления:
a=10; b=B'1010'; c=b'1010'     ! Результат работы программы :
d=O'12'  ; e=o'12'             ! a= 10 b= 10 c= 10 d= 10 e= 10 f= 10 g= 10
f=Z'A'   ; g=z'A'              !=========================================
write(*,'(7(a,i3))') ' a=', a,' b=',b,' c=', c,' d=',d,&
&                    ' e=', e,' f=',f,' g=', g
end
```

```
program testbit2; implicit none ! ФОРТРАН-программа, выясняющая есть ли
integer  n, f, k, w             !  у введенного целого единица в разряде
character*8 a(0:1) /'  Нет!  ','  Есть! '/ !       его   двоичных сотен.
write(*,*) 'введи целое:'; read(*,*) n; write(*,*) ' n=',n
w=n;                          ! 1-ый вариант:  сохранили n
k=0;                          !     a) обнуление номера двоичного разряда.
do; f = mod(w, 2)             !     b) нашли текущую младщшую двоичную цифру,
  w = w / 2                   !     c) подготовились к выделению очередной
  k=k+1                       !     d) нашли номер ее разряда.
  if (w.eq.0 .or. k>2) exit   !     e) как только найден бит двоичных
enddo                         !        сотен --> выйти из данного цикла.
write(*,*)  a(f)
f=iand(n, 4);    write(*,*) a(f/4)  ! 2-й вариант
f=iand(n, z'4'); write(*,*) a(f/4)  ! 3-й вариант.
write(*,1001) ' n=',n         !  Двоичный код однобайтового числа
write(*,1001) ' 4=',4         !  можно вывести по формату b8,
1001 format(1x,a,3x,b8)       !  а четырёхбайтового по формату b32.
end
```

Варианты пропуска программы для n=16 и n=36

```
 введи целое:                              введи целое:
 16                                        36
   n=          16                            n=          36
    Нет!                                      Есть!
    Нет!                                      Есть!
    Нет!                                      Есть!
   n=      10000  <--    двоичные    -->   n=     100100
   4=        100  <-- коды на печати -->   4=        100
```

```
program testbit5     ! Реализация на ФОРТРАНе обмена содержимым двух
implicit none        ! целых переменных без третьей переменной двумя
integer a, b, c, d;  ! способами: арифметическим и посредством
write(*,*) 'Введите два целых значения';     ! исключающего ИЛИ.
read (*,*) a, b                              !.....................
c=a; d=b;
write(*,*) ' a=', a, ' b=',b,'    c=',c,' d=',d
a=a+b;  c=ieor(c,d); write(*,*)' a=a+b = ',a,'    c=ieor(c,d)=',c
b=a-b;  d=ieor(c,d); write(*,*)' b=a-b = ',b,'    d=ieor(c,d)=',d
a=a-b;  c=ieor(c,d); write(*,*)' a=a-b = ',a,'    c=ieor(c,d)=',c
write(*,*) ' a=', a, ' b=',b,'    c=',c,' d=',d
end
```

```
 Введите два целых значения                    <--= Результат пропуска
 17  43                                             программы testbit5
   a= 17 b= 43    c= 17 d= 43
   a=a+b =  60    c=ieor(c,d)= 58
   b=a-b =  17    d=ieor(c,d)= 17
   a=a-b =  43    c=ieor(c,d)= 43
   a= 43 b= 17    c= 43 d= 17
```

1.2.8 Ещё один пример на побитовые

Решаем задачу подсчёта числа единиц в записи целого значения после его перевода в двоичную систему счисления). Обычное школьное решение: после ввода целого значения **N** алгоритм сводят к циклу с предусловием, не используя побитовые операции (здесь приведены соответствующие ФОРТРАН- и СИ-программы):

```
program testbit0; implicit none; integer, parameter :: kmax=100000000
integer N, m, k, s; real t0, t1
read(*,*) N; write(*,*) 'N=',N; call cpu_time(t0)
do k=1,kmax
  s=0; m=N; do while (m>0); s=s+mod(m,2);  m=m/2; enddo
enddo
call cpu_time(t1);    write(*,*) ' s=',s,' time=',t1-t0
end program testbit0
```

```
#include <stdio.h>
#include <time.h>
int main(void)
{ const int kmax=100000000; int N, m, k, s; clock_t t0, t1;
  scanf("%d",&N); printf(" N=%d\n",N);
  t0=clock();
  for (k=1; k<=kmax; k++) { s=0; m=N; while (m>0) {s=s+m%2; m=m/2;} }
  t1=clock();
  printf(" s=%d time=%7.2f",s, (double)(t1-t0)/CLOCKS_PER_SEC); return 0;
}
```

Время работы алгоритмов **testbit0** зависит от числа цифр в числе. В таблице ниже, в колонках **Ф** (ФОРТРАН) и **С** (СИ), приведены в секундах временные затраты $\mathbf{10^8}$-кратного расчёта количества единиц для двух значений **N=16** и **N=2147483647**:

Оптимизация:			-O0		-O1	
Алгоритм	N	s	Ф	СИ	Ф	СИ
testbit0	$\mathbf{2^4}$	1	2.6	2.0	1.5	1.2
	$\mathbf{2^{31}-1}$	31	14.5	14.4	8.1	8.7

Вообще говоря, ничего удивительного. Для **N=16** тело цикла будет повторяться 5 раз, а для **N=2147483647** — 31 раз. Поэтому временны́е затраты для последнего **N** должны быть примерно в шесть раз больше чем для первого, что и наблюдается. Теперь продемонстрируем насколько быстрее работает приведённый в [16, 15] алгоритм, который использует побитовые операции.

Время его работы не зависит от количества цифр во вводимом числе, так, что результат для любого 32-битного целого получается за время чуть ли не вдвое меньшее времени, затраченного **testbit0** при **N=16**.

```
program testbit1; implicit none; integer, parameter :: kmax=100000000
integer N, m, k, s; real t0, t1
read(*,*) N; write(*,*) 'N=',N
call cpu_time(t0)
do k=1,kmax
    s=N;
    s=iand(s,z'55555555')+iand(ishft(s, -1),z'55555555')
    s=iand(s,z'33333333')+iand(ishft(s, -2),z'33333333')
    s=iand(s,z'0f0f0f0f')+iand(ishft(s, -4),z'0f0f0f0f')
    s=iand(s,z'00ff00ff')+iand(ishft(s, -8),z'00ff00ff')
    s=iand(s,z'0000ffff')+iand(ishft(s,-16),z'0000ffff');
enddo
call cpu_time(t1); write(*,*) ' s=',s,' time=',t1-t0
end program testbit1
```

```
#include <stdio.h>
#include <time.h>
int main(void)
{ unsigned int w, N;
  const int kmax=100000000; int k, s; clock_t t0, t1;
scanf("%i",&N);  printf(" w=%i  %x\n",N,N);
t0=clock();
for (k=1;k<=kmax;k++)
  {
    s=N; s=(s&0x55555555)+((s>> 1)& 0x55555555);
         s=(s&0x33333333)+((s>> 2)& 0x33333333);
         s=(s&0x0f0f0f0f)+((s>> 4)& 0x0f0f0f0f);
         s=(s&0x00ff00ff)+((s>> 8)& 0x00ff00ff);
         s=(s&0x0000ffff)+((s>>16)& 0x0000ffff);
  }
t1=clock();
printf(" s=%d time=%7.2f",s, (double)(t1-t0)/CLOCKS_PER_SEC); return 0;
}
```

Соответствующие результаты для тех же параметров расчёта:

Оптимизация:			-O0		-O1	
Алгоритм	N	s	Ф	СИ	Ф	СИ
testbit1	$\mathbf{2^4}$	1	1.1	1.3	0.06	0.05
	$\mathbf{2^{31}-1}$	31	1.1	1.3	0.06	0.05

Идея алгоритма testbit1. На входе четырёхбайтовое целое **N** (или s), записанное 32-мя двоичными цифрами: какие-то из них 0, какие-то 1. Количество последних нужно узнать.

1. Пусть исходное $\mathbf{N} = \mathbf{10111100011000110111111011111111}_{(\mathbf{2})}$ (см. [16]).

2. Любую цифру двоичной записи можно рассматривать и как число, обозначающее сколько единиц находится в данном двоичном разряде: 0 — нуль единиц; 1 — одна единица.

3. Побитовая операция умножения и сдвига позволяют получить в качестве слагаемых наборы битов из чётных и нечётных позиций исходного **N**. Их одно сложение даст сумму каждой из 16 пар соседних битов, т.е. количество единиц в соответствующей паре:

```
1  2  2  0  1  1  0  2  1  2  1  2  2  2  2  2
                                                 Двоичный код этих 16 сумм
01 10 10 00 01 01 00 10 01 10 01 10 10 10 10 10 Именно его даёт алгоритм.
```

4. Аналогично за одно сложение организуется одновременный расчёт 8 сумм (по сумме на каждую пару соседних чисел, полученных в предыдущем пункте), находя, тем самым, количество единиц в соответствующих тетрадах исходного двоичного представления **N**.

```
   3    2    2    2    3    3    4    4
                                        Двоичные коды этих 8 чисел.
 0011 0010 0010 0010 0011 0011 0100 0100  Именно его получает алгоритм
```

5. Следующее очередное сложение находит суммы четырёх пар соседних чисел, полученных в предыдущем пункте. В итоге находятся количества единиц в каждом байте исходного представления N:

```
   5   4    6   8
                     Двоичные коды этих четырёх чисел.
 0101 0100 0110 1000  Именно их получает алгоритм.
```

6. По той же схеме за одно сложение находятся суммы 9 и 14

```
   9    14
                      Двоичные коды этих двух чисел.
 1001  1110           Именно их получает алгоритм.
```

7. Наконец, последнее (пятое) сложение даёт

```
 23       Одно число равное числу единиц в исходном N
          Двоичный код этого числа.
10111    Именно его получает алгоритм  (16+7=23).
```

Как получаются очередные слагаемые?

1. Побитовые операции выполняются над всеми битами ячейки одновременно. Побитовое умножение 32-битовой маски

   01010101010101010101010101010101010101010101010101010101010101

   (или 55555555 в шестнадцатеричной записи) на исходное s получит значение, которое будет иметь единицы в тех нечётных двоичных разрядах, в которых они были в исходном значении s:

```
   s=bc637eff 1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 Нашли все
   m=55555555 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 единицы из
------------------------------------------------------------------------------ нечётных
 s&m=14415455 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 битов.
```

2. При подвижке исходного значения s вправо на один бит (СИ-шная операция $s >> 1$) содержимое всех чётных битов сместится в соседние нечётные, так что умножение результата подвижки на прежнюю маску выделит единицы из чётных битов исходного N:

```
 s>>1=5e31bf7f 0 1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 Сдвиг на 1 бит
    m=55555555 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 Нашли все
------------------------------------------------------------------------------ единицы из
s>>1&m=54111555 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 чётных битов
```

3. Если теперь в качестве нового текущего значения s взять сумму

   $$s = s\&0x55555555 + (s >> 1)\&0x55555555\ ,$$

   то в очередной паре его битов будет находиться двубитовое двоичное число равное количеству единиц,содержащемуся в соответствующих складываемых битах исходного двоичного представления N. Нули в чётных битах маски в обоих случаях побитового умножения обеспечивают в чётных битах результата наличие нуля. Так что появление на его месте *единицы в уме* (если оно потребуется при двоичном сложении) корректно представит найденные двоичные суммы однобитовых чисел.

4. После сложения 32-битовая ячейка будет хранить 16 пар двузначных двоичных чисел (количества единиц в складываемых битах):

```
   s&m=14415455 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 нечётные биты;
s>>1&m=54111555 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 чётные биты
 + ------------------------------------------------------------------------------
    s=685269aa 0 1 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0 1 1 0 1 0 0 1 1 0 1 0 1 0 1 0 Их сумма в виде
               /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  / 16 пар двузначных
    s=       01 10 10 00 01 01 00 10 01 10 10 01 10 10 10 10 двоичных чисел.
  23= 5*1+9*2= 1 + 2 + 2 + 0 + 1 + 1 + 0 + 2 + 1 + 2 + 2 + 1 + 2 + 2 + 2 + 2
```

5. Значение самого младшего бита, исчезнувшее при сдвиге, не потребуется уже никогда, так как оно вошло в сумму единиц, хранящуюся в самом правом двузначном двоичном числе.

6. Складываем двузначные числа, выделяемые новой маской, которая нацелена на выделение двузначных двоичных чисел:

   00110011001100110011001100110011001100110011001100110011001100110011

   (или 33333333 в шестнадцатеричной записи). Результат её побитового умножения на текущее s — 8 двузначных двоичных чисел, находящихся в нечётных (считая справа) парах битов:

```
  s=685269aa 01 10 10 00 01 01 00 10 01 10 10 01 10 10 10 10 Находим числа
  m=33333333 00 11 00 11 00 11 00 11 00 11 00 11 00 11 00 11 равные коли-
---------------------------------------------------------- честву единиц
s&m=20122122 00 10 00 00 00 01 00 10 00 10 00 01 00 10 00 10 в нечётных
               /     /     /     /     /     /     /     / парах двоичных
           0010  0000  0001  0010  0010  0001  0010  0010  чисел текущего
             2     0     1     2     2     1     2     2   значения S
```

7. После сдвига текущего s вправо на два бита содержимое всех чётных двубитовых чисел переместится в соседние нечётные, так что умножение результата этого смещения на текущую маску выделит двузначные двоичные числа, имеющиеся в текущем s в чётных двубитовых парах:

```
    s>>2=1a149a6a 00 01 10 10 00 01 01 00 10 01 10 10 01 10 10 10 Сдвиг.
       m=33333333 00 11 00 11 00 11 00 11 00 11 00 11 00 11 00 11 Выборка
------------------------------------------------------------------ чисел
(s>>2)&m=12101222 00 01 00 10 00 01 00 00 00 01 00 10 00 10 00 10   из
                    /     /    /     /     /     /     /     / чётных
                 0001  0010  0001  0000  0001  0010  0010  0010  пар
                   1  0  2  0  1  0  0  0  1  0  2  0  2  0  2
```

8. Работа оператора $\mathbf{s = s\&0x33333333 + s >> 2\&0x33333333}$; получает в каждой из восьми тетрад количества единиц, которые хранятся в соответствующих тетрадах исходного значения N.

```
    s&m=20122122  0010 0000 0001 0010 0010 0001 0010 0010  Сложение чисел
(s>>2)&m=12101222  0001 0010 0001 0000 0001 0010 0010 0010  из нечётных и
 +   ---------------------------------------------------   чётных пар
      s=32223344  0011 0010 0010 0010 0011 0011 0100 0100
                    3  + 2 +  2 +  2 +  3 +  3 +  4 +  4  =  23.
```

9. Новая маска **00001111000011110000111100001111** (шестнадцатеричное $0f0f0f0f$) для выделения нечётных тетрад

```
      s=32223344  0011 0010 0010 0010 0011 0011 0100 0100   Выделяем
      m=0f0f0f0f  0000 1111 0000 1111 0000 1111 0000 1111   нечётные
      ---------------------------------------------------  тетрады
    s&m=02020304  0000 0010 0000 0010 0000 0011 0000 0100

   s>>4=03222334  0000 0011 0010 0010 0010 0011 0011 0100   Выделяем
      m=0f0f0f0f  0000 1111 0000 1111 0000 1111 0000 1111   чётные
  -----------------------------------------------------
(s>>4)&m=03022304  0000 0011 0000 0010 0000 0011 0000 0100
```

10. Так что после сложения

$$\mathbf{s = s\&0x0f0f0f0f + s >> 4\&0x0f0f0f0f};$$

получаем в каждой из четырёх восьмибитовых значений количества единиц, которые храняться в соответствующих байтах исходного значения N.

```
    s&m=02020304  0000 0010 0000 0010 0000 0011 0000 0100
(s>>4)&m=03022304  0000 0011 0000 0010 0000 0011 0000 0100
  +   -----------------------------------------------------
      s=          0000 0101 0000 0100 0010 0110 0000 1000
                         /         /         /         /
                  00000101  00000100  00000110  00001000
                       5    +    4    +    6    +   8   =  23
```

11. Очередная маска для выделения восьмибитовых чисел

$$\mathbf{00000000111111110000000011111111}$$

(шестнадцатеричная запись $00ff00ff$) даёт:

```
      s= 00000101  00000100  00000110  00001000
      m= 00000000  11111111  00000000  11111111
         ------------------------------------
   s&m= 00000000  00000100  00000000  00001000

  s>>8= 00000000  00000101  00000100  00000110
      m= 00000000  11111111  00000000  11111111
(s>>8)&m= 00000000  00000101  00000000  00000110
```

12. Так что после сложения

$$\mathbf{s = s\&00ff00ff + s >> 4\&0x00ff00ff};$$

получаем в каждом из двух шестнадцатибитовых двоичных числах количества единиц, которые храняться в соответствующих полусловах байтах исходного значения N.

```
    s&m= 0000000  00000100  00000000  00001000
(s>>8)&m= 0000000  00000101  00000000  00000110
  +   ----------------------------------------
      s= 0000000  00001001  00000000  00001110
                         /                  /
          000000000001001    0000000000001110
                      9    +              14   =  23
```

13. Наконец, используя маску **00000000000000001111111111111111** (шестнадцатеричная запись $00ff00ff$) получаем возможность сложить два шестнадцатибитовых числа:

```
       s= 0000000000001001 0000000000001110
       m= 0000000000000000 1111111111111111
          --------------------------------
     s&m= 0000000000000000 0000000000001110

   s>>16= 0000000000000000 0000000000001001
       m= 0000000000000000 1111111111111111
(s>>16)&m= 0000000000000000 0000000000001001

     s&m= 0000000000001001 0000000000001110
(s>>16)&m= 0000000000000000 0000000000001001
  +   ----------------------------------
          0000000000000000 0000000000010111 =1*2^4+7=16+7=23
```

1.2.9 Неявное преобразование типов в выражении

Неявное преобразование типов (т.е. при отсутствии в программе каких-либо явных инструкций по этому поводу) проводится, когда операнды разных типов:

a) **участвуют в бинарной операции**. Интуитивно ясно, что тип арифметического результата должен совпадать с типом того из операндов, значение которого требует в машинном представлении большего количества байт.

b) **находятся слева и справа от знака операции присваивания**. Иногда может потребоваться преобразование и более "*длинного*" данного в более "*короткое*"; например, получение значения типа **int** из типа **double**).

Для выбора направления неявного преобразования операндов определенно понятие **старшинства** или **ранга** типа. Всегда перед проведением операции операнд низкого ранга преобразуется к типу операнда более высокого (старшего).

СИ	**ФОРТРАН**	**Примечания**
	complex(16)	самый старший
	complex(8)	
long double	**real(16)**	
double	**real(8)**, **double precision**	
float	**real(4)** real	
long int	**integer(8)**	
unsigned long	Нет аналога *unsigned*	В ФОРТРАНе
int	**integer(4)**, integer	все целые
unsigned int	Нет аналога *unsigned*	типы
short int	**integer(2)**	знаковые
unsigned short	Нет аналога *unsigned*	
char	**integer(1)**	самый младший

Часто человек забывает, что операции деления для операндов целых и вещественных типов – разные, хотя и записывются одним и тем же значком (/). Вид операции выясняется из контекста: если оба операнда целого типа, то – деление целочисленное; если же хотя бы один из операндов – вещественный, то – вещественное. В приведённой ниже программе

демонстрируются как эффект целочисленного и вещественного делений при неявном приведении типов, так и эффект переполнения при попытке записи значения типа **signed int** в переменную типа **signed char**.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{ double x=1.0;  signed char m=3, n=5;
printf("   m/n*x=%f\n",m/n*x);   //  результат деления нацело m/n=0
printf(" x*(m/n)=%f\n",x*(m/n)); //  -"--"--"--"--"--"--"--"--"-
printf("   m*x/n=%f\n",m*x/n);   //  m и n неявно преобразуются к double
printf("   x*m/n=%f\n",x*m/n);   //  -"--"--"--"--"--"--"--"--"-
x=m*50; printf(" x=m*50=%f\n",x);//  m --> int --> double
n=m*50; printf(" n=m*50=%d\n",n);//  m --> int --> signed char,
return 0;
}
```

Результат работы программы:

```
   m/n*x=0.000000
 x*(m/n)=0.000000
   m*x/n=0.600000
   x*m/n=0.600000
 x=m*50=150.000000
 n=m*50=-106
```

Здесь переменные **m** и **n** однобайтовые знаковые, т.е. наибольшее положительное число, которое может быть в них записано равно

$$(\mathbf{127})_{\mathbf{10}} = (\mathbf{01111111})_{\mathbf{2}}.$$

Но значение $\mathbf{m * 50 = 3 * 50 = 150} = (\mathbf{10010110})_{\mathbf{2}}$ требует для своей двоичной записи (как положительное число) более одного байта (например, два):

$$(0000000010010110)_2 = 1\cdot 2^7 + 1\cdot 2^4 + 1\cdot 2^2 + 1\cdot 2^1 = 128 + 16 + 4 + 2 = 150.$$

Поскольку его нужно трактовать как значение типа **signed char**, то старшая единица будет трактоваться как знак числа, а остальные биты как дополнительный код некоторого отрицательного числа. Для получения его десятичного эквивалента все его биты следует инвертировать: 01101001 (получая обратный код) и добавить к последнему единицу: $(01101010)_2 = -(64 + 32 + 8 + 2) = -(106)_{10}$

Аналогичная ФОРТРАН-программа (gfortran) и ее результаты:

```
program tstpriv1; implicit none
integer(2) nn; integer(1) m, n; real(8) x
x=1d0; m=3; n=5                    !   РЕЗУЛЬТАТ:
write(*,*) '   m/n*x=',m/n*x       !    m/n*x=   0.00000000000000
write(*,*) ' x*(m/n)=',x*(m/n)     !  x*(m/n)=   0.00000000000000
write(*,*) '   m*x/n=',m*x/n       !    m*x/n=  0.59999999999999998
write(*,*) '   x*m/n=',x*m/n       !    x*m/n=  0.59999999999999998
x=m*50; write(*,*) '  x=m*50=',x   !   x=m*50=   150.000000000000
n=m*50; write(*,*) '  n=m*50=',n   !   n=m*50= -106
write(*,1000)  transfer(n,nn),&    ! nn = 0000000010010110 = 150
&              transfer(n,nn)
1000 format(1x,' nn= ',B16.16 ,' = ',i4)
end
```

Здесь встроенная функция **transfer(x, y[,size])** (см. [2]), преобразует данное **x** одного типа в данное **y** другого типа без изменения их физического представления (т.е. значения двоичных разрядов обоих типов совпадают).

Неявные СИ-преобразования младших типов в старшие и обратно проводятся по определенным правилам. Новые старшие разряды целых типов для отрицательного значения заполняются знаком числа, так как отрицательные числа записываются в дополнительном коде. Преобразование целого в вещественное и обратно выполняется специальными функциями. Переход от значения старшего типа к значению младшего корректен, когда результат не оказывается вне диапазона, допускаемого младшим типом. Аналогично и переход от значения младшего типа к значению старшего корректен, если переводимое значение находится в пределах значений допускаемых старшим типом. Например, попробуйте пропустить программу:

```
#include <stdio.h>
int main()                                      //  Результат:
{  signed char p;  unsigned char q;             //
   p=-126; q=p; printf(" p=%d  q=%d\n",p,q); //  p=-126  q=130
   q= 254; p=q; printf(" p=%d  q=%d\n",p,q); //  p=-2  q=254
  return 0;
}
```

1.2.10 Операции явного приведения типа

Смешивание в одном выражении операндов разных типов не рекомендуется, так как на их неявное преобразование тратится время. Кроме того, невнимательное отношение к записи выражения чревато и побочными эффектами (см. предыдущий пункт). Поэтому лучше избегать смешивания типов в выражении, а при необходимости осуществлять **явное приведение операндов к нужному типу**.

В **СИ** для этой цели служит операция **(имя нового типа)**, которая применяется к стоящему после закрывающейся круглой скобки операнду старого типа (но можно и так **имя нового типа (операнд старого)**):

```
#include <iostream>
using namespace std;
int main(void)
{ int    i=1, j=3;  float   f1, f2;  double  d1, d2;
 f1=i / j; f2=(float)(i/j);            cout<<" f1="<<f1<<" f2="<<f2<<endl;
           f2=(float)i/j;              cout<<"      f2="<<f2<<endl;
           f2=(float) i / (float) j;   cout<<"      f2="<<f2<<endl;
             f2=float(i)/j;            cout<<"      f2="<<f2<<endl;
 d1=i / j;   d2=(double)(i/j);         cout<<" d1="<<d1<<" d2="<<d2<<endl;
             d2=(double)i/j;           cout<<"      d2="<<d2<<endl;
             d2=(double)i / (double)j; cout<<"      d2="<<d2<<endl;
             d2=(double)i / double(j); cout<<"      d2="<<d2<<endl;
 cout.setf(ios::floatfield, ios::scientific); // флаги: вещ., с порядком
 cout<<" f1="<<f1<<" f2="<<f2<<endl;  cout<<" d1="<<d1<<" d2="<<d2<<endl;
 cout.precision(15);                          // число цифр мантиссы
 cout<<" f1="<<f1<<" f2="<<f2<<endl; cout<<" d1="<<d1<<" d2="<<d2<<endl;
 cout.setf(ios::fixed);                       // флаг: форма с ФИКС. запятой
      cout<<" f1="<<f1<<" f2="<<f2<<endl; cout<<" d1="<<d1<<" d2="<<d2<<endl;
return 0; }
```

Форма вывода вещественных чисел через поток **cout** задаётся явной установкой нужных **флагов формата** через метод **setf** класса **ios** пространства имен **std** (см., [9] гл. 8 или [8] гл. 13). Вывод программы, приведённой выше:

```
 f1=0 f2=0            |
      f2=0.333333     |
      f2=0.333333     | f1=0.000000e+00 f2=3.333333e-01
      f2=0.333333     | d1=0.000000e+00 d2=3.333333e-01
 d1=0 d2=0            | f1=0.000000000000000e+00 f2=3.333333432674408e-01
      d2=0.333333     | d1=0.000000000000000e+00 d2=3.333333333333333e-01
      d2=0.333333     | f1=0 f2=0.333333343267441
      d2=0.333333     | d1=0 d2=0.333333333333333
```

Подобная ФОРТРАН-программа и ее результаты:

```
program testpriv; implicit none
integer i /1/, j /3/
real   f1, f2;
real*8 d1, d2, d3;
f1= i / j; f2=float(i/j);  write(*,*) ' f1=',f1,' f2=',f2;
           f2= float(i)/j; write(*,*) '         f2=',f2;
           f2=float(i)/float(j); write(*,*) ' float:  f2=',f2;
           f2=real(i)/real(j);   write(*,*) ' real :  f2=',f2;
                                 write(*,1000) f1, f2
           write(*,'('' f1='',e15.7,'' f2='',e15.7)/') f1, f2
d1= i / j; d2=dfloat(i/j); write(*,*) ' d1=',d1,'  d2=',d2;
           d2=dfloat(i)/j; write(*,*) '          d2=',d2;
           d2= dfloat(i)/dfloat(j); write(*,*) ' dfloat:  d2=',d2
           d2= dble(i)  / dble (j); write(*,*) ' dble  :  d2=',d2
           d2= dreal(i) / dreal(j); write(*,*) ' dreal :  d2=',d2;
           d2= dfloat(i)/dfloat(j); write(*,1001) d2
           d2= dble(i)  / dble (j); write(*,1002) d2
           d2= dreal(i) / dreal(j); write(*,1003) d2
           d2= float(i) / float (j); write(*,1004) d2;
           d2= real(i)  / real(j);   write(*,1005) d2;
 1000 format(1x,' f1=', e15.7,' f2=', e15.7)
 1001 format(1x,' dfloat : d2=',d20.13)
 1002 format(1x,' dble   : d2=',d20.13)
 1003 format(1x,' dreal  : d2=',d20.13)
 1004 format(1x,' float  : d2=',d20.13)
 1005 format(1x,' real   : d2=',d20.13)
end
```

```
 f1=  0. f2=  0.                  ! В g77 преобразовать целое значение в
         f2=  0.333333343         ! в вещественное одинарной точности можно
         float:  f2=  0.333333343 ! посредством функций float и real.
 real :  f2=  0.333333343
 f1=  0.0000000E+00 f2=  0.3333333E+00
 f1=  0.0000000E+00 f2=  0.3333333E+00
 d1=  0.  d2=  0.                  ! Преобразовать целое в вещественное
          d2=  0.333333333         ! удвоенной точности можно посредством
 dfloat:  d2=  0.333333333         ! функций  dfloat, dble, dreal
 dble  :  d2=  0.333333333         !
 dreal :  d2=  0.333333333
dfloat : d2= 0.3333333333333E+00   ! но можно и посредством float и real,
dble   : d2= 0.3333333333333E+00   ! которые являются родовыми.
dreal  : d2= 0.3333333333333E+00
float  : d2= 0.3333333333333E+00
real   : d2= 0.3333333333333E+00
```

Если попробовать откомпилировать эту программу на **gfortran**, то получим:

```
 In file testpriv.for:18
           d2= dreal(i) / dreal(j); write(*,*) ' dreal :  d2=',d2;
                     1
 Error: Type of argument 'a' in call to 'dreal' at (1) should be COMPLEX(8),
 not INTEGER(4)
In file testpriv.for:22
            d2= dreal(i) / dreal(j); write(*,1003) d2
                      1
Error: Type of argument 'a' in call to 'dreal' at (1) should be COMPLEX(8),
not INTEGER(4)
```

Аргумент у **dreal** должен быть обязательно типа **complex(8)**. Другими словами, имя **dreal** – специфическое, это имя функции, нацеленной на работу исключительно со значениями комплексного типа. В **ФОРТРАНе** есть целое семейство встроенных функций для преобразования типов данных. Приведем некоторые примеры:

```
program testpriv1;  implicit none
complex   cs, ca; complex*16 cd; complex(8)   cb
real  recs, imcs, are, aim; real(8) imcd, recd, bre, bim; integer(1) k
cs=( 3.5,5.6 ); write(*,*) ' cs=',cs
imcs=aimag(cs); write(*,*) ' cs.im=',imcs ! извлечение мнимой части
cd=(3.5d0,5.6d0); write(*,*)' cd=', cd
            write(*,'('' cd='',d23.15,''+i*'',d23.15)') cd
imcd=aimag(cd);write(*,*) ' cd.im=',imcd  ! aimag - родовое имя для
                                          ! аргумента комплексного типа
recs=real(cs); write(*,*) ' cs.re=',recs  !  real - родовое имя для
recd=real(cd); write(*,*) ' cd.re=',recd  ! аргументов типа real,
               write(*,*) real(5,4)       ! integer, complex
               write(*,*) real(5,8)       !
 k=129; write(*,*) real(k,8)              !  Почему на печати -127.0?
        write(*,*) ' int(cs)=',int(cs)
        write(*,*) ' int(cd)=',int(cd); write(*,*) ' int(5.3)=',int(5.3)
        write(*,*) ' int(5.3d0)=',int(5.3d0)
are=4.7; aim=3.9; ca=cmplx(are,aim,8); write(*,*) ' ca=',ca
                    write(*,'('' ca='',d23.15,''+i*'',d23.15)') ca
ca=cmplx(are,aim,4); write(*,*) ' ca=',ca
ca=cmplx(are,aim);   write(*,*) ' ca=',ca
bre=4.2d0; bim=7.3_8; cb=cmplx(bre,bim);   write(*,*) '    cb=',cb
                      cb=cmplx(bre,bim,8); write(*,*) ' 8: cb=',cb
                      cb=dcmplx(bre,bim); write(*,*) ' cb=',cb
write(*,*) ' dble(ca)=',dble(ca); write(*,*) ' dfloat(ca)=',dble(ca)
write(*,*) ' dble(cb)=',dble(cb); write(*,*) ' dfloat(cb)=',dble(cb)
end
```

Компиляция **testpriv1** на **gfortran**-компиляторе должна проводиться при включении **-fno-range-check** и **-Wall** (*Почему?*).

Назначение функций, которые встречаются в программе **testpriv1**:

Функция	**Ее назначение**
aimag(z)	возвращает мнимую часть комплексного аргумента **z**. Результат имеет вещественный тип той же длины, что и мнимая часть аргумента. Тип аргумента **complex** или **complex(8)**, или **complex*16**
cmplx(x[,y] [,kind])	возвращает для заданного значения аргумента **x** любого числового типа, соответствующее значение типа **complex** разновидности **kind**. Аргумент **y** **y** при наличии комплексного **x** не требуется.
dble(x) **dfloat(x)**	возвращает для заданного значения аргумента **x** любого числового типа соответствующее значение вещественного типа удвоенной точности.
dcmplx(x,[,y])	может использоваться наряду с **cmplx**, когда необходима удвоенная точность результата, а указывать параметр разновидности лень.
float(x)	возвращает для заданного значения аргумента целого типа соответствующее значение вещественного.
int(a[,kind])	возвращает значение аргумента **a** любого числового типа, приведенным к значению целого типа длиной **kind** байтов, путем отбрасывания дробной части. При отсутствии параметра разновидности **kind** (длины типа результата), последний имеет стандартную длину принятую транслятором по умолчанию (обычно это 4 байта, если при вызове транслятора не указана опция /**4I2** или в тексте исходного модуля не применена директива **$integer:2**). Единственное ограничение: величина аргумента не должна конфликтовать с диапазоном значений типа результата. (Подробнее см. например, [3] [4]; [6]).
real(x,[kind])	возвращает для заданного значения аргумента любого числового типа, соответствующее значение вещественного разновидности **kind**.
transfer(x,y[,size]) (см., например, [2])	возвращает значение аргумента **x** любого типа без изменения его физического представления как значение типа аргумента **y** (см. раздел 12. Функция передачи типа **transfer**).

Замечание:

При переходе с одного компилятора на другой полезно проверить, что функции **real** и **float** при работе с данными типа **complex** дают результаты, согласующиеся на обоих компиляторах.

1.2.11 СИ-операция последовательного вычисления ,

В [10, 6.2] эта операция названа *близким родственником блока*, поскольку позволяет связать несколько выражений в одно, не заводя блока:

```
#include <iostream>

using namespace std;
int main(void)
{
  int i=10, j;                                          // Результат:
  i= (j=4, i+j); cout<<" i="<<i<<"  j="<<j<<endl;  //  i=14  j=4
  i=  j=4, i+j ; cout<<" i="<<i<<"  j="<<j<<endl;  //  i= 4  j=4
                 cout<<" i+j="<<i+j<<endl;          //  i+j=8
return 0;
}
```

В этой программе в операторе присваивания **i=(j=4,i+j)** операторы **j=4 и i+j** заключены в круглые скобки. Операция "**,**" выполняется до присваивания значения переменной **i** с результатом равным по определению значению оператора **i+j** из правой части, т.е. **i=10+4=14**.

Без круглых скобок **i=j=4,i+j** операция **i+j** тоже выполняется (правда, не совсем понятно зачем, так как из-за более высокого приоритета операции присваивания сначала **j=4**, а затем и **i=4**. Таким образом, левая часть конструкции **i=j=4,i+j**, расположенная до запятой, отработает полностью (т.е. слева от запятой не останется ни присваивания,ни имени приемщика суммы). Так что, выражение **i=j=4,i+j** синтаксически верно, но значение суммы **i+j**, которая всё-таки вычислиться, не будет присвоено никакой переменной.

Цель оператора **,** – обеспечить одновременное выполнение нескольких операторов, если по какой-то причине (например, по синтаксису языка) допустим только один.

Например, в операторе цикла с параметром (**for**) в качестве параметра цикла обычно используется одна переменная, для которой первым делом задается начальное значение, после чего ставится **;**, отделяющая шаг инициализации от условия продолжения цикла. Однако, нередки ситуации, когда в одном операторе **for** удобно использовать несколько аналогов параметра цикла. Рассмотрим примитивный вариант программы, выясняющей, является ли строка перевертышем:

```
#include <iostream>
#include <cmath>
using namespace std;
int main(void)
{int  i, j;  bool p, q;

char *s="пилвиноонивлип\0";
char *c="пилвиноенивлип\0";

for (i=0,j=13,p=true;(i<=j)&& (p=s[i]==s[j]);i++,j--);

if (p) cout<<s<<" - палиндром"<<endl;
  else cout<<s<<" - не палиндром"<< endl;

for (i=0,j=13,q=true;(i<=j)&& (q=c[i]==c[j]);i++,j--);

if (q) cout<<c<<" - палиндром"<<endl;
  else cout <<c<<" - не палиндром"<< endl;
return 0;
}
```

Если в этой программе после **i=0** поставить **;**, то уже не сможем в заголовке цикла инициализировать **j** и **p** (или **q**). Ещё один пример:

```
#include <iostream>   // В программе используются функции sumxy1 и sumxy2
using namespace std;  // соответственно с одним и двумя формальными типа
int sumxy1(int);      // int аргументами. В качестве единственного факти-
int sumxy2(int,int);  // ческого аргумента первой (sumxy1) подается выражение
int main(void)        // (x,x+y). В контексте наличия круглых скобок запятая
{int x=100, y=200;                            // между x и x+y трактуется
cout<<" x + y  ="<<sumxy1( (x,x+y) )<<endl; // компилятором как операция
                                              // последовательного вычисления
cout<<" x + y  ="<<sumxy2(x,x+y)<<endl;     // с результатом: x + y = 300.
return 0; }                                   // В контексте вызова sumxy2
int sumxy1(int x)                             // "запятая" толкуется как
{cout<<"sumxy1: x="<<x<<endl;                 // обычный разделитель аргументов
 return x;}                                   // так, что через имя функции
int sumxy2(int x, int y)                      // при аргументах x=100 и y=200
{cout<<"sumxy2: x="<<x<<"   y="<<y<<endl;   // для возвращается результат:
 return x+y;                                  // x + (x+y) = 400.
 }
```

Вывод: использование операции последовательного вычисления требует неослабного внимания и высокой аккуратности, поскольку позволяет (как видели из первого примера) легко получить синтаксически верную конструкцию, делающую совсем не то, что имелось ввиду.

1.2.12 Приоритет операций

	Название операции	Символ C, C++	Примеч.	Символ Ф-77 Ф>90	Примеч.
1	Обращение к функции	()	Слева направо	()	Слева направо
	Выделение элемента массива	[]	– ” –	()	– ” –
	Выделение поля структурной переменной	.	– ” –	. %	– ” –
	Выделение поля струк. перемен ной по указате лю на ее начало	–>	– ” –	=>	– ” –
	Возведение в степень a^q	**pow(a,q)**	функция	**	СПРАВА налево
2	Логическое отрицание	!	Справа налево	**.not.**	Справа налево
	Поразрядное логическое НЕ	∼	–”–	**not(i)**	Функция. **not(9)=6**
	Изменение знака	–	–”–	–	Справа налево
	Инкремент	++	–”–	НЕТ	**k=k+1**
	Декремент	−−	–”–	НЕТ	**k=k-1**
	Определение адреса переменной	&	–”–	НЕТ	
	Обращение к памяти по значению указателя	*	–”–		
	Преобразование к типу	(**имя**) **типа**	–”–	Набор функций	
	Определение размера в байтах	**sizeof**			Справа налево

	Название операции	Символ С, С++	Примеч.	Символ Ф-77 Ф>90	Примеч.
3	Умножение	*	Сл.напр.	*	Сл.напр.
	Деление	/	– ” –	/	– ” –
	Ост. от дел.	%	– ” –	**mod(a,b)**	Функция
4	Сложение	+	Сл.напр.	+	Сл.напр.
	Вычитание	–	–”–	–	–”–
5	Сдвиг влево	$\ll$	–”–	$\boldsymbol{ishft(i,s)}$	s>=0
					Функция
	Сдвиг вправо	$\gg$	–”–	$\boldsymbol{ishft(i,s)}$	s<=0
6	Меньше	<	Сл.напр.	**.lt.** <	Сл.напр.
	$\leq$	<=	–”–	**.le.** <=	–”–
	Больше	>	–”–	**.gt.** >	–”–
	$\geq$	>=	–”–	**.ge.** >=	–”–
7	Равно	==	Сл.напр.	**.eq.** ==	–”–
	Не равно	**!** =	–”–	**.ne.** / =	–”–
8	Поразр.конъ.	&	Сл.напр.	$\boldsymbol{iand(i,j)}$	Функция
9	Поразр.искл. ИЛИ (xor)	$\wedge$	Слева направо	$\boldsymbol{ieor(i,j)}$	Функция
10	Поразр.дизъ.	|	Сл.напр.	$\boldsymbol{ior(i,j)}$	Функция
11	Логическое И	&&	Сл.напр.	**.and.**	Сл.напр.
12	Логическое ИЛИ	||	Слева направо	**.or.**	Слева направо
	Искл. лог. ИЛИ	НЕТ		**.xor.**	
	Эквив.	НЕТ		**.eqv.**	
	Неравн.	НЕТ		**.neqv.**	
13	Условная операция	**? :**	Слева направо	НЕТ	
14	Присваивание	= %=	Справа налево	=	Справа налево
		+ = – =		НЕТ	
		* = / =	–”–	НЕТ	
		$\ll=$ $\gg=$	–”–	НЕТ	
		| = & =	–”–	НЕТ	
		^ =	–”–	НЕТ	
15	Опер. запятая	,	Сл.напр.	НЕТ	

Левая часть таблицы содержит перечень всех операций языка СИ в порядке убывания приоритета. Группы **Си**-операций одинакового приоритета разделены тонкой чертой. Справа даны соответствующие операции (или функции) ФОРТРАНа. Правда, **ФОРТРАН-** и **СИ-**приоритеты не всегда совпадают.

Приоритет выполнения операций ФОРТРАНа:

1	2	3	4	5	6	7	8
**	*	.lt. <	.eq. ==	.not.	.and.	.or.	.xor.
	/	.le. <=	.ne. /=				.eqv.
		.gt. >					.neqv.
		.ge. >=					

Видим, что операции отношения в ФОРТРАНе имеют более высокий приоритет чем любая из логических операций, в то время как в СИ операция отрицания **!** по приоритету выше любой операции отношения. Поэтому, например, **на ФОРТРАНе**: **(.not.0>1)** есть **true**, но на **СИ**: **(!0>1)** есть **0**, то есть **false** (см. последний оператор следующего примера). Именно поэтому в сомнительных случаях не пренебрегаем скобками. При использовании сложного булева выражения важно помнить, что его расчет прекращается, когда результат уже ясен, т.е. операнды после первой булевой операции **могут** и не вычислятся: Обратимся к примеру:

```
#include <iostream>
using namespace std;
int main(void)
{int i,j;
i=0;  j=5;
if((i++)&&(++j)) cout<<"  j="<<j<<endl;
                 cout<<" i="<<i<<"   j="<<j<<endl;
if((++i)&&(j++)) cout<<" i="<<i<<"   j="<<j<<endl;
cout<<"   (!0>1)="<<(!0 >1 )<<endl;
cout<<" (!(0>1))="<<(!(0>1))<<endl;
return 0; }
```

Результат работы этой программы:

```
              // Почему нет вывода, соответствующего первому условному
              // оператору программы?
 i=1   j=5    // Почему не отработало (++j)?
 i=2   j=6    // Почему отработало (j++)?
   (!0>1)=0
 (!(0>1))=1
```

Аналогичный результат получает и программа на ФОРТРАНе-77.

```
program tstbool0
implicit none
interface
  function bfun(j) result(w); logical w; integer j; end function bfun
end interface
logical  i
integer j
i=.false.
j=5
if ( i.and.bfun(j) ) write(*,*) ' i=',i,'    j=',j
i=.true.;            write(*,*) ' i=',i,'    j=',j
if ( i.and.bfun(j) ) write(*,*) ' i=',i,'    j=',j
write(*,*) ' .not. 0>1 =',.not.0>1
write(*,*) ' .not.(0>1)=',.not.0>1
end program tstbool0
function bfun(j) result(w)
implicit none
logical w
integer j;
w=j<1000; j=j+1;
end function bfun
```

Правда, в ФОРТРАНе невозможна проверка (.not.0)>1. *Почему?*

```
 i= T     j=           5
 i= T     j=           6
 .not. 0>1 = T
 .not.(0>1)= T
```

1.3 О чем узнали из первой главы? (2-ой семестр)

1. О СИ-операциях инкремента ++ и декремента −.
2. О **префиксной** и **постфиксной** формах их использования.
3. **Префиксная форма** сначала использует текущее значение операнда, а уж затем изменяет его значение.
4. **Постфиксная форма**, наоборот, – сначала изменяет текущее значение своего операнда, а уж затем использует.
5. В СИ операция нахождения остатка от деления двух целых обозначается символом **%**, а в ФОРТРАНе имеется похожая по сути встроенная функция **mod**, имя которой является родовым.
6. **Операция присваивания** в СИ не только пересылает значение, но и *понимает его* как свой результат. Поэтому в СИ допустима, например, запись **a=b=c=d=5**. В ФОРТРАНе оператор присваивания последним свойством не обладает.
7. О наличии в СИ **комбинированной операции присваивания**, которой тоже нет в ФОРТРАНе.
8. Про СИ-операции сдвига $\ll$ и $\gg$, которые C++ по контексту отличает от операции направления в потоки ввода/вывода, и про соответствующую одновременно им обеим ФОРТРАН-функцию **ishift**.
9. СИ-операция **sizeof** позволяет получить количество байт необходимое для размещения данного любого типа. Операндом этой операции может быть и значение, и имя типа, и имя переменной. В современном ФОРТРАНе для подобной цели можно использовать функции **kind**, **size** и **sizeof**.
10. О поразрядных логических операциях СИ и о соответствующих им встроенных функциях современного ФОРТРАНа.
11. О явном и неявном преобразовании типов данных.
12. Приоритеты операций в ФОРТРАНе и СИ не всегда одинаковы. Любая операция отношения ФОРТРАНа имеет более высокий приоритет чем любая из логических операций. В СИ приоритет операции отрицания выше приоритета любой из операций отношения.

1.4 Первое домашнее задание (2-ой семестр)

1. Продемонстрировать в дисплейном классе работоспособность лекционных СИ- и ФОРТРАН-программ по теме «ВЫРАЖЕНИЯ и ОПЕРАЦИИ»

   (пункты:

   - 1.2.3 — тернарная операция;
   - 1.2.4 — операция присваивания;
   - 1.2.5 — операция **sizeof**;
   - 1.2.6 — операции сдвига;
   - 1.2.7 — поразрядные логические операции.

).

2. Сдать преподавателю список вопросов по неясным моментам исходных СИ- и ФОРТРАН-текстов, рассмотренных в лекции.

3. Разработать С- и ФОРТРАН-функции подсчёта количества единиц в двоичном представлении целого неотрицательного числа.

4. Разработать (на ФОРТРАНе и С) наиболее оптимальный алгоритм (в виде функции) решения предыдущей задачи.

5. Усовершенствовать функцию из предыдущего пункта так, чтобы она позволяла находить количество единиц в двоичном представлении любой допускаемой компилятором разновидности типа целого типа (т.е. -1, -2, -4, -8, -16).

2 Немного о рекурсии в программировании.

Рекурсия – определение какого–нибудь понятия через само это понятие. Например, таким свойством обладает каждое из "бесконечной"череды изображений человека в настенном зеркале, если он держит в руках другое зеркало, отражающее изображение первого. Каждое из изображений определяется через свое предыдущее отражение. Налицо свойство рекурсивности.

Рекурсия часто используется в математических определениях (см., например, [17]):

1. **Натуральное число:** Число – это аксиоматическое (т.е. принимаемое без доказательства) понятие математики, используемое для отражения количественных соотношений. Интуитивно под натуральным числом понимаются числа, предназначенные для счёта.

   Рекурсивное определение:

   а) единица – натуральное число;

   б) целое число, следующее за натуральным, есть натуральное.

2. **n! :**.

   Нерекурсивное определение : $n! = 1 \cdot 2 \cdot 3 \cdots n$.

   Рекурсивное определение:

   а) $0! = 1$.

   б) если $n > 0$, то $n! = n \cdot (n-1)!$.

3. **Произведение двух натуральных чисел через сложение:**

   Нерекурсивный вариант: $a \cdot b = \underbrace{a + a + \cdots + a}_{b \text{ раз}} = \sum_{i=1}^{b} a.$

   Рекурсивное определение:

   а) $a \cdot 1 = a$;

   б) Если же $b \neq 1$, то $a \cdot b = a + a \cdot (b-1)$.

4. **Определение алгоритма Евклида** (поиск наибольшего общего делителя двух натуральных чисел):

```
Если a>=b, то     если a%b==0, то    НОД(a,b)=b,
                                иначе
                                     НОД(a,b)=НОД(b,a%b)
              иначе НОД(a,b)=НОД(b,a)
```

Во всех четырех примерах определяемое в пунктах б) (натуральное число, понятие факториала, произведения двух целых и алгоритм поиска наибольшего общего делителя двух натуральных чисел) выражалось через самого себя, то есть выбранное нами определение обладает свойством рекурсивности (так нам захотелось).

Рекурсия применима не только для определения алгоритмов, но и для определения **структур данных**. В реальной жизни многие явления схематически можно изобразить в виде **структурной системы**, части которой связаны отношениями включения и подчинения. Такие системы называются **иерархическими**, их части – **узлами**, связи между ними – **связями** или **ссылками**.

Примеры иерархических структур: Вселенная (Метагалактика – галактика – звездное скопление – звезда – планетная система); государство (республика – область – район – населенный пункт); завод (службы управления и обеспечения – цеха – участки); родословная (ее схематическое изображение – генеалогическое дерево); армейское соединение (дивизия – полк – батальон – рота – взвод – отделение).

В дискретной математике иерархическим структурам сопоставляются модельные математические понятия **списка**, **дерева**, **графа**.
Список может служить моделью очереди к врачу в поликлинике;
дерево – моделью родословной (генеалогическое дерево) или, например, моделью очередности встреч спортсменов по олимпийской системе;
граф – моделью станций и путей сообщения метрополитена и т.д.

Элемент списка (узел,звено) состоит как минимум из двух частей:
одна – содержательная (например, фамилия больного);
вторая – связующая (каждый должен помнить, за кем занял очередь).
Рекурсия позволяет определять подобные иерархические структуры.

- **Рекурсивное определение списка:**

  а) **Список** – это либо пустая структура;

  б) либо – элемент списка, указывающий на **список** из последующих элементов;

- **Рекурсивное определение дерева:**

  а) **дерево** – это либо пустая структура;

  б) либо – узел, связанный с конечным числом **деревьев**.

В обоих определениях налицо рекурсивность: понятие **списка** определяется через понятие **списка**; понятие **дерева** — через понятие **дерева**.

Если, например, сопоставить себе корень генеалогического дерева, то две дуги из корня приведут к матери и отцу, по две дуги от каждого из родителей приведут к бабушкам и дедушкам и так далее.

Во всех языках программирования есть понятие процедуры, которое сопоставляет алгоритму имя так, что, описав алгоритм один раз, получаем возможность многократно вызывать его по имени.

Аналогично: во многих языках программирования можно из имеющихся элементарных базовых типов строить более сложные структуры данных (массивы, структуры, динамические структуры; см. следующую тему *"Немного о массивах, структурах и указателях"*).

СИ, СИ++ и **современный ФОРТРАН** предоставляют возможность использовать и **рекурсивные** функции, и **рекурсивные** структуры данных, то есть описывать алгоритм или структуру, используя внутри описания обращение (или ссылку) к самому описываемому объекту.

В *"древних"* версиях ФОРТРАНа рекурсивные функции и подпрограммы НЕ ДОПУСКАЛИСЬ. В частности, при вызове в главной программе функции однократного интегрирования **y=trap(f,a,b,n)** (здесь **f** — процедура расчёта подынтегральной функции), в описание алгоритма **f** нельзя было включать вызов **trap** (что означало бы рекурсию) для расчёта интеграла, который мог бы присутствовать в подынтегральной функции. Приходилось описывать две функции (**trap** и **trap1**) однократного интегрирования, совпадающие во всех деталях кроме имён функций (и в главной программе, например, вызывать **trap**, а при расчёте подынтегральной функции **trap1**).

2.1 Рекурсивные процедуры (простые примеры).

Процедура (функция или подпрограмма) называется **рекурсивной**, если ее описание явно или неявно содержит обращение к ней же самой.

2.1.1 Нерекурсивная и рекурсивная СИ-функции расчета n!.

```
#include <iostream>                          //  Файл  tstfactor.cpp
using namespace std;
double factorn (int); double factorr1(int); double factorr2(int);
int main()
{int n;  double fn, fr1, fr2;
 cout<<"введите n:"<<endl; cin>>n; cout<<" n="<<n<<endl;
 fn=factorn (n);   fr1=factorr1(n);   fr2=factorr2(n);
 cout<<" nn!="<<fn<<"   nr1!="<<fr1<<"   nr2!="<<fr2<< endl;
 return 0;
}
```

```
double factorn(int n)                  //  Файл  factor.cpp
{ int i; double p;                     // Нерекурсивная функция расчета n!
  p=1; for (i=1;i<=n;i++) p*=i;        // Находит n! посредством оператора
  return p; }                            // цикла.
double factorr1(int n)                 // Рекурсивная функция расчета n!
{ if (n==0) return 1; else    return n*factorr1(n-1); }
double factorr2(int n)                 // Рекурсивная функция расчета n!
{ return(n==0 ? 1 : n*factorr2(n-1));}// (через тернарную операцию)
```

Нерекурсивная функция **factorn(n)** вызывается один раз и за счет внутреннего оператора цикла находит искомое произведение. Рекурсивная **factorr1(n)** в главной программе также вызывается один раз, оператора цикла не содержит, но внутри своего тела вызывает сама себя до тех пор, пока не получит в качестве текущего аргумента 0. Каждый очередной рекурсивный вызов не только требует дополнительного времени, но и захватывает новую ячейку памяти под формальный параметр **n**. Так что при расчете **3!** из оперативной памяти будет зафрахтовано четыре дополнительные ячейки для хранения **3** (первый вызов), **2** (второй вызов), **1** (третий вызов) и **0** (четвертый вызов). После завершения очередного вызова соответствующая зафрахтованная ячейка перейдет в ведение операционной системы.

```
введите n:        //  Результаты вызовов функций
3                 //  factorn(n), factorr1(n) и factorr2(n) при n=3.
 n=3
  nn!=6   nr1!=6   nr2!=6
```

2.1.2 Нерекурсивная и рекурсивная ФОРТРАН-функции расчета n!.

```
program tstfactor; implicit none                !  Файл tstfactor.for
interface
  function factorn(n) result(w); real(8) w; integer n
  end function factorn
  recursive function factorr(n) result(w); real(8) w; integer n;
  end function factorr
  function factornB(n); real(8) factornB; integer n
  end function factornB
end interface
integer n; real(8) rn, rr, rb
write(*,*) 'введи n:'; read (*,*) n; write(*,*) ' n=',n
rn=factorn(n); rr=factorr(n); rb=factornB(n); write(*,1000) rn, rr, rb
rn=factorn(n); rr=factorr(n); rb=factornB(n); write(*,1000) rn, rr, rb
1000 format(1x,' rn=',g15.7,'  rr=',g15.7,'  rb=',g15.7)
end program tstfactor

function factorn(n) result(w); implicit none     ! factorn - нерекурсивная
real(8) w; integer n, i                          !      функция расчета n!
w=1; do i=1,n; w=w*i; enddo
end function factorn
recursive real(8) function factorr(n) result(w); ! factorr1 - рекурсивная.
implicit none; integer n
data w /1.0d0/
if (n.eq.0) then;  w=1; else;  w=n*factorr(n-1); endif
end function factorr
function factornB(n); implicit none              ! factornB - нерекурсивная
real(8) w,  factornB; integer n, i               !      функция расчета n!
data w /1.0d0/                                   !  ОПАСНЫЙ вариант !!!
do i=1,n; w=w*i; enddo
factornB=w
end function factornB

 введи n:
  n=           5
  rn=   120.0000      rr=   120.0000      rb=   120.0000
  rn=   120.0000      rr=   120.0000      rb=   14400.00
```

Внимание! В ФОРТРАН-учебниках можно встретить фразу, что оператор **data** предназначен для задания начальных значений. Иногда его применяют для задания начального значения какой-нибудь локальной переменной той или иной процедуры (например, расчёта факториала: **data w /1.0d0/**), что при повторном вызове процедуры может привести к неожиданным результатам (см., результат **второго** вызова процедуры **factornB(5)**, который оказался равен **1440**).

Дело в том, что присваивание **data w /1.0/** выполняется компилятором, а не при работе программы. Поэтому после первого вызова **factornB** в **w** окажется не единица, а **120**, а после второго — **1440**.

На примере **factornB** видно, что описание **result(w)**, имеющееся в **factorn** и **factorr**, принципиально важно, так как в этом случае компилятор на наличие в тексте процедуры оператора **data w /1.0/**, сообщит:

```
data w /1.0d0/
    1
    ошибка: DATA attribute conflicts with RESULT attribute in 'w' at (1)
```

Нельзя значение переменной задавать оператором **data**, если она предназначена для возврата результата.

2.1.3 Схема вызовов при расчете $factorr1(3)$.

```
fr1:=factorr1(3)
              :=-->3* factorr1(2)
                               :=-->2*factorr1(1)
                                                 :=-->1*factr(0)
                                                      1 * 1   найден factr(0);
                                 2 *     1 <-----------------   найден factr(1);
               3 *    2    <-----------------------------      найден factr(2);
        6   <-------------------------------------------      найден factr(3),
fr:=    6   это значение и возвращается в главную программу.
```

Замечание: Часто спрашивают: **"Почему 0!=1 и 1!=1 ?"**

Факториал представляет собой частный случай широко использующейся в математике **Гамма-функции** (интеграл Эйлера):

$$\mathbf{\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt} \quad ,$$

значение которой при натуральном **x** равно соответствующему факториалу, именно

$$\mathbf{\Gamma(x+1) = x!} \quad .$$

В частности,

$$\mathbf{0! = \Gamma(1) = \int_0^\infty e^{-t} dt = 1} \quad , \quad \mathbf{1! = \Gamma(2) = \int_0^\infty e^{-t} \cdot t dt = 1} \quad .$$

2.1.4 Достоинства и недостатки рекурсивного описания.

1. Рекурсивное описание функции можно заменить нерекурсивным.
2. Рекурсивность – это не свойство функции, а свойство ее описания.
3. *"Какой способ описания выгоднее?"* – **решать программисту!**

Достоинства	Недостатки
определяет сколь-угодно много объектов через конечное высказывание;	как правило, работает медленнее нерекурсивного из–за временных затрат на повторные обращения;
записывается фактически прямо по описанию алгоритма;	не всегда переход на нерекурсивное описание так же прост, как при описании $factorn(n)$;
текст легче воспринимается пользователем	отладка может оказаться непростой, особенно при сложной рекурсии

2.2 Понятие глубины рекурсии. Алгоритм Евклида.

Глубина рекурсии – это количество промежуточных вызовов функции в процессе ее вычисления для данного аргумента.

Например, **factorr1(3)** помимо начального вызова активирует еще три рекурсивных: factorr1(2), factorr1(1), factorr1(0), т.е. рекурсия имеет глубину в три уровня. Для функции **n!** глубина рекурсии при любом **n** очевидна. К сожалению, это – исключение. Даже в простом алгоритме Евклида поиска наибольшего общего делителя двух натуральных чисел **a** и **b** глубина рекурсии не очевидна.

2.2.1 Нерекурсивная и рекурсивная СИ-функции поиска НОД.

В соответствие с определением 4 из введения к этой главе имеем:

```
#include <iostream>
using namespace std;
int nodn(int,int); int nodr(int,int);
int main()
{int a, b, nn, nr;
              cout<<"введите a, b:"<<endl;
 cin>>a>>b;   cout<<" a="<<a<<"   b="<<b<<endl;
 nn=nodn(a,b);
 nr=nodr(a,b);
 cout<<" nodn="<<nn<<"   nodr="<<nr<< endl;
 return 0;
}

int nodn(int a, int b)                    // Нерекурсивная функция
{int r;                                   // расчета НОД(a,b).
 while (b)
  { r=a%b; a=b; b=r; }
 return a;
}
int nodr(int a, int b)                    // Рекурсивная функция
{ if (a<b) return nodr(b,a);              // расчета НОД(a,b).
    else
      { if (b) return nodr(b,a%b);
          else return a;
      }
}

введите a, b:       // Результат C++ тестирования nodn и nodr:
85
51
 a=85   b=51
 nodn=17   nodr=17
```

2.2.2 Нерекурсивная и рекурсивная ФОРТРАН-функции поиска НОД.

```
program tstevclid; implicit none       !  Программа тестирующая
interface                              !  нерекурсивную  и
  integer function nodn(a,b) result(w) !  рекурсивную функции
  integer a, b                         !  поиска НОД(a,b)
  end function nodn
  recursive integer function nodr(a,b) result (w); integer a, b
  end function nodr
  integer function nodn1(a,b); integer a, b; end function nodn1
end interface
integer a, b, nn, nr, n1, n2; write(*,*) 'введи a, b:'; read (*,*) a, b
write(*,'(i29,i7)') a, b
write(*,'(17x,"НОД(a,b)",3x,"a",5x,"b")')
nn=nodn(a,b);      write(*,'(a,i5,i7,i7)') '  nodn ( a , b )=',nn, a, b
nr=nodr(a,b);      write(*,'(a,i5,i7,i7)') '  nodr ( a , b )=',nr, a, b
n1=nodn1((a),(b)); write(*,'(a,i5,i7,i7)') '  nodn1((a),(b))=',n1, a, b
n2=nodn1(a,b);     write(*,'(a,i5,i7,i7)') '  nodn1( a , b )=',n2, a, b
nn=nodn(a,b);      write(*,'(a,i5,i7,i7)') '  nodn ( a , b )=',nn, a, b
end program tstevclid
```

```
integer function nodn(aa,bb) result(w)    !  Нерекурсивная функция
implicit none; integer aa, b, bb, r       !  поиска НОД (aa,bb)
w=aa; b=bb; do while (b/=0); r=mod(w,b);   w=b;   b=r; enddo
end function nodn
recursive  function nodr(a,b) result(w)  ! Рекурсивная функция
implicit none; integer a, b, w           ! поиска НОД(aa,bb)
if (a.lt.b) then; w=nodr(b,a)
            else; if (b.ne.0) then
                    if (mod(a,b).ne.0) then; w=nodr(b, mod(a,b))
                                       else; w=b
                    endif;    else; w=a
                  endif
endif
end function nodr
integer function nodn1(a,b); implicit none !  Нерекурсивная функция
integer a, b, r                            !     поиска НОД (aa,bb)
do while (b/=0); r=mod(a,b);   a=b;   b=r; enddo
nodn1=a
end function nodn1
```

```
 введи a, b:                           ! Вспомнить о разных принципах
                          85     51    ! передачи аргументов процедурам
                НОД(a,b)   a     b     ! в ФОРТРАНе и СИ.  Ответить:
  nodn ( a , b )=   17     85     51    ! Почему NODN и NODR не меняют
  nodr ( a , b )=   17     85     51    ! значения аргументов после
  nodn1((a),(b))=   17     85     51    ! завершения работы, а NODN1 ---
  nodn1( a , b )=   17     17      0    ! только, если фактические
  nodn ( a , b )=   17     17      0    ! заключены в круглые скобки.
```

2.3 Пример задачи неразрешимой без рекурсии.

Напечатать вводимые числа в обратном порядке.
Если можно использовать массив, то нерекурсивное решение дается просто распечаткой элементов массива от последнего до первого. Однако, использование массива **запрещено** и заранее **не известно** сколько чисел будет вводится. Выберем в качестве признака окончания последовательности вводимых чисел ввод нуля. Кажется, что задача неразрешима. Но с рекурсией решение до удивления просто.

2.3.1 СИ-решение:

```
int revers();
int main()
{ revers();  return 0; }
```

```
#include <iostream>
using namespace std;
int revers()
{ int a;  cin>>a; if (a) { revers();  cout<<a<<"  "; } }
```

2.3.2 ФОРТРАН-решение:

```
program tstrevers; implicit none; call revers; end program tstrevers
```

```
recursive subroutine revers; implicit none
integer a
read(*,'(i2$)') a; if (a.ne.0) then; call revers; write(*,'(i2$)') a; endif
end subroutine revers
```

Символ **$** в форматном фрагменте операторов ввода(вывода) обеспечивает возможность набора(вывода) данных в одной строке.

```
 1 2 3 4 5 6 0                  //  Результат работы tstrevers
 6 5 4 3 2 1
```

Как работает revers?

1. Тело процедуры **revers** состоит из раздела описания переменной **a** и раздела операторов, выполняемых при вызове процедуры.
2. Переменная **a** – внутренняя локальная переменная процедуры.
3. При очередном вызове **revers** под **a** выделяется новая ячейка оперативной памяти. Это выделение происходит при каждом очередном рекурсивном вызове процедуры, то есть динамически.

4. При завершении очередного вызова **revers** ячейка, хранящая текущее **a**, высвобождается из под ведения и процедуры, и вызывающей её программы, и может быть, например, передана операционной системой вообще какой–то другой программе).

5. При очередном выходе из **revers** высвобождается как раз та ячейка, которая была *зафрахтована* последней.

6. Структура данных из последовательно выделяемых в оперативной памяти ячеек под переменную **a** при рекурсивных вызовах **revers** или освобождаемых при завершении таковых называется:

   стек или **магазин** или **одноконцовая очередь**.

   Стек – single top ended queue.

2.4 Немного о стеке.

Стек – это динамическая структура данных, которая состоит из переменного числа компонент одинакового типа. Компоненты стека извлекаются по принципу **LIFO:**

последним вошел (last in) – первым вышел (first out)

Принцип **FIFO:** первым вошел (first in) – первым вышел (first out) определяет другую динамическую структуру – привычную нам очередь.

Стек – одна из наиболее употребительных стуктур данных в системном программировании.

При вызове рекурсивных процедур стековая структура данных образуется автоматически, не требуя своего явного описания.

В языках ПАСКАЛЬ, СИ (а теперь и современный ФОРТРАН) есть средство, позволяющее непосредственно организовывать в оперативной памяти всякие динамические структуры данных (в частности, и стек).

Примерами стековой организации могут служить:

1. известный во всем мире рожок (магазин) автомата Калашникова;

2. тупиковое ответвление от железнодорожного полотна (выгодное для переприцепки тепловоза к противоположному концу поезда);

3. хорошо знакомая с детских лет каждому игрушка – пирамидка из колечек.

2.5 Задача о ханойской башне.

Задача излагается во многих учебных пособиях (например, [17], [18]). Её рекурсивное решение несравнимо проще нерекурсивного.

Задача. Даны три стержня. На первом в виде пирамиды надето n колец в строгом порядке увеличения их диаметра к основанию пирамиды. Необходимо всю пирамиду перенести с первого стержня на третий (второй стержень рабочий), соблюдая два закона:

1. Кольца можно перемещать только по одному.
2. Никогда нельзя класть большее кольцо на меньшее.

Требуется разработать программу, которая печатала бы: с какого стержня на какой нужно переносить очередной диск, чтобы добиться в конце концов полного перемещения пирамиды с первого стержня на третий.

2.5.1 Уяснение ситуации.

Рассмотрим для начала простейшие частные случаи, которые помогут разработать стратегию переноса:

1) на первом стержне одно кольцо. Пирамида из одного кольца перемещается за один перенос.

2) на первом стержне два кольца (сначала переносим меньшее кольцо с 1-го на 2-ой; затем самое большое кольцо с 1-го на 3-ий и, наконец, меньшее кольцо со 2-го на 3-ий). Так что перенос двуколечной пирамиды осуществлен за три переноса.

3) на первом стержне три кольца (перемещение за семь переносов):

I перенос кольца с первого стержня на третий;

II перенос кольца с первого стержня на второй;

III перенос кольца с третьего стержня на второй;

IV перенос кольца с первого стержня на третий;

V перенос кольца со второго стержня на первый;

VI перенос кольца со второго стержня на третий;

VII перенос кольца с первого стержня на третий.

Заметим, что перемещения **I** – **III** по сути дела привели к переносу башенки из двух верхних колец с первого стержня на второй. Затем был осуществлен перенос самого нижнего и единственного кольца первого стержня на третий стержень. После этого осталось лишь перенести башенку из двух колец со второго стержня на третий, используя свободный первый стержень в качестве рабочего. Ясно, что это ровно столько же перемещений, сколько было при переноске **I** – **III**. Так что перенос n колец распадается на три этапа:

1. Перенос башенки из $n-1$ кольца с первого стержня на второй;
2. Перенос одного кольца с первого стержня на третий;
3. Перенос башенки из $n-1$ кольца со второго стержня на третий.

Так что башню из $n-1$ кольца придется переносить дважды: один раз с 1-го стержня на 2-ой, используя в качестве рабочего 3-ий, а затем еще раз со 2-го на 3-ий, используя в качестве рабочего 1-ый.

Таким ообразом, алгоритм должен *уметь* переносить кольца с заданного колышка, например, с номером p на колышек с номером q, используя в качестве рабочего оставшийся колышек. Легко заметить, что номер этого рабочего колышка просто вычисляется по формуле $w = 6 - p - q$.

Видим, что алгоритм распадается на две задачи:

1. одну простую – перенос одного кольца;
2. другую – перенос башни.

Решение обеих оформим процедурами. Простую назовем **ring**. У нее два входных аргумента: номера исходного и принимающего колышков.

У второй процедуры (назовем ее **tower**) будет три входных аргумента: **m** (количество колец в башне) и номера колышков **p** и **q** (поставляющего кольца и принимающего их соответственно). Заметим, что рекурсивная процедура **tower** должна дважды вызывать сама себя так как башню из $\mathbf{m-1}$ кольца придется переносить дважды.

2.5.2 СИ-решение:

```
#include <iostream>
using namespace std;
void hanoi(int, int, int); void ring(int,int);
int main()
{ int n; cout<< " введи число колец"<<endl; cin>>n; cout<<"   n="<<n<<endl;
  hanoi(n,1,3);
  return 0;
}


#include <iostream>
using namespace std;
void ring(int from, int where)
{ cout<< " с "<<from<<" на "<<where<<endl;}
void hanoi(int n, int p, int q)
{ if (n==1) ring(p,q); else { hanoi(n-1,p,6-p-q); ring(p,q);
                              hanoi(n-1,6-p-q,q);
                            }
}
```

2.5.3 ФОРТРАН-решение:

```
program tsthanoi; implicit none
integer n
write(*,*) 'введи число колец:'; read (*,*) n; write(*,*) ' n=',n
call hanoi(n,1,3)
end

subroutine ring(from,where); implicit none
integer from, where
write(*,*) ' с ',from,' на ',where
end subroutine ring
recursive  subroutine hanoi(n,p,q); implicit none
integer n, p, q
if (n.eq.1) then; call ring(p,q)
            else; call hanoi(n-1,p,6-p-q); call ring(p,q);
                  call hanoi(n-1,6-p-q,q)
endif
end subroutine hanoi

  введи число колец     // Результат СИ-пропуска для m=4:
4
    n=4
1. с 1 на 2   4. с 1 на 2   7. с 1 на 2   A. с 2 на 1   D. с 1 на 2
2. с 1 на 3   5. с 3 на 1   8. с 1 на 3   B. с 3 на 1   E. с 1 на 3
3. с 2 на 3   6. с 3 на 2   9. с 2 на 3   C. с 2 на 3   F. с 2 на 3
```

2.6 Время работы рекурсивной и нерекурсивной функций.

Проведем некоторое сравнение временных затрат при работе рекурсивной и нерекурсивной функций, фиксируя моменты начала и завершения **n**-кратного обращения к ним. Рассмотрим некоторые примеры.

2.6.1 Расчет факториала (временные замеры для СИ).

```
#include <stdio.h>                    //  Файл  timefact.c
#include <time.h>
double factorn(int); double factorr(int);
int main()
{ double fn, fr, t01,t12,t23,t34; long int n=10000000, i,k, t0,t1,t2,t3,t4;
 printf("число нерекурсивных вызовов (n)=%ld\n",n);
 printf("введи аргумент факториала:\n");
 scanf("%d",&k); printf(" k=%d\n",k);
 t0=clock(); fn=factorn(k);  t1=clock(); fr=factorr(k);
 t2=clock(); for (i=1;i<=n;i++) fn=factorn(k);
 t3=clock(); for (i=1;i<=n;i++) fr=factorr(k);
 t4=clock();
 t01=((double)(t1-t0))/CLOCKS_PER_SEC; t12=((double)(t2-t1))/CLOCKS_PER_SEC;
 t23=((double)(t3-t2))/CLOCKS_PER_SEC; t34=((double)(t4-t3))/CLOCKS_PER_SEC;
 printf("             Нерекурсивная       Рекурсивная \n");
 printf("Результат      %15.7le    %15.7le\n",fn,fr);
 printf("Время 1 вызова %15.7le    %15.7le\n", t01, t12);
 printf("Время n вызова %15.7le    %15.7le\n", t23, t34); return 0; }

double factorn(long int n)            //                    Файл  factor.c
{ long int i; double p;               // Нерекурсивная функция расчета n!
  p=1; for (i=1;i<=n;i++) p*=i;       // Находит n! посредством оператора
  return p; }                         // цикла.
double factorr(long int n)            // Рекурсивная функция расчета n!
{ if (n==0) return 1; else  return n*factorr(n-1); }
```

```
$ gcc timefact.c factor.c -O0                  // ..................... -O1
$ time ./a.out                                 // $ time ./a.out
число нерекурсивных вызовов (n)=10000000       // Замеры при оптимизации -O1
введи аргумент факториала:                     //
10                                             //
               Нерекурсивная    Рекурсивная    // Нерекурсивная   Рекурсивная
Результат      3.6288000e+06   3.6288000e+06
Время 1 вызова 0.0000000e+00   0.0000000e+00
Время n вызова 4.6000000e-01   8.4000000e-01   // 2.4000000e-01  2.7000000e-01
real    0m6.520s                               //  real    0m2.192s
user    0m1.294s                               //  user    0m0.517s
sys     0m0.004s                               //  sys     0m0.001s
```

2.6.2 Расчет факториала (временные замеры для gfortran).

```
program timefact;  implicit none                ! Файл timefact.f95
integer(8) :: n=10000000, i, k                  !...................
real*8 fn, fr; real artime(2), t0,t1,t2,t3,t4, t01,t12,t23,t34
interface
  real(8) function factorn(n); integer(8) n; end function factorn;
  recursive real*8 function factorr(n) result(w); integer(8) n
                    end function factorr
end interface
write(*,*) 'число нерекурсивных вызовов (n)=',n
write(*,*) 'введи аргумент факториала:'; read (*,*) k;
call etime(artime,t0); fn=factorn(k); call etime(artime,t1)
                       fr=factorr(k); call etime(artime,t2)
do i=1,n; fn=factorn(k); enddo;       call etime(artime,t3)
do i=1,n; fr=factorr(k); enddo;       call etime(artime,t4)
t01=t1-t0; t12=t2-t1; t23=t3-t1; t34=t4-t3;
write(*,1000); write(*,1001) fn, fr;
write(*,1002) t01, t12; write(*,1003) t23, t34
1000 format(17x,' Нерекурсивная     Рекурсивная')
1001 format(1x,'Результат',6x,e15.7,3x,e15.7)
1002 format(1x,'Время 1 вызова ',e15.7,3x,e15.7)
1003 format(1x,'Время n вызова ',e15.7,3x,e15.7)
end program timefact

function factorn(n); implicit none               !       Файл factor.f95
real(8) factorn, p; integer(8) n, i              !......................
      p=1; do i=1,n; p=p*i; enddo; factorn=p
end function factorn
recursive real(8) function factorr(n) result(w); implicit none
integer(8) n
if (n.eq.0) then;  w=1; else;  w=n*factorr(n-1); endif
end function factorr

gfortran timefact.f95 factor.f95 -O0          // ..................... -O1
time ./a.out                                  // time ./a.out
 число нерекурсивных вызовов (n)=10000000     //  Замеры при оптимизации -O1
 введи аргумент факториала:                   //
10                                            //
                Нерекурсивная    Рекурсивная
 Результат      0.3628800E+07   0.3628800E+07
 Время 1 вызова 0.4999805E-05   0.1000008E-05
 Время n вызова 0.4760330E+00   0.8589249E+00 // 0.2355780E+00  0.3363059E+00
real   0m2.876s                               // real  0m2.536s
user   0m1.336s                               // user  0m0.572s
sys    0m0.002s                               // sys   0m0.002s
```

Как видно, **gfortran**-программа чуть-чуть уступает **gcc**-программе.

2.6.3 C++ расчет n-го числа Фибоначчи

Последовательность Фибоначчи — числовая последовательность, у которой первые два элемента равны единице, а каждый последующий равен сумме двух предыдущих:

$$\mathbf{1}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{5}, \mathbf{8}, \mathbf{13}, \mathbf{21}, \mathbf{34}, \mathbf{55}, \cdots$$

Записи алгоритма расчета **n**-го числа Фибоначчи и в нерекурсивной, и рекурсивной формах достаточно просты:

```
int fibon(int n)                    //                        Файл fibo.cpp
{int i, f1, f2, f3;                 // с описанием fibon, fibor1 и fibor2
 f1=1; f2=1; f3=1;
 for (i=2;i<n;i++) { f3=f1+f2; f1=f2; f2=f3; }  return f3;
}
int fibor1(int n) { if (n<3) return 1;
                       else { return fibor1(n-1)+fibor1(n-2);}
                  }
int fibor2(int k, int n, int fk1, int fk)
{ if (k==n) return fk; else  { return fibor2(k+1,n,fk, fk+fk1);} }
```

fibor1 осуществляет при работе два рекурсивных вызова, причём каждый будет многократно рекурсивно перевычислять числа Фибоначчи, вычисляемые ранее, так как любое из них (после его учёта при расчёте текущего) не запоминается в памяти (кроме последнего).
fibor2 получает конечный результат за счёт одного рекурсивного вызова, так как запоминает предыдущее число в качестве аргумента **fk1**.

```
#include <iostream>
int fibon(int); int fibor1(int); int fibor2(int,int,int,int);
using namespace std;
int main()
{int krep=10000000, i, n, fn, fr1, fr2, t1,t2,t3,t4;
 double t12, t23, t34;
 cout<<" Число нерекурсивных вызовов (krep)="<<krep<<endl;
 cout<<"введите n:"<<endl; cin>>n; cout<<" n="<<n<<endl; t1=clock();
 for (i=1;i<=krep;i++) fn=fibon(n);                      t2=clock();
 for (i=1;i<=krep;i++) fr1=fibor1(n);                    t3=clock();
 for (i=1;i<=krep;i++) fr2=fibor2(1,n,0,1);              t4=clock();
 cout<<"Функция    fibon   "<<"fibor1   "<<"fibor2"<<endl;
 cout<<"Значение      "<< fn <<"       "<< fr1<<"       "<<fr2<<endl;
 t12=double(t2-t1)/CLOCKS_PER_SEC;
 t23=double(t3-t2)/CLOCKS_PER_SEC;
 t34=double(t4-t3)/CLOCKS_PER_SEC;
 cout<<"Время       "<<t12<<"     "<<t23<<"     "<<t34<<endl;
 return 0;
}
```

Ниже даны фрагменты вывода приведённой выше программы при ключах оптимизации **-O0** (слева) и **-O1** (справа).

```
g++ tstfibo.cpp fibo.cpp -c -O0
./a.out
 n=11
Функция    fibon   fibor1   fibor2
Значение      89       89       89
Время       0.44    12.58     0.53
```

```
// g++ tstfibo.cpp -c -O1
// ./a.out
//
//  Функция    fibon   fibor1   fibor2
//
//  Время       0.12     8.25     0.21
```

Работу функции **fibor2** можно ускорить. Назовём её модифицированный вариант **fibor3**):

```
int fibor3(int n, int f[100])
{ int t;
  if ( (n==1) || (n==2) ) { f[n]=1; return 1; }
  if (f[n]) return f[n];
    else {t=fibor3(n-1,f)+fibor3(n-2,f); f[n]=t; return t;}
}
```

fibor3 (в отличие от **fibor2** запоминает не два последовательных числа Фибоначчи, а все ранее вычисленные в соответствующих элементах её второго аргумента — массива, который состоит из достаточного количества элементов. Число Фибоначчи, найденное посредством сложения двух предыдущих при рекурсивном вызове, записывается в элемент массива, так что, когда оно потребуется в последующих рекурсиях, то его можно просто извлечь из этого элемента. Перед вызовом **fibor3** массив следует проинициализировать нулями.

Вообще, для **n**-го числа Фибоначчи известно и явное выражение:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Кажется, что по этой простой формуле можно без всякой рекурсии очень быстро вычислить **n**-ое число Фибоначчи. Это обманчивое впечатление.

```
double fibof(int n)
{ const double x=(1.0+sqrt(5.0))*0.5; const double y=(1.0-sqrt(5.0))*0.5;
  return ( pow(x,(double)n)-pow(y,(double)n) )/sqrt(5.0);
}
```

Нерекурсивная **fibof** требует для расчёта 10-го числа Фибоначчи гораздо больше времени чем **fibon**, **fibor2** или **fibor3**

```
#include <iostream>
#include <cmath>
#define sec(x,y)  ((double)((y)-(x)))/CLOCKS_PER_SEC
using namespace std;
int fibon(int); int fibor1(int); int fibor2(int, int, int, int);
int fibor3(int, int []);  double fibonf(int);
int main()
{int krep=10000000, i, n, fn, fr1, fr2, fr3, t1,t2,t3,t4,t5,t6;
 double t12, t23, t34, t45, t56,ff;
 int work[100];
 for (i=0;i<=100;i++)  work[i]=0;
 cout<<" Число нерекурсивных вызовов (krep)="<<krep<<endl;
 cout<<"введите n:"<<endl; cin>>n; cout<<" n="<<n<<endl;
 t1=clock(); for (i=1;i<=krep;i++)  fn=fibon(n);
 t2=clock(); for (i=1;i<=krep;i++) fr1=fibor1(n);
 t3=clock(); for (i=1;i<=krep;i++) fr2=fibor2(1,n,0,1);
 t4=clock(); for (i=1;i<=krep;i++) fr3=fibor3(n,work);
 t5=clock(); for (i=1;i<=krep;i++)  ff=fibonf(n);  t6=clock();
 cout<<"Функция    fibon   "<<"fibor1   "<<"fibor2   "<<"fibo3    "
                           <<"fibonf   "<< endl;
 cout<<"Значение    "<< fn <<"       "<< fr1<<"       "
                      <<fr2 <<"       "<< fr3<<"       "<<ff<<endl;
 t12=sec(t1,t2);  t23=sec(t2,t3); t34=sec(t3,t4); t45=sec(t4,t5);
 t56=sec(t5,t6);
 cout<<"Время       "<<t12<<"     "<<t23<<"     "<<t34<<
              "     "<<t45<<"     "<<t56<<endl;
 return 0;
}
```

```
 Число нерекурсивных вызовов (krep)=10000000                // Сводная таблица
      Функция    fibon   fibor1   fibor2   fibor3    fiborf // временных затрат
n=10  Время       0.53     7.75     0.79     0.12     3.89  // последней
n=20  Время       1.18  1072.86     1.82     0.12     3.90  // программы.
```

Вывод:

При выборе рекурсивной формы записи функции полезно подумать об экономии времени(сравните времена **fibor1** и **fibor2**).

Вопрос:

"Почему рекурсивная fibor3 гораздо быстрее нерекурсивной fibon?"

2.6.4 ФОРТРАН-расчет n-го числа Фибоначчи.

ФОРТРАН-функции, аналогичные СИ-функциям из пункта **2.6.3**:

```
function fibon(n) result(f3)      ! Нерекурсивная ФОРТРАН-функция
implicit none                     ! расчета n-го  числа Фибоначчи
integer  f1, f2, f3,  n, i        ! через последовательное сложение
f1=1d0; f2=1d0; f3=1d0            ! каждых двух предыдущих чисел
do i=3,n                          ! (ФОРТРАН-аналог C-функции fibon).
  f3=f1+f2; f1=f2; f2=f3
enddo
end
recursive integer function fibor1(n) result(w)   ! Рекурсивный
implicit none                                    ! ФОРТРАН-аналог
integer n                                        ! C-функции
if (n<3) then;  w=1                              !        fibor1.
         else;  w=fibor1(n-1)+fibor1(n-2)
      endif
end
recursive integer function fibor2(k,n,fk1,fk) result(w) ! Рекурсивный
implicit none                                           ! ФОРТРАН-аналог
integer n, k;  integer fk1, fk                          ! C-функции
if (k.eq.n) then; w=fk                                  !        fibor2.
            else; w=fibor2(k+1,n,fk, fk+fk1)
endif
end
recursive integer function fibor3(n,f) result(w)   ! Рекурсивный
implicit none; integer n                           ! ФОРТРАН-аналог
integer f(100)                                     ! C-функции
if ((n.eq.1).or.(n.eq.2)) then; f(n)=1; w=1; endif !          fibor3.
if (f(n).ne.0) then
                     w=f(n)
               else
                     f(n)=fibor3(n-1,f)+fibor3(n-2,f)
                     w=f(n)
endif
end
real(8) function fibof(n) result(w)    ! Нерекурсивная ФОРТРАН-функция
implicit none                          ! расчета n-го числа Фибоначчи
integer n                              ! по формуле, выражающей его
real(8), parameter :: c5=sqrt(5d0)     ! зависимость от номера числа
real(8), parameter :: x=(1d0+c5)*0.5d0;! (ФОРТРАН-аналог
real(8), parameter :: y=(1d0-c5)*0.5d0;!  C-функции      fibof).
w=(x**n-y**n)/c5
end
```

Тестирующая программа для подсчёта времени использует встроенную функцию **second** (см. **Приложение III**):

```
program tstfibo4; implicit none; integer, parameter :: krep=10000000
integer  i, n, fn, fr1, fr2, fr3, fk1, fk, j
real  t1,t2,t3,t4,t5,t6, t12, t23, t34, t45, t56; real*8 ff
integer work(100);
interface; integer function fibon(n); integer n;            end function fibon
 recursive integer function fibor1(n) result(w); integer n; end function fibor1
 recursive integer function fibor2(k,n,fk1,fk) result(w)
                   integer k, n, fk1, fk;                    end function fibor2
 recursive integer function fibor3(n,f) result(w);
                  integer n;  integer f(100);                end function fibor3
 real(8) function fibof(n); integer n;                      end function fibof
end interface
work=0;   write(*,*) ' Число нерекурсивных вызовов (krep)=', krep
          write(*,*) ' введите n:'; read(*,*) n; write(*,*) ' n=',n
call second(t1);    do i=1,krep;   fn=fibon(n);            enddo
call second(t2);    do i=1,krep;   fr1=fibor1(n);          enddo
fk1=0d0; fk=1d0
call second(t3);    do i=1,krep;   fr2=fibor2(1,n,fk1,fk); enddo
call second(t4);    do i=1,krep;   fr3=fibor3(n,work);     enddo
call second(t5);    do i=1,krep;   ff=fibof(n);            enddo
call second(t6);
write(*,1000); write(*,1001) fn, fr1, fr2, fr3, ff
t12=t2-t1;  t23=t3-t2; t34=t4-t3; t45=t5-t4; t56=t6-t5
write(*,1002) t12, t23, t34, t45, t56
 1000 format( 1x,'Функция',5x,'fibon',7x,'fibor1',7x,'fibor2',&
                         7x,'fibor3',7x,'fiborf')
 1001 format(1x,'Значение',i10,2x,i10,2x,i10,2x,i10,6x,e11.4)
 1002 format(1x,'Время',3x,e11.4,2x,e11.4,2x,e11.4,2x,e11.4,2x,e11.4)
 end
```

Таблица сравнения временных затрат g++ и gfortran:

```
 Число нерекурсивных вызовов (krep)=10000000.              Оптимизация -O0
      Функция    fibon   fibor1   fibor2    fibo3    fiborf
n=10  Время       0.53     7.75     0.79     0.12     3.89    g++
      Время       0.60     8.42     0.95     0.12     1.250   gfortran
n=20  Время       1.18   1072.86    1.82     0.12     3.90    g++
      Время       1.38   1103.      2.24     0.12     1.35    gfortran

      Функция    fibon   fibor1   fibor2    fibo3    fiborf   Оптимизация -O1
n=10  Время       0.17     7.82     0.71     0.09     3.39    g++
      Время       0.18     8.14     0.97     0.09     0.98   gfortran
n=20  Время       0.36   1023.      1.82     0.08     3.65    g++
      Время       0.38   1044.      2.26     0.09     1.08    gfortran
```

Компилятор **gfortran** в режиме **O0** слегка уступает компилятору **g++** при работе с данными типа **integer**; при работе с одномерным массивом вообще не уступает, а с данными типа **real(8)** (см. колонку **fibof**) операция возведения в степень **gfortran**а работает чуть ли не втрое быстрее.

2.6.5 Поиск корня методом дихотомии (ФОРТРАН)

Среди численных методов поиска изолированного на промежутке **[a,b]** корня уравнения **f(x)=0** наиболее надежным (но не слишком быстрым) является метод деления пополам отрезка, содержащего корень. Суть метода в сужении этого отрезка вдвое до тех пор, пока не получим искомый корень с требуемой точностью (см. приложение **bisec**).

```
! Процедура bisecn0(aa,bb,epsx,epsy,res,iter,ier) находит методом деления
! отрезка [aa,bb] пополам изолированный на нём корень уравнения f(x)=0.
! Входные аргументы: aa - левая граница промежутка;  bb - правая граница;
!                  epsx - точность по аргументу; epsy - точность по функции.
! Выходные аргументы: res - найденная величина корня;
!   k  - число уточнений, за которое достигнуты требуемые критерии точности;
!  ier - код причины завершения работы bisecn0:
!   0  - достигнуты критерии точности и по функции, и по аргументу;
!   1  - число дроблений отрезка > kmax=32000 (но точность не достигнута);
!   2  - корень не искался, т.к. f(aa) и f(bb) одного знака (нет корня).
subroutine bisecn0(aa,bb,epsx,epsy,res,k,ier); implicit none
real(8) aa, bb, epsx, epsy, res, f, a, b, c, fa, fb, fc
integer k, ier, kmax / 1000 /; a=aa; b=bb; fa=f(a); fb=f(b)
if ( (fa*fb).le.0d0 ) then   ! Если f(x) перескает ось абсцисс, то:
  k=0                        ! Обнуляем число сужений промежутка.
  do; c=(a+b)*0.5d0; fc=f(c) ! Находим середину промежутка (c) и f(c).
    if ( fa*fc.gt.0d0) then  !   Если корень в правой половине отрезка,
      a=c; fa=fc;      else  !                     смещаем левую границу
      b=c; fb=fc             !   если в левой -   смещаем правую.
    endif                    !
    k=k+1                    !   Увеличиваем счетчик сужений
    if ((k.gt.kmax).or.&        ! Если число сужений больше допустимого
&     (dabs(b-a).lt.epsx.and.&  ! ИЛИ одновременно достигнуты точности
&      dabs(fc ).lt.epsy)) exit ! и по аргументу и по функции, то
  enddo                      ! выходим из цикла.
  res=(a+b)*0.5d0            ! Фиксируем найденное значение корня.
  if (k.le.kmax) then; ier=0 ! Если число сужений в норме, то код = 0
                 else; ier=1 !  иначе код завершения = 1
  endif
  else; ier=2          ! Если f(x) НЕ ПЕРЕСЕКАЕТ ось абсцисс, код = 2
endif
end subroutine bisecn0
```

Здесь пересылки **a=aa** и **b=bb** обеспечивают неизменность исходных значений **aa** и **bb**. Пересылки были бы излишни, если бы ФОРТРАН допускал передачу параметров по значению. Обойтись без пересылок можно, если при вызове **bisecn0** заключить фактические аргументы (a) и (b) в круглые скобки.

В рекурсивной форме алгоритм выглядит еще проще:

```
! Рекурсивная подпрограмма bisecr0(aa,bb,epsx,epsy,res,iter,ier) находит
! методом деления отрезка [aa,bb] пополам изолированный на нем корень
! уравнения f(x)=0.
! Входные аргументы: aa - левая граница промежутка;   bb - правая граница;
!                   epsx - точность по аргументу;    epsy - точность по функции;
! Выходные аргументы: res - найденная величина корня;
!  k  - число рекурсий, за которое достигнуты требуемые критерии точности;
! ier - код причины завершения работы bisecr0:
!  0  - достигнуты критерии точности и по функции, и по аргументу;
!  1  - число дроблений отрезка [a,b] > kmax=1000, но точностные
!       критерии либо не достигнуты, либо достигнут лишь один из них;
!  2  - корень не искался, так как на концах промежутка [aa,bb]
!            функция f(x) имеет значения одного знака.
recursive subroutine bisecr0(a,b,epsx,epsy,res,k,ier);  implicit none
real(8) aa, bb, epsx, epsy, res, f, a, b, c, fa, fb, fc
integer k, ier, kmax / 1000 /
!      a=aa;  b=bb;
fa=f(a);  fb=f(b)
if ( (fa*fb).le.0d0 ) then               ! Пересекает ли f(x) ось абсцисс?
  res=(a+b)*0.5d0; c=res; fc=f(c)        ! Да! Находим середину
  if ( (dabs(b-a).lt.epsx) .and.&         ! Если выполнены точностные
&      (dabs( fc).lt.epsy) ) then; ier=0 ! критерии, то ier=0
    else
      if (k.gt.kmax) then; ier=1    ! Если число уточнений > kmax, то ier=1
        else; c=(a+b)*0.5d0; fc=f(c) ! В противном случае продолжаем
              k=k+1;                 ! уточнения.
              if (f(a)*fc.gt.0) then
                call bisecr0(c,b,epsx,epsy,res,k,ier)
                else; call bisecr0(a,c,epsx,epsy,res,k,ier)
              endif
      endif
  endif
  else; ier=2 ! f(x) не пересекает ось абсцисс.
endif
end subroutine bisecr0
```

В качестве пробного примера выберем решение уравнения **3*x-1=0**.

```
! Подпрограмма-функция f(x) для заданного аргумента x вычисляет значение
! левой части уравнения 3*x-1=0.
function f(x) result(w)
implicit none
real(8) w, x
w=3*x-1
end
```

Программа, тестирующая процедуры **bisecn0** и **bisecr0**, вводит из файла **input** текущей директории: границы отрезка **[aa,bb]** и абсолютные погрешности поиска корня по абсциссе и ординате (**epsx** и **epsy** соответственно). Для оценки времени работы каждая из процедур вызывается **krep=1000** раз. Ниже приводится текст тестирующей программы и сводная табличка некоторых результатов.

```
! Программа предназначена для тестирования двух подпрограмм поиска
! изолированного корня уравнения f(x)=0 методом деления отрезка пополам:
!    1) нерекурсивной bisecn(aa,bb,epsx,epsy,res,iter,ier)
!    2) рекурсивной   bisecr(aa,bb,epsx,epsy,res,iter,ier)
program tstbis; implicit none
real(8) f, aa, bb, epsx, epsy, resn, resr; real t1, t2, t3, t4
character(8) mess(2) /'bisecn0','bisecr0'/
integer itern, iterr, iern, ierr, i
integer ninp / 5 /,  nres / 6 /, krep / 1000000 /
open (unit=ninp,file='input')
open (unit=nres,file='result',status='replace')
read (ninp, '(d15.7)') aa, bb, epsx, epsy
write(nres,1100) aa, bb, epsx, epsy, krep
call second(t1); do i=1,krep
                   call bisecn0(aa,bb,epsx,epsy,resn,itern,iern)
                 enddo; call second(t2);
write(nres,1101) resn, f(resn), itern, iern, t2-t1, mess(1)
call second(t3); do i=1,krep; iterr=0
                   call bisecr0(aa,bb,epsx,epsy,resr,iterr,ierr)
                 enddo;  call second(t4)
write(nres,1101) resr, f(resr), iterr, ierr, t4-t3, mess(2); close(nres)
 1100 format(1x,'   a=',d15.7,'    b=',d15.7/ 1x,'epsx=',d15.7,&
&  ' epsy=',d15.7,'  krep=',i10/13x,'res',14x,'f(res)',3x,'iter',3x,'ier',&
&       3x,'Время',3x,'Подпрограмма')
 1101 format(1x,d25.17,2x,d9.2,2x,i5,2x,i2,2x,f7.3,6x,a)
end
```

```
    a=  0.0000000D+00    b=  0.1000000D+01       krep=1000
         res                f(res)   iter ier   Время  Подпрограмма  epsx  epsy
0.33333206176757812D+00  -0.38D-05   17   0    0.002     bisecn0    1d-5  1d-5
0.33333206176757812D+00  -0.38D-05   17   0    0.005     bisecr0

0.33333333333333337D+00   0.11D-15 32001  1    3.148     bisecn0    1d-25 1d-4
0.33333333333333337D+00   0.11D-15 32001  1   10.948     bisecr0

0.33333333333333337D+00   0.11D-15  1001  1    0.102     bisecn0    1d-5  1d-18
0.33333333333333337D+00   0.11D-15  1001  1    0.319     bisecr0

0.33333333333333348D+00   0.44D-15   50   0    0.005     bisecn0    1d-15 1d-15
0.33333333333333348D+00   0.44D-15   50   0    0.015     bisecr0
```

2.6.6 Поиск корня методом дихотомии (C++)

Ниже приводятся соответствующие задаче исходные C++ тексты: тестирующей программы, функции расчета левой части решаемого уравнения, нерекурсивной и рекурсивной функций, реализующих метод деления отрезка пополам, а также результаты пропуска с оценкой временных затрат **g++**- и **gfortran**-исполнимых кодов (без оптимизации), решающих уравнение **3*x-1=0** миллион раз.

```
#include <iostream>                                          // Файл tstbis.cpp
#include <fstream>                                           // ===============
#include <iomanip>
#define sec(x,y)  ((double)((y)-(x)))/CLOCKS_PER_SEC
using namespace std;
double f(double);
void bisecn0(double,double,double,double,double&,int&,int&);
void bisecr0(double,double,double,double,double&,int&,int&);
int main()
{double a, b, epsx, epsy, resn, resr; int t1, t2, t3, t4;
 char* mess[2]={"bisecn0","bisecr0"};
 int itern, iterr, iern, ierr, i, krep=1000000;
 ifstream ninp ("cinput"); if (!ninp) { cout<<"Файл input НЕ ОТКРЫТ !!!"<<endl;
                                        return 1;}
 ofstream nres("result"); if (!nres){ cout<<"Файл result НЕ ОТКРЫТ !!!"<<endl;
                                        return 2;}
 ninp >> a >> b >> epsx >> epsy; ninp.close();
 nres <<scientific; nres.precision(13);
 nres << "    a=" <<  a  << "    b=" <<   b  << "  krep=" << krep << endl;
 nres << " epsx="<< epsx << " epsy=" << epsy << endl;
 nres << "        res    "<<"             f(res)    "<< "       k  "
      << " ier "<< "   Время  "<<" Функция    "<<endl;
 t1=clock(); for (i=1;i<=krep;i++)
                { bisecn0(a,b,epsx,epsy,resn,itern,iern);}
 t2=clock(); nres<<scientific<<resn<<"   "<<f(resn)
                 <<"   "<<itern<<"   "<<iern<<"   "
                 <<fixed<<setw(7)<<setprecision(3)<<sec(t1,t2)
                 <<"   "<<mess[0]<<endl;
 t3=clock(); for (i=1;i<=krep;i++)
           { iterr=0; bisecr0(a,b,epsx,epsy,resr,iterr,ierr); }
 t4=clock(); nres<<scientific<<setprecision(13)<<resr<<"   "<<f(resr)
                 <<"   "<<iterr<<"   "<<ierr<<"   "
                 <<fixed<<setw(7)<<setprecision(3)<<sec(t3,t4)
                 << "   "<<mess[1]<<endl;
 nres.close();
}

double f(double x) {return 3*x-1;}
```

```
#include <cmath>
double f(double);
void bisecn0(double a, double b, double epsx, double epsy,
                              double &res,int &k, int &ier)
{double c, fa, fb, fc;
 const int kmax=1000;   fa=f(a); fb=f(b);
 if ( (fa*fb)<=0.0)
   { k=0; do { c=(a+b)*0.5; fc=f(c);
               if ( fa*fc>0.0) { a=c; fa=fc;} else { b=c; fb=fc;} k+=1;
             }
     while ( (k<=kmax) && ( (fabs(b-a)>epsx) || ( fabs(fc)>epsy)) );
     res=(a+b)*0.5;
     if (k<=kmax) ier=0; else ier=1;
    }
 else ier=2;
}
```

```
#include <cmath>
double f(double);
void bisecr0(double a,double b,double epsx,double epsy,
                           double &res,int &k,int &ier)
{ double  c, fa, fb, fc; int kmax=1000;  fa=f(a);  fb=f(b);
  if ( (fa*fb)<=0.0 )
    { res=(a+b)*0.5; c=res; fc=f(c);
        if ( (fabs(b-a)<epsx) && (fabs( fc)<epsy) ) ier=0;
        else { if (k>kmax) ier=1;
                 else { c=(a+b)*0.5; fc=f(c);
                        k=k+1;
                        if (f(a)*fc>0.0) bisecr0(c,b,epsx,epsy,res,k,ier);
                          else           bisecr0(a,c,epsx,epsy,res,k,ier);
    }       }         }
  else ier=2;
}
```

```
                                                                  g++
    a=0.0000000000000e+00    b=1.0000000000000e+00  krep=1000000  -----
 epsx=1.0000000000000e-05 epsy=1.0000000000000e-05
        res                  f(res)           k   ier    Время   Функция
3.3333206176758e-01   -3.8146972656250e-06   17   0     1.760   bisecn0
3.3333206176758e-01   -3.8146972656250e-06   17   0     4.310   bisecr0

    a=  0.0000000D+00    b=  0.1000000D+01                      gfortran
 epsx=  0.1000000D-04 epsy=  0.1000000D-04  krep=   1000000     --------
             res              f(res)   iter   ier   Время   Подпрограмма
   0.33333206176757812D+00  -0.38D-05     17   0    1.895      bisecn0
   0.33333206176757812D+00  -0.38D-05     17   0    5.354      bisecr0
```

2.7 Понятие о хвостовой рекурсии

Хвостовая рекурсия — особый вид рекурсивного описания функции, когда рекурсивный вызов оказывается её последним действием. В этом случае рекурсию можно заменить на итерацию (т.е. на оператор цикла, исключая временные затраты на рекурсивные вызовы и не используя стековую память для хранения адресов возврата), что автоматически реализуется во многих компиляторах, если при компиляции, указываются соответствующие **опции оптимизации**. Продемонстрируем эффект использования хвостовой рекурсии на примере расчёта факториала.

2.7.1 СИ (компилятор gcc)

```
typedef double real;

#include "mytype.h"
real factorn(long int n)              //                   Файл  factorx.c
{ long int i; real p;                 // Нерекурсивная функция расчета n!
  p=1.0; for (i=1;i<=n;i++) p*=i;     // Находит n! посредством оператора
  return p; }                         // цикла.
real factorr1(long int n)             // Рекурсивная функция расчета n!
{ if (n==0) return 1.0;               // по формуле f=n*f(n-1).
    else    return n*factorr1(n-1); }
real factorr2(long int n)             // Рекурсивная функция расчета n!
{ if (n==0) return 1.0;               // по формуле f=f(n-1)*n.
    else    return factorr2(n-1)*n; }
real ftimes(long int n, real res)  // ХВОСТОВАЯ РЕКУРСИЯ: Функция умножения
{ if (n==0) return res;            // второго  аргумента на все натуральные
    else return ftimes(n-1,res*n); //  от 1  до значения первого.
}
real factorxx(long int n)          // Процедура вызова рекурсивной функции
{ return ftimes(n,1);              // с хвостовой рекурсией.
}
```

В файле **factorx.c** помещены описания пяти функций:

1. **factorn** — нерекурсивная;
2. **factorr1** — рекурсивная (по схеме n!=n*factorr1(n-1));
3. **factorr2** — рекурсивная (по схеме n!=factorr2(n-1)*n);
4. **ftimes** — рекурсивная с **хвостовой рекурсией**;
5. **factorx** — функция вызова рекурсивной **ftimes**.

Каждая из них вызывается в главной программе по 10000000 раз для расчёта значений **k!** (**k=50, 75, 100, 125, 150**) с вычислением соответствующих временных затрат.

```
#include <stdio.h>                    //  Файл  timefact.c
#include <time.h>
#include "mytype.h"
real factorn (long int); real factorr1(long int);
real factorr2(long int); real ftimes  (long int,real);
real factorxx(long int);
int main()
{ real  fn, fr1, fr2, fr3, frx;
  double t01,t12,t23,t34,t45;
  double s01,s12,s23,s34,s45;
  long int n=10000000, i, k, t0,t1,t2,t3,t4,t5;
  s01=s12=s23=s34=s45=0;
  printf(" Число нерекурсивных вызовов (n)=%ld\n",n);
  printf("%25s %30s\n"," Форма описания функции:",
                 "Нерекурсивная      Рекурсивная");
  printf("%73s\n","-1-       -2-       -3-       -4-");
  for (k=50; k<=150; k+=25)
    {
      t0=clock(); for (i=1;i<=n;i++) fn=factorn (k);
      t1=clock(); for (i=1;i<=n;i++) fr1=factorr1(k);
      t2=clock(); for (i=1;i<=n;i++) fr2=factorr2(k);
      t3=clock(); for (i=1;i<=n;i++) fr3=ftimes(k,1.0);
      t4=clock(); for (i=1;i<=n;i++) frx=factorxx(k);
      t5=clock();
      t01=((double)(t1-t0))/CLOCKS_PER_SEC;
      t12=((double)(t2-t1))/CLOCKS_PER_SEC;
      t23=((double)(t3-t2))/CLOCKS_PER_SEC;
      t34=((double)(t4-t3))/CLOCKS_PER_SEC;
      t45=((double)(t5-t4))/CLOCKS_PER_SEC;
      s01+=t01; s12+=t12; s23+=t23; s34+=t34; s45+=t45;
      printf(" Время расчёта %5ld!: %10.2lf%10.2lf%10.2lf%10.2lf%10.2lf\n",
                  k,t01,t12,t23,t34,t45);
  }
 printf(" Итоги по колонкам   : %10.2lf  %8.2lf  %8.2lf  %8.2lf  %8.2lf\n",
                      s01, s12, s23, s34, s45);
 printf(" Полное время  :%10.2lf\n", s01+s12+s23+s34+s45);
 printf("\n     Значение   150!  =   %8.1le  %8.1le  %8.1le  %8.1le  %8.1le\n",
        fn,fr1,fr2,fr3,frx);
 return 0;
}
```

На следующей странице приведены результаты пропусков этой программы при различных уровнях оптимизации.

```
Оптимизация -O0.          factorn   factorr1  factorr2   ftimes  factorrx
----------------
 Время расчёта    50!:      2.62      2.94      3.09      8.60      8.74
 Время расчёта    75!:      4.30      4.55      4.73     13.13     13.27
 Время расчёта   100!:      5.70      6.12      6.29     17.41     17.56
 Время расчёта   125!:      7.08      7.59      7.81     21.70     21.84
 Время расчёта   150!:      8.48      9.10      9.41     25.99     26.12
 Итоги по колонкам   :     28.18     30.30     31.33     86.83     87.53
       Полное время  :   264.17

Оптимизация -O1.          factorn   factorr1  factorr2   ftimes  factorrx
----------------
 Время расчёта    50!:      1.05      2.78      2.80      3.82      3.93
 Время расчёта    75!:      2.04      4.46      4.46      5.88      5.99
 Время расчёта   100!:      2.67      5.90      5.92      7.76      7.89
 Время расчёта   125!:      3.29      7.24      7.24      9.65      9.78
 Время расчёта   150!:      3.93      8.60      8.61     11.55     11.67
 Итоги по колонкам   :     12.98     28.98     29.03     38.66     39.26
       Полное время  :   148.91

Оптимизация: -O2.         factorn   factorr1  factorr2   ftimes  factorrx
-----------------
 Время расчёта    50!:      1.04      2.85      2.85      1.00      0.99
 Время расчёта    75!:      2.00      4.35      4.35      2.00      2.03
 Время расчёта   100!:      2.62      5.74      5.74      2.64      2.65
 Время расчёта   125!:      3.26      7.09      7.10      3.27      3.28
 Время расчёта   150!:      3.89      8.53      8.51      3.90      3.91
 Итоги по колонкам   :     12.81     28.56     28.55     12.81     12.86
       Полное время  :    95.59

Оптимизация: -O3.         factorn   factorr1  factorr2   ftimes  factorrx
-----------------
 Время расчёта    50!:      1.04      2.13      1.71      1.00      0.99
 Время расчёта    75!:      2.00      3.15      2.63      2.01      2.02
 Время расчёта   100!:      2.63      4.30      3.59      2.64      2.65
 Время расчёта   125!:      3.26      5.27      4.47      3.26      3.28
 Время расчёта   150!:      3.89      6.39      5.58      3.89      3.91
 Итоги по колонкам   :     12.82     21.24     17.98     12.80     12.85
 Полное время  :     77.69
```

Как видим, при опциях оптимизации **-O2** и **-O3** время работы функции с хвостовой рекурсией практически совпадает со временем работы нерекурсивной.

2.7.2 ФОРТРАН-95 (компилятор gfortran)

Поместим ФОРТРАН-аналоги СИ-функций **factorn**, **factorr1**, **factorr2**, **ftimes** и **factorrx** в модуль **myfactor.f95**:

```
module myfactor
implicit none
contains
  function factorn(n)
  real   (8) factorn, p
  integer(8) n, i
  p=1d0; do i=1,n
           p=p*i
         enddo
  factorn=p
  end function factorn
  recursive real(8) function factorr1(n) result(w)
  integer(8) n
  if (n.eq.0) then;  w=1d0
              else;  w=n*factorr1(n-1)
  endif
  end function factorr1
  recursive real(8) function factorr2(n) result(w)
  integer(8) n
  if (n.eq.0) then; w=1d0
              else; w=factorr2(n-1)*n
  endif
  end function factorr2
  recursive real(8) function ftimes(n,res) result(w)
  integer(8) n
  real(8)res
  if (n.eq.0) then; w=res
              else; w=ftimes(n-1,res*n)
  endif
  end function ftimes
  real(8) function factorxx(n) result(w)
  integer(8) n
  w=ftimes(n,1d0)
  end function factorxx
  end module myfactor
```

```
 program timefact; use myfactor; implicit none   !  Файл timefact.f9
  integer(8) :: n=10000000, i, k                 !...................
 real(8) fn, fr1, fr2, fr3, frx
 real t0,t1,t2,t3,t4,t5, t01,t12,t23,t34,t45
 real s01,s12,s23,s34,s45
 s01=0.0; s12=0.0; s23=0.0; s34=0.0; s45=0.0
 write(*,*) ' Число нерекурсивных вызовов (n)=',n
 write(*,*) ' Форма описания функции :  Нерекурсивная      Рекурсивная'
 write(*,'(a75)') '-1-       -2-       -3-       -4-'
 do k=50,150,25
   call cpu_time(t0); do i=1,n; fn=factorn(k);     enddo
   call cpu_time(t1); do i=1,n; fr1=factorr1(k);   enddo
   call cpu_time(t2); do i=1,n; fr2=factorr2(k);   enddo
   call cpu_time(t3); do i=1,n; fr3=ftimes(k,1d0); enddo
   call cpu_time(t4); do i=1,n; frx=factorxx(k);   enddo
   call cpu_time(t5);
   t01=t1-t0; t12=t2-t1; t23=t3-t2; t34=t4-t3; t45=t5-t4
   s01=s01+t01; s12=s12+t12; s23=s23+t23; s34=s34+t34; s45=s45+t45
   write(*,&
&   '('' Время расчёта '',i5,''!....:'',f10.2,f10.2,f10.2,f10.2,f10.2)')&
&                                              k,t01,t12,t23,t34,t45
 enddo
 write(*,'(" Итоги по колонкам       :",&
&  f10.2,2x,f8.2,2x,f8.2,2x,f8.2,2x,f8.2)') s01, s12, s23, s34, s45
 write(*,'(" Полное время  :",f10.2)') s01+s12+s23+s34+s45
 write(*,'(/"          Значение",i5,''!  =  '',1P,e8.1,2x,e8.1,2x,e8.1,&
&            2x,e8.2,2x,e8.1/)')  k,fn,fr1,fr2, fr3, frx
 end

Оптимизация: -O3.         factorn   factorr1  factorr2   ftimes  factorrx
 Время расчёта  50!....:    5.63      6.29      2.83      5.75      5.80
 Время расчёта  75!....:    8.48      9.50      4.21      8.59      8.65
 Время расчёта 100!....:   11.25     12.61      5.58     11.56     11.53
 Время расчёта 125!....:   14.03     15.70      6.96     14.38     14.37
 Время расчёта 150!....:   16.80     18.92      8.54     17.33     17.48
 Итоги по колонкам     :   56.19     63.02     28.11     57.61     57.83
 Полное время  :    262.77
```

Ключ оптимизации **-O3** при использовании компилятора **gfortran** реализует наиболее быстрый способ расчёта факториала на основе рекурсивной функции **factorr2**, работающей по схеме: **n!=(n-1)!*n** (вдвое быстрее !!! чем нерекурсивная **factorn**). Использование хвостовой рекурсии (хоть и сократило время расчёта до времени работы **factorn**), тем не менее, привело к более медленному алгоритму нежели схема **factorr2**, которая в свою очередь оказалась вдвое медленней исполнимого СИ-кода с явным оформлением хвостовой рекурсии.

2.8 О чем узнали из второй главы? (2-ой семестр)

1. **Рекурсия** – способ определения какого-либо понятия через само это понятие.

2. **Рекурсивная процедура** – это процедура явно или неявно (т.е. через посредника), обращающаяся сама к себе.

3. Всякий алгоритм может быть описан как в рекурсивной форме, так и в нерекурсивной. Способ выбирает программист.

4. В нерекурсивном алгоритме аналог рекурсии моделируется оператором цикла.

5. Старые версии **ФОРТРАН**а не допускали рекурсиных процедур.

6. Современный ФОРТРАН допускает рекурсивные процедуры, однако, в отличие от СИ, ФОРТРАН-процедура по умолчанию считается нерекурсивной. (рекурсивности описания ФОРТРАН-процедуры необходимо указать особо служебным словом **recursive**).

7. В C и C++ изначально предполагается возможность рекурсии.

8. При описании рекурсивной функции всегда должно быть задано условие прекращения ее рекурсивного вызова.

9. Глубина рекурсии – число промежуточных рекурсивных вызовов функции.

10. Запись рекурсивной функции обычно выглядит проще нерекурсивной. Однако время её работы может оказаться значительно больше (из-за временных затрат на каждый рекурсивный вызов), а отладка – сложнее. Поэтому при выборе рекурсивного описания функции выгодно уменьшать количество рекурсивных вызовов, запоминая (если возможно) их результаты (например, в массиве).

11. **Хвостовая рекурсия** — описание рекурсивного алгоритма так, что его рекурсивный вызов оказывается последним выполняемым действием.

2.9 Второе домашнее задание (2-ой семестр)

Каждая программа, тестирующая процедуры, должна оценивать время работы соответственно обоих вариантов расчета (нерекурсивного и рекурсивного), обеспечивая ввод исходных данных из файла и вывод результата в файл.
make-файл должен инициировать выполнение программы, используя утилиту **time**.

1. Разработать на ФОРТРАНе и СИ нерекурсивную и рекурсивную процедуры

   a) целочисленного деления одного целого числа на другое, моделируя операцию деления операциями вычитания;

   b) нахождения остатка от целочисленного деления одного целого на другое, моделируя операцию нахождения остатка операциями вычитания;

   c) перевода введенного целого в двоичную систему счисления

   d) возведения основания в неотрицательную целочисленную степень, моделируя операцию возведения в степень операциями умножения.

   e) перевода введенного неотрицательного меньшего единицы вещественного числа в двоичную систему счисления;

   f) которые выясняют, является ли заданное натуральное простым;

2. Продемонстрировать на ФОРТРАНе и СИ работоспособность алгоритма **Ханойская башня** (см. лекцию).

3. Продемонстрировать на ФОРТРАНе и СИ работоспособность всех лекционных алгоритмов, реализующих расчет чисел Фибоначчи.

4. Продемонстрировать на ФОРТРАНе и СИ работу лекционных алгоритмов поиска изолированного корня уравнения f(x)=0 методом половинного деления.

5. Добавить к программе тестирования рекурсивной и нерекурсивной функций моделирования операции возведения в степень через посредство операции умножения вызов соответствующей рекурсивной функции, использующей схему хвостовой рекурсии.

3 Немного о массивах, структурах и указателях

3.1 Массивы

В программировании одни простые переменные не всегда удобны. При необходимости сопоставить одному имени много значений, одновременно хранимых в оперативной памяти (например, матрицу) используют структуру данных **массив** – набор конечного количества нумерованных элементов **одинакового типа**. Массив называют **одномерным** (или вектором), если в его описании указано, что все элементы распределены вдоль одного измерения (указать элемент можно его порядковым номером). Так положение точки задаётся вектором, содержащим три элемента (первый — абсцисса, второй — ордината, третий — апликата):

3.1.1 ФОРТРАН

```
program tvector; implicit none !                    Файл tvector.f95
real  a(3), c(3), d(3)         ! Число в скобках - число элементов.
real  b(3) / 1.0, 2.0, 3.0 /   ! Старый способ инициализации вектора;
data  c / 4.0, 5.0, 6.0 /      ! ещё один старый способ инициализации;
data  d / 3 * 8.0 /            ! ещё один старый способ инициализации;
real, dimension (3) :: g=3*8.0 ! Описание массива возможное в ФОРТРАНе-90
integer i; real :: e(3)=(/ 7.0, 8.0, 9.0 /) ! Пример инициализации массива
real, dimension (3) :: f=(/(5*i,i=3,1,-1)/) ! через его конструктор (/ /).
a=(/ 10, 11, 12 /)                          ! Справа неименованный массив!
write(*,*) ' a=',a; write(*,*) ' b=',b      ! Фортран позволяет для вывода
write(*,*) ' c=',c                          ! всего массива указать только
write(*,*) (a(i),i=1,3); write(*,*) (b(i),i=1,3,2) ! его имя, хотя возможен
write(*,*) (c(i),i=3,1,-1)                  ! и поэлементный вывод в виде
write(*,*) ' d=',d;                         ! циклоподобного списка вывода.
write(*,*) ' g=',g                          !
end
```

```
  a=   10.000000       11.000000       12.000000    ! Результат
  b=   1.0000000       2.0000000       3.0000000    ! программы tvector
  c=   4.0000000       5.0000000       6.0000000
   10.000000       11.000000       12.000000
   1.0000000       3.0000000
   6.0000000       5.0000000       4.0000000
  d=   8.0000000       8.0000000       8.0000000    ! Сравните содержимое
  g=   24.000000       24.000000       24.000000    !      d  и  g
```

В ФОРТРАНе описание массива допускает изменение нумерации (индексации) элементов. Например, **real a(-1:1)** или **real b(11:13)**. Правда, вряд ли подобное изменение удобно для описания положения точки.

3.1.2 СИ

```
#include <stdio.h>  // СИ-пример аналогичного описания координат точки.
int main(void)
{ float  a[3];                    // Число в скобках - число элементов.
  float  b[3]={ 1, 2, 3 };        // Инициализация вектора при описании.
  float e[]={4, 5, 6};            // В СИ число элементов может определиться
  int i;                          // по числу инициализирующих значений.
  a[0]=10.0; a[1]=11.0; a[2]=12.0;        printf(" a=%p\n",a);
  printf(" b=%e %e %e\n",b[0],b[1],b[2]);
  for (i=0;i<=2;i++)
  { printf("%e...",e[i]);}  printf("\n");
  printf("Адрес a[0]=%p, a содежимое a[0]=%e \n",    a , *a);
  printf("Адрес a[0]=%p, a содежимое a[0]=%e \n", &a[0], *a);
  printf("Адрес a[1]=%p, a содежимое a[1]=%e \n", (a+1), *(a+1));
  printf("Адрес a[1]=%p, a содежимое a[1]=%e \n", &a[1], *(a+1));
  printf("Адрес a[2]=%p, a содежимое a[2]=%e \n", (a+2), *(a+2));
  printf("Адрес a[2]=%p, a содежимое a[2]=%e \n", &a[2], *(a+2));
  return 0;
}
```

1. СИ-индексация массива всегда начинается с нуля.

2. Имя СИ-массива есть **указатель** (адрес) его начального элемента. По имени массива в сочетании с операцией разыменования выводится значение лишь начального элемента (в ФОРТРАНе — все).

3. Функция форматного вывода **printf** предоставляет для вывода указателя спецификатор **p**.

4. Синтаксически доступ к содержимому нужного элемента массива можно оформить двояко: по традиции **a[0]**, **a[1]**, **a[2]** или через операцию разыменования указателя ***(a+i)**. Запись ***(a+i)**, где **i** — **индекс** элемента, означает:

   1) извлечь адрес **&a[0]** начального элемента переменной **a**;
   2) увеличить его на **i*sizeof(тип_элемента)**, т.е. на число байт, нужное для **i** элементов (с 0 по (i-1)), вычисляя адрес **a[i]**-го;
   3) разыменовать указатель ***(a+i)**, получая доступ к **a[i]**.

5. Операция выделения элемента массива посредством заключения его индекса в квадратные скобки имеет более высокий приоритет нежели операция **&** (взятия адреса). Поэтому, например, при записи выражения **&a[0]** можно **a[0]** не заключать в круглые скобки.

6.
```
 a=0xbf99afd0                                          // Результат пропуска
  b=1.000000e+00 2.000000e+00 3.000000e+00             // СИ-программы из
4.000000e+00...5.000000e+00...6.000000e+00...         // этого пункта.
Адрес a[0]=0xbf99afd0, a содежимое a[0]=1.000000e+01 // printf по формату
Адрес a[0]=0xbf99afd0, a содежимое a[0]=1.000000e+01 // %p выводит значение
Адрес a[1]=0xbf99afd4, a содежимое a[1]=1.100000e+01 // указателя
Адрес a[1]=0xbf99afd4, a содежимое a[1]=1.100000e+01 // в шестнадцатеричном
Адрес a[2]=0xbf99afd8, a содежимое a[2]=1.200000e+01 // виде с префиксом 0x
Адрес a[2]=0xbf99afd8, a содежимое a[2]=1.200000e+01 //
```

3.2 Структуры

В программировании наряду с массивами (наборами элементов одинакового типа) широко востребован именованный тип данных, который характеризуется набором элементов разных типов.

Например, строка каталога звёзд содержит имя звезды, её координаты, звёздную величину, спектральный класс и т.д. Обработка каждой характеристики требует операций, соответствующих её типу данных, хотя для человека удобна возможность обозначить весь набор, относящийся к одной звезде, одним именем. В случае массива элементы однотипны и состоят из одинакового числа байтов (адрес элемента просто вычислить по его индексу). В случае разнородной информации размер памяти, отводимой под элементы разных типов, может оказаться разным (расчёт адреса по номеру элемента будет не так прост как в случае массива).

Упорядочение элементов разных типов по номеру неудобно и для человека, т.к. обезличивает их смысловую нагрузку. Вместо числового индекса гораздо важнее видеть имя, указывающее назначение данного (**a.ra** *right ascention* — прямое восхождение звезды; **a.dec** *declination* — её склонение) и т.д. Подобный подход иногда предпочтительнее даже, когда вполне могли бы обойтись и массивом. Например, при написании программы, работающей с координатами точки, удобнее иметь дело с обозначениями **a.x**, **a.y** и **a.z** вместо **a(1)**, **a(2)** и **a(3)**.

Языки программирования для организации именованной совокупности возможно разнотипных элементов предоставляют синтаксическую конструкцию, которая называется в ФОРТРАНе — **производным типом** (служебное слово **type**, в СИ/С++ — **структурой** (служебное слово **struct**). Эта конструкция определяет новое имя, которое "*понимается*" программой как имя **типа** данных, а также имена и типы, входящих в него элементов. Последние часто называют **полями** структуры.

3.2.1 ФОРТРАН

В программе **tstcoord0** описан тип **coord**, с полями **x**, **y**, **z** и инициализированы **coord**-переменные:

```
program tstcoor0; implicit none
type coord          ! В качестве имени нового типа ---  слово coord.
  real x, y, z      !                  x, y, z - его поля типа real.
end type coord      ! Завершение описания типа coord.
type cube           ! cube - имя ещё одного типа.  v - имя его поля,
  type (coord) v(8) ! являющегося массивом из 8 элементов, каждый из
end type cube       ! которых, в свою очередь есть структура с именем coord.
type (coord) :: a, b, c, w    ! Описание четырёх переменных типа coord.
type (cube) ::  t             ! Описание одной переменной типа cube.
integer j
a=coord(1.0, 2.0, 3.0); b=coord(4.0, 5.0, 6.0)  ! Присваивание значений
c=coord(7.0, 8.0, 9.0)                          ! переменным типа coord
write(*,*) ' a: ',a; write(*,*) ' b: ',b; write(*,*) ' c: ',c
w=a; a=c; c=w
write(*,*) ' a: ',a; write(*,*) ' b: ',b; write(*,*) ' c: ',c         !   и
t=cube ((/ coord(4,4,4), coord(4,4,5), coord(4,5,5), coord(4,5,4),&   ! типа
           coord(5,4,4), coord(5,4,5), coord(5,5,5), coord(5,5,4)/))  ! cube
write(*,'((a,i5,3e15.7))') ' t: ',1,t%v(1), ('    ',j,t%v(j),j=2,4),&
&                                           ('    ',j,t%v(j),j=5,8)
end
```

В **tstcoord0** помимо типа **coord** описан и тип **cube** с одним полем типа одномерного массива из восьми элементов типа **coord**. При инициализации значения полей типа **coord**, разделяются запятыми. Их список заключается в круглые скобки и предваряется именем типа. При инициализации поля **v** типа **cube** необходимо использовать конструктор массива, заключая весь список значений типа **coord** в символы (/ и /).

```
  a:    1.0000000       2.0000000       3.0000000    ! Результат работы
  b:    4.0000000       5.0000000       6.0000000    ! tstcoord0.
  c:    7.0000000       8.0000000       9.0000000
  a:    7.0000000       8.0000000       9.0000000
  b:    4.0000000       5.0000000       6.0000000
  c:    1.0000000       2.0000000       3.0000000
 t:     1  0.4000000E+01  0.4000000E+01  0.4000000E+01
        2  0.4000000E+01  0.4000000E+01  0.5000000E+01
        3  0.4000000E+01  0.5000000E+01  0.5000000E+01
        4  0.4000000E+01  0.5000000E+01  0.4000000E+01
        5  0.5000000E+01  0.4000000E+01  0.4000000E+01
        6  0.5000000E+01  0.4000000E+01  0.5000000E+01
        7  0.5000000E+01  0.5000000E+01  0.5000000E+01
        8  0.5000000E+01  0.5000000E+01  0.4000000E+01
```

1. Компилятор **gfortran** в качестве разделителя имён структурного объекта и его поля использует значок процента (%), а не точки, хотя некоторые ФОРТРАН-компиляторы имеют опции, допускающие в качестве такого разделителя и точку.

2. Правильный выбор производного типа может существенно упростить тексты исходных файлов, из которых генерируется исполнимый код, если описание новых типов помещать в **модуль**, подключаемый к нужным единицам компиляции через оператор **use**.

3. В тот же модуль удобно включать и описания процедур, формальные аргументы которых являются объектами указанных производных типов. Так вывод объекта производного типа удобно оформить процедурой, с тем, чтобы главная программа имела дело только с именем этого объекта, а работой с именами его полей ведала бы процедура:

```
module modcoor1; implicit none
type coord;   real x, y, z;       end type coord
type cube ;   type (coord) v(8);  end type cube
contains
  subroutine prtcoor(a)
  type (coord) a
  write(*,'(3e15.7)') a%x, a%y, a%z
  end subroutine prtcoor
  subroutine prtcube(t)
  integer j
  type (cube) t
  do j=1,8;    call prtcoor(t%v(j));   enddo
  end subroutine prtcube
end module modcoor1


program tstcoor1; use modcoor1
implicit none
type (coord) :: a, b, c, w;   type (cube) ::  t
integer j
a=coord(1.0, 2.0, 3.0); b=coord(4.0, 5.0, 6.0); c=coord(7.0, 8.0, 9.0)
call prtcoor(a);  call prtcoor(b); call prtcoor(c); write(*,*)
w=a; a=c; c=w
call prtcoor(a);  call prtcoor(b); call prtcoor(c); write(*,*)
t=cube ((/ coord(4,4,4), coord(4,4,5), coord(4,5,5), coord(4,5,4),&
           coord(5,4,4), coord(5,4,5), coord(5,5,5), coord(5,5,4)/))
call prtcube(t)
end
```

3.2.2 СИ

Для первого знакомства с описанием и инициализацией переменных типа **struct** в СИ используем предыдущий пример: задание положения точки в пространстве тремя её координатами. Тогда оно описывалось одномерным вектором из трёх элементов типа **float**. Сейчас для описания используем тип данных **struct** с тремя полями **x**, **y** и **z** типа **float**. Рассмотрим соответствующий аналог программы **tvector.c**:

```
#include <stdio.h>
int main(void)    //                                    Файл tstruct.c
{
  struct coord    // В качестве нового имени типа определено слово coord.
    { float x;    // x, y, z - его поля типа float. Некоторые компиляторы
      float y;    // не допускали объявления однотипных полей одним
      float z;    // оператором float x, y, z; (gcc и g++ допускают).
    };
  struct coord a;                  // Простое описание переменной типа coord.
  struct coord b={1,2,3};          // Полная инициализация.
  struct coord c={.x=8, .z=88.0};  // Частичная инициализация (только в СИ,
                                   //                        но не в C++ !!!).
  printf(" a=%e %e %e\n", a.x, a.y, a.z);
  a.x=10.0; a.y=11.0; a.z=12.0;
  printf(" a=%e %e %e\n"  , a.x, a.y, a.z);
  printf(" b=%e %e %e\n\n", b.x, b.y, b.z);
  printf("Адрес a.x=%p, a содежимое a.x=%e \n"    , &a.x, a.x);
  printf("Адрес a.y=%p, a содежимое a.y=%e \n"    , &a.y, a.y);
  printf("Адрес a.z=%p, a содежимое a.z=%e \n\n"  , &a.z, a.z);
  printf("Адрес c.x=%p, a содежимое c.x=%e \n"  , &c.x, c.x);
  printf("Адрес c.y=%p, a содежимое c.y=%e \n"  , &c.y, c.y);
  printf("Адрес c.z=%p, a содежимое c.z=%e \n"  , &c.z, c.z);
  return 0;
}
```

Результат её работы:

```
 a=1.048576e+06 1.883795e-305 0.000000e+00
 a=1.000000e+01 1.100000e+01 1.200000e+01
 b=1.000000e+00 2.000000e+00 3.000000e+00

Адрес a.x=0xfee27730, a содежимое a.x=1.000000e+01
Адрес a.y=0xfee27734, a содежимое a.y=1.100000e+01
Адрес a.z=0xfee27738, a содежимое a.z=1.200000e+01

Адрес c.x=0xfee27710, a содежимое c.x=8.000000e+00
Адрес c.y=0xfee27714, a содежимое c.y=0.000000e+00
Адрес c.z=0xfee27718, a содежимое c.z=8.800000e+01
```

1. После фигурной скобки, завершающей описание полей структуры, необходим символ **;** (точка с запятой).

2. СИ допускает разные варианты именования структурных типов и описания переменных типа **struct**. Например,

```
struct coord            // Описание имени типа coord и его структуры.
 { float x,y,z;};       // (именно так тип coord описан в tstruct.c)

struct coord
  { float z,y,z;} a,b;  // Вместе с типом описаны и переменные a и b
                        // типа coord, под которые выделяется память
```

3. Помещать описание структурного типа в тело главной программы невыгодно. Если переменные типа **coord** в дальнейшем окажутся фактическими аргументами каких-то функций, то он потребуется и при описании их прототипов. Выгодно обеспечить доступ к нему из файлов с исходными текстами этих функций, описав тип **coord** в отдельном файле (например, **tstruct1.h**)

```
struct coord //                                          Файл  tstruct1.h
{            // Имя типа coord должно быть известно  и главной программе,
  float x;   // и функциям. Помещать описания этих функций в этом файле
  float y;   // tstruct1.h можно, но невыгодно, так как, компилируя главную
  float z;   // программу, компилятор будет их перекомлировать заново.
};
```

   Посредством директивы #**include** “**tstruct1.h**” содержимое файла **tstruct1.h** подключается к нужной функции.

4. Аналогичную директиву включим и в файл **tstrfun1.c** с функциями, которые реализуют ввод и вывод данных типа **coord**:

```
#include <stdio.h>                                      // Файл tstrfun1.c
#include "tstruct1.h"
void rdrcoord0(struct coord *a)
{ scanf("%e %e %e", &((*a).x), &((*a).y), &(a->z));}
void rdrcoord1(struct coord *a)  // При желании можно распределить работу
{ float u, v, w;                 // на два этапа:
  scanf("%e%e%e", &u, &v, &w);   // 1) ввод значений простых переменных;
  (*a).x=u;  a->y=v;  a->z=w;  } // 2) их присваивание полям  структуры.
void prtcoord(struct coord a)
{ printf("%e %e %e\n",a.x,a.y,a.z);
         a.x=555; a.y=666; a.z=777; printf("%e %e %e\n",a.x,a.y,a.z);
}
```

5. **Внимание!** Тип формального аргумента и функции **rdrcoord0** и функции **rdrcoord1** есть **указатель** на структуру. В этом случае возможны две эквивалентных формы доступа к её полям:

   - через разыменование указателя перед **.имя_поля**:

     (*a).x, (*a).y, (*a).z

   - посредством конструкции **имя_указателя->имя_поля:**

     a->x, a->y, a->z

6. Функции **scanf** (по её интерфейсу) при указании списка ввода требуются адреса вводимых переменных, т.е. **&((*a).x)** или **&(a->x)**).

7. В отличие от **rdrcoord0** и **rdrcoord1** формальный аргумент процедуры **prtcoord** является значением типа **coord**, а не адресом. Процедура **prtcoord** нацелена лишь на вывод значения, переданного фактическим аргументом формальному, а не на измененеие значения фактического. В предпоследней строке **prtcoord** моделируется изменение формального аргумента, но фактический не изменяется:

```
#include <stdio.h>
#include "tstruct1.h"            // Указываем имя используемой структуры.
void rdrcoord0(struct coord *);  // Указываем прототипы (интерфейс)
void rdrcoord1(struct coord *);  // используемых функций.
void prtcoord (struct coord );
int main(void)
{ struct coord a, x, p;          // Описываем переменные типа coord
  struct coord b={1,2,3};        // Инициализация полная
  struct coord c={.x=8, .z=88.0};//              и частичная.
  a.x=10.0; a.y=11.0; a.z=12.0;
  prtcoord(a); printf("%f    %f    %f\n", a.x, a.y, a.z);
  prtcoord(b); printf("%f    %f    %f\n", b.x, b.y, b.z);
  prtcoord(c); printf("%f    %f    %f\n", c.x, c.y, c.z);
  printf("введите x\n"); rdrcoord0(&x);
  prtcoord(x);           printf("%f    %f    %f\n", x.x, x.y, x.z);
  printf("введите p\n"); rdrcoord1(&p);
  prtcoord(p);           printf("%f    %f    %f\n", p.x, p.y, p.z);
return 0;
}
```

8. Результаты пропуска этой программы:

```
1.000000e+01 1.100000e+01 1.200000e+01 //prtcoord: 1-ый вывод a.x, a.y, a.z
5.550000e+02 6.660000e+02 7.770000e+02 //        : 2-ой вывод
10.000000    11.000000    12.000000    // main   :            a.x, a.y, a.z
1.000000e+00 2.000000e+00 3.000000e+00 //prtcoord: 1-ый вывод b.x, b.y, b.z
5.550000e+02 6.660000e+02 7.770000e+02 //        : 2-ой вывод
1.000000    2.000000    3.000000       // main   :            b.x, b.y, b.z
8.000000e+00 0.000000e+00 8.800000e+01 //prtcoord: 1-ый вывод c.x, c.y, c.z
5.550000e+02 6.660000e+02 7.770000e+02 //        : 2-ой вывод
8.000000    0.000000    88.000000      // main   :            c.x, c.y, c.z
введите x
4  44
                     444
4.000000e+00 4.400000e+01 4.440000e+02 //prtcoord: 1-ый вывод x.x, x.y, x.z
5.550000e+02 6.660000e+02 7.770000e+02 //        : 2-ой вывод
4.000000    44.000000    444.000000    // main   :            x.x, x.y, x.z
       введите p
9
   99
         999
9.000000e+00 9.900000e+01 9.990000e+02 //prtcoord: 1-ый вывод p.x, p.y, p.z
5.550000e+02 6.660000e+02 7.770000e+02 //        : 2-ой вывод
9.000000    99.000000    999.000000    // main   :            p.x, p.y, p.z
```

Ещё раз: передача формального аргумента по значению не может изменить значения соответствующего фактического аргумента.

9. Правда, есть объективное основание для описания формального аргумента процедуры **prtcoord** *указателем* на значение типа **coord** (при замене списка вывода **printf** либо на **a->x, a->y, a->z**, либо на **(*a).x, (*a).y, (*a).z**. Обоснование заключается в том, что передача одного адреса начала структуры занимает гораздо меньше времени чем копирование значений всех полей структуры. Короче,

“ Думаем сами, решаем сами: иметь или не иметь”.

10. В СИ и C++ есть оператор **typedef**, сопоставляющий имени, придуманному нами, любой уже существующий тип, что может существенно упростить модификацию и унифицировать описание программы в целом без описания придуманного имени в директиве препроцессора.

3.2.3 Об операторе typedef.

Оператор **typedef** определяет псевдоним имени существующего типа:

1. **typedef float real** позволит вместо **float a, h;** использовать **real a, h;**, и переход на тип **double** сведётся лишь к **typedef double real**

```
#include <iostream>                    //  Файл tsttrap.c
using namespace std;                   //
typedef long double real;              //  real            Результат:
int main()                             //  float              inf
{ real a=1.0e250, h=1.0e303, s=a*h;    //  double             inf
  cout <<s<<endl; return 0;    }       //  long double        1e+553
```

2. При описании переменных типа массив бывает неудобно после каждого имени явно указывать структуру массива посредством конструктора **[]**. Можно поступить так:

```
typedef double xyz[3];       // Определили имя типа xyz
int main()                   // и описали два массива a и b типа xyz, хотя
{ xyz a={1.2,2.3,3.7}, b ; } // могли бы объявить и так: double a[3], b[3].
```

3. Аналогично вместо директивы препроцессора, подключающей заголовочный файл с описанием типа структуры, можно определить синоним нужной структуры посредством оператора **typedef**:

```
typedef struct  { float x, y, z;}  coord;
```

   Теперь слово **coord** — эквивалент типа структуры с тремя полями типа **float**, и описание переменных типа **coord** выглядит проще:

```
coord a, b, c;     //  (вместо    struct coord a, b, c;)
```

4. Через **typedef** определяется псевдоним **указатель на функцию**, имя которой служит аргументом другой функции. Так, прототип

```
double trap (double*(double), double, double, double);
```

   после описания псевдонима **pf**: **typedef double (*pf)(double);** — запишется проще:

```
double trap (pfun, double, double, int);
```

3.3 Указатели в СИ

В предыдущих пунктах уже касались понятия **указатель** (это производный тип, значением которого служит адрес ячейки, предназначенной для хранения данного нужного типа). В СИ по умолчанию типы указателей на объекты разных типов различаются. Например,

```
#include <stdio.h>
int main(void)
{ int u, *a, k; double *b;
  a=b; return 0;
}
```

1. Звёздочка перед именем переменной **при описании типа** указывает, что **a** и **b** — **указатели**, т.е. будут хранить **адреса** ячеек, отведённых под значения типов **int** и **double** соответственно, а **u** и **k** — просто для значений типа **int**.

2. Для хранения адреса значения любого типа ячейки всегда достаточно 8 байтов. Однако, прямая попытка присвоить указателю, ссылающемуся на значение **int**, значение указателя, ссылающегося на **double**, приведёт к сообщению компилятора:

   ```
   tpointm1.c: In function ‘main’:
   tpointm1.c:4: warning: assignment from incompatible pointer type
   ```

   т.е. тип указателя из левой части оператора присваивания отличен от типа указателя из правой части. При желании возможна операцию приведения типа: **a=(int*)b;**, но нужно ли? — решать нам.

3. Для выделения области оперативной памяти нужного размера есть функция **malloc**, которая (при успешном вызове) возвращает через своё имя адрес начального байта области. Освобождение памяти осуществляет функция **free**. Прототипы обеих функций обычно из **stdlib.h**: **void *malloc(size_t size);** **void free(void *ptr);**.

4. **malloc** выделяет **size** байтов под переменную типа **size_t** и возвращает через имя **malloc** указатель (типа **void**) на выделенную память. Поэтому при необходимости приписать указателю нужный тип используем любую из операций приведения типа (например, **(int*)**, **(double*)**). При невозможности выделить память **malloc** возвращает указатель **NULL**.

5. Область оперативной памяти, находимая **malloc** или освобождаемая **free** при их вызове, не имеет имени. Для доступа к содержимому этой области используется операция разыменования указателя.

6. Переменная, описанная в программе в качестве указателя, имеет имя, память под неё выделяется при компиляции и находится полностью в ведении программы. Область же памяти, адресуемая через указатель, определяется и высвобождается в процессе работы программы (и в этом смысле является **динамической**).

```
#include <stdio.h>
#include <stdlib.h>  // Не забывать при использовании malloc!!!
int main(void)
{ int *pa; double *pb; char *pc; pa=NULL; pb=NULL; pc=NULL;
  printf("&pa=%p  &pb=%p  &pc=%p \n",&pa, &pb, &pc);// What now in
  printf(" pa=%p   pb=%p   pc=%p \n", pa,  pb,  pc);// pa, pb, pc?
  pa=(int*) malloc(sizeof(int));
  pb=(double*) malloc(sizeof(double));
  pc=(char*)   malloc(sizeof(char));
  printf("&pa=%p  &pb=%p  &pc=%p\n",&pa, &pb, &pc); // What now in
  printf(" pa=%p   pb=%p   pc=%p\n", pa,  pb,  pc); // pa, pb, pc?
  *pa=5; *pb=1.2345678901234567891; *pc='q';
  printf(" &pa=%p  &pb=%p  &pc=%p\n",&pa, &pb, &pc);   // What now in
  printf("*pa=%d  *pb=%30.18le  pc=%c\n",*pa,*pb,*pc);//*pa,*pb,*pc?
  free((void*)pa);
  printf("&pa=%p &pb=%p  &pc=%p\n",&pa, &pb, &pc);
  printf(" pa=%p  pb=%p   pc=%p\n", pa,  pb,  pc);    //What now in
  printf("*pa=%d *pb=%30.18le  *pc=%c\n",*pa, *pb, *pc);//      pa?
  return 0;
}
```

7. Небрежное использование функии **malloc** может привести к образованию в оперативной памяти областей недоступных программе из-за ошибочной потери содержимого соответствующих указателей. В итоге, и сами не используем динамическую память, захваченную своей же программой, и другим не даём.

8. Напомним (см., пункты **3.3.5** и **3.3.6**; первый семестр), что в СИ для изменения содержимого **по адресу** фактического аргумента соответствующий формальный по типу должен быть только **указателем**. В C++ для той же цели используем описание формального аргумента **ссылкой** (*скрытым указателем*), переходя к обычному СИ-указателю лишь по необходимости.

3.3.1 СИ

```
#include <stdio.h>
#define pintdef int*   // pintdef  - имя последовательности int*

typedef  int* pinttyp; // pinttype - синоним типа указатель на целое.
int main(void)
{ int*    pa, a;   // pa - указатель;  a - переменная типа int.
  pintdef pb, b;   // pb - указатель;  b - переменная типа int.
  pinttyp pc, c;                        // pc и c - указатели.
  printf(" &pa=%p  pa=%p\n"  ,&pa, pa); // У переменных pa и pb адреса есть,
  printf(" &pb=%p &pb=%p\n\n",&pb, pb); // но содержимое pa и pb не задаётся.
  printf(" &a=%p  a=%d\n",   &a, a);    // У переменных a и b адреса есть,
  printf(" &b=%p  b=%d\n\n", &b, b);    // но содержимое a и b не задаётся.
  a=5; b=7;
  printf(" &a=%p  a=%d\n"  , &a, a);    // Переменным a и b присвоены значения.
  printf(" &b=%p  b=%d\n\n", &b, b);
  pa=&a;     // переменной pa присвоено значение - это адрес переменной a;
  pb=&b;     // переменной pb присвоено значение - это адрес переменной b;
  printf(" &pa=%p  pa=%p  *pa=%d\n",  &pa, pa, *pa); // *pa - операция
  printf(" &pb=%p  pb=%p  *pb=%d\n\n",&pb, pb, *pb); //      разыменования
  pc=pa; c=pb;
  printf(" &pc=%p  pc=%p  *pc=%d\n", &pc, pc, *pc);
  printf("  &c=%p   c=%p   *c=%d\n\n",&c, c,  *c);
  c=NULL; printf(" c%p  &c%p\n",c, &c);
  return 0;
}
```

```
 &pa=0xbfdb30dc  pa=0x64fff4
 &pb=0xbfdb30d4 &pb=0x8048310

 &a=0xbfdb30d8  a=134514059
 &b=0xbfdb30d0  b=134514048

 &a=0xbfdb30d8  a=5
 &b=0xbfdb30d0  b=7

 &pa=0xbfdb30dc  pa=0xbfdb30d8  *pa=5
 &pb=0xbfdb30d4  pb=0xbfdb30d0  *pb=7

 &pc=0xbfdb30cc  pc=0xbfdb30d8  *pc=5
  &c=0xbfdb30c8   c=0xbfdb30d0   *c=7

 c(nil)  &c0xbfdb30c8
```

NULL — адресная константа (нулевой указатель). Совпадение указателя с **NULL** означает, что указателю уже не на что указывать (например, список окончился).

/*

3.3.2 Ещё раз о способе передачи аргумента Си-функции

По умолчанию передача фактического аргумента СИ-функции всегда происходит **по значению** (см., например, пункты **3.3.5** и **3.3.6** первого семестра, уже упоминаемые чуть выше). При передаче **по значению** происходит копирование значения, подставляемого в качестве фактического аргумента в ячейку формального. Далее вся работа СИ-функции идёт с формальным аргументом, а не с фактическим. Поэтому изменить содержимое фактического аргумента СИ-функция не может никогда (в частности, не может изменить и адрес переменной, передаваемый фактическим аргументом). Однако, она может изменить содержимое переменной, расположенной по этому адресу.

Типы значений фактического и формального аргументов должны соответствовать друг другу. Именно, если в качестве фактического аргумента выступает адрес переменной типа **int**, то в качестве типа формального, который может обеспечить изменение её содержимого должен быть тип **int***.

Пример №1.

Рассмотрим работу функции **sub1(int,int*)** типа **void**, которая значение первого аргумента типа **int** присваивает переменной, адрес которой указан во втором аргументе типа **int***. В исходный текст функции **sub1(int,int*)**, который приводится ниже специально включены операторы вывода значений и адресов формальных аргументов для того, чтобы легче было уяснить суть происходящего при передаче аргументов по значению. В качестве выполняемой части тела функции **sub1** выступает один единственный оператор ***Y=X**. Здесь ***Y** — операция разыменования указателя **Y**, т.е. обеспечение доступа к той области памяти, адрес которой содержится в указателе **Y** (т.е., на которую **Y** указывает).

```
#include <stdio.h>
void sub1(int,int*);
int main()
{ int A=55, B=1; printf("  main: A=%2d     &A=%p\n",A,&A);
                 printf("  main: B=%2d     &B=%p\n",B,&B);
sub1(A,&B);      printf("  main: B=%2d     &B=%p\n",B,&B); return 0;
}
```

```
void sub1(int X, int* Y)          //         Вывод:
{ printf("sub1_1:   X=%d\n",X);  // 1) значения формального аргумента X
                                  //    (т.е. целого значения, которое
                                  //    передано X фактическим аргументом
                                  //    при вызове sub1 из  main. В нашей
                                  //    main таким фактическим аргументом
                                  //    служит значение 55 переменной A).

  printf("sub1_1:   Y=%p\n",Y);  // 2) значения формального аргумента Y,
                                  // (т.е. адреса памяти, который передан
                                  // формальному аргументу Y фактическим
                                  // аргументом при вызове sub1 из main.
                                  // В нашей main таким фактическим
                                  // аргументом служит адрес переменной B.

  printf("sub1_1:  &X=%p\n",&X); // 3) адреса формального аргумента X;
  printf("sub1_1:  &Y=%p\n",&Y); // 4) адреса формального аргумента Y;
  printf("sub1_1: *Y=%d\n",*Y);  // 5) содержимого той области памяти,
                                  //    на которую ссылается Y (т.е.
                                  //    значение переменной A).

  *Y=X; // Изменение содержимого области памяти, на которую ссылается Y.

  printf("sub1_1: *Y=%d\n",*Y);  // 6) Вывод этого нового содержимого.
}
```

Результат работы программы:

```
  main: A=55     &A=0x7fff2a67ce4c // Значение переменной A и её АДРЕС.
  main: B= 1     &B=0x7fff2a67ce48 // Значение переменной B и её АДРЕС.
sub1_1:   X=55                    // Значение формального аргумента X
                                  // (целое число, скопированное из A)
sub1_1:   Y=0x7fff2a67ce48        // Значение формального аргумента Y
                                  // --- адрес переменной B, ведь
                                  // именно он (&B) служит фактическим
                                  // аргументом и поэтому именно он
                                  // скопирован в Y.
sub1_1:  &X=0x7fffa5c3069c        // Адрес формального аргумента X.
sub1_1:  &Y=0x7fff2a67ce20        // Адрес формального аргумента Y.
                                  // (&X и &Y ОТЛИЧНЫ от &A и &B).
sub1_1: *Y=1                      // Значение, хранящееся в области
                                  // памяти, адрес которой указан в Y,
                                  // т.е. изначальное значение B.
sub1_1: *Y=55                     // Новое значение области памяти,
                                  // адрес которой хранится в Y,
                                  // полученное посредством оператора
                                  //             *Y=X.
  main: B=55     &B=0x7fff2a67ce48 // Действительно, значение B изменено!
```

Пример №2.

Рассмотрим пример, когда первый аргумент функции хранит значение типа **int***, а второй аргумент должен передать это значение (имеется ввиду не значение типа **int**, а именно значение **int***, т.е. изменить значение адреса, хранящегося в указателе, описанном в главной программе.

Возражение о том, что главная программа и без этого знает адрес первого аргумента (ведь именно она обеспечила его подачу в качестве первого аргумента функции) — НЕ ПРИНИМАЕТСЯ. Сейчас важно ответить на вопрос: "*Как нужно описать формальный аргумент функции, чтобы, через него можно было передать вызывающей программе адрес, т.е. значение типа УКАЗАТЕЛЬ?*"

Задача похожа на предыдущую (отличие лишь в типах передаваемых аргументов). Первый формальный аргумент хранит указатель на **адрес** значения целого типа; тип этого указателя — **int***.

Второй формальный аргумент не может быть типа **int***, т.к.

1) должен адресовать НЕ значение типа **int**, а значение типа **int***;

2) если же всё-таки проигнорировать 1) и сопоставить второму формальному аргументу тип **int***, то окажемся у разбитого корыта, поскольку значение адреса соответствующего фактического аргумента будет передано формальному *по значению*, так что изменение формального аргумента внутри функции НЕ изменит значения фактического. Что делать?

Поскольку переменная, содержимое которой должно быть изменено, имеет тип **int***, то формальный аргумент функции, обеспечивающий возможность такого изменения, должен иметь тип **int****, т.е. тип указателя на (указатель типа **int***).

Рассмотрим пример, моделирующий ситуацию.

```
#include <stdio.h>
#include <stdlib.h>
void sub3(int *,int **);
int main()
{ int *v;          // v - рабочий указатель на int
  int A=55, B=3;
  printf(" &v=%p\n",&v);    printf(" v=%p\n",v);
  printf(" &A=%p\n",&A);    printf(" A=%d\n",A);
  printf(" &B=%p\n",&B);    printf(" B=%d\n",B);
  v=&B;                     printf(" v=%p\n",v);
  sub3(&A,&v);
```

```
  printf("  v=&A=%p\n", v);
  printf(" *v= A=%d\n",*v);     return 0;
}

void sub3(int *X, int **Y)
{
  printf("sub3_1:   &X=%p\n", &X);
  printf("sub3_1:    X=%p\n",  X);
  printf("sub3_1:   *X=%d\n", *X);
  printf("sub3_1:   &Y=%p\n", &Y);
  printf("sub3_1:    Y=%p\n", Y);
  printf("sub3_1:   *Y=%p\n",*Y);  // (Разыменование_Y )=*Y=&B.
  printf("sub3_1:  **Y=%d\n",**Y); // (Разыменование_*Y)=**Y=B=3
  *Y=X;                           // Теперь *Y стал указателем на &A.
  printf("sub3_2:   *Y=%p\n",*Y);  // (Разыменование_Y )= *Y=&A
  printf("sub3_2:  **Y=%d\n",**Y); // (Разыменование_*Y)=**Y=55
}
```

Результат работы программы:

```
 &v=0x7ffff1ffeca8            // Адрес указателя  v.
 v=(nil)                      // Значение........ v.
 &A=0x7ffff1ffeca4            // Адрес переменной A.
 A=55                         // Значение........ A.
 &B=0x7ffff1ffeca0            // Адрес переменной B.
 B=3                          // Значение........ B.
 v=0x7ffff1ffeca0             // Значение v после v=&B

sub3_1:   &X=0x7ffff1ffec88   // Адрес формального аргумента X
sub3_1:    X=0x7ffff1ffeca4   // Значение..........  X = &A
sub3_1:   *X=55               // (Разыменование_X)= *X = 55

sub3_1:   &Y=0x7ffff1ffec80   // Адрес формального аргумента Y
sub3_1:    Y=0x7ffff1ffeca8   // Значение..........   Y = &v
sub3_1:   *Y=0x7ffff1ffeca0   // (Разыменование_Y )= *Y = &B
sub3_1:  **Y=3                // (Разыменование_*Y)=**Y =  3
                              //   После  *Y=X
sub3_2:   *Y=0x7ffff1ffeca4   // (Разыменование_Y )= *Y = &A
sub3_2:  **Y=55               // (Разыменование_*Y)=**Y = 55

  v=&A=0x7ffff272a794         // Значение v после отработки sub3
 *v= A=55                     // (Разыменование_v )= *v = 55
```

Укажем полезную статью [21] «Нежданчики» языка ФОРТРАН:

https://habr.com/en/company/intel/blog/254235/

где излагаются моменты, с которыми сталкивается СИ-программист при переходе на ФОРТРАН, и ФОРТРАН-программист при переходе на СИ.

Пример №3.

Указатели в примерах №1,2 были связаны со значениями элементарного типа **int**. Приведём теперь пример, когда функция должна присвоить фактическому аргументу значение типа

struct vec { double x; double y; double z;} ,

где структура используется в качестве модели, задающей координаты точки окончания вектора, исходящего из начала координат.

Напишем функцию **addvec(a,b,c)**, которая находит сумму двух таких векторов **A** и **B**, и помещает её в структуру **C**

```
struct coord   //                                  Файл  tstruct1.h
{              // Имя типа coord должно быть известно и главной
  double x;    // программе, и функциям. Поэтому помещаем его описание
  double y;    // отдельном файле, содержимое которого к любой функции
  double z; }; // можно подсоединить посредством #include <tstruct.h>.


                                //                     Файл tstvec2.c
#include <stdio.h>
#include "tstruct2.h"           // Доступ к имени типа структуры

void rdrcoord(struct coord*);   // Указываем  прототипы  (интерфейсы)
void prtcoord(struct coord );   //                          используемых
void addvec(struct coord, struct coord, struct coord*); //   функций.
int main(void)
{
 struct coord A, B, C;          // Описываем переменные типа coord и
 struct coord *PA,*PB,*PC;      // указатели на переменные типа coord.
 rdrcoord(&A);   printf("A="); prtcoord(A); // Демонстрация работы с
 rdrcoord(&B);   printf("B="); prtcoord(B); // переменными типа
 addvec(A,B,&C); printf("C="); prtcoord(C); //          struct coord.
 printf("\n");
 PA=&A; printf("A="); prtcoord(*PA);  // Теперь указатели PA, PB и PC
 PB=&B; printf("B="); prtcoord(*PB);  // адресуют переменные A, B и C
 PC=&C; printf("C="); prtcoord(*PC);  // соответственно, и PA, PB и PC
 printf("\n");                       // используем для работы с A, B и C.
 rdrcoord(PA);       printf("A="); prtcoord(*PA); // При вызове addvec
 rdrcoord(PB);       printf("B="); prtcoord(*PB); // используем
 addvec(*PA,*PB,PC); printf("C="); prtcoord(*PC); // разыменование.
 printf("\n");                                   // указателей PA и PB.
 PA=&B;              printf("B="); prtcoord(*PA); // При желании можем
```

```
 PB=&C;              printf("C="); prtcoord(*PC); // изменить адрес, храня-
 addvec(*PA,*PC,PB); printf("B="); prtcoord(*PB); // щийся в указателе.
  return 0; }
```

В программировании часто употребляются термины АДРЕС, ССЫЛКА и УКАЗАТЕЛЬ.

И ссылка, и указатель содержат адрес того объекта, на который ссылаются. Различие между ними в том, что ссылка не может изменить тот адрес, на который ссылается, а указатель может.

Например, адреса переменных A, B и C, описанных в главной программе, это ссылки, устанавленные компилятором — их изменить нельзя, в смысле нельзя изменить адреса этих переменных. Однако можно изменить содержимое тех переменных, чьи адреса хранятся по адресам расположения ссылок A, B и C.

Аналогично, НЕЛЬЗЯ изменить адреса ячеек памяти, установленные компилятором для хранения указателей PA, PB и PC. Однако содержимое любого из них изменить можно (ситуация аналогична предыдущей).

Так что после PA=&B, в PA будет храниться адрес структуры B.

```
//                                                          Файл addvec.c
#include <stdio.h>
#include "tstruct2.h"
void rdrcoord(struct coord* a)
{
  scanf("%lg  %lg  %lg\n",&(*a).x,&(*a).y,&(*a).z);
}
void prtcoord(struct coord a)
{
  printf("%lg  %lg  %lg\n",a.x,a.y,a.z);
}
void addvec(struct coord a, struct coord b, struct coord* c)
{
 (*c).x=a.x+b.x; (*c).y=a.y+b.y; (*c).z=a.z+b.z;
}
```

Результат работы программы:

```
A=1  2  3
B=1  1  1
C=2  3  4

A=1  2  3
```

```
B=1  1  1
C=2  3  4

A=1  2  3
B=1  1  1
C=2  3  4

B=1  1  1
C=2  3  4
B=3  4  5
```

3.3.3 Пример построения стека

```
                                // Содержимое файла    myinter.h
typedef struct candidat* link;  // link --- синоним указателя на
                                //         звено списка candidat
struct candidat                 //   описание структуры candidat
  {
    int info;
    link next;
  };
void prtlist(link);             // прототип функции вывода списка

                                              // Содержимое файла  testlist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"

int main(void)
{
  link top,  cur;
  int i, k, ier;
  top=NULL;                                    // инициализация головы списка.
  i=1;                                         // --"-- номера вводимого числа
  do
    {
      printf(" input %d-st number > 0\n",i);  // подсказка о вводе i-го числа
      scanf("%d",&k);                          // ввод i-го числа
      printf("%d-st number is %d\n", i,k);    // контрольный вывод i-го числа
      if (k!=0)
        { cur=(link) malloc(sizeof (struct candidat)); // адрес нового звена
                                                       //             списка
          cur->info= k;    // в поле info помещаем k.
          cur->next=top;   // в поле next --"-- адрес того звена, на которое
                           //             пока ссылается top (голова списка).
          top=cur;         // теперь в top помещаем адрес звена, под которое
                           // выше была выделена память и заполнены её поля.
        }
      i++;                 // увеличение номера вводимого числа.
    }
  while (k!=0);
  prtlist(top);
  return 0;
}
```

```
                      // Содержимое файла          prtlist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"
void prtlist(link top)  // Функция prtlist выводит содержимое списка,
{                       // адрес головы которого указан в top.
  link cur;             // Указатель на текущее звено списка
  int i;                // Номер звена
  i=0;                  // Инициализация номера звена
  cur=top;              // Инициализация текущего звена
  while (cur!=NULL)     // Пока указатель текущего звена информирует
    {                   // о существовании следующего:
      i++;                                          // номер текущего звена
     printf("%d-e ",i);                             // вывод этого номера
     printf(" число в списке равно %d\n",cur->info); // вывод поля info
     cur=cur->next;                        // теперь в текущем указателе
                                           // адрес следующего звена.
    }
}
```

Содержимое файла input

```
23
45
67
98
100
0
```

Содержимое файла result

```
 input 1-st number > 0
1-st number is 23
 input 2-st number > 0
2-st number is 45
 input 3-st number > 0
3-st number is 67
 input 4-st number > 0
4-st number is 98
 input 5-st number > 0
5-st number is 100
 input 6-st number > 0
6-st number is 0
1-e  число в списке равно 100
2-e  число в списке равно 98
3-e  число в списке равно 67
4-e  число в списке равно 45
5-e  число в списке равно 23
```

3.3.4 Пример добавления нового звена в вершину стека

Если речь идёт о добавлении нового звена главной программой, то добавление аналогично соответствующим операторам из тела цикла:

```
                                              // Содержимое файла  testlist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"

int main(void)
{
  link top,  cur;
  int i, k, ier;
  top=NULL;                                    // инициализация головы списка.
  i=1;                                         // --"-- номера вводимого числа
  do
    {
      printf(" input %d-st number > 0\n",i);  // подсказка о вводе i-го числа
      scanf("%d",&k);                          // ввод i-го числа
      printf("%d-st number is %d\n", i,k);    // контрольный вывод i-го числа
      if (k!=0)
        { cur=(link) malloc(sizeof (struct candidat)); // адрес нового звена
                                                       //             списка
          cur->info= k;    // в поле info помещаем k.
          cur->next=top;   // в поле next --"-- адрес того звена, на которое
                           //             пока ссылается top (голова списка).
          top=cur;         // теперь в top помещаем адрес звена, под которое
                           // выше была выделена память и заполнены её поля.
        }
      i++;                 // увеличение номера вводимого числа.
    }
  while (k!=0);
prtlist(top);
printf(" input %d-st number > 0\n",i);  // подсказка о вводе i-го числа
scanf("%d",&k);                         // ввод i-го числа
printf("%d-st number is %d\n", i,k);    // контрольный вывод i-го числа
if (k!=0)
  { cur=(link) malloc(sizeof (struct candidat)); // адрес нового звена списка
    cur->info= k;
    cur->next=top;
    top=cur;
  }
printf("Добавили ещё одно звено в голову списка\n");
prtlist(top);
return 0;
}
```

Содержимое файла input

```
23 45 67 98 100 0 777
```

Содержимое файла result

```
 input 1-st number > 0
1-st number is 23
 input 2-st number > 0
2-st number is 45
 input 3-st number > 0
3-st number is 67
 input 4-st number > 0
4-st number is 98
 input 5-st number > 0
5-st number is 100
 input 6-st number > 0
6-st number is 0
1-e  число в списке равно 100
2-e  число в списке равно 98
3-e  число в списке равно 67
4-e  число в списке равно 45
5-e  число в списке равно 23
 input 7-st number > 0
7-st number is 777
Добавили ещё одно звено в голову списка
1-e  число в списке равно 777
2-e  число в списке равно 100
3-e  число в списке равно 98
4-e  число в списке равно 67
5-e  число в списке равно 45
6-e  число в списке равно 23
```

Некоторое затруднение может возникнуть, когда добавление нового элемента (звена стека) должна выполнить функция типа **void**, т.е. возвращающая результат через свои аргументы. Пусть, например, её вызов имеет вид **addelem0(top,k);**. Здесь **top** — указатель на вершину стека, а **k** — целая переменная, хранящая информацию, которая должна оказаться в поле **info** головного звена. Описание функции **addelem0** поместим в тот же файл, что и функцию **prtlist** (назовём его **worklist**, отражая тот факт, что в будущем в нём будут собраны все функции, обеспечивающие работу со стеком).

```
                                              // Содержимое файла  testlist3.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"
int main(void)
{ link top,  cur; int i, k, ier;
  top=NULL;                                    // инициализация головы списка.
  i=1;                                         // --"-- номера вводимого числа
  do { printf(" input %d-st number > 0\n",i);  // подсказка о вводе i-го числа
       scanf("%d",&k);                         // ввод i-го числа
       printf("%d-st number is %d\n", i,k);    // контрольный вывод i-го числа
       if (k!=0)
         { cur=(link) malloc(sizeof (struct candidat)); // адрес нового звена
                                                        //             списка
           cur->info= k;    // в поле info помещаем k.
           cur->next=top;   // в поле next --"-- адрес того звена, на которое
                            //             пока ссылается top (голова списка).
           top=cur;         // теперь в top помещаем адрес звена, под которое
                            // выше была выделена память и заполнены её поля.
         }
       i++;                 // увеличение номера вводимого числа.
    }
  while (k!=0);
  printf("Первый список:\n"); prtlist(top);
  addelem0(top,777); printf("Второй       :\n"); prtlist(top);
  return 0;
}

                        // Содержимое файла          prtlist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"
void prtlist(link top)  // Функция prtlist выводит содержимое списка,
{                       // адрес головы которого указан в top.
  link cur;             // Указатель на текущее звено списка
  int i;                // Номер звена
  i=0;                  // Инициализация номера звена
  cur=top;              // Инициализация текущего звена
  while (cur!=NULL)     // Пока указатель текущего звена информирует
    {                   // о существовании следующего:
      i++;                                          // номер текущего звена
     printf("%d-e ",i);                             // вывод этого номера
     printf(" число в списке равно %d\n",cur->info); // вывод поля info
     cur=cur->next;                         // теперь в текущем указателе
                                            // адрес следующего звена.
    }
}
```

```
void addelem0(link top, int num) // Функция ADDELEM0 добавляет (???)
{                                // новое звено к вершине стека.
  link cur;
  cur=(link) malloc(sizeof(struct candidat));
  cur->info=num;
  cur->next=top;
  top=cur;
}
```

Содержимое файла input

```
23 45 67 98 100 0
```

Содержимое файла result

```
 input 1-st number > 0
1-st number is 23
 input 2-st number > 0
2-st number is 45
 input 3-st number > 0
3-st number is 67
 input 4-st number > 0
4-st number is 98
 input 5-st number > 0
5-st number is 100
 input 6-st number > 0
6-st number is 0
Первый список:
1-е  число в списке равно 100
2-е  число в списке равно 98
3-е  число в списке равно 67
4-е  число в списке равно 45
5-е  число в списке равно 23  // Видим, что несмотря на то, что в тексте
Второй       :                // главной программы явно указано добавление
1-е  число в списке равно 100 // нового звена посредством вызова addlist0,
2-е  число в списке равно 98  // функция prtlist выводит тот же самый
3-е  число в списке равно 67  // список, который был и до вызова
4-е  число в списке равно 45  // addlist0 (т.е. addlist0 работает не так.
5-е  число в списке равно 23  // как требуется.     П О Ч Е М У   ???
```

Для уяснения происходящего поместим в тело **addlist0** вспомогательные печати. Назовём копию функции **addlist0** с этими печатями **addlist1**.

```
                                        // Содержимое файла  testlist4.c
#include <stdio.h>
#include <stdlib.h>
```

```
#include "myinter.h"

int main(void)
{
  link top,  cur; int i, k, ier;
  top=NULL;                                    // инициализация головы списка.
  i=1;                                         // --"-- номера вводимого числа
  do
    { printf(" input %d-st number > 0\n",i);  // подсказка о вводе i-го числа
      scanf("%d",&k);                          // ввод i-го числа
      printf("%d-st number is %d\n", i,k);    // контрольный вывод i-го числа
      if (k!=0)
        { cur=(link) malloc(sizeof (struct candidat)); // адрес нового звена
                                                       //             списка
          cur->info= k;    // в поле info помещаем k.
          cur->next=top;   // в поле next --"-- адрес того звена, на которое
                           //             пока ссылается top (голова списка).
          top=cur;         // теперь в top помещаем адрес звена, под которое
                           // выше была выделена память и заполнены её поля.
        }
      i++;                 // увеличение номера вводимого числа.
    }
  while (k!=0);
                     printf("Первый список:\n"); prtlist(top);
  addelem1(top,555); printf("Второй       :\n"); prtlist(top);
  return 0;
}

                       // Содержимое файла          prtlist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"
void prtlist(link top)  // Функция prtlist выводит содержимое списка,
{                       // адрес головы которого указан в top.
  link cur;             // Указатель на текущее звено списка
  int i;                // Номер звена
  i=0;                  // Инициализация номера звена
  cur=top;              // Инициализация текущего звена
  while (cur!=NULL)     // Пока указатель текущего звена информирует
    {                   // о существовании следующего:
      i++;                                          // номер текущего звена
     printf("%d-e ",i);                             // вывод этого номера
     printf(" число в списке равно %d\n",cur->info); // вывод поля info
     cur=cur->next;                        // теперь в текущем указателе
                                           // адрес следующего звена.
    }
}
```

```
void addelem1(link top, int num)
{
  link cur;
  cur=(link) malloc(sizeof(struct candidat)); printf("%p\n",cur);
  cur->info=num;                              printf("%d\n",cur->info);
  cur->next=top;                              printf("%p\n",cur->next);
  top=cur;                                    printf("%p\n",top);
}
```

Содержимое файла input

```
23 45 67 98 100 0
```

Содержимое файла result

```
 input 1-st number > 0
1-st number is 23
 input 2-st number > 0
2-st number is 45
 input 3-st number > 0
3-st number is 67
 input 4-st number > 0
4-st number is 98
 input 5-st number > 0
5-st number is 100
 input 6-st number > 0
6-st number is 0
Первый список:
1-е  число в списке равно 100
2-е  число в списке равно 98
3-е  число в списке равно 67
4-е  число в списке равно 45
5-е  число в списке равно 23
0x18310b0                       // Это адрес нового текущего звена.
555                             // Это содержимое его info-поля.
0x1831090                       // Это адрес старого головного звена.
0x18310b0                       // Это адрес нового головного звена.
Второй       :                  // И, тем не менее, для главной
1-е  число в списке равно 100   // программы функция addlist1
2-е  число в списке равно 98    // отработала вхолостую ---
3-е  число в списке равно 67    //   ничего не изменилось !!!
4-е  число в списке равно 45    //        П О Ч Е М У     ???
5-е  число в списке равно 23
```

Дело в том, что мы забыли, что в СИ аргументы функций всегда передаются **по значению**. Это означает, что значение любого фактического аргумента сначала копируется в ячейку формального, и далее

вся работа внутри функции ведётся исключительно с формальным аргументом. В частности, передача внутрь функции адреса вершины стека тоже происходит по значению, т.е. новое значение адреса вершины, оказавшееся по адресу формального аргумента, никогда не будет передано вызывающей программе. Что же делать? Сделать надо то же самое, что мы уже делали в первом семестре, когда требовалось изменить значение фактического аргумента. Вспомним программу **testsub1.c** и функцию **subfun1.c** из пункта **6.3.5** первого семестра:

```
#include <stdio.h>                                      // файл testsub1.c
void subfun(double,double*);
int main()
{ double x, y, z, a, b;
  double resx, resy, resz, resa, resb;
  x=0.51; y=1; z=0.75; a=1.5; b=1.499999999999999;
  subfun(x,&resx);  printf("         x=%23.16e  resx=%23.16e\n",x,resx);
  subfun(y,&resy);  printf("         y=%23.16e  resy=%23.16e\n",y,resy);
  subfun(z,&resz);  printf("         z=%23.16e  resz=%23.16e\n",z,resz);
  subfun(a,&resa);  printf("         a=%23.16e  resa=%23.16e\n",a,resa);
  subfun(b,&resb);  printf("         b=%23.16e  resb=%23.16e\n",b,resb);
  return 0;
}

void subfun(double x, double* y)                        // файл subfun1.c
{ (*y)=2*x-3;
  printf(" subfun: x=%23.16le     y=%23.16le\n",x,*y);
}
```

Вызов функции **subfun1(x,&res)** должен изменить значение **res** переменной главной программы. Другими словами, в качестве соответствующего фактического аргумента следует использовать адрес переменной **res**, а в качестве соответствующего формального — **указатель** на область памяти, где расположен фактический аргумент.

Аналогично следует поступить и для изменения адреса вершины стека: подать в качестве фактического аргумента не адрес вершины стека (он будет передан по значению), а адрес ячейки, в которой адрес вершины стека записан. Адрес этой ячейки тоже будет передан по значению и его изменить функция не сможет, но сможет изменить содержимое ячейки, в которой записан адрес вершины стека. Неважно, значение какого типа хранится в области памяти, на которую указывает указатель — важно, что для изменения её содержимого в качестве фактического аргумента следует подать её адрес:

Имя переменной	Тип фактического аргумента	Тип формального аргумента	Вызов
double res;	&res	double*	subfun(x,&res);
link top;	& top	link*	addelem(&top,int);

```
                                                // Содержимое файла  testlist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"

int main(void)
{
  link top,  cur;
  int i, k, ier;
  top=NULL;                                      // инициализация головы списка.
  i=1;                                           // --"-- номера вводимого числа
  do
    {
      printf(" input %d-st number > 0\n",i);  // подсказка о вводе i-го числа
      scanf("%d",&k);                         // ввод i-го числа
      printf("%d-st number is %d\n", i,k);    // контрольный вывод i-го числа
      if (k!=0)
        { cur=(link) malloc(sizeof (struct candidat)); // адрес нового звена
                                                       //             списка
          cur->info= k;    // в поле info помещаем k.
          cur->next=top;   // в поле next --"-- адрес того звена, на которое
                           //             пока ссылается top (голова списка).
          top=cur;         // теперь в top помещаем адрес звена, под которое
                           // выше была выделена память и заполнены её поля.
        }
      i++;                 // увеличение номера вводимого числа.
    }
  while (k!=0);
                    printf("Первый список:\n"); prtlist(top);
  addelem(&top,555); printf("Второй       :\n"); prtlist(top);
  addelem(&top,444); printf("Третий       :\n"); prtlist(top);
  addelem(&top,333); printf("Четвёртый    :\n"); prtlist(top);
  return 0;
}
```

```
                      // Содержимое файла          worklist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"
void prtlist(link top)  // Функция prtlist выводит содержимое списка,
{                       // адрес головы которого указан в top.
  link cur;             // Указатель на текущее звено списка
  int i;                // Номер звена
  i=0;                  // Инициализация номера звена
  cur=top;              // Инициализация текущего звена
  while (cur!=NULL)     // Пока указатель текущего звена информирует
    {                   // о существовании следующего:
      i++;                                          // номер текущего звена
     printf("%d-e ",i);                             // вывод этого номера
     printf(" число в списке равно %d\n",cur->info); // вывод поля info
     cur=cur->next;                        // теперь в текущем указателе
                                           // адрес следующего звена.
    }
}
void addelem(link* top, int num) // Функция ADDELEM добавляет к вершине
{                                // стека новое звено, помещая в его
                                 // поле INFO число NUM.
// Важно осознать, что  добавление нового звена должно привести к
// изменению содержимого указателя на вершину стека.
// Если в качестве первого аргумента ADDELEM использовать просто
// указатель на вершину стека, то (в силу того, что передача аргументов
// в СИ происходит по значению) изменение содержимого этого указателя
// (как формального аргумента) не приведёт к изменению фактического.
// Для изменения фактического аргумента-указателя необходимо в качестве
// типа первого аргумента ADDELEM использовать тип
// <указатель_на_указатель>. Тогда через операцию разыменования первого
// указателя можем осуществить изменение второго, т.е. определить тем
// самым адрес новой вершины стека.
  link cur;
  cur=(link) malloc(sizeof(struct candidat));
  cur->info=num;
  cur->next=(*top);
  (*top)=cur;
}
```

Содержимое файла input

```
23 45 67 98 100 0
```

Содержимое файла result

```
 input 1-st number > 0
1-st number is 23
 input 2-st number > 0
2-st number is 45
 input 3-st number > 0
3-st number is 67
 input 4-st number > 0
4-st number is 98
 input 5-st number > 0
5-st number is 100
 input 6-st number > 0
6-st number is 0
Первый список:
1-е  число в списке равно 100
2-е  число в списке равно 98
3-е  число в списке равно 67
4-е  число в списке равно 45
5-е  число в списке равно 23
Второй       :
1-е  число в списке равно 555
2-е  число в списке равно 100
3-е  число в списке равно 98
4-е  число в списке равно 67
5-е  число в списке равно 45
6-е  число в списке равно 23
Третий       :
1-е  число в списке равно 444
2-е  число в списке равно 555
3-е  число в списке равно 100
4-е  число в списке равно 98
5-е  число в списке равно 67
6-е  число в списке равно 45
7-е  число в списке равно 23
Четвёртый    :
1-е  число в списке равно 333
2-е  число в списке равно 444
3-е  число в списке равно 555
4-е  число в списке равно 100
5-е  число в списке равно 98
6-е  число в списке равно 67
7-е  число в списке равно 45
8-е  число в списке равно 23
```

3.3.5 Пример удаление звена из вершины стека

Решать вопрос о изменении адреса вершины стека приходится и при удалении головного звена.

```
                                              // Содержимое файла  testlist6.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"

int main(void)
{
  link top,  cur;
  int i, k, ier;
  top=NULL;                                    // инициализация головы списка.
  i=1;                                         // --"-- номера вводимого числа
  do { printf(" input %d-st number > 0\n",i); // подсказка о вводе i-го числа
      scanf("%d",&k);                          // ввод i-го числа
      printf("%d-st number is %d\n", i,k);    // контрольный вывод i-го числа
      if (k!=0)
        { cur=(link) malloc(sizeof (struct candidat)); // адрес нового звена
                                                       //             списка
          cur->info= k;    // в поле info помещаем k.
          cur->next=top;   // в поле next --"-- адрес того звена, на которое
                           //             пока ссылается top (голова списка).
          top=cur;         // теперь в top помещаем адрес звена, под которое
                           // выше была выделена память и заполнены её поля.
        }
       i++;                // увеличение номера вводимого числа.
     }
  while (k!=0);
  printf("Первый список   :\n");
  printf("АДРЕС ВЕРШИНЫ СТЕКА: %p\n",top); prtlist(top);
  printf("ДОБАВЛЕНИЕ звеньев:\n");
  addelem(&top,555); printf("АДРЕС ВЕРШИНЫ СТЕКА: %p\n",top);
                     printf("Содержимое стека   :\n"); prtlist(top);
                     printf("Второй список      :\n");
  addelem(&top,444); printf("АДРЕС ВЕРШИНЫ СТЕКА: %p\n",top);
                     printf("Содержимое стека   :\n"); prtlist(top);
                     printf("Третий список      :\n");
  addelem(&top,333); printf("АДРЕС ВЕРШИНЫ СТЕКА: %p\n",top);
                     printf("Содержимое стека:\n"); prtlist(top);
  printf("УДАЛЕНИЕ звеньев:\n");
  do { delelem(&top); printf("АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: %p\n",top);
                      printf("Содержимое стека:\n"); prtlist(top); }
  while (top!=NULL);
  return 0;
}
```

```
                      // Содержимое файла          worklist.c
#include <stdio.h>
#include <stdlib.h>
#include "myinter.h"
void prtlist(link top)  // Функция prtlist выводит содержимое списка,
{                       // адрес головы которого указан в top.
  link cur;             // Указатель на текущее звено списка
  int i;                // Номер звена
  i=0;                  // Инициализация номера звена
  cur=top;              // Инициализация текущего звена
  while (cur!=NULL)     // Пока указатель текущего звена информирует
    {                   // о существовании следующего:
      i++;                                          // номер текущего звена
     printf("%d-e ",i);                             // вывод этого номера
     printf(" число в списке равно %d\n",cur->info); // вывод поля info
     cur=cur->next;                        // теперь в текущем указателе
                                           // адрес следующего звена.
    }
}
void addelem(link* top, int num) // Функция ADDELEM добавляет к вершине
{                                // стека новое звено, помещая в его
                                 // INFO-поле целое число NUM, и
  link cur;                      // устанавливая НОВЫЙ адрес вершины.
  cur=(link) malloc(sizeof(struct candidat));
  cur->info=num;
  cur->next=(*top);
  (*top)=cur;
}
void delelem(link* top)
{ link cur;
  if ((*top)==NULL) printf("Стек пуст.\n");
    else { cur=(*top);
           (*top)=cur->next; free(cur);}
}
```

Содержимое файла input

```
23
45
67
98
100
0
```

Содержимое файла result

```
 input 1-st number > 0
1-st number is 23
 input 2-st number > 0
2-st number is 45
 input 3-st number > 0
3-st number is 67
 input 4-st number > 0
4-st number is 98
 input 5-st number > 0
5-st number is 100
 input 6-st number > 0
6-st number is 0
Первый список   :
АДРЕС ВЕРШИНЫ СТЕКА: 0x8ae090
1-е  число в списке равно 100
2-е  число в списке равно 98
3-е  число в списке равно 67
4-е  число в списке равно 45
5-е  число в списке равно 23
ДОБАВЛЕНИЕ звеньев:
АДРЕС ВЕРШИНЫ СТЕКА: 0x8ae0b0
Содержимое стека   :
1-е  число в списке равно 555
2-е  число в списке равно 100
3-е  число в списке равно 98
4-е  число в списке равно 67
5-е  число в списке равно 45
6-е  число в списке равно 23
Второй список      :
АДРЕС ВЕРШИНЫ СТЕКА: 0x8ae0d0
Содержимое стека   :
1-е  число в списке равно 444
2-е  число в списке равно 555
3-е  число в списке равно 100
4-е  число в списке равно 98
5-е  число в списке равно 67
6-е  число в списке равно 45
7-е  число в списке равно 23
Третий список      :
АДРЕС ВЕРШИНЫ СТЕКА: 0x8ae0f0
Содержимое стека:
1-е  число в списке равно 333
2-е  число в списке равно 444
3-е  число в списке равно 555
4-е  число в списке равно 100
5-е  число в списке равно 98
```

```
6-е  число в списке равно 67
7-е  число в списке равно 45
8-е  число в списке равно 23
УДАЛЕНИЕ звеньев:
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: 0x8ae0d0
Содержимое стека:
1-е  число в списке равно 444
2-е  число в списке равно 555
3-е  число в списке равно 100
4-е  число в списке равно 98
5-е  число в списке равно 67
6-е  число в списке равно 45
7-е  число в списке равно 23
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: 0x8ae0b0
Содержимое стека:
1-е  число в списке равно 555
2-е  число в списке равно 100
3-е  число в списке равно 98
4-е  число в списке равно 67
5-е  число в списке равно 45
6-е  число в списке равно 23
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: 0x8ae090
Содержимое стека:
1-е  число в списке равно 100
2-е  число в списке равно 98
3-е  число в списке равно 67
4-е  число в списке равно 45
5-е  число в списке равно 23
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: 0x8ae070
Содержимое стека:
1-е  число в списке равно 98
2-е  число в списке равно 67
3-е  число в списке равно 45
4-е  число в списке равно 23
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: 0x8ae050
Содержимое стека:
1-е  число в списке равно 67
2-е  число в списке равно 45
3-е  число в списке равно 23
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: 0x8ae030
Содержимое стека:
1-е  число в списке равно 45
2-е  число в списке равно 23
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: 0x8ae010
Содержимое стека:
1-е  число в списке равно 23
АДРЕС НОВОЙ ВЕРШИНЫ СТЕКА: (nil)
Содержимое стека:
```

3.3.6 Задача о считалке

При её решении полезно сочетать и тип данных **struct**, и **указатель** на значение типа **struct**)

n детей собираются играть в прятки и выясняют кому водить, произнося считалку из **m** слов. Для простоты каждому участнику сопоставим **номер**. Требуется написать программу, которая моделирует процесс выяснения номера водящего. Есть несколько подходов к решению:

1. Найти формулу, которая по **m** и **n** находит номер водящего (нас она пока не интересует). Нам интересно как можно ближе к реальности смоделировать процесс выявления водящего.

2. Описать массив из **n** элементов булева типа. Инициализировать их значением **true** (любой из играющих может оказаться водящим). Затем, перебрав **m** слов считалки, поместить в **m**-ый элемент значение **false**, что будет признаком выбывания из **списка** кандидатов на водящего. При подобном моделировании всё равно придётся проверять: "*А не выбыл ли уже играющий из списка кандидатов?*". В реальности этого нет. Говорящий считалку просто ориентруется только на тех, кто ещё остался в круге из кандидатов на водящего.

3. Смоделировать процесс, используя указатели на структуру данных

```
struct candidat
  { int number;              // поле с номером кандидата на вождение;
    struct candidat* next;  // поле с указателем на соседнего кандидата.
  };
```

   одно звено которой состоит из двух полей:

   (a) первое типа **int** с номером участника;

   (b) второе типа **указателя** на соседа (ещё не выбывшего) из списка

   Внимание! При описании поля **next** типа **struct candidat** указано, что значение, помещаемое в **next**, есть указатель на тип **struct candidat**. Другим словами, описание структуры **struct candidat** — рекурсивное, т.е. при описании типа **struct candidat** используем ссылку на сам описываемый тип. Подобная рекурсивность допускается только при описании указателей на него.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct candidat* link; // Тип link - указатель на данное типа candidat
struct candidat                // Тип candidat - структура из двух полей:
       { int number;           //  number - для данного целого типа и
         link next;            //  next - для хранения указателя на тип
       };                      //                                  candidat.
int main(void)
{  int j, k, m, n;
  link first, tek, nov;
  n=5; m=3;                    // число кандидатов и число слов в считалке

//         СОЗДАНИЕ КОЛЬЦЕВОГО СПИСКА ИГРАЮЩИХ.

  first=(link) malloc(sizeof(struct candidat)); // выделение памяти под
                                                // значение типа candidat
                   // её начальный адрес хранится в указателе first,
  first->number=1;  // а в её поле number запомнен номер первого игрока.
  tek=first;        // Сейчас  tek  смотрит туда же куда и first.
  for (k=2;k<=n;k++)                            // Фрахтуем ячейку под
    {     nov=(link) malloc(sizeof(struct candidat)); // k-го участника.
          nov->number=k;                          // Запоминаем его номер.
          tek->next=nov;    // Запоминаем в поле next (предыдущего игрока)
                            // указатель на зафрахтованную ячейку k-го.
          tek=nov;          // теперь tek смотрит на k-го игрока.
    }
    tek->next=first;        // После последнего игрока следует первый.

                            //    КОЛЬЦЕВОЙ СПИСОК ИГРАЮЩИХ СОЗДАН.
// ПОИСК НОМЕРА ВОДЯЩЕГО:

   for (k=n;k>=2; k--)      // Пока в списке не останется один человек
      {                     // (водящий) повторяем:
       for (j=1;j<=m-1;j++) // проговор m-1 слова считалки, получая в tek
         {                  // указатель на игрока, который должен выбыть.
           tek=tek->next;   //
          }                 // Выбытие достигается присваиванием полю
                            // tek->next соседа предшествующего выбывающему
                            // указателя, хранящегося в поле next
        tek->next=tek->next->next; // выбывающего. При этом звено
      }                             // выбывающего не отдаётся операционной
    //  системе, хотя и становится недоступным данной программе (ведь его
    // адрес, хранящийся в next-поле предыдущего игрока, уже затёрт).

     printf(" Водит игрок под номером %d\n",  // Вывод номера
               tek->number);                  // водящего игрока
  return 0;
}
```

В этой програме через оператор **typedef struct candidat* link** для типа указателя на **struct candidat** определён синоним **link** и описание поля **next** имеет вид: **link next;**. Причина в желании упростить и унифицировать описание типа рабочих переменных **first**, **tek** и **nov** (типа **struct candidat***) и операцию приведения к типу **struct candidat*** указателя типа **void** (получаемого функцией **malloc**). Использование псевдонима **link** (вместо **struct candidat***), хоть и незначительно, но упростит исходный текст.

3.4 Понятие о указателях ФОРТРАНа

В простейшем случае указатель в ФОРТРАНе можно рассматривать как переменную, значением которой является ссылка на другой объект или указание об отсутствии такой ссылки.

1. Посредством указателей можно во время работы программы создавать в оперативной памяти **динамические объекты**, списки, деревья и т.д. Для объявления указателя в ФОРТРАНе используется атрибут **pointer**. Например,

```
integer, pointer :: p, q
real, dimension (:), pointer :: inp
pointer :: ptr(:,:)
```

2. Связь указателя с объектом можно установить:

   1) либо посредством оператора **allocate** (при создании динамического объекта);
   2) либо посредством оператора присваивания

3. Вид оператора присваивания для указателя отличается от обычного оператора присваивания и имеет вид

```
указатель => адресат
```

   Здесь **адресат** либо **указатель**; либо объект, на который можно ссылаться через указатель, либо ссылка на функцию, возвращающую в вызывающую её единицу компиляции **указатель**.

4. Значением операнда-указателя служит значение объекта, связанного с ним в момент обращения.

5. Оператор **nullify** уничтожает связь указателя с объектом.

6. Оператор **deallocate** уничтожает и размещённые объекты, и связь с ними.

В качестве примеров использования указателей в ФОРТРАНе приведём две простые программы. Первая строит в оперативной памяти связный список и выводит его. Вторая решает задачу о считалке. Полезно сопоставить её ФОРТРАН-решение с СИ-решением.

3.4.1 Построение и вывод простого связного списка

```
program testlist; use modlist; implicit none
type (link), pointer :: top, cur
integer i, k
top=>NULL()                                    ! инициализация указателя
i=1; do; write(*,*) ' введи ',i,'-е число > 0'
         read (*,*) k
         write(*,*) i,'-е число равно ', k
         i=i+1; if (k/=0) then; allocate (cur)
                              cur%info= k    !/ или же так:
                              cur%next=>top  !\   cur=link(k,top)
                                   top=>cur
                         else; exit
                endif
enddo
call prtlist(top)
end

module modlist; implicit none
type link                          ! описание производного типа link (звено)
    integer info                   ! имя информационного поля звена
    type (link), pointer :: next ! имя указательного   поля звена
end type link                      ! завершение описания типа link
contains
  subroutine prtlist(top)          ! описание подпрограммы вывода списка
  type(link), pointer :: top, cur ! top --- указатель на голову списка
  cur=>top                         !         (формальный аргумент prtlist).
  do while (associated(cur))       ! cur --- локальная переменная для
    write(*,*) cur%info            !         хранения указателя на
    cur => cur%next                !         очередное звено списка.
  enddo                            ! Пока cur"/="null() выводим cur%info
  end subroutine prtlist
end module modlist
```

NULL([mold]) — функция связности указателя. Её вызов без указания аргумента эквивалентен инициализации указателя, что означает установку статуса связности указателя на значение *не связан*. Функция **NULL()** отсутствует в ФОРТРАНе-90 (в нём используется оператор **nullify(pointer)** — обнуление указателя **pointer**, точнее разрывает связь указателя с объектом).

При наличии аргумента функция NULL возвращает значение имеющее тот же статус связности, что и аргумент (другими словами выражение **p=>NULL(q)** преобразует статус связности указателя **p** на статус связности указателя **q**). Пока будем вызывать **NULL()** без аргумента.

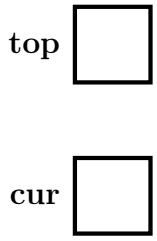
3.4.2 Схема функционирования программы testlist

testlist строит односвязный список вида **LIFO** (*Last In First Out*, т.е. последним вошёл — первым вышел). Модуль **modlist** содержит описание производного типа **link** и подпрограммы **prtlist** вывода списка. **testlist** в цикле вводит положительные значения целочисленной переменной **k** (*нуль* — признак завершения ввода). Для каждого введённого положительного программа выделяет область памяти под соответствующее звено списка, помещая адрес звена в указатель **cur**. Далее **info**-полю звена присваивается значение **k**, а **next**-полю — значение указателя на предыдущее звено. Результат работы **testlist** при вводе чисел **4, 3, 2, 1, 0**:

```
введи            1 -е число > 0
         1 -е число равно            4
введи            2 -е число > 0
         2 -е число равно            3
введи            3 -е число > 0
         3 -е число равно            2
введи            4 -е число > 0
         4 -е число равно            1
введи            5 -е число > 0
         5 -е число равно            0
         1
         2
         3
         4
```

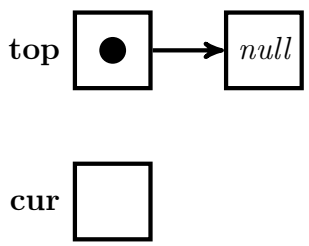
Как видно, содержимое информационных полей выводится **prtlist** в порядке обратном порядку ввода чисел (последним вошёл — первым вышел). Аргументом процедуры служит указатель на **головное звено** списка (т.е. последнее заполненное).

1. **F: type(link), pointer :: top, cur** | **C: link top, cur;**



Отведены переменные **top** и **cur** для размещения указателей на данные типа **link**. Значения **top** и **cur** пока не заданы.

2. **F: top=>NULL()** | **C: top=NULL;**

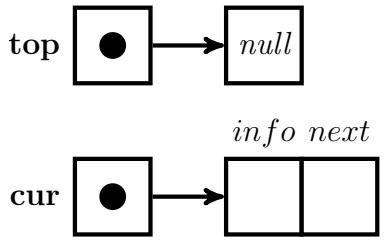


Теперь **top** *смотрит* на **null** — код завершения любого списка. Значение указателя **cur** по-прежнему не задано.

3. **F: i=1; read(*,*) k**

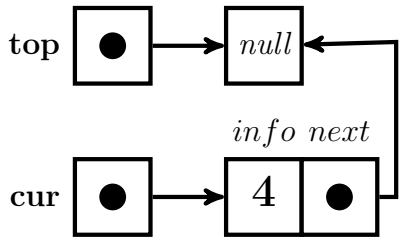
   C: scanf(”%d”,&k); Первое введённое значение переменной **k**, равно **4**. Далее предстоит занести его в **info**-поле данного типа **link**, под которое память пока не выделена.

4. **allocate(cur)**



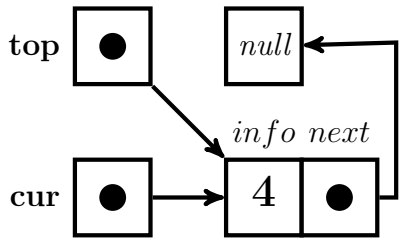
top по-прежнему *смотрит* на *null*. Но **cur** теперь *смотрит* на область памяти, предназначенную для размещения нового данного типа **link** (первого звена списка), т.е. структуры из двух полей $info$ и $next$. Правда, пока в эти поля ничего не занесено (но место под них выделено и адресовано оператором *allocate*).

5. **cur=link(k,top)** или, что то же, **cur%info=k; cur%next=>top**



Теперь звено, адресуемое **cur**, заполнено: в $info$-поле — значение переменной **k**, а в $next$-поле — значение указателя **top** (т.е. $next$-поле первого звена *смотрит* туда же, куда и **top** — в *null*, который фиксирует окончание строящегося списка). Поэтому в **top** теперь можно поместить адрес первого звена списка, который пока хранится в **cur**.

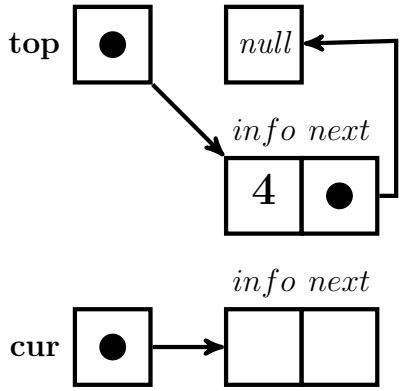
6. **top=>cur**.



После **top=>cur** указатель **top** уже адресует голову списка. Её же пока адресует и **cur**. Однако, теперь **cur** может быть использовать для фрахтовки места под новое звено, если таковое потребуется.

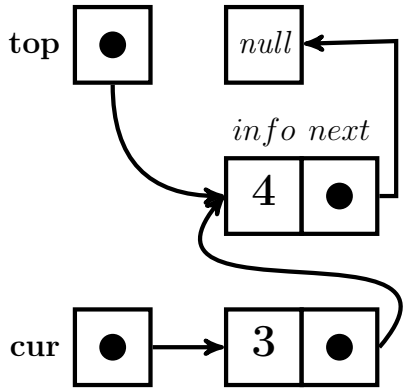
7. **i=2; read(*,*) k**. Второе введённое значение переменной **k** равно **3**.

8. **allocate(cur)**



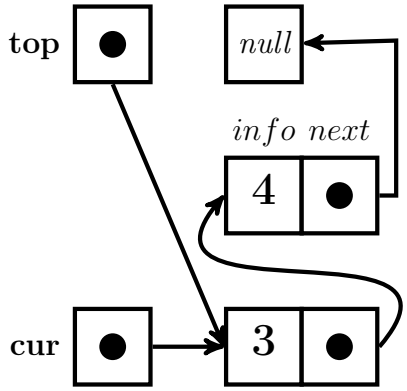
top по-прежнему хранит адрес первого звена, т.е головы предшествующего списка. Однако **cur** уже адресует новое звено и теперь его нужно заполнять.

9. **cur=link(k,top)** или, что то же, **cur%info=k; cur%next=>top**



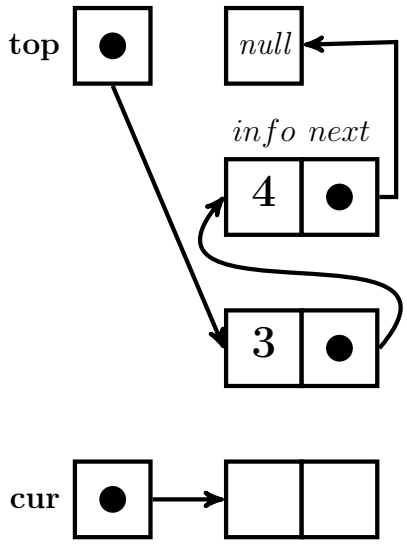
top пока хранит адрес головы старого списка. Но поля нового (второго) звена уже заполнены. Так что, если поместить в **top** адрес начала второго звена, то **top** (как и **cur**) станет адресовать список из уже двух звеньев.

10. **top=>cur)**



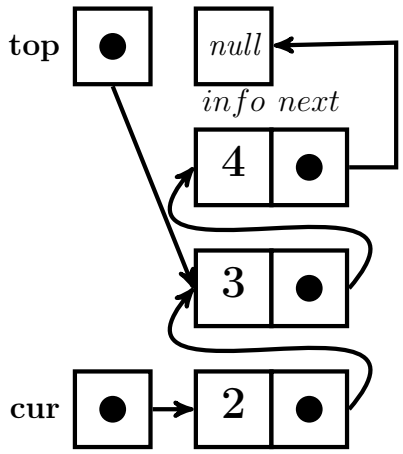
Теперь **top** уже адресует голову двузвенного списка, и, значит, можно (если нужно) использовать **cur** для адресации нового звена типа **link**.

11. **i=3; read(*,*) k; allocate(cur)**.



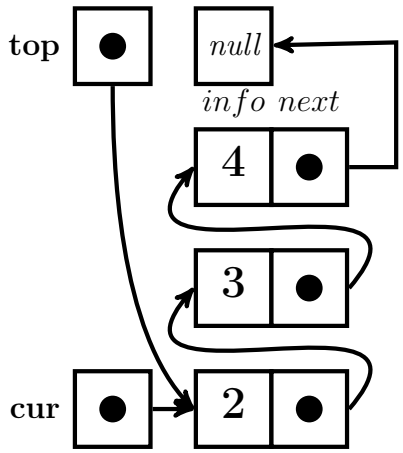
Третье введённое значение переменной **k** равно **2**. **allocate(cur)** находит место для нового (третьего) звена типа **link**.

12. **cur=link(k,top)** или, что то же, **cur%info=k; cur%next=>top**



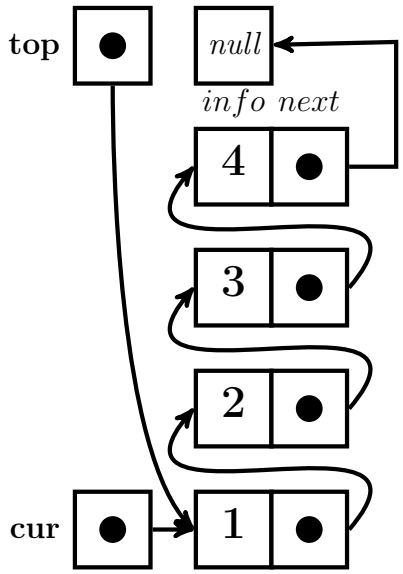
Трёхзвенный список, по существу дела, уже имеется. Правда, он адресуется указателем **cur**, который используется в качестве рабочего. Для адресации нового головного звена через указатель **top** необходимо присвоить **top** значение указателя **cur**.

13. **top=>cur**



Добавим ещё одно звено в голову списка, приведя окончательную схему к виду, соответствующему данным **4, 3, 2, 1**, указанным в нвчале этого параграфа.

14. **i=4; read(*,*) k; allocate(cur); cur=link(k,top); top=>cur**.

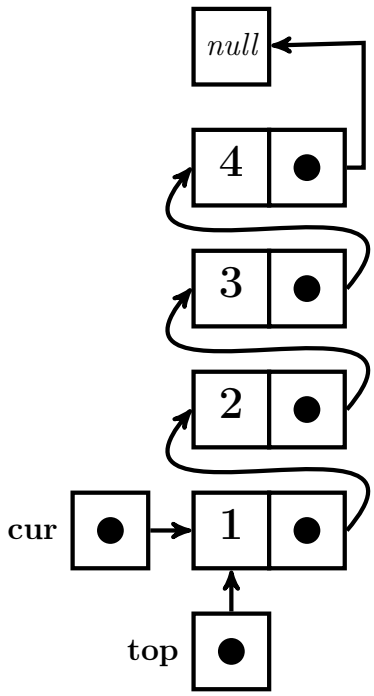


Теперь **top** адресует голову списка, состоящего из четырёх звеньев.

15. **i=5; read(*,*) k**. Пятое введённое значение переменной **k** равно **0**, который инициирует выход из цикла.

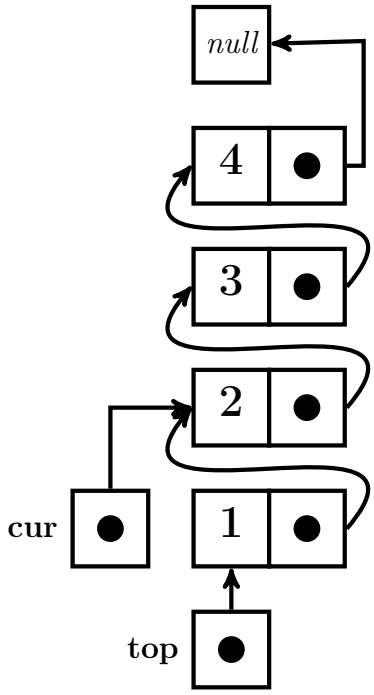
3.4.3 Схема функционирования подпрограммы prtlist

1. **cur=>top; do while (associated(cur)**



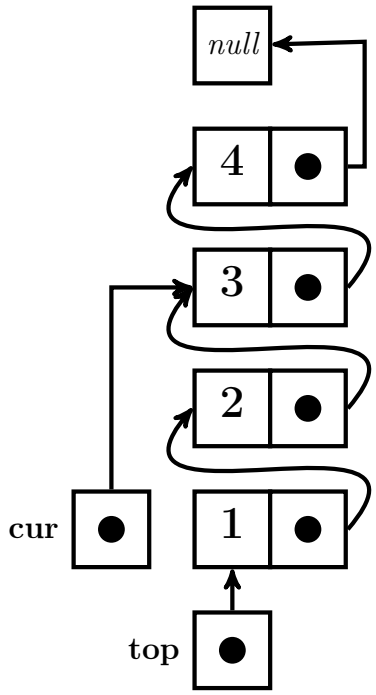
В качестве аргумента **prtlist** использует **top** — указатель на голову списка. Не будем использовать непосредственно **top** для адресации очередного звена списка — ведь фактичекие аргументы ФОРТРАН-процедур передаются по ссылке (*а не по значению, как в СИ*). Поэтому внутри **prtlist** описан вспомогательный локальный указатель **cur**, который и используется для адресации. Далее работает заголовок оператора цикла с предусловием, вызывающим встроенную булеву ФОРТРАН-функцию **associated**, которая проверяет статус связности указателя **cur** с каким-либо адресатом. Если **cur** связан с полноправным очередным звеном списка, то возвращается **.true.**; если же **cur** указывает на *null* — **.false.**. При **.true.** оператор **write(*,*) cur%info** выведет **1**, после чего оператор **cur=>cur%next** переключит **cur** на обзор предыдущего звена.

2. **cur=>cur%next**



После того как выведено содержимое информационного поля текущего звена списка, **cur** перенастраивается на адресацию следующего звена. Поскольку и для него **associated**(**cur**) ≡ .**true.**, то после **write(*,*) cur%info** выведет **2**.

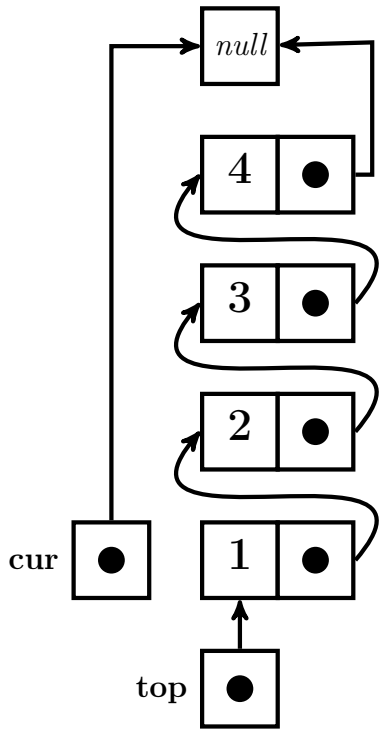
3. **cur=>cur%next**



Теперь будет выведено **3**.

4. После очредного **cur=>cur%next** оператор **write(*,*) cur%next** выведет **4**.

5. **cur=>cur%next**



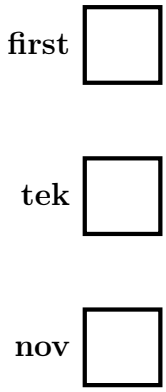
Наконец, когда **associated**(**cur**) = .**false**. цикл прекратится и подпрограмма **prtlist** завершит свою работу.

3.4.4 Задача о считалке

```
program testring; implicit none
type candidat
  integer number
  type (candidat), pointer :: next
end type candidat
integer j, k, m, n, nfirst;
type (candidat), pointer :: first, tek, nov
  n=5               ! число кандидатов
  m=3               ! число слов в считалке
  allocate(first)   ! Адресуем память под первое звено типа candidat.
  first%number=1    ! В поле number запомнен номер звена.
  first%next=>first ! в поле next запомнен указатель на первое звено,
                    ! т.е. кольцевой список из одного звена построен.
  tek=>first        ! Указатель на текущее звено (вначале первое).
  do k=2,n
    allocate(nov)   ! Адресуем память под новое (k-ое) звено.
    nov%number=k    ! В number-поле k-ого звена --- его номер.
    tek%next=>nov   ! В next-поле (k-1)-ого --- указатель на k-ое.
    tek=>nov        ! Теперь текущим становится k-ое звено.
  enddo
                    ! Сейчас в tek указатель на последнее (n-ое) звено.
  tek%next=>first   ! В next-поле n-го звена --- указатель на первое.
  nullify(nov,first)! Разрыв связи указателей nov и first с адресатами
                    ! с целью показать, что это не помешает дальнейшему
                    ! поиску.
  do k=n,2,-1       ! Пока не останется один человек (водящий)
                    ! повторяем:
    do j=1,m-1      ! проговор m-1 слова считалки, получая в tek
      tek=>tek%next ! указатель на игрока, который должен выбыть.
    enddo
                             ! Выбытие достигается присваиванием
    tek%next=>tek%next%next ! next-полю предшествующего звена
                             ! значения указателя из next-поля
                             ! удаляемого. При этом звено, удаляемое из
                             ! списка становится недоступным и данной
  enddo                      ! программе, и операционной системе (ведь
                             ! указатель на него, ранее хранящийся
                             ! в next-поле предыдущего звена, уже
                             ! затёрт).
   write(*,*) ' Водит игрок под номером ',&  ! Вывод номера
              tek%number                     ! водящего игрока
end
```

3.4.5 Схема построения кольцевого списка

1. **type (candidat), pointer :: first, tek, nov**

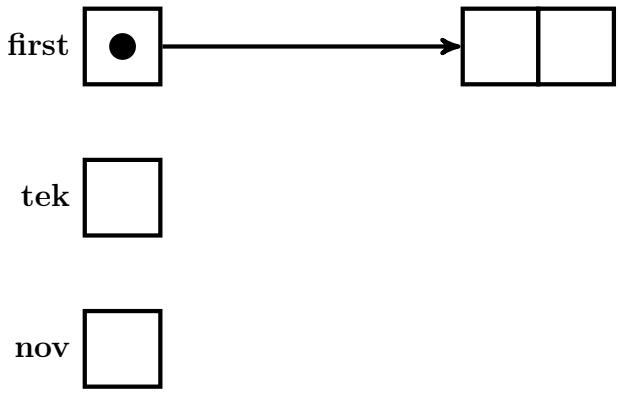


Описаны переменные **first**, **tek** и **nov** для размещения указателей на звенья структуры типа **type(candidat)**. Значения **first**, **tek** и **nov** пока не заданы.

2. **n=5; m=3**

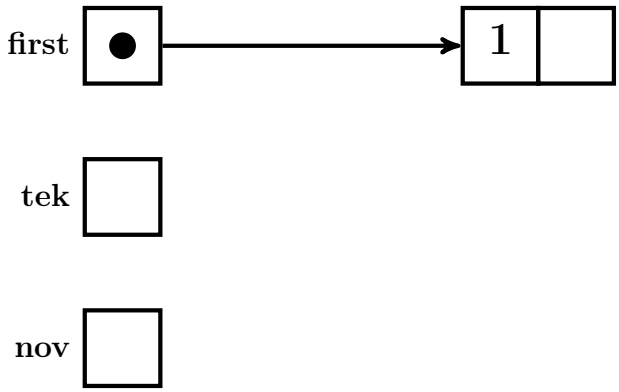
n=5 — число кандидатов. **m=3** — число слов.

3. **allocate(first)**



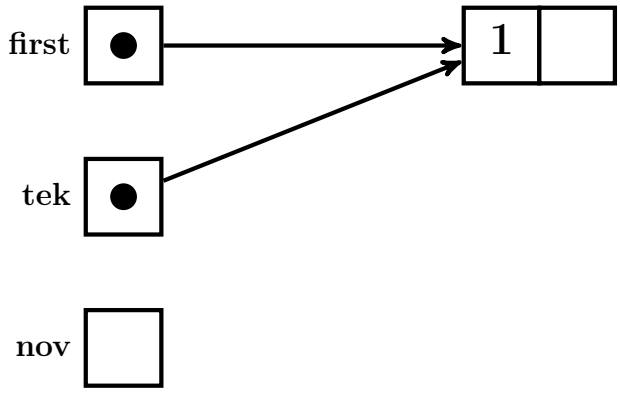
first адресует место для первого звена, которое в дальнейшем должно будет хранить информацию о **первом** кандидате, точнее, о том кандидате, с которого начнём проговаривать считалку (информация в поля звена пока не занесена)

4. **first%number=1**



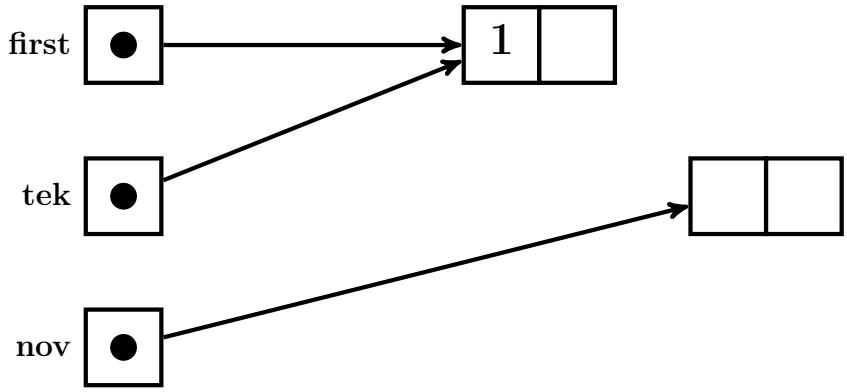
В *number*-поле занесён номер первого кандидата (можно было бы построить кольцевой список из одного звена посредством **first%next=>first**).

5. **tek=>first**



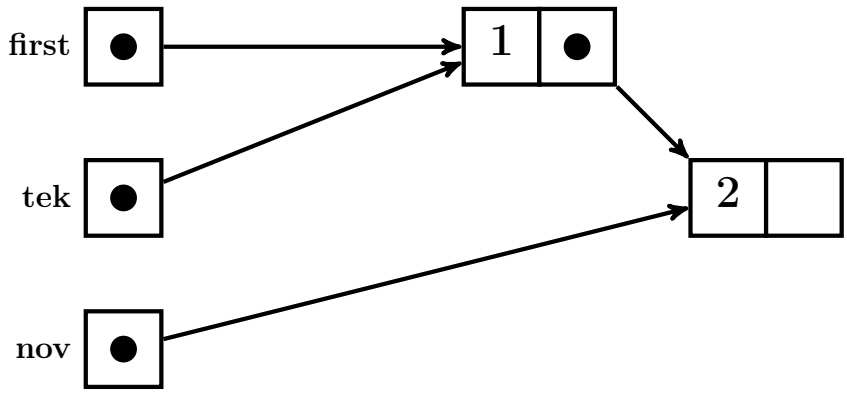
Для построения кольцевого списка используем указатели **tek** и **nov**. **first** потребуется лишь для ссылки последнего звена кольцевого списка на первое. Теперь **tek** обозревает то же звено, что и **first**.

6. **k=2; allocate(nov)**



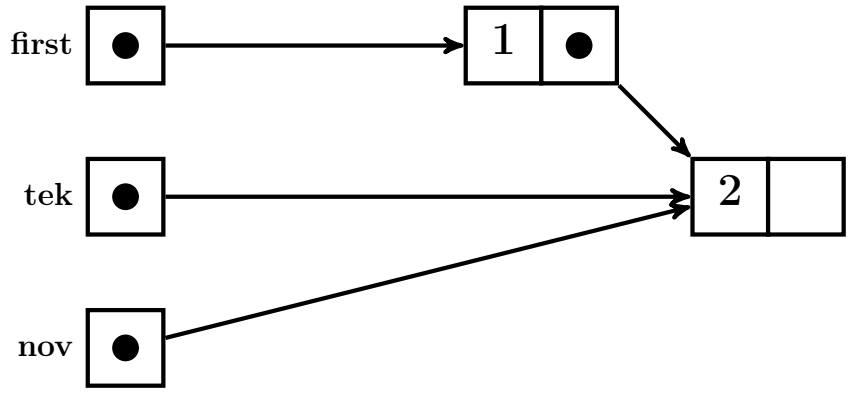
Место для нового звена (второго игрока) адресует **nov**, но содержимое полей этого звена пока не задано.

7. **nov%number=k; tek%next=>nov**



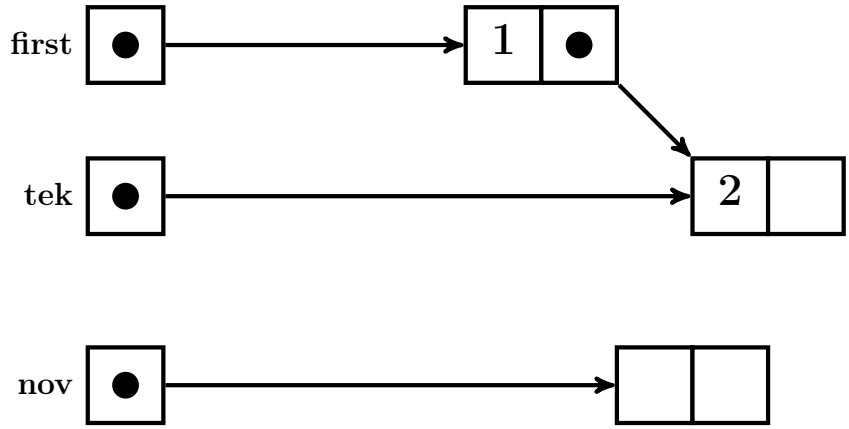
Теперь номер второго игрока записан в *number*-поле второго звена и в *next*-поле первого звена записан указатель на второе.

8. **tek=>nov**



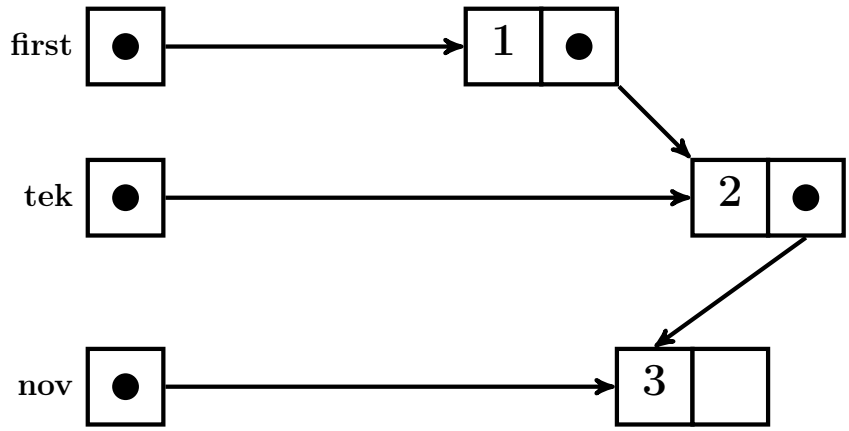
tek адресует уже второе звено, так что **nov** готово к поиску места для третьего.

9. **k=3; allocate(nov)**



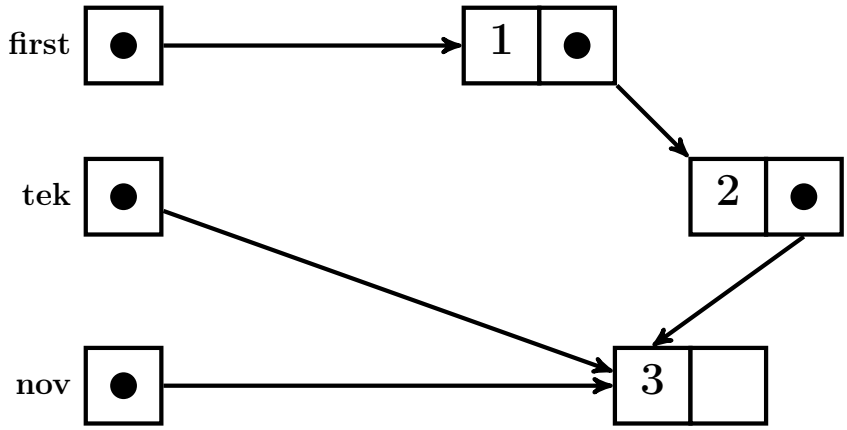
nov адресует третье звено (пока не заполненное).

10. **nov%number=k; tek%next=>nov**



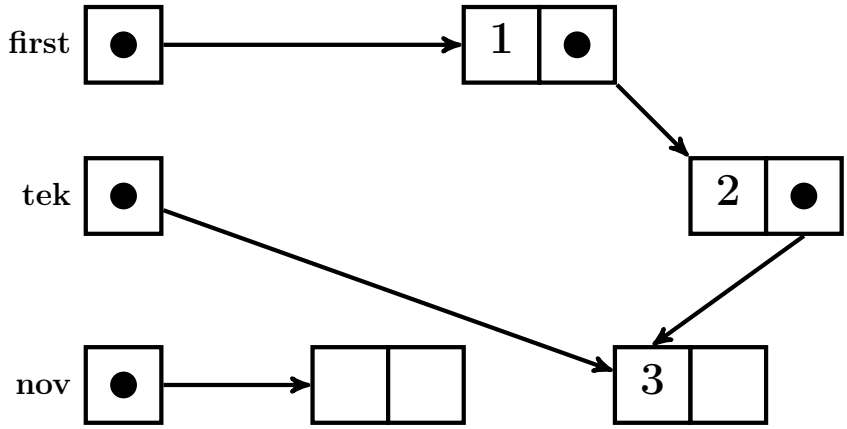
В *number*-поле звена, адресуемого **nov**, записан номер игрока, а в *next*-поле второго звена помещён указатель на третье звено.

11. **tek=>nov**



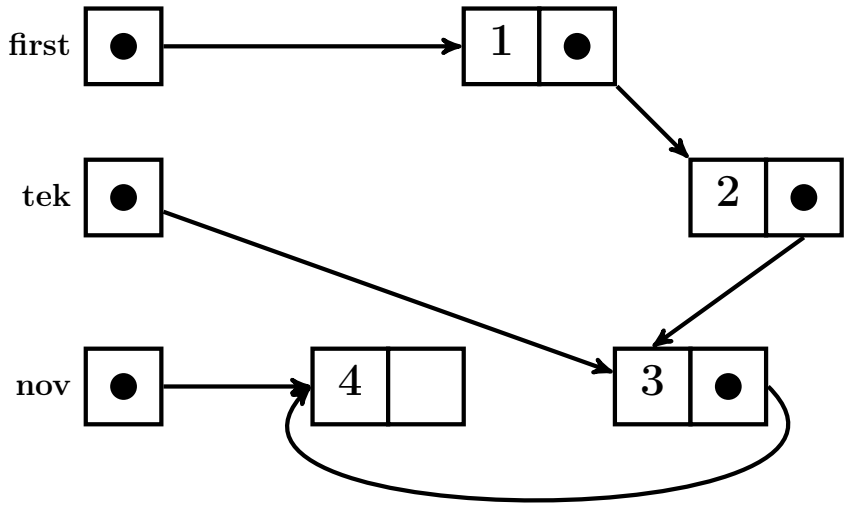
tek переключён на обзор третьего звена, чтобы **nov** можно было переключить на поиск четвёртого.

12. **k=4; allocate(nov)**



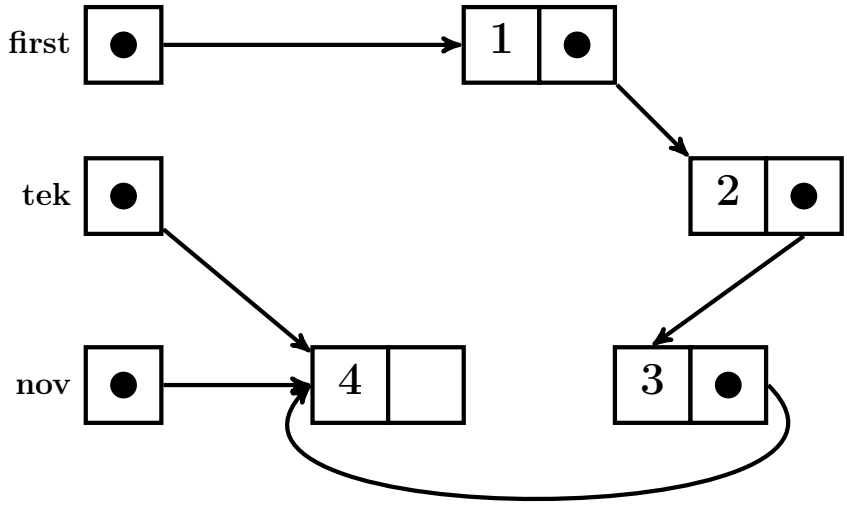
nov хранит указатель на четвёртое звено (пока пустое).

13. **nov%number=k; tek%next=>nov**



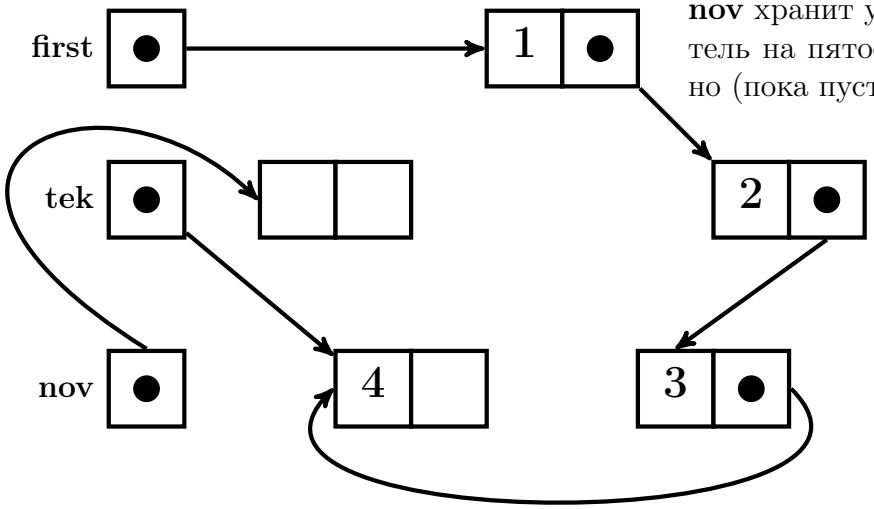
Теперь заполнены *number*-поле четвёртого звена и *next*-поле третьего.

14. **tek=nov**



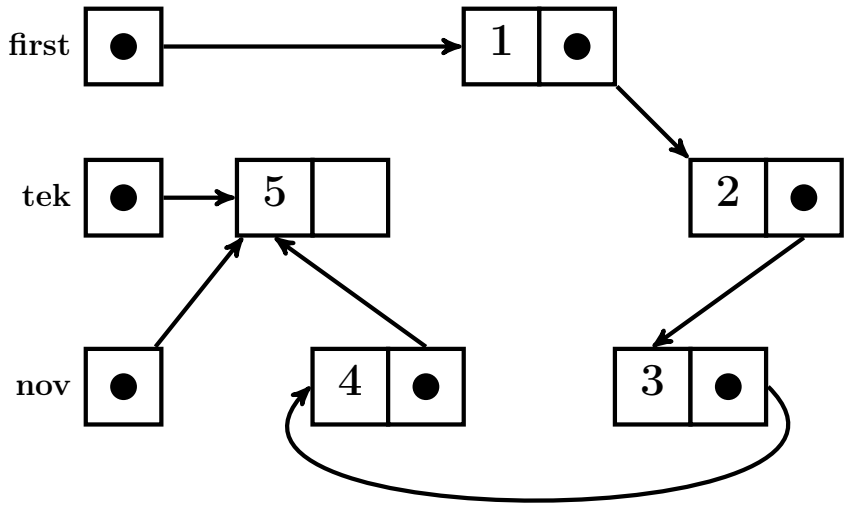
Теперь **nov** можно будет использовать для адресации нового (пятого) звена, поскольку после **tek%next=>nov** указатель **tek** хранит ссылку на четвёртое звено.

15. **k=5; allocate(nov)**



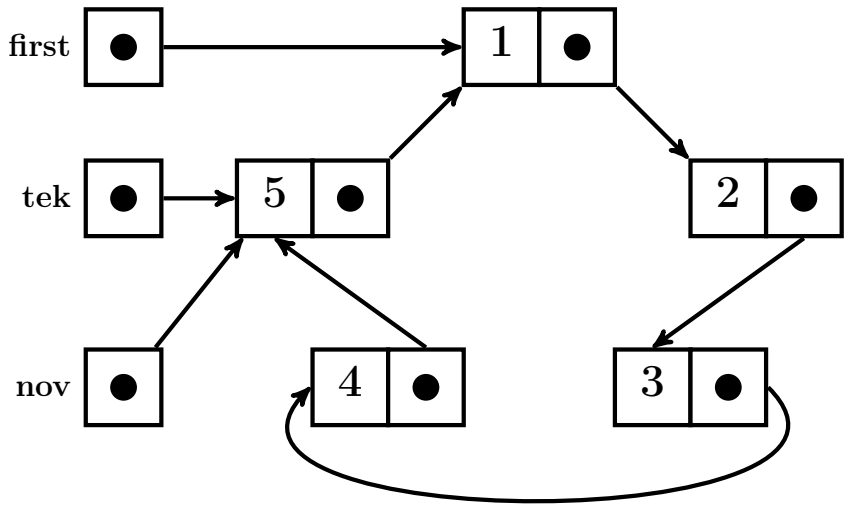
nov хранит указатель на пятое звено (пока пустое).

16. **nov%number=k; tek%next=>nov**



Занесён номер пятого игрока в *number*-поле пятого звена и указатель на пятое звено в *next*-поле четвёртого звена.

17. **tek%next=>first**

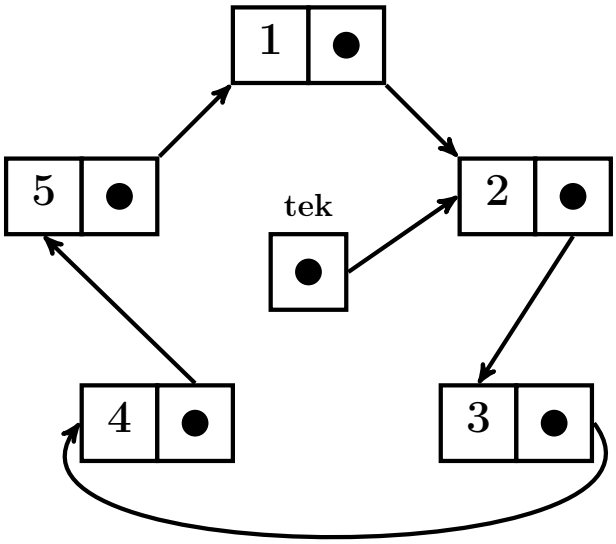


next-поле пятого звена хранит указатель на первое — кольцевой список из пяти звеньев завершён.

18. **nullify(nov)**. Для поиска номера водящего, вообще говоря, указатель **nov** не нужен. Посредством оператора **nullify** разорвём его связь с пятым звеном (чтобы не загромождать в дальнейшем схему ссылок поиска водящего). Значение указателя **tek**, указывающее на последнее звено кольца, предполагает, что проговор читалки начнётся с первого звена. Другими словами, поиск водящего может обойтись и без указателя **first** — ведь звено, указываемое через посредство **first**, указывается ещё и через посредство **tek%next** последнего звена. Поэтому, чтобы не загромождать схему ссылок, разорвём связь **first** с первым звеном **nullify(first)**. Проверить разрыв связи указателя с объектом можно посредством логической функции **associated(имя_указателя)**. В ФОРТРАНе-95 наряду с оператором **nullify** имеется ещё и функция **null**, действие которой при отсутствии аргумента эквивалентно действию **nullify**.

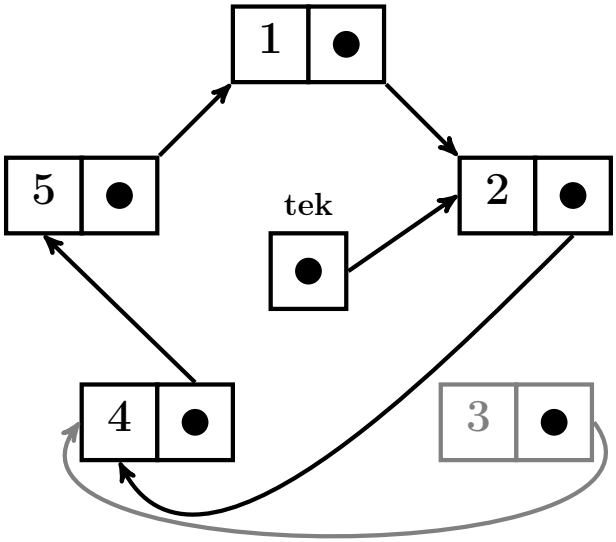
3.4.6 Схема функционирования поиска водящего

1. **k=5; do j=1,m-1; tek=>tek%next; enddo**.



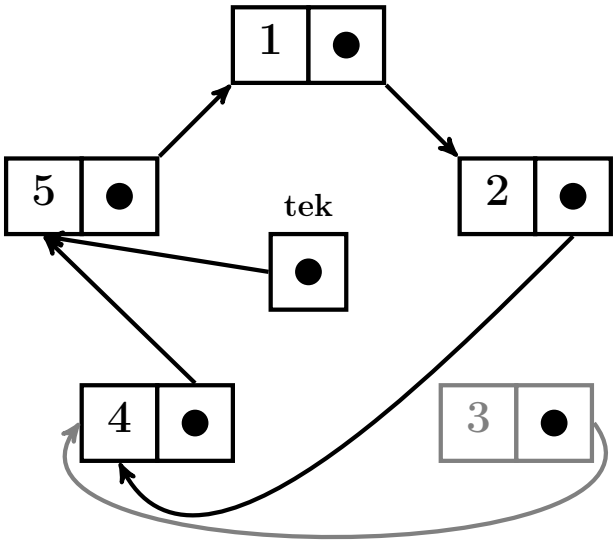
На предыдущем рисунке указатель **tek** адресовал последнее пятое звено. После отработки цикла **do j=1,2** содержимым **tek** будет указатель хранящийся в *next*-поле первого звена, т.е. **tek** станет адресовать второе звено (см. рисунок слева). Третье слово считалки — последнее. Поэтому третье звено нужно будет исключить из кольцевого списка, поместив в *next*-поле второго звена указатель из *next*-поля третьего.

2. **tek%next=>teh%next%next**



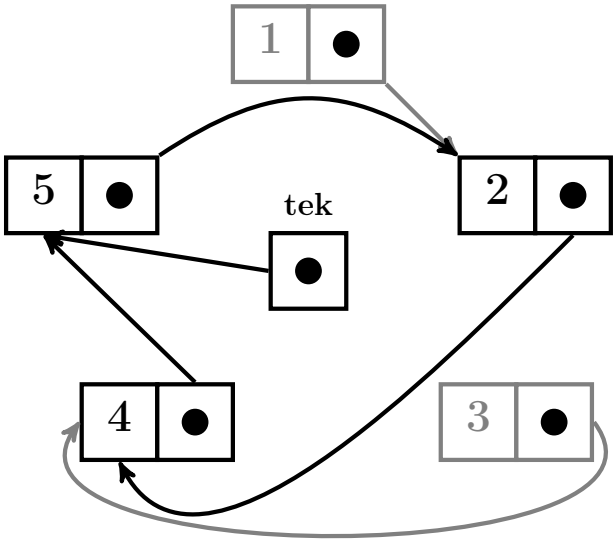
Третье звено исключено из кольцевого списка, но место в оперативной памяти оно продолжает занимать, причём указатель его расположения, ранее хранившийся в *next*-поле второго звена уже затёрт указателем на четвёртое звено. Поэтому наша программа и сама потеряла доступ к третьему звену, и не отдала его место операционной системе (короче, повела себя как собака на сене: *и сама не ест, и другим не даёт*). Тем не менее поиск водящего программа может продолжать.

3. **k=4; do j=1,m-1; tek=>tek%next; enddo**.



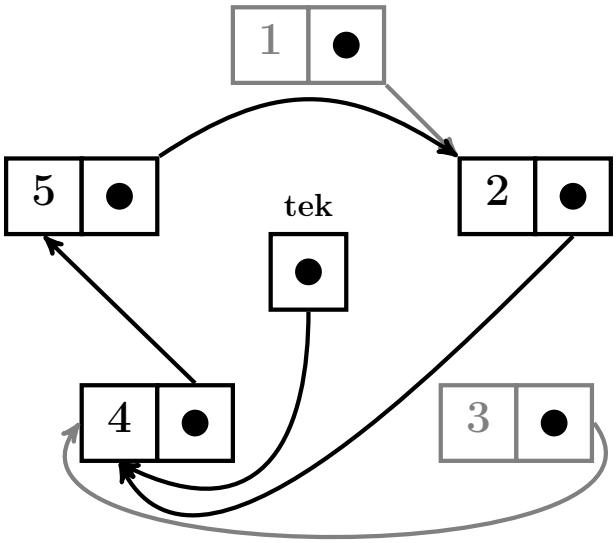
После исключения третьего звена в кольцевом списке осталось четыре участника **1,2,4,5**, причём очередная проговорка считалки начинается с четвёртого. Двукратный повтор оператора **tek=>tek%next** поместит в переменную **tek** указатель на пятое звено. Так что очередным кандидатом на исключение окажется первое звено.

4. **tek%next=>tek%next%next**



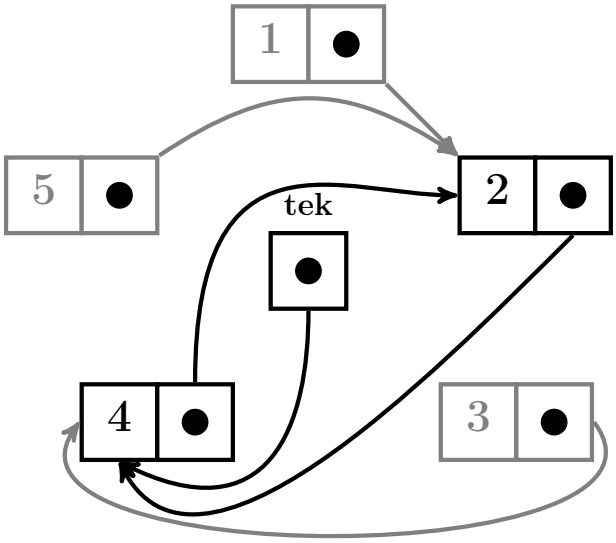
Первый участник исключён из кольцевого списка, но его звено продолжает занимать место в оперативной памяти. Теперь в кольцевом списке только три участника **2,4,5**.

5. **k=3; do j=1,m-1; tek=>tek%next; enddo**



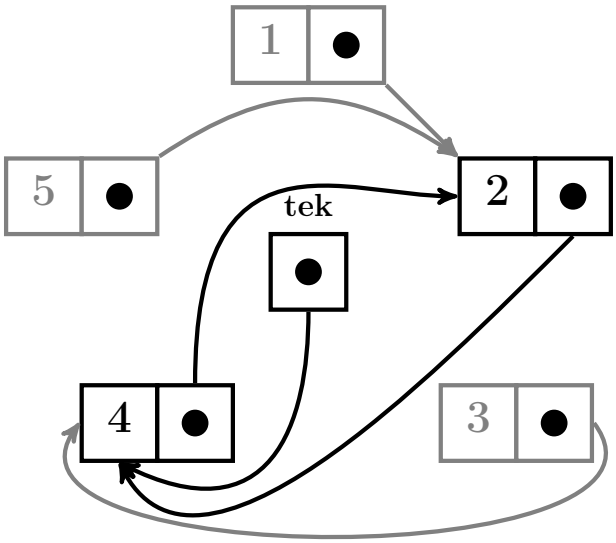
После очередного двукратного повторения оператора **tek=>tek%next** в переменную **tek** будет помещён указатель на четвёртое звено, т.е. очередным кандидатом на вылет окажется участник под номером **5**.

6. **tek%next=>tek%next%next**



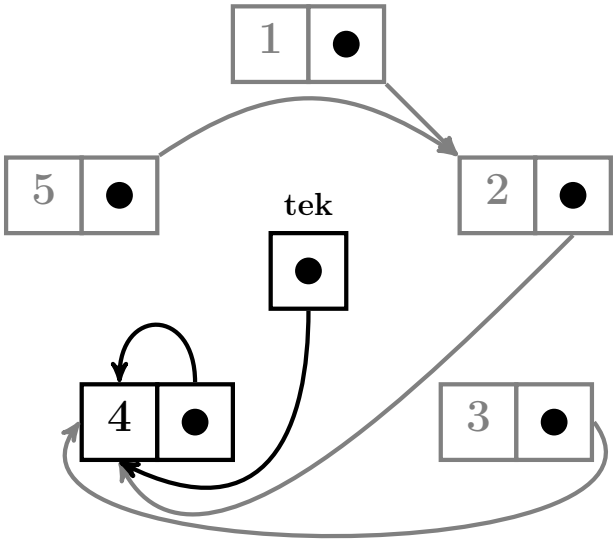
Пятое звено исключено, но продолжает занимать место в оперативной памяти. Теперь в кольцевом списке только два звена **2,4**.

7. **k=2; do j=1,m-1; tek=>tek%next; enddo**



До работы оператора цикла переменная **tek** хранит указатель на четвёртого участника. Однако теперь, когда пятый участник исключён, после четвёртого следует второй. Так что после отработки упомянутого цикла в переменной **tek** опять окажется указатель на четвёртого участника, т.е. последнее слово считалки приходится на второго участника, который должен будет вылететь.

8. **tek%next=>tek%next%next**



В итоге в кольцевом списке оказывается один участник под номером **4**, который и должен водить.

На самом деле, утверждая ранее, что для поиска водящего потребуется лишь одна переменная типа указатель на звено списка, мы не планировали экономое использование оперативной памяти. В результате место, отведённое под звенья всех *вылетевших* участников удерживается программой до завершения её работы, причём доступ к этим звеньям нашей программой утерян навсегда. Для экономного использования памяти можно *вылетающие* звенья возвращать из под *юрисдикции* программы под *юрисдикцию* операционной системы, используя оператор **deallocate**, который удаляет размещённые объекты.

3.4.7 Операторы nullify, deallocate и функция associated

nullify разрывает связь указателя с объектом. (т.е. **nullify(first, nov)** поместит в переменные **first** и **nov** признак отсутствия связи их со звеньями, на которые указывали они раньше). Восстановить прежнее значение указателя на *оторванное* звено можно лишь, если предварительно позаботиться о сохранении значений, уничтожаемых **nullify** в переменных **first** или **nov**, например, в других переменных-указателях.

```
program tpoint6; implicit none
type candidat
  integer number
  type (candidat), pointer :: next
end type candidat
type (candidat), pointer :: q, first, tek, nov  ! здесь значения указателей
                                                ! не определены, т.е. нам
! не известно, связан ли каждый из описанных указателей хоть с каким-нибудь
! адресом. Узнать об этом можно посредством встроенной функции булева типа
! associated(имя_указателя).
integer ier
write(*,*) associated(q),associated(first),associated(tek),associated(nov)

nullify(q,first,tek,nov)

write(*,*) associated(q),associated(first),associated(tek),associated(nov)

allocate(q,stat=ier); if (ier/=0) stop 1

write(*,*) associated(q),associated(first),associated(tek),associated(nov)

q%number=77; write(*,*) 'q%number=',q%number
nullify(q);  write(*,*) 'q%number=',q%number

allocate(q,stat=ier); if (ier/=0) stop 2
q%number=55; write(*,*) 'q%number=',q%number
tek=>q
nullify(q);  write(*,*) 'q%number=',q%number
             write(*,*) 'tek%number=',tek%number

allocate(q,stat=ier); if (ier/=0) stop 3
q%number=33; write(*,*) 'q%number=',q%number
tek=>q
deallocate(q);  write(*,*) 'q%number=',q%number
                write(*,*) 'tek%number=',tek%number
                write(*,*) associated(q),associated(tek)
end
```

Результаты работы программы **tpoint6**:

```
T F T T        ! Как видим, q, tek и nov связаны, а first --- нет.
                          ! Более того, даже можем вывести содержимое их
    q%number=  -843398271 ! number-полей, правда, лишь для того, чтобы
first%number=           0 ! убедиться, что в них действительно есть
  tek%number=   604801882 ! что-то, воспринимаемое компьютером как данные
  nov%number= -1077198184 ! типа integer.
                          ! Однако после nullify(q, first, tek, nov) все
F F F F                   ! все указатели не связаны ни с каким адресом.
    q%number=           0
first%number=           0
  tek%number=           0
  nov%number=           0

               !  После allocate(q) указатель q успешно адресует звено
T F F F        !  типа type(candidat). Пока поля этого звена не заданы.
q%number=         77    ! Теперь полю number присвоено значение.
q%number=          0    ! Поскольку связь разорвана, то выводится
                        ! значение, определяемое системой по умолчанию.
                        ! Однако, на самом деле звено, с которым
                        ! разорвана связь, хотя и недоступно программе
                        ! тем не менее существует в оперативной памяти.
q%number=         55    ! После tek=>q разрыв nullify(q) не помешает
q%number=          0    ! достать значение, занесённое в поле number,
tek%number=         55  ! через указатель tek посредством tek%number.

q%number=         33    ! Однако, если бы использовали не nullify(q), а
q%number=          0    ! deallocate(q), то не только разорвали связь, но
tek%number=          0  ! и уничтожили адресуемое звено, так что достать
                        ! из него содержимое number-поля уже невозможно
                        ! даже через посредство указателя tek, хранящего
                        ! адрес места расположения бывшего звена, о чём
F T                     ! свидетельствует associated(tek).
```

3.4.8 Атрибуты target и pointer

В программах **testlist** и **testring** указатели использовались для создания связного списка, когда связь одного звена с другим осуществлялась исключительно через поле с атрибутом **pointer**. Однако, указатели могут адресовать и обычные переменные (скаляры, массивы), имена которых явно указаны в разделе описаний.

Пусть нужно обменять значениями целые переменные **a** и **b**. В программе **tpoint7** приведены два способа такого обмена и третий способ — обмен значениями адресующих **a** и **b** указателей.

```
program tpoint7; implicit none
integer, target ::   a,  b,  c ! Объявили, что доступ к этим переменным
                               ! возможен не только посредством имён
                               ! a, b и c, НО и посредством указателей.
integer, pointer :: pa, pb, pc ! Место для указателей pa, pb, pc
                               ! отведено, но их значения пока неизвестны.
a=1; b=8                       ! Задание значений a и b
                                  ! ПЕРВЫЙ СПОСОБ ОБМЕНА  a и b значениями
                                  ! --------------------     (для него ни
                                  ! target, ни pa, ни pb, ни pc НЕ НУЖНЫ)
write(*,*) '  a=',a,'      b=',b   ! Вывод заданных значений a и b
c=a; a=b; b=c                     ! Обычный обмен переменных значениями
write(*,*) '  a=',a,'      b=',b,' после ПЕРВОГО СПОСОБА ОБМЕНА.'
write(*,'(40("-"))')
                                  ! ВТОРОЙ СПОСОБ ОБМЕНА a и b значениями
                                  ! -------------------- (через посредство
                                  ! указателей pa, pb и pc)
pa=>a                             ! Указатель pa настроен на адресацию a
pb=>b                             ! Указатель pb настроен на адресацию b
pc=>c                             ! Указатель pc настроен на адресацию c
!---------------------------------
! Другими словами, пока  pa, pb и pc не переопределены, то ФОРТРАН трактует
!---------------------------------  их просто ссылочными синонимами имён
                                  ! a, b и c, что и видно из вывода значений
write(*,*) '  a=', a,'      b=', b ! переменных  a и  b и их ссылочных
write(*,*) ' pa=',pa,'     pb=',pb ! синонимов pa и pb. Они пока отражают
                                  ! результат обмена ПЕРВЫМ СПОСОБОМ.
pc=pa                             ! И здесь имена pc, pa и pb (слева и справа)
pa=pb                             ! от обычного оператора присваивания (=)
pb=pc                             ! ФОРТРАН трактует просто как ссылочные
                                  ! синонимы имён  c, a и b  соответственно.
write(*,*) '  a=',a, '      b=', b,'  Результат ВТОРОГО СПОСОБА ОБМЕНА'
write(*,*) ' pa=',pa,'     pb=',pb,'  Результат ВТОРОГО СПОСОБА ОБМЕНА'
write(*,'(40("-"))')
```

```
                                  ! ТРЕТИЙ СПОСОБ - имитация обмена на
                                  ! -------------
                                  ! основе переприсваивания значений.
write(*,*) '  a=', a,'      b=', b ! указателей. Таков результат обмена
write(*,*) ' pa=',pa,'     pb=',pb !      a и  b    предыдущим способом.
pc=>pa      ! Здесь ситуация принципиально отлична от предыдущей.
pa=>pb      ! Здесь указателям присваиваются НОВЫЕ ЗНАЧЕНИЯ, т.е.
pb=>pc      ! pa, pb и pc перестают быть соответствующими синонимами
            ! первоначальных имён переменных a, b и c, т.е. происходит.
! обмен содержимым именно указателей, а не тех переменных, синонимами
! которых они были во ВТОРОМ СПОСОБе. В результате ТРЕТЬЕГО СПОСОБА
!  pa становится синонимом уже переменной b (но не a) и
!  pb     становится синонимом переменной a (но не b).
! Поэтому, содержимое переменных a и b остаётся таким же каким и было,
! НО, если забыть про новое толкование pa и pb, то может возникнуть
! иллюзия, что обменялись содержимым переменные a и b.
write(*,*) '  a=', a,'      b=', b, ' после ТРЕТЬЕГО СПОСОБА ОБМЕНА'
write(*,*) ' pa=',pa,'     pb=',pb, ' после ТРЕТЬЕГО СПОСОБА ОБМЕНА'
end
```

Результаты работы программы **tpoint7**:

```
   a=           1       b=           8
   a=           8       b=           1  после ПЕРВОГО СПОСОБА ОБМЕНА.
----------------------------------------
   a=           8       b=           1
  pa=           8      pb=           1
   a=           1       b=           8  после ВТОРОГО СПОСОБА ОБМЕНА
  pa=           1      pb=           8  после ВТОРОГО СПОСОБА ОБМЕНА
----------------------------------------
   a=           1       b=           8
  pa=           1      pb=           8
   a=           1       b=           8  после ТРЕТЬЕГО СПОСОБА ОБМЕНА
  pa=           8      pb=           1  после ТРЕТЬЕГО СПОСОБА ОБМЕНА
```

1. Атрибут **target** в описании переменных **a** и **b** означает, что на эти переменные программа *может* ссылаться не только посредством имён **a** и **b**, но и через посредство указателей. Объекты, снабжённые атрибутом **target** часто называют *адресатами*.

2. Указатель всегда определяется типом объекта, который предполагается адресовать, и атрибутом **pointer**. Естественно, содержимое указателя (ссылка на адресат) должно быть настроено соответствующим образом (сразу после описания указателя статус его связности неопределён; поэтому, результат работы, например, функции **associated**, выясняющей статус связности, непредсказуем).

3. Для помещения в указатель той или иной ссылки (прикрепления ссылки к адресату) в ФОРТРАНе используется оператор => (*не путать* с оператором обычного присваивания = — в противном случае компиляция может завершиться успешно, а на этапе выполнения получим аварийный останов). Во ВТОРОМ СПОСОБЕ перед обменом в указатель **pa** посредством **pa=>a** записывается ссылка на адресат **a**, а в **pb** — ссылка на адресат **b**. Тем самым **pa** и **pb** становяться ссылочными синонимами переменных **a** и **b**.

4. В отличие от **testlist**, **testring** и **tpoint6**, где слева и справа от оператора => стояли указатели (т.е. объекты, наделённые атрибутом **pointer**), правые части операторов **pa=>a** и **pb=>b** из **tpoint7** корректны исключительно из-за наличия в описании **a** и **b** атрибута **target**. Если слово **target** убрать из описания **integer, target :: a, b, c**, то при компиляции программы получим:

```
$ gfortran tpoint7.f95
tpoint7.f95:11.4:
pa=>a
    1
ошибка: Pointer assignment target is neither TARGET nor POINTER at (1)
tpoint7.f95:12.4:
pb=>b
    1
ошибка: Pointer assignment target is neither TARGET nor POINTER at (1)
```

5. Наличие слова **target** нисколько не мешает использовать имена **a** и **b** обычным образом, не обращаясь к указателям, если это предпочтительнее. Например, весь ПЕРВЫЙ СПОСОБ обмена осуществлён посредством обычного оператора присваивания (естественно, для такого способа можно обойтись и без атрибута **target**).

6. В языке СИ есть *операция разыменования*, обозначаемая в контексте выполняемых СИ-операторов звёздочкой перед именем СИ-указателя. Так что, образно говоря, ФОРТРАН-имена **pa**, **pb** можно понимать как СИ-имена ***pa** и ***pb**.

Когда-то резко критиковали ФОРТРАН за используемый им способ доступа процедуры к фактическому аргументу — по адресу (**по ссылке**). Критика (приводилась в первом семестре — см.**5.3.7**) была настолько объективно обоснованной, что в языках C и C++ был выбран способ

доступа — **по значению**. Правда, при этом возникла проблема возврата результата, которая удачно решалась посредством указателей. Указатель, передаваемый по значению, нельзя изменить внутри функции, но через него можно было изменить содержимое адресуемой им переменной. И вот в C++ появляется новая возможность по сравнению с СИ — **скрытый указатель** или параметр-ссылка (см. опять-таки первый семестр **5.3.6**).

В книге [9], в частности, отмечается (и с этим трудно не согласиться), что «по мнению многих программистов, параметры-ссылки предлагают более понятный и элегантный интерфейс, чем неуклюжий механизм указателей». Для уяснения сути дела приведём аналоги двух программ из [9] на СИ и C++, демонстрирующие удобство **параметра-ссылки**:

```
#include <iostream>
using namespace std;
void swap_latent(int&,int&);
void swap_pointr(int*, int*);
int main()
{ int a, b;
  a=1; b=8;        cout<<" a="<<a<<"     b="<<b<<endl;

  swap_latent(a,b);

                   cout<<" a="<<a<<"     b="<<b<<endl;

   swap_pointr(&a,&b);

                   cout<<" a="<<a<<"     b="<<b<<endl;
  return 0;
}
void swap_latent(int &a, int &b)
{ int c;
  c=a;     a=b;    b=c;
}
void swap_pointr(int *a, int *b)
{ int c;
   c=*a;  *a=*b;  *b=c;
}
```

Результаты работы этой программы после запуска исполнимого файла, полученного посредством **g++ swap.cpp -lm**:

```
 a=1      b=8
 a=8      b=1
 a=1      b=8
```

Мало кто предпочтёт функцию **swap_pointr** варианту **swap_latent**. Для вызова последней и адреса переменных **a** и **b** определять не надо, и внутри не нужно явно заботиться об операции разыменования, и значок **&** амперсанда встречается только в заголовке функции. Именно он означает, что нужно не копировать значения переменных **a** и **b** главной программы в формальные аргументы, а просто работать по адресам фактических, в неявной форме (скрытно) аппелируя к механизму указателей, но формально в тексте, используя только имена переменных. Другими словами, ... — поступать так, как это испокон веков делал ФОРТРАН (см. ниже функцию **swap_var**, не пугая прикладников страшилками-адресов и указателей):

```
program tpoint8
implicit none
interface
  subroutine swap_poi1(pa,pb); integer, pointer :: pa, pb
  end subroutine swap_poi1
  subroutine swap_poi2(pa,pb); integer, pointer :: pa, pb
  end subroutine swap_poi2
end interface
integer, target :: a, b
integer, pointer :: pa, pb
a=1; b=9
pa=>a
pb=>b
write(*,*) '       a=',a,'   b=',b
call swap_var(a,b)
write(*,*) 'var:   a=',a,'   b=',b
write(*,'(40("-"))')
call swap_poi1(pa,pb)
write(*,*) 'poi1:  a=', a,'   b=',b
write(*,*) 'poi1: pa=',pa,'  pb=',pb
write(*,'(40("-"))')
call swap_poi2(pa,pb)
write(*,*) 'poi2:  a=', a,'   b=',b
write(*,*) 'poi2: pa=',pa,'  pb=',pb
end
```

```
subroutine swap_var(a,b)
implicit none
integer a,b,c
c=a; a=b; b=c
end

subroutine swap_poi1(pa, pb)
implicit none
integer, pointer :: pa, pb
integer c
c=pa;  pa=pb; pb=c
end

subroutine swap_poi2(pa, pb)
implicit none
integer, pointer :: pa, pb, pc
pc=>pa;  pa=>pb; pb=>pc
end
```

Результаты работы **tpoint8**:

```
        a=           1    b=           9
 var:   a=           9    b=           1
---------------------------------------
 poi1:  a=           1    b=           9
 poi1: pa=           1   pb=           9
---------------------------------------
 poi2:  a=           1    b=           9
 poi2: pa=           9   pb=           1
```

В **tpoint8** вызываются три подпрограммы **swap_var**, **swap_poi1** и **swap_poi2**, реализующие алгоритмы обмена, ранее продемонстрированные программой **tpoint7**.

1. **swap_var(a,b)** реализует обычный алгоритм обмена содержимым аргументов.
2. **swap_poi1(pa,pb)** — реализует (по сути) тот же алгоритм, но в качестве аргументов использует указатели на переменные **a** и **b** главной программы.
3. **swap_poi2(pa,pb)** — обменивает ссылками аргументы-указатели.
4. Конечно, программируя на ФОРТРАНе, вряд ли кто-то предпочтёт вместо процедуры **swap_var** процедуру **swap_poi1** (разве что в качестве простого упражнения по вызову процедур с аргументами-указателями).

5. Более того, при использовании **swap_poi1** и **swap_poi2** (в качестве внешних процедур) архиважно явно описать в главной программе их интерфейс:

```
interface
  subroutine swap_poi1(pa,pb); integer, pointer :: pa, pb
  end subroutine swap_poi1
  subroutine swap_poi2(pa,pb); integer, pointer :: pa, pb
  end subroutine swap_poi2
end interface
```

При исключении его из текста на этапе комиляции получим сообщения вида:

```
t1.f95:18.14:
call swap_poi1(pa,pb)
               1
ошибка: Dummy argument 'pa' of procedure 'swap_poi1' at (1) has
an attribute that requires an explicit interface for this procedure ...
```

6. В случае программы на **C++** при вызове **swap_latent** в качестве аргументов явно указывались имена переменных главной программы, и только из описания прототипа функции **swap_latent** (её интерфейса) главная программа узнавала, что имена аргументов внутри функции следует толковать как параметры-ссылки. В случае ФОРТРАНа, когда формальными аргументами **swap_poi1** являются именно указатели, нельзя при её вызове поступить как в C++ — указать в качестве фактических аргументов имена переменных **a** и **b**, хотя значения указателей и являются их синонимами. Так в программе **tpoint8** вызов **call swap_poi1(pa,pb)** нельзя заменить на вызов **call swap_poi1(a,b)** — при компиляции получим:

```
tpoint8.f95:18.15:
call swap_poi1(a,b)
               1
ошибка: Actual argument for 'pa' must be a pointer at (1)
```

ФОРТРАН, в данном случае, требует полного соответствия атрибутов фактического и формального аргументов.

7. Формальные аргументы **swap_var(a,b)** были описаны без атрибута **target**. Его можно включить в их описание. Однако, тогда потребовалось бы включить в интерфейсный блок и явное описание интерфейса **swap_var(a,b)**. Вообще говоря, при переходе на версии современного ФОРТРАНа всегда полезно явно указывать интерфейс всех вызываемых процедур (как — решать нам: либо посредством оператора **interface**, либо модульно).

8. Вызов **swap_poi2** тоже можно рассматривать как соответствующее упражнение по освоению работы с указателями в ФОРТРАНе. Однако, именно обмен указателей ссылками при решении соответствующих задач способен реально сократить затраты процессорного времени. Так, если бы **a** и **b** были массивами с большим количеством элементов, то время обмена данными между ними в случае работы аналогов **swap_var** и **swap_poi1** было бы пропорционально этому количеству. Осуществляя же один обмен ссылками двух указателей, адресующих эти массивы, можем использовать вместо одного массива другой практически без каких-либо временны́х затрат, касающихся пересылки их элементов (см. следующий пункт).

3.4.9 Достоинство работы с указателями

Использование указателей требует неослабного внимания, что, конечно, вряд ли можно отнести к достоинству. Однако, временны́е затраты при работе с указателями могут оказаться значительно меньше временных затрат программы, работающей без указателей. Рассмотрим задачу обмена содержимым двух одномерных массивов.

В старых версиях ФОРТРАНа не было типа данного **указатель**. Поэтому обмен массивов своим содержимым свелся бы к полномасштабной *перекачке* содержимого одного массива в другой. Программа **tpoint9** (приведена ниже) вызывает процедуры **swap1** и **swap2**, которые оценивают временны́е затраты двух вариантов **kmax**=**100001**-кратной *перекачки* массивов из **n**=**100000** элементов целого типа.

Оценка временны́х затрат осуществлется и процедурой **swap3**, которая, как бы, имитирует обмен содержимым двух массивов посредством обмена значениями соответствующих указателей на эти массивы.

```
program tpoint9; use swap; implicit none
integer i
write(*,'(7x,a,i10)') ' kmax=',kmax
write(*,'(7x,a,i10)') '    n=',n
a=(/(i,i=1,n)/)         ! Задание значений
b=(/(i,i=n,1,-1)/)      ! векторов a и b.
call swap1              ! c=a; a=b; b=c
call swap2              ! цикл по обмену каждого из n элементов
call swap3              ! обмен обычными ФОРТРАН-ссылками на вектора
end
```

Получение исполнимого файла достигается командой

```
gfortran swap.f95 tpoint9.f95
```

```
    Оптимизация -O0                Оптимизация -O1       ! Результаты
                                                         ! программы
       kmax=    100001                 kmax=    100001   !  tpoint9
          n=    100000                    n=    100000

 swap1: time= 0.923E+01          swap1: time= 0.927E+01
 swap2: time= 0.503E+02          swap2: time= 0.111E+02
 swap3: time= 0.717E-03          swap3: time= 0.751E-03
```

Если требование о копировании данных можно заменить копированием ссылок на данные, то преимущество указателей очевидно.

Исходный текст модуля swap

```
module swap; implicit none
integer, parameter ::    n=100000   ! количество элементов в массивах
integer, parameter :: kmax=100001   ! количество повторов обмена
integer, target :: a(n), b(n)
integer, pointer :: pa(:), pb(:), pc(:)
contains
subroutine swap1                    ! пересылка массивов без
integer c(n), k                     ! цикла по всем их элементов
real t1, t2
call cpu_time(t1);
do k=1,kmax; c=a; a=b; b=c; enddo
call cpu_time(t2);
write(*,'(a,e10.3)') ' swap1: time=',t2-t1
end subroutine swap1
subroutine swap2                    ! пересылка массивов с
integer  k, i, m                    ! циклом по всем их элементам
real t1, t2
call cpu_time(t1);
do k=1,kmax
  do i=1,n
     m=a(i); a(i)=b(i); b(i)=m
  enddo
enddo
call cpu_time(t2); write(*,'(a,e10.3)') ' swap2: time=',t2-t1
end subroutine swap2
subroutine swap3
integer k, i, m
real t1, t2
pa=>a                                           ! Настройка pa и pb.
pb=>b                                           !
call cpu_time(t1);
do k=1,kmax                                     ! kmax-кратный обмен
  pc=>pa; pa=>pb; pb=>pc                        !  pa и pb ссылками.
enddo
call cpu_time(t2); write(*,'(a,e10.3)') ' swap3: time=',t2-t1 ! Его время
end subroutine swap3
end module swap
```

Ещё мéньших затрат времени можно достичь, используя, так называемые, *целочисленные* ФОРТРАН-указатели.

О целочисленных показателях. Целочисленные указатели обеспечивают переход к СИ-указателям при совмещённом программировании, хотя могут использоваться и сами по себе в ФОРТРАН-программах.

В ФОРТРАНе-90 их нет, но в ФОРТРАН-95 и **gfortran** они включены. В **gfortran** использование целочисленных указателей требует при компиляции включения опции **-fcray-pointer**. Целочисленные указатели всегда имеют тип **integer(4)**.

1. Значение целочисленного указателя (т.е. именно адреса) можно вывести в виде некоторого целого числа.

2. При объявлении целочисленного указателя имя типа (на значение которого ссылается указатель) не конкретизируется — только слово **pointer** за которым в круглых скобках следует пара

   pointer(имя_указателя, имя_адресной_переменной).

   Адресная переменная (см. [4, 5]) связана с областью памяти, адрес которой установлен в указатель, так что через посредство указателя

   Значения из адресной переменной передаются адресату.

   Значения из адресата передаются адресной переменной.

```
program tpoint10; implicit none; integer, parameter :: n=5   ! Пример
integer i
real(8) ::   a(n)=(/(dble(i),i=1,n)/)
real(8) :: var(n), x     ! var и x - адресные переменные, на которые
pointer (p,var), (q,x)   !           нацеливаются указатели p и q.
p=loc(a)                 ! loc(a) находит адрес (a) в виде целого числа
write(*,*) '1)'
p=loc(a)
write(*,*) p
do i=1,n                              ! Тут просто выводим значение
  write(*,*)  p, loc(a(i)), a(i)      ! указателя p, вычисляемое по p=p+8
    p=p+8                             ! и через loc(a(i)), а затем и
enddo                                 ! значение a(i), адресуемое p
write(*,*) '2)'                       ! Однако, если хотим осуществить
p=loc(a)                              ! аналогичный вывод адресной
write(*,*) p                          ! переменной (т.е. через приращение,
do i=1,n                              ! указателя), то помним, что её
  write(*,*)  p, loc(var(1)), var(1) ! элементы адресуются относительно
  p=p+8                               ! адреса её начала, каковым является
enddo                                 ! в данном случае элемент var(1).
```

```
write(*,*) '3)'                       ! При формальной замене во фрагменте
p=loc(a)                              ! 1) обозначения a(i) на var(i)
write(*,*) p                          ! результат (кроме var(1))
do i=1,n                              ! не совпадёт с результатом 1),
  write(*,*)  p, loc(var(i)), var(i) ! так как в качестве начального
  p=p+8                               ! адреса var будет использован не
enddo                                 ! loc(a(1)), a   loc(a(i)+8) !!!

write(*,*) '4)'           ! Здесь целочисленный указатель параметр - цикла.
do q=loc(a),loc(a)+32,8   ! Поскольку с q связана адресная переменная x, и,
  write(*,*) q, loc(x), x ! в q послан "адрес" массива A, то теперь x ---
enddo                     ! просто другое имя текущего элемента a(i).
end program tpoint10
```

Получение исполнимого файла и его активация

```
gfortran -fcray-pointer tpoint10.f95 -o tpoint10
./tpoint10 > tpoint10.res
```

```
 1)                           ! Результат работы tpoint10
             6299776
             6299776             6299776   1.0000000000000000
             6299784             6299784   2.0000000000000000
             6299792             6299792   3.0000000000000000
             6299800             6299800   4.0000000000000000
             6299808             6299808   5.0000000000000000
 2)
             6299776
             6299776             6299776   1.0000000000000000
             6299784             6299784   2.0000000000000000
             6299792             6299792   3.0000000000000000
             6299800             6299800   4.0000000000000000
             6299808             6299808   5.0000000000000000
 3)
             6299776
             6299776             6299776   1.0000000000000000
             6299784             6299792   3.0000000000000000
             6299792             6299808   5.0000000000000000
             6299800             6299824   0.0000000000000000
             6299808             6299840   0.0000000000000000
 4)
             6299776             6299776   1.0000000000000000
             6299784             6299784   2.0000000000000000
             6299792             6299792   3.0000000000000000
             6299800             6299800   4.0000000000000000
             6299808             6299808   5.0000000000000000
```

Ещё один пример. Программа tpoint11.

```
program tpoint11; implicit none; integer, parameter :: n=5
real(8)    a(n) /n*0d0/, b(n) /n*8.0_8/
real(8) var (n), x
integer i
pointer (p,var)
pointer (q,x)
p=loc(a)
write(*,'(a,10f5.1)') '                  a(1:5)=',  a(1:5)
write(*,'(a,10f5.1)') '                var(1:5)=',var(1:5)
var(2)=0.3_8
write(*,'(a,10f5.1)') ' var( 2 )=0.3 -->  a(1:5)=',  a(1:5)
write(*,'(a,10f5.1)') '                var(1:5)=',var(1:5)
a(5)=5.7_8
write(*,'(a,10f5.1)') '   a( 5 )=5.7 -->  a(1:5)=',  a(1:5)
write(*,'(a,10f5.1)') '                var(1:5)=',var(1:5)
p=loc(b)
write(*,'(a,10f5.1)') '                  b(1:5)=',  b(1:5)
write(*,'(a,10f5.1)') '                var(1:5)=',var(1:5)
var(1:5:2)=(/(i,i=1,5,2)/)
write(*,'(a,10f5.1)') '   var(1:5:2)=(/(i,i=1,5,2)/) -->  b(1:5)=',  b(1:5)
write(*,'(a,10f5.1)') '                                  var(1:5)=',var(1:5)
q=loc(var)
write(*,*) p,q
end
```

1. В **tpoint11** описаны три массива **a(5)**, **b(5)** и **var(5)** типа **real(8)**).
2. На этапе компиляции массивы **a** и **b** заполняются константами **0.0** и **8d0** соответственно.
3. Массив **var(5)** — адресная переменная указателя **p**.
4. Оператор **pointer (p,var)** должен располагаться в разделе описаний программной единицы
5. Адресную переменную **var** нельзя инициализировать при описании её типа или в операторе **data**.
6. **p=loc(a)**: в **p** помещается указатель на начало массива **a**.
7. Вывод элементов массивов **a** и непосредственно, и через адресную переменную **var**. Имя **var(i)** элемента адресной переменной — синоним имени **a(i)** до тех пор, пока не переопределим указатель **p**.

8. Изменение **var(2)** эквивалентно изменению **a(2)**, что подтверждается последующим выводом.

9. Аналогично, и изменение элемента **a(5)** эквивалентно изменению того, на что ссылается **var(5)**.

10. **p=loc(b)**: переключение указателя **p** на адресацию массива **b**.

11. Теперь адресная переменная **var** адресует элементы массива **b**, что и подтверждается соответствующим выводом.

Результат работы tpoint11

```
$ gfortran tpoint11.f95 -fcray-pointer -o tpoint11
$ ./tpoint11
                  a(1:5)=  0.0  0.0  0.0  0.0  0.0
                var(1:5)=  0.0  0.0  0.0  0.0  0.0
 var( 2 )=0.3 -->  a(1:5)=  0.0  0.3  0.0  0.0  0.0
                var(1:5)=  0.0  0.3  0.0  0.0  0.0
   a( 5 )=5.7 -->  a(1:5)=  0.0  0.3  0.0  0.0  5.7
                var(1:5)=  0.0  0.3  0.0  0.0  5.7
                  b(1:5)=  8.0  8.0  8.0  8.0  8.0
                var(1:5)=  8.0  8.0  8.0  8.0  8.0
  var(1:5:2)=(/(i,i=1,5,2)/) -->  b(1:5)=  1.0  8.0  3.0  8.0  5.0
                              var(1:5)=  1.0  8.0  3.0  8.0  5.0
```

3.5 О чем узнали из третьей главы? (2-ой семестр)

1. При разработке и написании программ использовать переменные только простых элементарных типов (**integer**, **int**, **real(8)**, **double**, и т.д) не всегда удобно.

2. И ФОРТРАН, и СИ предоставляют программисту возможность оперировать более сложными структурами данных (**массивы**, **структуры**, **указатели**).

3. **Массив** — именованный набор данных, состоящий из конечного количества элементов одинакового типа. Доступ к конкретному элементу осуществляется по номеру (индексу) элемента.

4. **Производный тип** (**структура**) — именованный набор данных из конечного количества элементов (**полей**) возможно разных типов. Доступ к нужному полю осуществляется по имени поля.

5. **Имя поля** придумываем сами так, чтобы оно мнемонически ясно отражало смысловую нагрузку данного, хранимого в поле.

6. **Указатель** — тип данных, предназначенный для хранения адреса.

7. Массив называется **одномерным** (или вектором), если для указания его элемента достаточно одного индекса.

8. **Индексация СИ-массива** всегда начинается **с нуля**.

9. **Индексация ФОРТРАН-массива** (если она не изменена при его описании) всегда начинается с **единицы**.

10. Размер одномерного массива в СИ обычно указывается в квадратных скобках (называемых конструктором массива) после имени описываемого массива.

11. Размер одномерного массива в ФОРТРАНе можно указать в круглых скобках, которые пишутся после имени массива при описании типа его элементов, либо в операторе **dimension**.

12. Для отделения имени структуры от имени её поля используется: в СИ **точка**, а в стандарте ФОРТРАНа-95 значок **процента**, хотя некоторые компиляторы допускают и точку.

13. Доступ к содержимому поля СИ-структуры при использовании указателя на структуру возможен только через операцию разыменования указателя (например, либо **((*a).x)**, либо **a->x**, где **x** — имя поля, **a** - указатель на структуру).

14. В ФОРТРАНе операция присваивания значения указателю обозначается парой символов => (это вовсе не ссылка на поле структуры).

15. Указатели и производные типы данных позволяют создавать в оперативной памяти **динамические** структуры данных.

16. В СИ полезен оператор **typedef** позволяющий описать синоним существующего типа данных.

3.6 Третье домашнее 3а-задание (2-ой семестр)

1. Написать СИ-программу, которая: моделирует координаты точки пространства вектором из трёх элементов, вводит координаты двух точек и вызывает функцию расчёта расстояния между ними.

   Описания главной программы и функции должны быть расположены в разных файлах.

2. Написать СИ-программу, которая, используя производный тип

   ```
   struct coord
   {  float x, y, z; };
   ```

   вызывает процедуры:

   - **rdr_coord** ввода данного типа **struct coord** для чтения координат точки **A**,
   - **wrt_coord** вывода данного типа **struct coord** для печати координат точки **A**,
   - **rdr_coord** ввода данного типа **struct coord** для чтения координат точки **B**,
   - **wrt_coord** вывода данного типа **struct coord** для печати координат точки **B**,
   - расчёта расстояния между точками **A** и **B**,
   - выводит результат расчёта.

   Тип struct coord должен быть определён в одном файле;

   Описание процедур — в другом файле;

   Описание главной программы — в третьем.

   Обеспечить ввод исходных данных из файла **input** и вывод результатов в файл **result**.

   Обеспечить работу с программой посредством Make-файла.

3. Задача та же (что и 2), **ОДНАКО** главная программа и процедуры должна использовать вместо типа **struct coord** тип **struct coord***.

3.7 Третье домашнее 3b-задание (2-ой семестр)

1. Написать ФОРТРАН-программу, которая, используя тип

```
type coord4
  real(4) x, y, z
end type coord4
```

   - вызывает процедуру **rdr4** ввода данного типа **coord4** для чтения координат точки **A**,
   - вызывает процедуру **wrt4** вывода данного типа **coord4** для печати координат точки **A**,
   - вызывает процедуру **rdr4** ввода данного типа **coord4** для чтения координат точки **B**,
   - вызывает процедуру **wrt4** вывода данного типа **coord4** для печати координат точки **B**,
   - вызывает процедуру **dist4** расчёта расстояния между точками **A** и **B**,
   - выводит результат расчёта.

   Тип type coord4 должен быть определён в одном файле;

   Описание процедур — в другом файле;

   Описание главной программы — в третьем.

   Обеспечить ввод исходных данных из файла **input** и вывод результатов в файл **result**.

   Обеспечить работу с программой посредством Make-файла.

2. Задача та же, но поместить описания всех процедур в единицу компиляции типа **module**.

3. Задача та же, но дополнить **module** описаниями соответствующих процедур, работающих с координатами типа **real(8)**. Добавить в главную программу вызов новых функций и вывод их результата.

4. Задача та же, но ОБЕСПЕЧИТЬ возможность вызова процедур **rdr4** и **rdr8** по одному имени **rdr**; процедур **wrt4** и **wrt8** по одному имени **wrt** и процедур **dist4** и **dist8** по одному имени **dist**.

5. Задача. Модифицировать СИ- и ФОРТРАН-решения задачи о считалке так, чтобы оперативная память, отведённая изначально для кандидатов на водящего, передавалась по мере их выбывания операционной системе.

6. Написать две СИ-функции расчёта значения полинома **n**-ой степени по заданным значениям его коэффициентов, хранящихся в массиве, и аргументу. В первой описание формального параметра-массива оформить в терминах конструкции **[]**; во второй — через указатель. Сформировать своё мнение о достоинствах и недостатках обоих способов. Продемонстрировать работу соответствующей тестирующей программы.

4 Массивы (введение).

4.1 Уяснение ситуации.

В программировании использовать только простые переменные не всегда удобно. Например, вряд ли удобно описывать **2550** различных переменных для хранения коэффициентов системы линейных уравнений с **50**-ю неизвестными. Структура данных **массив** предоставляет возможность доступа к любому коэффициенту по одному имени.

1. **Массив** – набор конечного количества нумерованных элементов **одинакового типа**, расположенных в оперативной памяти непосредственно друг за другом в порядке возрастания номера.

2. Массив называют **одномерным** (или вектором), если в его описании указано, что все элементы распределены вдоль одного измерения. Поэтому для выборки элемента достаточно указать его порядковый номер (одно целое число; например, **a(3)** или **a(k)**).

3. Массив называют **двумерным** (или матрицей), если в его описании указано, что элементы распределены по двум измерениям (т.е. для выборки элемента массива нужно два целых числа (первое — номер строки матрицы, второе – номер столбца).

4. Массивы могут быть и **многомерными**. Книгу удобно моделировать **трехмерным** массивом. Положение буквы в ней определяется тремя числами — номерами страницы, строки и знака в строке.

5. Массив называют **индексированной** переменной. Иногда нумерацию элементов выгодно с нуля или с отрицательного числа. Поэтому вместо слова *номер* часто используется термин **индекс**.

6. По адресу начального элемента массива, индексу и типу элемента просто вычислить адрес элемента, указанного индексом. Чем больше индексов имеет элемент массива, тем большее время требуется для расчета его адреса.

7. Число измерений массива называют **рангом** (или размерностью) массива, а число элементов массива – его **размером** (не путаем *размер* с *размерностью*). Массив ранга 1 – вектор; ранга 2 – матрица (допускаются массивы до 7-го ранга включительно).

8. **Форма**(или **конфигурация)** массива определяется его **рангом** и количеством элементов по каждому измерению. Это количество называют также протяженностью или **экстентом** измерения. Например, описание **real b(2,7,10)** или же **real, dimension (2,7,10) :: b** определяют массив ранга **3** и размером **140** элементов ($2 \times 7 \times 10$).

9. **Конфигурация** массива – вектор целого типа, размер которого равен рангу исходного массива, а элементы равны размерам соответствующих измерений. Находит форму встроенная функции **shape**:

```
program testshape; implicit none        !  Результат пропуска testshape:
integer conf(2)                         !  $ gfortran testshape.f95
real(8) a(-5:5,0:20), b(2,7,10)         !  $ ./a.out
conf=shape(a)                           !
write(*,'(a,2i3)')  shape(a)=', conf    !  shape(a)= 11 21
write(*,'(a,3i3)') shape(b)=', shape(b) !  shape(b)=  2  7 10
end
```

10. Массивы одинаковой конфигурации называют **согласованными** (так массивы **real(8) d(-3:-2, -3:3,-4:5), b(2,7,10)** согласованы).

11. В современном ФОРТРАНе можно задать массив без имени в виде последовательности скаляров посредством **конструктора** массива:

```
integer a(3)          ! Здесь набор чисел 111, 222, 333, заключённый
a=(/111, 222, 333/) ! в символы конструктора массива (/ и /), есть
write(*,*) a          ! массив без имени. Результат работы:
end                   !         111         222         333
```

12. **Секция** массива (начиная с ФОРТРАНа -90) – часть массива, задаваемая посредством имени массива и списка индексов секции:

```
program testsect; implicit none          ! Результаты пропуска testsect:
integer a(3,5)                           ! $ gfortran a2.f95
integer b(5), c(5), e (3), f(3)          ! $ ./a.out
do i=1,3                                 !  11  21  31
  do j=1,5                               !  12  22  32
    a(i,j)=10*i+j                        !  13  23  33
  enddo                                  !  14  24  34
enddo                                    !  15  25  35
write(*,'(3i4)') a                       !
b=a(1,:); write(*,'(a,5i5)') ' b=',b     ! b=   11   12   13   14   15
c=a(3,:); write(*,'(a,5i5)') ' c=',c     ! c=   31   32   33   34   35
e=a(:,2); write(*,'(a,5i5)') ' e=',e     ! e=   12   22   32
f=a(:,4); write(*,'(a,5i5)') ' f=',f     ! f=   14   24   34
end
```

13. При описании матрицы первым указывается количество её строк, а затем количество столбцов. Например, описание **integer a(3,5)** означает, что матрица состоит из трёз строк и пяти столбцов. При использовании в программе её элемента **a(i,j)** первым индексом **i** указывается индекс строки, а вторым **j** – индекс столбца. В СИ с точностью до синтаксиса имеем такую же очередность записи размерностей матрицы и индексов её элементов.

    Однако СИ-матрица упорядочена в оперативной памяти по по строкам, в то время **ФОРТРАН-матрица** упорядочена по столбцам. Именно поэтому в первой строке вывода матрицы **a** оказались числа **11**, **21**, **31**, которые расположены в её первом столбце.

14. В общем случае размещение в оперативной памяти многомерного ФОРТРАН-массива происходит по правилу:

    чем левее индекс, тем быстрее он меняется.

```
program test3d; implicit none ! Результаты работы test3d:
integer a(4,3,2)              ! $ gfortran test3d.f95 ! Наряду с кратными
do i=1,4                      ! $ ./a.out             ! циклами DO
  do j=1,3                    !                       ! в ФОРТРАНе-95
    do k=1,2                  !  111 211 311 411      ! можно использовать
      a(i,j,k)=100*i+10*j+k   !  121 221 321 421      ! оператор forall
    enddo                     !  131 231 331 431
  enddo                       !  112 212 312 412 forall (i=1:4,j=1:3,k=1:2)
enddo                         !  122 222 322 422   a(i,j,k)=100*i+10*j+k
write(*,'(4i4)') a            !  132 232 332 432 endforall
end
```

    Размещение же СИ-массивов следует правилу:

    чем правее индекс, тем быстрее он меняется.

```
#include <stdio.h>                  //         Результат работы:
int main()                          //  111  112    121  122    131  132
{ int i, j, k, a[4] [3] [2];        //  211  212    221  222    231  232
  int *pa;                          //  311  312    321  322    331  332
  for(i=0;i<=3;i++)                 //  411  412    421  422    431  432
  for(j=0;j<=2;j++)
  for(k=0;k<=1;k++)
      a[i][j][k]=100*(i+1)+10*(j+1)+(k+1); pa=&a [0] [0] [0];
  for (i=1;i<=4;i++)
     {printf("%4d %4d   %4d %4d   %4d %4d\n", *pa,     *(pa+1),
                                               *(pa+2), *(pa+3),
                                               *(pa+4), *(pa+5));

      pa=pa+6;
     }
  return 0;
}
```

15. В СИ многомерный массив описывается последовательным указанием его конструкторов (квадратных скобок) по очередному измерению **int a[4] [3] [2]**. В ФОРТРАНе **integer a(4,3,2)**.

16. Совпадение размеров СИ- и ФОРТРАН-массивов (когда размер последнего указан просто числом элементов) не означает совпадения значений соответствующих индексов. Индексация СИ-массива всегда начинается с нуля, а ФОРТРАН-массива (если не оговорено иное) — с единицы. Так последний элемент СИ-массива **int a[4] [3] [2]** доступен по имени **a[3][2][1]**, а Фортран-массива **integer a(4,3,2)** — по имени **a(4,3,2)**.

17. Доступ к элементу СИ-массива можно выразить, как заключая соответствующие индексы в квадратные скобки, так и используя эти индексы в адресных выражениях языка указателей. Например,:

```
#include <stdio.h>
int main()
{ double a[2][3];  int i, j; a[0][0]=  0;  a[0][1]=  1;  a[0][2]=  2;
                               a[1][0]= 10;  a[1][1]= 11;  a[1][2]= 12;
for (i=0; i<=1;i++)
  { printf("%5.1lf %5.1lf %5.1lf\n",     a[i][0],    a[i][1],     a[i][2]);
    printf("%5.1lf %5.1lf %5.1lf\n", *(  a[i]+0),*(  a[i]+1),  *( a[i]+2));
    printf("%5.1lf %5.1lf %5.1lf\n\n",*(*(a+i)+0),*(*(a+i)+1),*(*(a+i)+2));
  }
for (i=0; i<=1;i++)
  { printf("%p %p %p\n",    &(a[i][0]),  &(a[i][1]), &(a[i][2]));
    printf("%p %p %p\n",     a[i]+0 ,   a[i]+1,  a[i]+2);
    printf("%p %p %p\n\n", *(a+i)+0 , *(a+i)+1, *(a+i)+2);
  }
return 0; }
```

В случае двумерного массива обозначение **a[i]** трактуется как адрес начального элемента **i**-ой строки. Имя матрицы (**a**) можно трактовать как указатель на начальный элемент одномерного массива с элементами **a[0]**, **a[1]** и т.д., которые хранят адреса нулевых элементов каждой из строк.

Запись ***(a[i]+2)** означает:

1) **a[i]** — адрес начального байта **i**-ой строки;

2) **a[i]+2** — увеличить его на количество байт **2*sizeof(double)** необходимое для размещения двух ее элементов (0-го и 1-го), находя, тем самым, адрес элемента **a[i][2]**;

3) ***(a[i]+2)** — через операцию разыменования адреса **a[i]+2** получить доступ к его содержимому.

Запись ***(*(a+i)+2)** означает:

1) **a** — получить адрес начального байта **нулевой** строки (адрес, хранящийся в **a[0]**);

2) **a+i** — увеличить его на **i*sizeof(double)** байт, находя адрес элемента **a[i]**, который хранит адрес начального элемента **i**-ой строки матрицы;

3) ***(a+i)** — извлекаем из **a[i]** через операцию разыменования (внутренняя *) адрес начального элемента **i**-ой строки;

4) ***(a+i)+2** — вычисляем адрес элемента **a[i][2]** по формуле

&a[0]+i*sizeof(double) +2*sizeof(double);

5) ***(*(a+i)+2)** через операцию разыменования найденного адреса получаем доступ к содержимому элемента **a[i][2]**.

Результат работы последней программы:

```
 0.0   1.0   2.0
 0.0   1.0   2.0
 0.0   1.0   2.0

10.0  11.0  12.0
10.0  11.0  12.0
10.0  11.0  12.0
                                                  // Легко заметить, что
0x7fff07a6ed10 0x7fff07a6ed18 0x7fff07a6ed20 // адрес элемента a[0][1]
0x7fff07a6ed10 0x7fff07a6ed18 0x7fff07a6ed20 // на 8 байт больше адреса
0x7fff07a6ed10 0x7fff07a6ed18 0x7fff07a6ed20 // элемента a[0][0].
                                                  // П О Ч Е М У (на первый
0x7fff07a6ed28 0x7fff07a6ed30 0x7fff07a6ed38 // взгляд) адрес a[0][2]
0x7fff07a6ed28 0x7fff07a6ed30 0x7fff07a6ed38 // больше адреса a[0][1]
0x7fff07a6ed28 0x7fff07a6ed30 0x7fff07a6ed38 // только на 2 байта ???
```

18. Функция форматного вывода **printf** позволяет отпечатать адрес, по которому хранится значение переменной (в частности, и элемента массива, расположенного в оперативной памяти). Для вывода адреса используется спецификатор формата **%p** (от pointer – указатель):

```
#include <stdio.h>                          // Файл matra.c

int main()
{ int a[3][4]   ={ { 0,1,2,3}, { 10, 11, 12, 13},  {20, 21, 22, 23} };
  int i, j;                          // Если a - имя массива, то обозначение
  printf("a= %p  &a=%p\n",a, &a);  // &a тождественно обозначению a и означает
  printf("%s \n","  Индекс  ");    // адрес начального элемента массива.
  printf("%s  %s  %s\n","i-й строки","     &a[i]    ", "    a[i]  ");
  for (i=0;i<3; i++)                              // Если a - матрица, то &a[i]
    printf("%5i  %17p  %13p\n",i,&(a[i]),a[i]); // тождественно a[i] и означает
                                                  // адрес элемента a[i] [0].
  printf("\n%10s  %15s  %15s  %15s  %15s\n","   \\  j  ",
         "0-й столбец","1-й столбец", "2-й столбец","3-й столбец");
  printf("%7s  %17s  %15s %16s %16s\n","i  \\ ",
         "   &  a[i][j]","   &  a[i][j]", "   &  a[i][j]","   &  a[i][j]");
  printf("%7s  \n", "   \\");
  for (i=0;i<3;i++)            // Пример вывода адресов элементов матрицы и
    { printf("%d-я строка",i); // содержимого этих элементов:
      for (j=0;j<4;j++) printf("  %12p  %2d ", &a[i][j], a[i][j]); printf("\n");
    }
  printf("\n &i=%p   i=%d\n",&i,i); // Пример вывода адресов и содержимого
  printf(  " &j=%p   j=%d\n",&j,j); // простых переменных.
  return 0;
}
```

Результат работы этой программы:

```
a= 0xfee3f290  &a=0xfee3f290
  Индекс
i-й строки       &a[i]          a[i]
    0         0xfee3f290     0xfee3f290
    1         0xfee3f2a0     0xfee3f2a0
    2         0xfee3f2b0     0xfee3f2b0

    \  j        0-й столбец      1-й столбец      2-й столбец      3-й столбец
  i  \          &  a[i][j]       &  a[i][j]       &  a[i][j]       &  a[i][j]
      \
0-я строка  0xfee3f290   0   0xfee3f294   1   0xfee3f298   2   0xfee3f29c   3
1-я строка  0xfee3f2a0  10   0xfee3f2a4  11   0xfee3f2a8  12   0xfee3f2ac  13
2-я строка  0xfee3f2b0  20   0xfee3f2b4  21   0xfee3f2b8  22   0xfee3f2bc  23

 &i=0xfee3f28c   i=3
 &j=0xfee3f288   j=4
```

Значения адресов, приведённые здесь, — четырёхбайтовые, которые использовались на ранних версиях компиляторов.

19. Различие в очередности расположения в оперативной памяти элементов многомерных ФОРТРАН- и СИ-массивов надо учитывать при совмещённом программировании. ФОРТРАН-функция **reshape** позволяет изменять форму массива (в частности, преобразовать очерёдность ФОРТРАН-расположения в СИ-расположение.

```
program test3d2; implicit none; integer a(4,3,2), c(2,3,4), b (4,3,2)
integer i,j,k                                ! Обратите внимание на удобство
forall (i=1:4,j=1:3,k=1:2)                   ! использования оператора
      a(i,j,k)=100*i+10*j+k                  ! forall вместо написания
endforall                                    ! трёхкратного вложенного цикла
write(*,*) ' ФОРТРАН-расположение       :'; write(*,'(4i4)') a
write(*,*) ' СИ-расположение из ФОРТРАНа:'
c=reshape(a, (/2,3,4/),order=(/3,2,1/));    write(*,'(6i4)') c
write(*,*) ' ФОРТРАН-расположение из  СИ:'
b=reshape(c,(/4,3,2/), order=(/3,2,1/));    write(*,'(4i4)') b
end
```

20. В зависимости от момента выделения памяти под массив (на этапе компиляции или во время выполнения программы) различают **статические** и **динамические** массивы соответственно.

4.2 Одномерные массивы (простые примеры)

1. Описания ФОРТРАН-массива и варианты его вывода.
2. Варианты описания одномерного массива в СИ.
3. Изменение диапазона индексов ФОРТРАН-массива.

4.2.1 Описание и вывод одномерного ФОРТРАН-массива.

```
      program exarr1                    ! В программе описывается статический
      implicit none                     ! массив k из четырех элементов целого
      integer k(4) / 40, 30, 20, 10 /   ! типа. k(1)=40, ..., k(4)=10.
      integer i                         ! Удобно при выводе:
      write(*,*) k                      !           массива в одну строку;
      write(*,*) (k(i),i=1,4)           !           линейной выборки из массива;
      write(*,*) (i,k(i),i=4,1,-1)      !           элементов в обратном порядке;
      write(*,1000)
      write(*,1001) (i, k(i), i=1,4,1) ! Удобно при форматированном выводе.
      write(*,1000)
      do i=4,1,-1                     ! Удобно, если между заголовком цикла и
        write(*,1001) i, k(i)         ! оператором вывода есть расчетная часть
      enddo
 1000 format(5x,'  индекс       значение'/5x,'    i           k[i]    ')
 1001 format(1x,'  ',i7,i14)
      end
```

```
Результат работы exarr1.out    !  Задать число элементов в описании массива
  40 30 20 10                  !  можно не только целым числом, но и имено-
  40 30 20 10                  !  ванной константой целого типа, например:
  4 10 3 20 2 30 1 40          !        integer n
       индекс       значение   !        parameter (n=4)
         i           k[i]      !        real a(n), b(n), c(n)
         1            40       !
         2            30       !  Ясно, что изменить число элементов проще
         3            20       !  один раз в операторе именования константы.
         4            10       !  Если же оно встречается и в теле программы
       индекс       значение   !  то выигрыш от именованной константы очевиден
         i           k[i]      !
         4            10       !  ФОРТРАН-95 позволяет описать и задать
         3            20       !  именованную константу одним оператором:
         2            30       !
         1            40       !        integer, parameter :: n=4
```

1. Число в круглых скобках после имени ФОРТРАН-массива при его описании, равно и количеству элементов массива, и индексу последнего элемента массива, так как по умолчанию номер начального элемента равен **единице**.

2. При описании одномерного массива в СИ число в квадратных скобках тоже трактуется как число элементов, но индекс начального элемента всегда полагается равным **нулю** (поэтому, в частности, С-индекс последнего элемента на единицу меньше числа элементов).

3. В ФОРТРАНе задать начальные значения элементов массива можно не только операторами описания, но и посредством оператора **data**, который выполняется только **при компиляции**.

```
      program exarr2      !  Задать начальные значения элементам массива
      implicit none       ! можно и посредством невыполняемого при работе
      integer n           ! программы оператора   data.
      parameter (n=5000)
      integer k(n), i
      data k / n*10 /
      write(*,1000); write(*,1001) (i, k(i), i=1,4,1)
 1000 format(5x,'  индекс       значение'/5x,'    i           k[i]   ')
 1001 format(1x,'  ',i7,i14)
      end
                              ! Правда, при этом размер исполнимого
       индекс       значение   ! кода в байтах окажется равным:
         i           k[i]      !        n       g77     gfortran
         1            10       !      5000     26470      26115
         2            10       !     10000     48390      48067
         3            10       !     20000     88390      88067
         4            10       !     40000    166470     166115
```

4.
```
      program exarr3; implicit none ! При компиляции без инициализации
      integer n                     ! размер исполнимого кода МЕНЬШЕ:
      parameter (n=5000)            !        n        g77     gfortran
      integer k(n), i               !      5000      6546       6113
      do i=1,n                      !     10000      8390       8039
        k(i)=i                      !     20000      8390       8039
      enddo                         !     40000      6534       6127
      write(*,1000); write(*,1001) (i, k(i), i=1,4,1)
 1000 format(5x,'  индекс       значение'/5x,'    i           k[i]   ')
 1001 format(1x,'  ',i7,i14)
      end
```

5.
```
program exarr4   ! При "лобовой" попытке описать массив, размер
implicit none    ! которого должен задаваться не константой, а
integer n, i     ! переменной, ФОРТРАН-компиляторы сообщат об ошибке!!!
write(*,*) 'Введи размер массива'
read (*,*) n     ! In file rdrarr0c.f90:6
integer k(n)     ! integer k(n)
end              !         1  Unexpected data declaration statement at (1)
```

6. Массив можно описать не только посредством операторов описания типа, но и оператором **dimension**:

```
integer k(4)           ! Вектор целого типа из четырех элементов
real a(10)             ! Вещ. вектор из 10 элементов одинарной точности
logical b(-5:5)        ! Вектор из 11 элементов булева типа
real(8) w(0:100)       ! Вещ. вектор из 101 элемента удвоенной точности
double precision v(50) ! --"--"--"--"--  50 --"--"--"--"--"--"--"--
character(33) c(13)    ! Каждый из 13 элементов вмещает до 33 литер
dimension q(153)       ! Все 153 элемента массива q типа REAL, т.к. в
                       !  данном контексте действует правило умолчания
```

Служебное слово **dimension** предназначено для явного описания именно структуры (формы) массива, не касаясь типа его элементов.

7. Современный ФОРТРАН допускает в одном операторе описания размещение через запятую всех атрибутов, указывающих свойства массива. Список атрибутов завершается **двойным двоеточием ::**, за которым следуют имена описываемых объектов.

```
program ar95; implicit none                        !   Конструкция вида:
integer, parameter  :: n=4                         ! (/ список значений /)
real, dimension (n) :: b=(/2.1,2.2,2.3,2.4/)       !     называется
integer, dimension(3), parameter :: a=(/3, 2, 1 /) ! конструктором массива.
integer c(5), i               ! Встроенный DO-цикл для указания в операторе
data (c(i),i=1,3) /3*1/       ! DATA значений НЕ ВСЕХ элементов   массива.
integer d(2:1), e(5,1000,3:2) ! ФОРТРАН-95 допускает массивы нулевой
                              ! длины, т.е. не содержащие ни одного
                              ! элемента, если хотя бы по одному
                              ! измерению нижняя граница больше верхней.
write(*,*) 'a=',a; write(*,*) 'b=',b; write(*,*) 'c=',c
write(*,*) ' Размер массива c равен',size(c)
write(*,*) ' Размеры массивов d и e равны:',size(d), size(e)
write(*,*) ' Содержимое массива d и e:', d, e
end
```

```
 a=           3           2           1                          ! Результат
 b=   2.100000       2.200000       2.300000       2.400000     ! работы
 c=           1           1           1           0           0 ! программы
  Размер массива c равен           5                            ! ar95
  Размеры массивов d и e равны:           0           0         !
  Содержимое массива d и e:                                     !
```

Оператор DATA без встроенного DO-цикла по индексу с необходимостью требует указания значений всех элементов массива.

4.2.2 Описание и вывод одномерного массива в СИ.

1. ```
   #include <stdio.h>
   int main()
   {int k[4]={40, 30, 20, 10}, i; // Квадратные скобки - конструктор массива
    printf(" k:  %i\n",*k);
    printf(" k:  %3i %3i %3i %3i\n",k[0],k[1],k[2],k[3]);
    printf("%3s  индекс    значение    значение\n"," ");
    printf("%3s     i        k[i]      *(k+i) \n"," ");
    for (i=3;i>=0;i--) printf(" %8i %1s %8i %1s %8i\n",i," ",k[i]," ",*(k+i));
    printf("%3s  индекс    значение    значение\n"," ");
    printf("%3s     i        k[i]      *(k+i) \n"," ");
    for (i=0;i<=3;i++) printf(" %8i %1s %8i %1s %8i\n",i," ",k[i]," ",*(k+i));
    return 0;
   }
    k:  40                              //  Результат работы этой программы.
    k:   40  30  20  10                 //
        индекс    значение    значение  //
           i        k[i]      *(k+i)    //   Помним, что в СИ по умолчанию
           3         10         10      //   идентификатор массива -- это
           2         20         20      //   указатель на его начальный
           1         30         30      //   элемент с индексом 0
           0         40         40      //   Поэтому  синтаксически доступ к
        индекс    значение    значение  //   нужному элементу массива можно
           i        k[i]      *(k+i)    //   оформить двумя способами:
           0         40         40      //    1) по традиции -- через k[i]
           1         30         30      //                или
           2         20         20      //    2) через операцию разыменования
           3         10         10      //       указателя  *(k+i)
   ```

   Запись ***(k+i)*** означает:

   (a) извлечь адрес **&k[0]** начального элемента переменной **k**;

   (b) увеличить его на **i* sizeof(int)** – число байт под **i** элементов типа **int**, находя адрес **k[i]**, и получить доступ

   (c) через операцию разыменования ***(k+i)** к содержимому **k[i]**.

   Функция **printf** позволяет указать в списке вывода через операцию разыменования идентификатор массива ***k**. Однако в отличие от ФОРТРАНа, который выводил весь массив, отпечатается только один нулевой элемент. Для вывода **n** элементов массива нужно написать функцию, например:

   ```
   void prtfor(int *a, int n)
   { int i;
     for (i=0;i<n;i++) printf("%i ",a[i]); printf("\n");
   }
   ```
   ```

2. Проинициализировать массив при его С-описании (т.е. задать начальные значения его элементов) можно двумя способами:

   - явно указывая число элементов массива в квадратных скобках (как в предыдущей программе) перед списком инициализации;
   - не указывая в квадратных скобках (при их наличии) количество элементов массива (в этом случае число элементов определяется компилятором по списку инициализации). Например,

```
#include <stdio.h>
int main()
{int k[]={40, 30, 20, 10};         // Описан массив k из 4 элементов.
 int i, num, num1;
 printf(" k:  %i\n",*k);
 printf(" k:  %3i %3i %3i %3i\n",k[0],k[1],k[2],k[3]);
 printf("%3s Индекс    значение    значение\n"," ");
 printf("%3s    i        k[i]      *(k+i) \n"," ");
 for (i=3;i>=0;i--) printf("%8i   %8i   %8i\n",i, k[i], *(k+i));
 printf(" %8i %8i \n", k[6], *(k+6)); // Элемента k[6] в массиве НЕТ.
 printf(" Одно int-значение размещается в %d байтах.\n",sizeof(int));
 printf(" Массив занимает   %d байт памяти.\n", sizeof(k));
 num=sizeof(k)/sizeof(int);
 printf(" В массиве %d элементов типа int.\n", sizeof(k)/sizeof(int));
 num1=num-1;
 printf(" Индекс последнего элемента = %d.\n", num1);
 printf("%3s    i        k[i]      *(k+i) \n"," ");
 for (i=num1;i>=0;i--) printf("%8i   %8i   %8i\n",i, k[i], *(k+i));
 return 0; }
 k:  40                            // СИ для НЕ КОНТРОЛИРУЕТ значения
 k:   40  30  20  10               // индексов массива на предмет их
    индекс    значение    значение // выхода за пределы, допускаемые
       i        k[i]      *(k+i)   // описанием. Поэтому есть шанс
       3         10         10     // заставить программу извлечь
       2         20         20     // содержимое несуществующего
       1         30         30     // элемента, например, k[6].
       0         40         40     //
 -17991432 -17991432               // <-- это его значение - СЛУЧАЙНО
                                   //     и может меняться от пропуска
 Одно int-значение размещается в 4 байтах.// к пропуску. Для выяснения
 Массив занимает   16 байт памяти.        // числа элементов в массиве
 Массив состоит из 4 элементов типа int.  // удобно использовать
 Индекс последнего элемента = 3.          // операцию
        i        k[i]      *(k+i)      //     sizeof(имя массива).
        3         10         10        // Она в случае применения
        2         20         20        // к имени массива возвращает
        1         30         30        // число байт  памяти, занятое
        0         40         40        // компилятором под массив.
```

3. В прикладных задачах иногда желательно объявить массив, размер которого задается переменной, вычисляемой внутри тела программы. Компилятор **gcc** предоставляет такую возможность:

```
#include <stdio.h>
int main()
{ int n, i;    printf("Введи число коэффициентов полинома:\n");
               scanf ("%d", &n); printf(" n=%d\n",n);
  double  a[n];
  printf(" double-значение занимает %d байтов.\n", sizeof(double));
  printf(" double-массив   занимает %d байтов.\n", sizeof(a));
  printf(" double-массив состоит из %d элементов типа double.\n",
                                    sizeof(a)/sizeof(double));
  for (i=0;i<n;i++)
    { printf("введи %d коэффициент\n",i); scanf("%le", &a[i]);}
  printf(" i          a[i]\n");
  for (i=0;i<n;i++) printf(" %i %23.16le\n",i,a[i]); return 0; }
Введи число коэффициентов полинома:
3
 n=3
 double-значение размещается в 8 байтах.
 double-массив   занимает      24 байт памяти.
 double-массив   состоит  из   3 элементов типа double.
введи 0 коэффициент
1                          // В отличие от предыдущих примеров массив (a)
введи 1 коэффициент        // в данном случае -- динамический, поскольку
1.3                        // его размер определяется во время работы
введи 2 коэффициент        // программы. Правда, освободить память, занятую
2.5e-2                     // им таким образом, можно только
 i          a[i]           //
 0  1.0000000000000000e+00 // заключив описание double a[n] с операторами,
 1  1.3000000000000000e+00 // использующими его в фигурные скобки, то есть
 2  2.5000000000000001e-02 // оформляя блок:
#include <stdio.h>
int main()
{ int n, i;     printf("Введи число элементов массива:\n");
                scanf ("%d", &n);  printf(" n=%d\n",n);
 { double  a[n];
  printf(" double-значение занимает %d байтов.\n", sizeof(double));
  printf(" double-массив   занимает %d байтов.\n", sizeof(a));
  printf(" double-массив состоит из %d элементов типа double.\n",
                                    sizeof(a)/sizeof(double));
 }
 double a[n+5]; printf(" double-значение : %d байтов.\n", sizeof(double));
                printf(" double-массив   : %d байт памяти.\n", sizeof(a));
printf(" double-массив из  %d элементов.\n", sizeof(a)/sizeof(double));
  return 0; }
```

```
Введи число элементов массива:     // Результат работы
3                                   // последней программы:
 n=3
 double-значение размещается в 8 байтах.
 double-массив   занимает      24 байт памяти.
 double-массив   состоит  из   3 элементов типа double.

 double-значение размещается в 8 байтах.
 double-массив   занимает      64 байт памяти.
 double-массив   состоит  из   8 элементов типа double.
```

Без организации блока компилятор выдаст сообщения:

```
rdrarr0b1.c:12: error: redeclaration of 'a' with no linkage
rdrarr0b1.c:7: error: previous declaration of 'a' was here
```

Вообще говоря, для организации динамических массивов в СИ используются функции **malloc()** и **free()** (#**include** <**stdlib.h**>; см., например, **man malloc**; подробнее см.**12.2**).

4. Как и в случае ФОРТРАНа, инициализация массива во время компиляции приводит к значительному увеличению исполнимого кода. Например,

```
int main()                          //    n      gcc     c++
{ int k[5000]={1,2,3,4,5,           //  5000   24740   24804
              6,7,8,9,10};          //  10000  44740   44804
                                    //  20000  84740   84804
 return 0;                          //  40000 164740  164804
}
```

Отсутствие инициализации при компиляции приводит к принципиально меньшему размеру исполнимого кода:

```
int main()                           //    n       gcc     c++
{ int k[5000];                       //  5000    4622    4686
                                     //  10000   4622    4686
                                     //  20000   4622    4686
 return 0;                           //  40000   4622    4686
}
```

5. C++ наряду с процедурно-ориентированными возможностями языка C, для работы с одномерными массивами предоставляет класс **vector**, который входит в библиотеку стандартных шаблонов, определяя динамический массив как класс (подробнее см., например, [9]).

4.2.3 Изменение диапазона индексов ФОРТРАН-массива.

В C и C++ индекс начального элемента массива всегда нулевой.
ФОРТРАН позволяет изменять диапазон допустимых индексов массива:

```
program rdrarr1; implicit none
real(8) a(-2:2) / 1.1, 1.2, 1.3, 1.4, 1.5/;    integer i
write(*,1000); write(*,1001) (i, a(i), i=-2,2)
 1000 format(5x,'  индекс       значение'/5x,'    i           a[i]    ')
 1001 format(1x,'  ',i7,d23.16)
end
```

```
       индекс       значение          ! Как следует исправить
         i           a[i]             ! текст исходной программы,
        -2 0.1100000023841858E+01     ! чтобы верными в пределах
        -1 0.1200000047683716E+01     ! шестнадцати значащих цифр
         0 0.1299999952316284E+01     ! оказались  значения всех
         1 0.1399999976158142E+01     ! заданных элементов массива, a
         2 0.1500000000000000E+01     ! не только a[2] ???
```

```
#include <stdio.h> // СИ-аналог ФОРТРАН-программы rdrarr1 и его
int main()         //                                  результаты.
{double a[5]={1.1, 1.2, 1.3, 1.4, 1.5};  int i;
 printf("%3s  индекс %5s   значение %15s   значение\n"," "," "," ");
 printf("%3s     i   %5s     a[i]   %15s   *(a+i)  \n"," "," "," ");
 for (i=0;i<=4;i++)
   printf(" %8i %1s %23.16e %1s %23.16e\n",i," ", a[i]," ", *(a+i));
 return 0;
}
```

```
    индекс          значение                    значение        // Помним, что
       i              a[i]                        *(a+i)         // по умолчанию
       0    1.1000000000000001e+00    1.1000000000000001e+00  // C-константы
       1    1.2000000000000000e+00    1.2000000000000000e+00  // задаются
       2    1.3000000000000000e+00    1.3000000000000000e+00  // с удвоенной
       3    1.3999999999999999e+00    1.3999999999999999e+00  // точностью.
       4    1.5000000000000000e+00    1.5000000000000000e+00
```

Поэтому, если придётся C-программу, ориентированную на задачу вычислительного характера, переводить на ФОРТРАН (в котором обычно вещественные константы по умолчанию полагаются четырёхбайтовыми), то НЕ будем игнорировать соответствующий ФОРТРАН-синтаксис. Например, не

real(8), parameter :: x=1.3

а **real(8), parameter :: x=1.3d0** или **real(8),parameter :: x=1.3_8**.

4.3 Ещё несколько простых ФОРТРАН примеров.

Рассмотрим задачу суммирования первых **n** элементов одномерного массива. Оформим алгоритм разными способами:

1. главной программой;
2. внешней функцией;
3. внутренней функцией;
4. воспользуемся встроенной ФОРТРАН-функцией **sum**.

Для краткости текста главной программы первые **n** элементов массива заполним числами натурального ряда, вводя только **n**.

Конечно, указанную сумму можно найти по известной формуле, не проводя суммирование "в лоб". Однако, на тривиальной задаче проще приобрести навыки работы с **одномерным массивом**, а совпадение результатов, полученных обоими способами, может служить критерием отсутствия ошибок и в записи алгоритма суммирования, и в записи упомянутой формулы.

4.3.1 Оформление простой программой

```
program testpsum; implicit none; integer, parameter :: nmax=1000
integer  n, i; real ar(nmax), s
write(*,*) 'Введи (n<=1000) - число заполняемых элементов массива'
read (*,*) n; write(*,*) ' n=',n
do i=1,n; ar(i)=i; enddo; write(*,1000) (i,ar(i),i=1,n)
  s=0
  do i=1,n; s=s+ar(i); enddo    ! Современный ФОРТРАН позволяет: s=sum(ar).
  write(*,*) ' s=',s,' по формуле:',n*(n+1.0)/2.0
 1000 format(1x,'  Номер       Содержимое  '/&
 &            1x,' элемента      элемента   '/(1x,i5,e20.7) )
end
```

```
 Введи (n<=1000) - число заполняемых элементов массива ! Результат работы
5                                                       ! исполнимого файла
  n=           5                                        ! программы testpsum
   Номер       Содержимое
  элемента      элемента
     1       0.1000000E+01
     2       0.2000000E+01
     3       0.3000000E+01
     4       0.4000000E+01
     5       0.5000000E+01
  s=   15.00000      по формуле:   15.00000
```

4.3.2 Оформление внешней функцией

В качестве аргументов функции примем имя массива и количество суммируемых элементов. Современный ФОРТРАН предоставляет несколько способов описания массива в качестве формального параметра. Используем способ явного задания формы и размера массива, служащего формальным параметром.

```
function fsum1(a,n) result(s); implicit none ! Функция fsum1(a,n) находит
integer n, i; real a(n), s                   ! сумму первых n элементов
s=0.0; do i=1,n; s=s+a(i); enddo             ! одномерного массива (a).
end function fsum1

program testfsum1; implicit none       ! Программа запоминает в первых n
integer, parameter :: nmax=1000        ! элементах вещественного вектора
real ar(nmax), fsum1, s                ! ar(nmax) первые n чисел
integer n, i                           ! натурального ряда, а затем вызывает
write(*,*) 'Введи (n) - число генерируемых элементов массива'
read (*,*) n; write(*,*) ' n=',n       ! функцию fsum1(a,n), которая находит
do i=1,n; ar(i)=i; enddo               ! сумму первых n элементов вектора a.
write(*,1000) (i,ar(i),i=1,n)
s=fsum1(ar,n)                          ! Обращение к внешней функции fsum1.
write(*,*) ' n=',n,' s=',s, ' по формуле:', n*(n+1.0)/2.0
 1000 format(1x,'  Номер       Содержимое  '/&
&            1x,' элемента      элемента   '/(1x,i5,e20.7) )
end

 Введи (n<=1000) - число заполняемых элементов массива ! Результат работы
5                                                       ! tstfsum1 совпадает с
  n=           5                                        ! результатом из 4.3.1.
   Номер       Содержимое
  элемента      элемента                 ! Заметим, однако, что в tstfsum1
     1       0.1000000E+01               ! массив ar состоит из 1000 элементов,
     2       0.2000000E+01               ! а функция fsum1 при обращении
     3       0.3000000E+01               ! fsum1(ar,n) будет "считать" его
     4       0.4000000E+01               ! 5-элементным. Ничего страшного
     5       0.5000000E+01               ! в этом нет. ВАЖНО, чтобы n
  s=   15.00000      по формуле:   15.00000          ! НЕ БЫЛО БОЛЬШЕ 1000.
```

1. В старых ФОРТРАН-программах описание массива (формального аргумента процедуры), могло иметь вид **real a(1)**.

   Не используем подобные описания: некоторые компиляторы *"рассматривают"* выход индекса (при работе программы) за объявленный диапазон как ошибку.

2. Если **формальный** аргумент – массив, то в качестве **фактического аргумента** может быть и элемент массива. Функцию **fsum1** без каких-либо изменений можно использовать для подсчёта суммы элементов с любого элемента до **n-го**:

```
program tsqfsum1;                  ! Программа демонстрирует возможность
implicit none                      ! подачи в качестве фактического аргу-
integer, parameter :: nmax=1000    ! мента не только имени массива, но и
real ar(nmax),fsum1, s             !  его элемента, несмотря на то, что
integer n, i                       !  соответствующий формальный аргумент
write(*,*) 'Введи (n) - число генерируемых элементов массива' ! описан как
read (*,*) n; write(*,*) ' n=',n                               ! массив
do i=1,n; ar(i)=i; enddo;
write(*,1000) (i,ar(i),i=1,n); s=fsum1(ar,n)          ! Передан: массив ar;
write(*,*)' n=',n,' s=',s,' по формуле:',n*(n+1.0)/2.0
s=fsum1(ar(3),n-2)                                          ! элемент ar(3);
write(*,1001) 3,5,s
do i=1,n; s=fsum1(ar(i), n-i+1); write(*,1001) i,n,s;enddo ! элемент ar(i).
 1000 format(1x,'  Номер       Содержимое  '/&
&            1x,' элемента      элемента   '/(1x,i5,e20.7))
 1001 format(1x,i2,'..',i2,': s=',e15.7)
end
 Введи (n) - число генерируемых элементов массива
 5
   n= 5
    Номер       Содержимое
   элемента      элемента
      1       0.1000000E+01
      2       0.2000000E+01
      3       0.3000000E+01
      4       0.4000000E+01
      5       0.5000000E+01
   n= 5 s=  15. по формуле:  15.
   3.. 5: s=  0.1200000E+02     <--= Здесь           a(3)+a(4)+a(5)
   1.. 5: s=  0.1500000E+02     <--=       a(1)+a(2)+a(3)+a(4)+a(5)
   2.. 5: s=  0.1400000E+02     <--=            a(2)+a(3)+a(4)+a(5)
   3.. 5: s=  0.1200000E+02     <--=                 a(3)+a(4)+a(5)
   4.. 5: s=  0.9000000E+01     <--=                      a(4)+a(5)
   5.. 5: s=  0.5000000E+01     <--=                           a(5)
```

Рекомендация: Подобную возможность ФОРТРАНа (*то есть подачу в качестве аргумента процедуры имени конкретного элемента массива вместо имени самого массива*) **лучше не использовать** – она требует от человека особого внимания. Однако в чужих программах указанный прием может встретиться. Поэтому полезно иметь представление о нем.

4.3.3 Оформление внутренней ФОРТРАН-функцией

Как и в предыдущем примере в качестве двух формальных аргументов функции выберем имя массива и имя переменной с числом суммируемых элементов.

```
program tinfsum0; implicit none      ! Программа демонстрирует описание
integer, parameter :: nmax=1000      ! внутренней функции пользователя
real ar(nmax), s                     !  fsum0(a,n), которая подсчитывает
integer n, i                         ! сумму первых n элементов вектора.
write(*,*) 'Введи (n) - число заполняемых элементов массива'
read (*,*) n; write(*,*) ' n=',n
ar=(/ (i,i=1,n) /)                   ! Справа от = конструктор массива
write(*,1000) (i,ar(i),i=1,n)
s=fsum0(ar,n)                        !<--= Возможные вызовы fsum0
write(*,1002) n, s, n*(n+1)/2.0
s=fsum0(ar(3),n-2); write(*,1001) 3,5,s                      !<--=
do i=1,n; s=fsum0(ar(i), n-i+1); write(*,1001) i,n,s; enddo !<--=
 1000 format( 1x,'  Номер       Содержимое  '/&
&             1x,' элемента      элемента   '/ (1x,i5,e20.7) )
 1001 format(1x,i2,'..',i2,': s=',e15.7)
 1002 format(1x,' n=',i2,' s=',e15.7,' по формуле:',e15.7)
contains                                  !
function fsum0(a,n) result(s)             !
real a(n), s; integer n, i                ! Описание внутренней
s=0; do i=1,n; s=s+a(i); enddo            ! ! функции.
end function fsum0                        !
end
```

1. Если главная программа при вызове внутренней функции всегда требует сложения всех элеменов массива до **n**-го включительно, то имя **n** можно исключить из списка формальных и фактических аргументов: ведь оно глобально по отношению к внутренней функции и потому доступно из неё. После исключения, однако, в функции придётся заменить описание **real a(n)** на **real a(nmax)**, т.к. имя переменной **n** без помещения в список формальных аргументов процедуры, нельзя использовать для указания размера массива.

2. Расчет суммы элементов вектора выгодно оформить внешней функцией – обращение к ней может потребоваться в самых разных программах. Именно поэтому в ФОРТРАН-95 включена соответствующая **встроенная** функция **sum**, позволяющая удобно и эффективно суммировать элементы массива (или его сечения) любой допустимой размерности.

4.3.4 Чуть-чуть о встроенной ФОРТРАН-функции SUM

Современный ФОРТРАН позволяет вообще не описывать функцию расчета суммы элементов вектора, так как имеет свою собственную соответствующую встроенную функцию **sum(имя массива)**. Поэтому на современном ФОРТРАНе должна пройти программа:

```
program testsum          ! Программа запоминает в первых n элементах
implicit none            ! вещественного вектора ar(nmax) первые n
integer nmax, n, i       ! чисел натурального ряда, а затем вызывает
parameter (nmax=1000)    ! встроенную функцию sum, которая посредством
real ar(nmax), fsum1, s  ! вызова sum(ar(i1:i2)) находит сумму  элементов
write(*,*) 'Введи (n) - число заполняемых элементов массива' !  с i1-го .
read (*,*) n; write(*,*) ' n=',n                              ! по i2-ой
do i=1,n; ar(i)=i; enddo;  write(*,1000) (i,ar(i),i=1,n);
s=sum(ar(1:n)); write(*,1002) n,s,n,s/n,(n+1)/2.0;          !   вызовы
s=sum(ar(3:n)); write(*,*) 'n=',n-2,' s=',s,' s/n=',s/(n-2) ! встроенной
do i=1, n                                                   !     sum
  s=sum(ar(i:n))                                            ! и печать
  write(*,*) ' n=',n-i+1,' s=',s,' s/n=',s/(n-i+1)          ! результатов
enddo
s=sum(ar(3:4)); write(*,*) ' s=',s           !<--= еще один вызов sum
 1000 format(1x,'  Номер       Содержимое  '/&
&            1x,' элемента      элемента   '/ (1x,i5,e20.7) )
 1002 format(1x,' n=',i2,' s=',e15.7,' s/',i2,'=',e15.7,&
&                                  ' по формуле:',e15.7)
end
```

1. Вызов встроенной **sum** более удобен, чем вызов **fsum1**. Например, если нужно просуммировать элементы с третьего по четвертый, то вызов **fsum1** следует записать так: **s=fsum1(ar(3),2)**. Здесь, хоть и просто, но второй аргумент (двойку) пришлось вычислить в уме. При вызове же **s=sum(ar(3:4))** даже мысли не возникает о целесообразности такого подсчета.

2. Наряду с функцией **sum** современный ФОРТРАН предоставляет целое семейство функций, нацеленных на работу с массивами (см., [2, 3, 4] или в данном пособии раздел **7**. Эти встроенные функции можно распределить по нескольким подгруппам, например: вычислительная работа с массивами (нахождение суммы, произведения элементов, скалярного произведения векторов, максимального или минимального элементов); преобразование массивов (многомерный в одномерный или наоборот); получение справочных данных о массиве (размер, форма, значение границ по каждому измерению).

4.4 ФОРТРАН-массив как формальный аргумент функции.

Современный ФОРТРАН предоставляет три способа описания массива в качестве формального аргумента:
1. Явное задание формы формального аргумента.
2. Заимствование формальным аргументом формы фактического.
3. Заимствовании формальным аргументом размера фактического.

4.4.1 Явное задание формы формального аргумента

Явное задание формы формального аргумента позволяет указать процедуре, что его ранг и размер, если нужно, могут отличаться от ранга и размера соответствующего фактического аргумента. Вспомним программу из пункта **4.3.2**, тестирующую внешнюю функцию **fsum1**:

```
program testsum; implicit none  ! Программа запоминает в первых n
integer, parameter :: nmax=1000 ! элементах вещественного вектора
real ar(nmax), fsum1, s         ! ar(nmax) первые n чисел
integer n, i                    ! натурального ряда, а затем вызывает
write(*,*) 'Введи (n) - число заполняемых элементов массива'
read (*,*) n; write(*,*) ' n=',n ! функцию fsum1, которая находит
do i=1,n; ar(i)=i; enddo         ! сумму первых n элементов вектора ar.
write(*,1000) (i,ar(i),i=1,n)
s=fsum1(ar,n); write(*,*) ' s=',s
 1000 format(1x,' Номер      Содержимое  '/(1x,i4,e20.7) )
end


function fsum1(a,n) result(s); implicit none ! Функция fsum1(a,n) находит
integer n, i; real a(n), s                   ! сумму первых n элементов
s=0.0; do i=1,n; s=s+a(i); enddo             ! одномерного массива (a).
end function fsum1
```

1. В главной программе массив **ar** состоит из **1000** элементов.
2. В функции **fsum1** формальный аргумент – массив **a** состоит из **n** элементов. При **n=5** функция *полагает* массив **ar** пятиэлементным.
3. Описание формального аргумента в виде **real a(n)** удобно, так как исходный текст функции не зависит от конкретного значения **n**.
4. Значение **n**, задаваемое в программе, вызывающей **fsum1**, не должно превышать числа элементов массива, заявленного в программе. Так при вводе **n=2000** получим **Segmentation fault**, а при **n=1001** — зацикливание.

Иногда выгодно указать процедуре, что **ранг** её формального аргумента (массива) отличается от **ранга** фактического. Пусть процедура должна найти сумму элементов **шестимерного** массива. Убедимся, что **одномерная индексация выгоднее шестимерной.**

Суммирование шестимерного массива **real a(10,10,10,10,10,10)** поручим функции **fsum3** с шестикратным циклом; одномерного — **fsum1**, а встроенная **sum** *разберётся* с размерностью по умолчанию.

```
program testsum3; implicit none           ! Программа подсчитывает kk
integer,parameter :: n=10, n6=n*n*n*n*n*n ! раз (kk вводится) сумму
real a (n,n,n,n,n,n) /1000000*1.0/        ! миллиона элементов шести-
real s1, s2, s3, fsum1, fsum3             ! мерного массива, вызывая
real t(2), r0, r1, r2, r3; integer k, kk  ! 1) fsum1(a,n6), полагающую
write(*,*) 'Сколько раз вычислим сумму?'  !    массив (a) одномерным;
read (*,*) kk; write(*,*) 'kk=', kk       ! 2) fsum3(a, n), полагающую
call etime(t,r0);                         !     массив (a) шестимерным;
do k=1,kk; s1=fsum1(a,n6); enddo;call etime(t,r1) !  3)   встроенную sum(a)
do k=1,kk; s2=fsum3(a,n ); enddo;call etime(t,r2) ! и находит время
do k=1,kk; s3=sum(a);      enddo;call etime(t,r3)             !   нужное
write(*,1000); write(*,'(a,3(f12.1))') 'Сумма',s1, s2, s3     ! для расчёта
write(*,'(a,3(f12.3))') 'Время', r1-r0, r2-r1, r3-r2          ! каждому из
 1000 format(1x,6x,'fsum1(a,n6)',2x,'fsum3(a,n)',3x,'sum(a)') !  способов.
end
function fsum3(a,n) result(s); implicit none ! Функция fsum3(a,n) находит
integer n, i1,i2,i3,i4,i5, i6                !  сумму  n**6  элементов
real a(n,n,n,n,n,n), s                       ! шестимерного массива (a),
s=0.0; do i1=1,n; do i2=1,n; do i3=1,n       ! используя шестикратный цикл.
       do i4=1,n; do i5=1,n; do i6=1,n
         s=s+a(i1,i2,i3,i4,i5,i6)
       enddo; enddo; enddo; enddo; enddo; enddo
end function fsum3
```

kk		fsum1(a,n6)		fsum3(a,n)		sum(b)	
	Сумма	1000000.0		1000000.0		1000000.0	
	Ключ	-O0	-OI	-O0	-O1	-O0	-O1
1	Время	0.015	0.007	0.068	0.052	0.011	0.003
2	Время	0.024	0.010	0.127	0.105	0.021	0.004
10	Время	0.107	0.030	0.645	0.524	0.105	0.024
100	Время	1.027	0.256	6.274	5.208	1.052	0.235

Для ключей оптимизации **-O0** и **-O1** экономия времени при замене многомерного массива одномерным налицо. Меньше всего времени на работу требуется, естественно, встроенной функции **sum**; внешней же **fsum3**, при работе с шестимерным массивом требуется в 5-6 раз больше времени при ключе оптимизации **-O0** и практически на порядок больше при ключе оптимизации **-O1**.

4.4.2 Заимствование формы фактического аргумента

Имеется ввиду, что форма массива, служащего формальным аргументом **заимствуется** у фактического. Признак **заимствования** — после двоеточия не указывается верхняя граница формального параметра.

Выгода: **из списка формальных аргументов можно исключить граничные индексы массива** (а, значит, и возможную ошибку их передачи при оформлении вызова процедуры).

Внутри процедуры граничные индексы массива (фактического аргумента) находим, используя элементные (т.е. способные возвращать в качестве результата массив) функции **lbound** и **ubound**, получающие границы измерений массива (нижние и верхние соответственно).

1. ```
   program testsum4; implicit none        ! Программа демонстрирует вызов
   interface
   function fsum4(a) result(s); real :: a(:), s; end function fsum4
   end interface
   integer,parameter  :: na=1000, nb=2000  ! функции fsum4(a) с формальным
   real a (na), b(nb), sa, sb; integer i   ! аргументом - массивом, который
   do i=1,na; a(i)=i; enddo;               !  по описанию a(:) перенимает
   do i=1,nb; b(i)=i; enddo;               ! форму у фактического аргумента.
   sa=fsum4(a); write(*,*) '  Сумма элементов (a) =', sa
   sb=fsum4(b); write(*,*) '  Сумма элементов (b) =', sb
   write(*,*) ' lbound(a)=',lbound(a), 'ubound(a)=',ubound(a)
   write(*,*) ' lbound(b)=',lbound(b), 'ubound(b)=',ubound(b)
   end
   function fsum4(a) result(s); implicit none ! Функция fsum4(a) находит сумму
   integer  n1(1), n2(1), i; real :: a(:), s  ! элементов одномерного  массива
   n1=lbound(a); n2=ubound(a)                 ! A, вычисляя граничные индексы
   write(*,1000) n1(1), n2(1)                 ! посредством встроенных
   s=0.0; do i=n1(1),n2(1); s=s+a(i); enddo   ! функций lbound и ubound
    1000 format(1x,' n1(1)=',i5, '  n2(1)=',i5,$)
   end
   ```

   Описание **a(:)** удобно, если нижний индекс фактического аргумента равен **1**, иначе **lbound** и **ubound** получат индексы массива лишь согласованного с фактическим, начав индексацию с **1**.

   При использовании процедур с аргументами-массивами перенимающими форму наличие явного интерфейса **необходимо**, в то время как, например, **testsum3** обходилась по старинке неявным.

   В современном ФОРТРАНе целесообразно в том или ином виде ВСЕГДА указывать интерфейс процедур.
   ```

2. Для заимствования формы массива и его индексации при нижней границе фактического аргумента отличной от единицы используем конструкцию

имя_массива (нижний_индекс :).

```
program testsum5; implicit none
interface
  function fsum5(x) result(s); real s; real, dimension (-3:) :: x
  end function fsum5
endinterface
real  a(-3:10),  b(-3:5), c(-3:0),  d(-3:-1)
real sa       , sb      , sc     , sd
integer i
write(*,100) '  main: lbound(a) =', lbound(a), ' ubound(a)=', ubound(a)
write(*,100) '        lbound(b) =', lbound(b), ' ubound(b)=', ubound(b)
write(*,100) '        lbound(c) =', lbound(c), ' ubound(c)=', ubound(c)
write(*,100) '        lbound(d) =', lbound(d), ' ubound(d)=', ubound(d)
a(-3:10)=(/ (i,i=-3,10) /)      ! Инициализация
b(-3: 5)=(/ (i,i=-3, 5) /)      !    векторов
c(-3: 0)=(/ (i,i=-3, 0) /)      !        посредством
d(-3:-1)=(/ (i,i=-3,-1) /)      !           конструктора  массива
sa=fsum5(a); sb=fsum5(b); sc=fsum5(c); sd=fsum5(d)
write(*,*) ' sa=',sa,' sb=',sb,' sc=',sc,' sd=',sd
100 format(a,i3,a,i3)
end program testsum5

function fsum5(x) result(s)  ! Функция fsum5(x), находя сумму элементов
implicit none                ! массива (x), заимствующего у фактического
integer  i                   ! аргумента с индексом начального элемента
real x(-3:), s               ! равным -3 ФОРМУ, для проверки правильности
integer n1(1), n2(1)         ! заимствования индексации выводит принятые
n1=lbound(x);                ! значения нижней и верхней границ
n2=ubound(x)
write(*,*) ' fsum5: lbound(x) =',lbound(x),' ubound(x) =',ubound(x)
s=0.0
do i=n1(1), n2(1); s=s+x(i); enddo
end function fsum5

main: lbound(a) = -3 ubound(a)= 10  !  Результат пропуска исполнимого кода,
      lbound(b) = -3 ubound(b)=  5  !  полученного на gfortran из
      lbound(c) = -3 ubound(c)=  0  !  gcc version 4.5.0
      lbound(d) = -3 ubound(d)= -1
fsum5: lbound(x) =          -3  ubound(x) =          10
fsum5: lbound(x) =          -3  ubound(x) =           5
fsum5: lbound(x) =          -3  ubound(x) =           0
fsum5: lbound(x) =          -3  ubound(x) =          -1
sa=   49.000000    sb=   9.0000000    sc=  -6.0000000    sd=  -6.0000000
```

3. Если индексация фактического аргумента отличается от индексации соответствующего формального, то в программе, которая вызывает процедуру, **необходимо** сообщать **явный интерфейс** процедуры. Без наличия явного интерфейса процедуры **fsum6** (при замене индексации формального аргумента с **x(-3:)** на **x(-2:)**) получаем:

```
program testsum6; implicit none; integer i
real  a(-3:10), b(-3:5), c(-3:0), d(-3:-1), sa, sb, sc, sd
write(*,100) '  main: lbound(a) =', lbound(a), ' ubound(a)=', ubound(a)
write(*,100) '        lbound(b) =', lbound(b), ' ubound(b)=', ubound(b)
write(*,100) '        lbound(c) =', lbound(c), ' ubound(c)=', ubound(c)
write(*,100) '        lbound(d) =', lbound(d), ' ubound(d)=', ubound(d)
a(-3:10)=(/ (i,i=-3,10) /);       c(-3: 0)=(/ (i,i=-3, 0) /)
b(-3: 5)=(/ (i,i=-3, 5) /);       d(-3:-1)=(/ (i,i=-3,-1) /)
sa=fsum6(a); sb=fsum6(b); sc=fsum6(c); sd=fsum6(d)
write(*,*) ' sa=',sa,' sb=',sb,' sc=',sc,' sd=',sd
  100 format(a,i3,a,i3)
end program testsum6
```

```
function fsum6(x) result(s) ! Функция fsum6(x) воспринимает фактический
implicit none;              ! аргумент, от которого формальный заимствует
integer  i                  ! ФОРМУ, индексированным не от -3, как указано
real x(-2:), s              ! в главной программе, а от -2).
integer n1(1), n2(1)        ! При отсутствии в последней явного описания
n1=lbound(x)                ! интерфейса вызов ubound внутри функции
n2=ubound(x)                ! завершается аварийно.
write(*,*) ' fsum6: lbound(x) =', lbound(x),' ubound(x) =', ubound(x)
s=0.0; do i=n1(1), n2(1);  s=s+x(i); enddo
end function fsum6
```

```
 gfortran tstfsum6.f95 fsum6.f95      # Компиляция завершится с кодом 0.
 ./a.out                              # Но запуск исполнимого кода
  main: lbound(a) = -3 ubound(a)= 10  # при вызове ubound завершится
        lbound(b) = -3 ubound(b)=  5  # аварийно
        lbound(c) = -3 ubound(c)=  0
        lbound(d) = -3 ubound(d)= -1
 fsum5: lbound(x) =          -2  ubound(x) =     8388606
 Ошибка сегментирования
```

```
  main: lbound(a) = -3 ubound(a)= 10  #  Результат пропуска исполнимого
        lbound(b) = -3 ubound(b)=  5  #  кода после указания явного
        lbound(c) = -3 ubound(c)=  0  #  интерфейса
        lbound(d) = -3 ubound(d)= -1
  fsum5: lbound(x) =          -2  ubound(x) =          11
  fsum5: lbound(x) =          -2  ubound(x) =           6
  fsum5: lbound(x) =          -2  ubound(x) =           1
  fsum5: lbound(x) =          -2  ubound(x) =           0
  sa= 49.000000   sb= 9.0000000   sc= -6.0000000   sd= -6.0000000
```

4.4.3 Заимствование размера фактического аргумента

Имеется ввиду, что массив, являющийся формальным параметром, заимствует у фактического параметра только его размер, при возможном отличии формы.

Признаком, определяющим рассматриваемый способ — **звездочка** при описании размера последнего измерения формального параметра.

Этот способ сохранён для совместимости с ФОРТРАНом-77 (см. [2]). Пользоваться им не рекомендуется (см. [4, 2]).

4.4.4 Выводы

1. При описании массива в качестве формального параметра внешней процедуры используем явное задание его формы, добавляя (по необходимости) в список параметров переменные, хранящие его заявленный и/или требуемый размеры.

2. Явное задание формы (конфигурации) массива, служащим формальным парметром процедуры, работает и в старых, и в современных версиях ФОРТРАНа.

3. Объявление массива формальным параметром, заимствующим форму фактического, выгодно

   (a) либо при описании начального индекса по каждому измерению **единичным** (по умолчанию);

   (b) либо при безукоризненной и удобной для пользователя работе функций **lbound** и **ubound**, которая в старых версиях компилятора **gfortran** вызывала определённые нарекания.

4. Стараемся не использовать режим заимствования размера.

4.5 Массив в СИ как формальный аргумент функции

В первом семестре (см. часть **I (3.3.7)** сообщалось, что передача аргумента функции в ФОРТРАНе происходит по адресу, а в СИ – по значению (там же проводилось обоснование выгоды передачи по значению).

Время на передачу одного значения простого типа (**int**, **float** или **double**) в ячейку формального аргумента, невелико. Копирование же всех элементов массива требует гораздо больше времени. Поэтому при использовании массива в качестве фактического аргумента передаётся лишь указатель на его начальный элемент.

Содержимое **i**-го элемента массива **a** доступно или по записи **a[i]**, или (см. часть **II (4.2.2)**) по записи ***(a+i)**, обозначающей операцию разыменования указателя на элемент **a[i]**. В соответствие со сказанным описание одномерного массива в качестве формального аргумента можно выполнить и в терминах конструктора массива, и на языке указателей.

1. Описание без указания количества элементов в массиве:

```
#include <stdio.h>
void c0(int [], int);
int main()
{ int i, n=100, a[100];         // В main значение переменной n равно 100.
  printf(" main 1: n=%d\n",n);  // Убедимся в этом до вызова функции.
  c0(a,n);                      // Функция не изменяет фактический аргумент
  printf(" main 2: n=%d\n", n); // n, хотя формальный меняется (n=5).
  for (i=0;i<=4;i++)            // Убедимся в этом после вызова функции.
  printf(" %d  %d\n", i, a[i]); // В отличие от n содержимое пяти элементов
  return 0; }                   // (a) изменено. Почему?
void c0(int a[],int n)          // По значению переданы: адрес начального
{ int i;                        // элемента массива и значение  (n)
 printf(" c0 1: n=%d\n",n);       // Вывод формального n при работе функции
  n=5; printf(" c0 2: n=%d\n",n); //  (до и после изменения n).
  for(i=0; i<=n-1; i++)
        a[i]=i+1;
}
```

```
 main 1: n=100     // Результат пропуска программы c0.c
 c0 1: n=100
 c0 2: n=5
 main 2: n=100
 0  1
 1  2
 2  3
 3  4
 4  5
```

2. Описание с указанием количества элементов в массиве:

```
#include <stdio.h>
void c2(int [100], int);  // Размер массива, указанный в конструкторе
int main()                // формального аргумента, НЕ ПЕРЕДАЁТСЯ
{ int na, a[100];         // внутрь функции.
  na=sizeof(a)/sizeof(a[0]);        // Здесь
  printf(" main:  na=%d\n",na);     //         na=100
  c2(a,na); return 0;
}                                   // Но внутри функции отношение
void c2(int a[100],int n)           //  sizeof(a)/sizeof(a[0])=1,
{ int nac2;                         // т.к. компилятор gcc отводит
  nac2=sizeof(a)/sizeof(a[0]);      // для хранения указателя 4 байта, т.е.
  printf(" c2 : nac2=%d\n", nac2);  // столько же, сколько и для хранения
  printf(" c2 :  n  =%d\n", n   ); }// одного элемента типа int.

 main:  na=100     // Как и предупреждалось, размер массива, вычисленный
 c2 : nac2=1       // внутри процедуры c2, равен единице, а переданный
 c2 :  n  =100     // через параметр равен хранимому в нем значению.
```

Так что, если размер массива, необходим внутри функции, то его нужно передать дополнительным параметром.

3. В качестве описателя одномерного массива **int []**, служащего формальным параметром, не менее элегантно выглядит и простое **int***.

```
#include <stdio.h>
void c1(int*, int);     // В СИ имя массива -- указатель на его начальный
int main()              // элемент. Поэтому при описании формального
{ int i, n=100, a[100]; // аргумента вместо int[] можно использовать int*
  printf(" main 1: n=%d\n", n);
       c1(a,n);
  printf(" main 2: n=%d\n", n);
  for (i=0;i<=4;i++) printf(" %d  %d\n", i, a[i]);  return 0;
}
void c1(int *a,int n)                  // Здесь описание int a[100]
{ int i;  printf(" c0 1: n=%d\n", n);  // возможно, но размер формального
  n=5;    printf(" c0 2: n=%d\n", n);  // аргумента (вектора a)
  for(i=0; i<=n-1; i++)     a[i]=i+1;} // в процедуру передан не будет.
```

4. Иногда описывать массив, указывая атрибуты его формы только через конструктор или указатель, неудобно. Часто выгодно сопоставить безымянному типу **int []** или **int*** синоним, отражающий проблемное назначение массива. Так, если одномерные массивы **a**, **b**, и **c** хранят коэффициенты полиномов, то может оказаться удобным их описание в виде

```
coef a, b, c;    // Здесь coef - имя, придуманное нами для обозначения
                 // типа указателя на одномерный массив.
```

Используя СИ-оператор **typedef**, нацеленный на определение подобных синонимов, программа вызова функции, суммирующей коэффициенты двух полиномов (при одинаковых степенях аргумента), и описание функции могут иметь вид:

```
#include <stdio.h>
typedef double coef[100] ;           // coef - синоним типа double [100]
void add(coef, coef, coef, int);     // описание прототипа (интерфейса) add
int main()
{ int i,n, nw=6;        coef a, b, c;
  for(i=0; i<=nw-1; i++)             // пример заполнения векторов a и b
    {  a[i]=i; *(b+i)=10*i; }        // обеими формами доступа к элементу
  n =sizeof(a)/sizeof(a[0]); printf(" n=%d\n",n);
  add(a,b,c, nw);
 for (i=0;i<=nw-1;i++) printf("%d  %le  %le  %le\n",i,a[i], b[i],c[i]);
 return 0;}
void add(coef a, coef b, coef c, int n)
{ int i;  printf(" add:  n=%d\n", n);
  for(i=0; i<=n-1; i++) c[i]=a[i]+b[i]; }
```

```
 n=100                                       // Результат работы программы
 add:  n=6                                   //                        c3.c
0  0.000000e+00  0.000000e+00  0.000000e+00
1  1.000000e+00  1.000000e+01  1.100000e+01
2  2.000000e+00  2.000000e+01  2.200000e+01
3  3.000000e+00  3.000000e+01  3.300000e+01
4  4.000000e+00  4.000000e+01  4.400000e+01
5  5.000000e+00  5.000000e+01  5.500000e+01
```

5. В пункте **4** главная программа и функция описаны в одном файле. Поэтому описание типа **coef** *видно* обеим. Если бы описание функции находилось в файле **add.c**, то

```
void add(coef a, coef b, coef c, int n)  // при компиляции gcc add.c -c
{ int i;  printf(" add:  n=%d\n", n);    //
  for(i=0; i<=n-1; i++) c[i]=a[i]+b[i];} // получили бы сообщения:

add.c:1: error: syntax error before "a"
add.c: In function ‘add’:
add.c:2: error: ‘n’ undeclared (first use in this function) <--=  что n не
add.c:2: error: (Each undeclared identifier is reported only once объявлено
add.c:2: error: for each function it appears in.)                 ошибка
add.c:3: error: ‘c’ undeclared (first use in this function)       наведена
add.c:3: error: ‘a’ undeclared (first use in this function) отсутствием
add.c:3: error: ‘b’ undeclared (first use in this function) описания типа coef
```

6. Обычно описание прототипов функций и имен синонимов типов данных помещают в **заголовочный файл**, подсоединяемый к нужным файлам директивой #**include** (см. част**I (1.6.2)**), например:

```
typedef double coef [100];                    // Файл mytype.h

#include <stdio.h>                            // Файл tstadd1.c
#include "mytype.h"
void add1(coef, coef, coef, int);
int main()
{ int i,n, nw; coef a, b, c;
  nw=6;
  for(i=0; i<=nw-1; i++) {  a[i]=i; b[i]=10*i; }
  n =sizeof(a)/sizeof(a[0]);
  printf(" n=%d\n",n);      add1(a,b,c, nw);
  for (i=0;i<=nw-1;i++)
  printf("%d  %le  %le  %le\n",i,a[i], b[i],c[i]);
  return 0; }


#include "mytype.h"                           // Файл add1.c
void add1(coef a, coef b, coef c, int n)
{ int i; // printf(" add1:  n=%d\n", n);
  for(i=0; i<=n-1; i++) c[i]=a[i]+b[i];
}
```

7. Если при описании типа **coef** не указать (по невнимательности) в операторе **typedef** тип значения, на которое должен ссылаться указатель типа **coef** (т.е. вместо **typedef double coef[100];** написать **typedef coef[100];**, то компилятор **gcc** не выдал бы никаких ошибок, но результат мог бы иметь вид:

```
n=100                                              // СИ при описании функции
0  0.000000e+00  -5.412876e+303  1.484567e-313    // без указания ее типа
1  2.121996e-313  -5.412876e+303  1.484567e-313   // полагает, что её тип
2  4.243992e-313  -5.412876e+303  1.484567e-313   // int. Видимо, это же
3  6.365987e-313  -5.412876e+303  1.484567e-313   // относится и к описанию
4  8.487983e-313  -5.412876e+303  1.484567e-313   // типа массива, т.к. при
5  1.060998e-312  -5.412876e+303  1.484567e-313   // замене %le на %d
                                                   // получаем нужный ответ.
```

g++-компиляция файла **tstadd1.c** приводит к сообщению:

```
mytype2.h:1: error: ISO C++ forbids declaration of ‘coef’ with no type
```

4.6 О чем узнали из четвёртой главы (второй семестр).

1. **Массив** – именованный набор конечного количества нумерованных элементов **одинакового типа**, располагаемых в оперативной памяти непосредственно друг за другом в порядке их следования.
2. Массив называют **одномерным** (или вектором), если в его описании указано о распределении его элементов вдоль одного измерения.
3. Массив называют **двумерным** (или матрицей), если в его описании указано, что все элементы распределены вдоль двух измерений.
4. Для доступа к элементу вектора нужен порядковый номер элемента.
5. Для выборки элемента двумерного массива (матрицы) нужно задать два числа – номер строки и номер столбца.
6. Номер элемента часто называют **индексом**.
7. ФОРТРАН-индекс начального элемента вектора по умолчанию =**1**. СИ-индекс начального элемента вектора всегда =**0**. Поэтому СИ-индекс последнего элемента вектора на **1** меньше числа элементов.
8. В ФОРТРАНе можно явно задать диапазоны изменения индексов.
9. Размер статического массива задается при его описании константой.
10. Размер динамического определяется при работе программы
11. При описании статического массива по умолчанию всегда указывается количество элементов по соответствующему измерению.
12. Имя СИ-массива — указатель на его начальный элемент.
13. Конструктор массива в СИ обозначается **[]**.
14. Если формальный параметром СИ-функции является массив, то при вызове ей передаётся адрес начального элемента соответствующего фактического аргумента.
15. В СИ обозначения **a[i]** и ***(a+i)** – синтаксически эквивалентны.
16. В современном ФОРТРАНЕ есть много встроенных функций, нацеленных на работу с массивами. Одна из них функция **sum**, находящая сумму элементов массива.

4.7 Четвёртое домашнее задание (второй семестр).

- **Решения дать на ФОРТРАНе-95 и С (fprintf, fscanf)**.
- ФОРТРАН-процедуры из задач под номерами

  с **1**-го по **4**-ый разместить в модуле **poly**.

  с **5** по **6** разместить в модуле **binrad**.
- Главная программа должна подсоединять модуль, вводить исходные данные из файла, вызывать нужную процедуру и выводить результат в файл вывода.
- **Инициировать запуск каждой программы должен make-файл**.

1. Разработать процедуру, которая по заданным **n+1** коэффициентам полинома **n**-й степени и его аргументу **x** вычисляет значение полинома **P(x)**.
2. Разработать процедуру, которая по заданным **n+1** коэффициентам полинома **n**-й степени и его аргументу вычисляет **два** значения полинома (**P(x)** и **P(-x)**) практически за время расчёта одного значения **P(x)** ([14]).
3. Первые **n** элементов одномерного массива содержат **n** целых чисел из диапазона **[-16,16]**. Разработать процедуру, которая подсчитывает количество встреч в массиве каждого из чисел соответственно.
4. Разработать процедуру, которая по заданному целому находит его двоичное представление, помещая очередную двоичную цифру в соответствующий элемент вектора (для отрицательных чисел представление должно быть получено в дополнительном коде).
5. Разработать процедуру, которая набор нулей и единиц, размещённых в элементах вектора переводит в значение типа **integer**
6. Разработать процедуру, преобразующую набор двоичных цифр, размещённых в элементах одномерного массива, в набор восьмеричных цифр так, чтобы целые значения, изображаемые указанными наборами, численно были равны.

7. Значение интеграла по промежутку **[a,b]** от подинтегральной функции **f(x)** приближенно вычисляется через квадратурную сумму формулы **средних прямоугольников** (подробнее см. учебное пособие «ПРАКТИКА ПРОГРАММИРОВАНИЯ» (приближённое вычисление интегралов) (сайт НИАИ им. В.В. Соболева WWW-ресурсы)):

$$\tilde{\mathbf{S}} = \mathbf{h} \cdot \sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{f}(\mathbf{x_i}), \quad \text{где} \quad \mathbf{h} = \frac{\mathbf{b-a}}{\mathbf{n}}; \quad \mathbf{x_i} = \mathbf{a} + \mathbf{h} \cdot (\mathbf{i} - \frac{\mathbf{1}}{\mathbf{2}}); \quad \mathbf{i} \in [\mathbf{1}, \mathbf{n}].$$

n – количество промежутков равномерного дробления отрезка **[a,b]**.

Разработать функцию **rectan(y,a,b,n)**, которая по набору из **n** значений подинтегральной функции, хранящихся в первых **n** элементах одномерного массива **y** вычисляет значение квадратурной суммы формулы средних прямоугольников. Описание функции **rectan** поместить в модуль **quarda**.

5 Одномерный динамический массив.

До ФОРТРАНа-90 динамические массивы в ФОРТРАНе не допускались.

5.1 ФОРТРАН

Современный стандарт ФОРТРАНа допускает несколько механизмов динамического выделения памяти под массив [2]:

1. **Автоматические** – массивы, память под которые выделяется при входе в процедуру и освобождается при выходе из неё.
2. **Размещаемые** – массивы объявленной фиксированной размерности, но с границами, вычисляемыми при выполнении программы.
3. **Создаваемые** – массивы, создаваемые без их явного объявления.

5.1.1 Автоматические массивы

Нередко внутри процедуры требуется локальный рабочий массив, который не хочется включать в список её формальных параметров. В то же время его размер зависит от формального параметра. Например, следуя [2, 4] рассмотрим процедуру обмена содержимым двух векторов:

```
subroutine obmen(u,v) ! Здесь w - локальный внутренний массив
implicit none         ! процедуры obmen. В ФОРТРАНе-77 начало
real u(:), v(:)       ! описания процедуры выглядело бы так:
real w(size(u))       ! subroutine obmen(u,v,w,n) real u(n), v(n), w(n)
w=u; u=v; v=w         ! что удвоило бы число аргументов
end
```

Ясно, что выгода от использования автоматических массивов налицо.

```
program testobmen; implicit none
interface
  subroutine obmen(u, v);real u(:), v(:); end subroutine obmen
end interface
integer j; real u(5) / 1.7, 2.7, 3.7, 4.7, 5.7 /,&
&               v(5) / 1.3, 2.3, 3.3, 4.3, 5.3 /
call obmen(u,v)
write(*,'(i3,f10.3,f10.3)') (j,u(j),v(j),j=1,5)
end program testobmen
```

1. Автоматические массивы не должны быть формальными аргументами или элементами общих областей (**COMMON**-блоков).

2. **size** — встроенная справочная функция, получающая (в случае фактического аргумента-массива) общее число его элементов.

3. Автоматические массивы *видны* лишь той процедуре, в которой описаны.

4. Их нельзя использовать в операторах **data**, **namelist**, **equivalence**.

5. Интерфейсный блок в программе, вызывающей процедуру с формальным аргументом-массивом, заимствующим форму, обязателен. Так, если закомментировать описание интерфейса

```
program testobmen; implicit none
!      interface
!       subroutine obmen(u, v);real u(:), v(:); end subroutine obmen
!      end interface
integer j; real u(5) / 1.7, 2.7, 3.7, 4.7, 5.7 /,&
&               v(5) / 1.3, 2.3, 3.3, 4.3, 5.3 /
call obmen(u,v)
write(*,'(i3,f10.3,f10.3)') (j,u(j),v(j),j=1,5)
end program testobmen
```

   то результат работы исполнимого файла оказался бы следующим:

```
Program received signal SIGSEGV: Segmentation fault - invalid memory
reference. Backtrace for this error:
#0 0x7FB4012E4467
#1 0x7FB4012E4AAE
#2 0x7FB4007EB65F
#3 0x400BC9 in obmen_
#4 0x4008DE in MAIN__ at tstobmen.f90:?
Ошибка сегментирования (core dumped)
```

6.
```
subroutine obmen1(u,v, n) ! В gfortrane для указания числа
real u(n), v(n)           ! элементов автоматического массива w
real w(n)                 ! можно использовать не только функцию
w=u; u=v; v=w             ! size, но и формальный аргумент n.
end subroutine obmen1     ! Какова может быть критика исходного текста?

program testobmen1; implicit none       !     Результат работы:
integer j, n                             !    1     1.300     1.700
real u(5) / 1.7, 2.7, 3.7, 4.7, 5.7 /,& !    2     2.300     2.700
&    v(5) / 1.3, 2.3, 3.3, 4.3, 5.3 /    !    3     3.300     3.700
n=3                                      ! Почему эта программу проходит
call obmen1(u,v,n)                       ! без явного описания интерфейса?
write(*,'(i3,f10.3,f10.3)') (j,u(j),v(j),j=1,n)
end program testobmen1
```

5.1.2 Размещаемые массивы

Размещаемые массивы объявляются с атрибутом **allocatable** (размещаемый). Их создание и уничтожение выполняется операторами **allocate** и **deallocate** соответственно, в отличие от автоматических массивов, которые создаются и уничтожаются при вызове процедуры автоматически.

```
program allocat0; implicit none
real(8), allocatable, dimension(:) :: a ! объявление размещаемого массива
integer n, n1, n2, ier, k
write(*,*) ' введи размер массива:'
read (*,*) n; write(*,*) ' n=', n       ! Значение n можно и вычислить.
allocate (a(n), stat=ier)               ! Выделяем память под массив.
if (ier.ne.0) stop 'Не могу выделить!'
do k=1,n; a(k)=dsqrt(dfloat(k)); enddo  ! Моделирование расчета.
write(*,1001) (k, a(k), k=1,n)          ! Вывод результата.
deallocate(a, stat=ier)                 ! Высвобождение памяти.
if (ier/=0) stop 'Не могу высвободить!'
write(*,*) 'введи граничные индексы:'
read (*,*) n1, n2                       ! Значения n1 и n2  программа
write(*,*) ' n1=', n1, ' n2=',n2        ! может и вычислить.
allocate(a(n1:n2), stat=ier)            ! Выделяем память под массив.
if (ier.ne.0) stop 'Не могу выделить!'
write(*,*) allocated(a)
do k=n1,n2; a(k)=dsqrt(dabs(dfloat(k))); enddo ! Моделирование расчёта.
write(*,1002) (k, a(k), a(k)**2, k=n1,n2)      ! Вывод результата.
deallocate(a, stat=ier)                        ! Высвобождение памяти.
if (ier/=0)  stop  'Не могу высвободить!'
if (.not.allocated(a)) write(*,*) 'Массив a не размещен!!!'
1001 format(1x,i3,3x, d23.16,3x); 1002 format(1x,i3,3x, d23.16,3x, d23.16)
end program allocat0
```

1. **2-я строка.** Объявляется **размещаемый** массив **a**, который должен обладать следующими **атрибутами**:

   элементы типа **real(8)**; размещаемый **allocatable**; одномерный **dimension (:)** (одно двоеточие в круглых скобках).

   Атрибуты разделяются запятыми.

2. **6-я строка.** Оператор **allocate a(n)** фрахтует память под необходимые **n** элементов (индекс начального элемента массива равен единице, т.е. определяется умолчанием начального индекса). Общий вид оператора **allocate**:

$$\mathbf{allocate}(\mathbf{a}, \mathbf{b}, \cdots, \mathbf{z}[, \mathbf{stat} = \text{имя\_переменной}])$$

Здесь **a**, **b**, $\cdots$, **z** — имена **размещаемых массивов** или **указателей**. После имени массива в круглых скобках указываются нижний и верхний индексы (по соответствующему измерению), разделенные двоеточием. Спецификатор **stat** – необязательный. Он присваивает указанной переменной код причины завершения **allocate**. Если код равен нулю, то размещение успешно (иначе – безуспешно).

3. **строки 10 и 20.** Оператор **deallocate**. Общая форма его записи подобна форме записи оператора **allocate**.

4. **15-я строка.** Оператор **allocate a(n1:n2)** и выделяет память, и явно модифицирует диапазон индексов размещаемого массива.

5. **17-я строка.** Вызывается функция **allocated**, которая возвращает **.true.**, если массив размещен, и **.false.**, если нет (см. 22 строку).

6. После компиляции и пропуска программы **allocat0** получим:

```
 введи размер массива:
 n=           2
  1    0.1000000000000000D+01
  2    0.1414213562373095D+01
введи граничные индексы:
 n1=          -2  n2=           2
T
 -2    0.1414213562373095D+01    0.2000000000000000D+01
 -1    0.1000000000000000D+01    0.1000000000000000D+01
  0    0.0000000000000000D+00    0.0000000000000000D+00
  1    0.1000000000000000D+01    0.1000000000000000D+01
  2    0.1414213562373095D+01    0.2000000000000000D+01
Массив a не размещен!!!
```

7. Изменение диапазона индексов при описании массива может потребовать соответствующих изменений в операторах, использующих граничные индексы.

8. Размещаемые массивы, как и автоматические, не могут использоваться в операторах **common**, **equivalence** и **namelist**, но в отличие от автоматических могут иметь атрибут **save**.

9. **Динамическое создаваемые** массивы (см. [2] **7.8**).

5.2 СИ.

Для динамического распределения памяти используем функции **malloc** и **free** (последняя освобождает память). Их описания можно посмотреть посредством **man malloc** и **man free**.

void *malloc(размер памяти в байтах); – выделяет память нужного размера. При успешном выделении возвращает указатель типа **void** на первый свободный байт выделенной памяти (в противном случае возвращается **NULL**). Например,

```
#include <stdio.h>     // Потребуются scanf и printf.
#include <stdlib.h>    // Потребуются malloc и free !!!
int main()
{ double *a;   // a - указатель на ячейку для значения типа double
  int n, k;    // (место под указатель выделено, но сам он не определен).
 printf(" введи размер массива:\n");     // Ввод и контрольная печать
 scanf ("%d", &n); printf(" n=%d\n", n); //          размера массива.
 a=(double*) malloc(n*sizeof(double));   // Поиск адреса, с которого можно
                                         // разместить n значений типа double.
 // Согласно описанию прототипа malloc возвращает указатель на значение типа
 // void. Нам же нужен указатель на значение типа double. Поэтому, используем
 // операцию (double*) приведения указателя (void*), выработанного malloc, к
 // типу  указателя на double*.
  if (!a) {printf("Не могу выделить!\n"); return 1;}     // Попытка неудачна.
  for (k=0;k<=n-1;k++)                      // При удаче работаем с массивом.
    { a[k]=sqrt((double)k);                 // Имеем право именовать элемент
     printf("  %d   %20.13le %20.13le \n",  // массива: и *(a+k), и a[k].
                   k,*(a+k), a[k]*a[k]); }
  free(a);  // После free(a) память, адресуемая a, отдана операционной системе.
  n=n/2;   printf(" n=%d\n", n);             // Еще один пример формировки
  a=(double*) malloc(n*sizeof(double));      // размещаемого массива:
  if (!a) {printf("Не могу выделить!\n"); return 1;}
  for (k=0;k<=n-1;k++) {a[k]=(double)k*k; printf("  %d   %20.13le\n", k,a[k]);}
  free(a); return 0; }
```

```
 введи требуемый размер массива:  //  malloc - НЕ ИНИЦИАЛИЗИРУЕТ
 5                                //           выделяемую память !!!
  n=5                             //            (см. man malloc)
   0    0.0000000000000e+00  0.0000000000000e+00    // Аналогично malloc
   1    1.0000000000000e+00  1.0000000000000e+00    // работает  и calloc.
   2    1.4142135623731e+00  2.0000000000000e+00    //
   3    1.7320508075689e+00  3.0000000000000e+00    // calloc ОБНУЛЯЕТ !!!
   4    2.0000000000000e+00  4.0000000000000e+00    // выделяемую память.
  n=3                                              // У calloc два аргумента:
    0    0.0000000000000e+00                       //  число элементов в массиве
    1    1.0000000000000e+00                       //  и
    2    4.0000000000000e+00                       //  число байт на один элемент.
```

5.3 О чем узнали из пятой главы? (второй семестр)

1. Современный ФОРТРАН предоставляют возможность использования **динамических массивов**.
2. **Динамический массив** – это массив, память под который выделяется в процессе работы исполнимого кода (не при компиляции).
3. Размер динамического массива задаётся при работе программы.
4. В ФОРТРАНЕ-95 существуют три вида динамических массивов:

   автоматические, **размещаемые** и **создаваемые**.

5. **Автоматические** массивы локальны по отношению к процедурам, использующим их, создаются при вызове процедуры, в которой описаны, и уничтожаются при выходе из неё **автоматически**.
6. Размер **автоматического** массива можно задать и константой, и переменной (глобальной или служащей формальным аргументом), и посредством функции **size**, в качестве аргумента которой выступает другой массив, известного размера.
7. **Автоматический** массив не инициализируем оператором **data** и операторами описания типа, не используем в операторах **namelist** и **equivalence**, и не снабжаем атрибутом **save**.
8. **Размещаемые** массивы определяются атрибутом **allocatable** (при описании) с указанием только ранга массива.
9. Распределение памяти под **размещаемый** массив осуществляется оператором размещения **allocate**, который требует задания размеров массива или его граничных индексов.
10. **deallocate** – оператор освобождения памяти, занимаемой размещаемым массивом.
11. **Автоматические** и **размещаемые** массивы не должны быть формальными аргументами.
12. **Автоматически размещаемые** массивы реализуются через **стек**.

13. **Динамически размещаемые** массивы реализуются через динамическую область **“куча”**.

14. В отличие от **автоматических** массивов, исчезающих после завершения работы процедуры, динамически **размещаемые** локальные массивы могут иметь атрибут **save** и тем самым сохранять свой статус при последующих обращениях к процедуре.

15. Если размещаемый массив не имеет атрибута **save**, то перед выходом из процедуры целесообразно явно освободить занимаемую ими память посредством оператора **deallocate**.

16. **Создаваемые** массивы — массивы, создаваемые динамически, без явного использования операторов описания массивов.

17. **Создаваемые** объекты (в том числе и массивы) не имеют имён так, что ссылаться на них можно только через **указатели**, которые содержат адреса созданных объектов, установленные операторами **allocate** и **deallocate**

18. В **C** для динамической работы с памятью используются функции **malloc** и **free**. Их прототипы **void *malloc(size_t size);** и **void free(void *ptr);** (описаны в **stdlib.h**; см. **man malloc**).

19. Вызов **malloc(размер_памяти_в_байтах);** – нацелен на выделение памяти нужного размера. При успешном выделении через имя функции возвращается указатель типа **void** на первый свободный байт выделенной памяти (иначе возвращается **NULL**). Память, выделенная функцией **malloc** не инициализируется.

20. Для выделения памяти под указатель на данное не типа **void** необходимо посредством операции приведения типа указатель, найденный **malloc**, привести к требуемому, например,

    (double*) malloc(размер_памяти_в_байтах);

21. **free(указатель_на_данное_типа_void)** освобождает область памяти, адресуемую указателем, полученную ранее ранее посредством **malloc()**.

5.4 Пятое домашнее задание (второй семестр).

Решения задач оформить на ФОРТРАНе с использованием **размещаемого массива**

- ФОРТРАН-решения задач с **1** по **7** должны подключать модуль **sort**, содержащий описания всех используемых в них процедур.
- ФОРТРАН-решения задач с **8** по **9** должны подключать модуль **quadra** (из предыдущего домашнего задания с описанием функции **rectan**), дополненный функциями **trap** (задача 8) и **sim** (задача 9).
- **Инициировать запуск каждой программы должен make-файл**, который наряду с целями компоновки исполнимого кода, получения объектных кодов единиц компиляции и запуска исполнимого кода, содержит и цель вывода результата из файла на экран.

1. Разработать функцию, определяющую число нечётных элементов вектора целого типа, и продемонстрировать её работоспособность на тестовых примерах (все элементы чётные, один элемент нечётный, три нечётных элемента, все элементы нечётные).
2. Разработать функцию и подпрограмму, которые объединяют два вектора в один с чередованием элементов исходных векторов целого типа, и продемонстрировать её работоспособность на тестовых примерах (в обоих векторах одинаковое количество элементов, больше элементов в первом векторе, больше элементов во втором векторе).
3. Разработать нерекурсивную и рекурсивную функции инвертирования вектора (перестановки элементов в обратном порядке).
4. Разработать функцию циклического сдвига элементов вектора, т.е. при сдвиге вправо содержимое **k**-го элемента переносится в **k+1**-й элемент (**k=1, ... , n**), причём содержимое **последнего** элемента переносится в **первый**; а при сдвиге влево содержимое **k**-го переносится в **k-1**-й для (**k=2, ... , n**), причём содержимое **первого** элемента переносится в **последний**.
5. Разработать функцию проверки вектора на упорядоченность.

6. Разработать нерекурсивную и рекурсивную функции поиска образца в упорядоченном по величине элементов векторе.

7. Разработать функцию и процедуру объединения двух упорядоченных массивов в один с сохранением упорядоченности.

8. Значение интеграла по промежутку **[a,b]** от подинтегральной функции **f(x)** приближенно вычисляется по формуле трапеций:

$$\tilde{\mathbf{S}} = \mathbf{h} \cdot \Big[\frac{\mathbf{f(a)} + \mathbf{f(b)}}{\mathbf{2}} + \sum_{\mathbf{i=2}}^{\mathbf{n}} \mathbf{f(x_i)}\Big], \quad \text{где}$$

$$\mathbf{h} = \frac{\mathbf{b-a}}{\mathbf{n}}; \qquad \mathbf{x_i = a + (i-1)h} \qquad \mathbf{i = 2, \cdots n}; \quad \mathbf{x_1 \equiv a} \;\; , \;\; \mathbf{x_{n+1} \equiv b}.$$

n – количество промежутков равномерного дробления отрезка **[a,b]**.

Разработать функцию **trap(y,a,b,n)**, которая по набору из **n+1** значений подинтегральной функции, хранящихся в первых **n+1** элементах одномерного массива **y** вычисляет значение квадратурной суммы формулы трапеций. Описание функции **trap** поместить в модуль **quarda** (тот же самый, в который помещена и процедура **rectan** из предыдущего домашнего задания).

Главная программа должна подсоединять соответствующий модуль, вводить исходные данные из файла, вызывать нужную процедуру и выводить результат +вместе с таблицей подинтегральной функции так, чтобы, получить её график, активируя цель **make plot**.

9. Для расчёта интегралов наряду с формулами прямоугольников и трапеций широко используется **составная формула Симпсона**:

$$\tilde{\mathbf{S}} = \frac{\mathbf{h}}{\mathbf{3}} \cdot [\mathbf{f(a)} + \mathbf{f(b)} + \mathbf{4} \cdot \mathbf{S_1} + \mathbf{2} \cdot \mathbf{S_2}]$$

$$\mathbf{S_1} = \mathbf{f_2} + \mathbf{f_4} + \cdots + \mathbf{f_{n=2m}} \;\; , \;\; \mathbf{S_2} = \mathbf{f_3} + \mathbf{f_5} + \cdots + \mathbf{f_{n-1}}.$$

$$\mathbf{f_i = f(x_i)} \;\; \mathbf{x_i = a + h \cdot (i-1)}, \;\; \mathbf{i = 1, 2, \cdots, (n = 2m), n+1}, \;\; \mathbf{h} = \frac{\mathbf{b-a}}{\mathbf{n}}$$

причём $\mathbf{n = 2 \cdot m}$ – число промежутков дробления – обязательно чётное. Учитывая требования к решению предыдущей задачи, разработать функцию **sim(y,a,b,n)**.

6 Операции ФОРТРАНА-95 над массивами.

В ФОРТРАНе-77 доступ к элементу массива был возможен единственным способом – явно через индекс элемента. Поэтому при работе с массивами приходилось использовать оператор цикла.

В современном ФОРТРАНе есть альтернативная возможность работы со всем массивом или его частью (см. [2, 3]). Сравните:

a=b	и	**do i=1,n** **a(i)=b(i)** **enddo**

6.1 Инициализация элементов массива

Дополним примеры инициализации массива из раздела **3.2**:

```
program sect0; implicit none; integer i,j           ! ФОРТРАН-инициализация
integer  A(5,7) / 11,12,13,14,15,  21,22,23,24,25,& ! массива через
                  31,32,33,34,35,  41,42,43,44,45,& ! 1) оператор
 51,52,53,54,55,  61,62,63,64,65,  71,72,73,74,75 / !        описания типа;
integer B(5,7)
data  B  / 11,12,13,14,15,  21,22,23,24,25,&        ! 2) оператор DATA;
           31,32,33,34,35,  41,42,43,44,45,&        !
 51,52,53,54,55,  61,62,63,64,65, 71,72,73,74,75 /
integer :: C(5,7)= reshape(&                        ! 3) конструктор
                   (/ ((10*j+i, i=1,5),j=1,7) /),&  !     массива и
                    shape=(/5,7/))                  !     функцию reshape
integer, dimension (5,7) :: D= reshape( &           !
      (/ ((10*j+i, i=1,5),j=1,7) /), shape=(/5,7/)) !
real E; dimension E(5); data E /1.0,2.0,3.0,4.0,5.0/! Так
dimension F(5); real F /1,2,3,4,5 /                 !     и
real, dimension(5) :: G=(/ 1,2,3,4,5 /)            ! так    можно!!!
!       real P; dimension P(5) / 1,2,3,4,5/         !   /  Но так
!       real, dimension :: Q(5)=(/1,2,3,4,5/)       !   \  НЕЛЬЗЯ!!!
write(*,'(5i6)') A            ! A, B, C, или D хранятся в памяти одинаково.
write(*,'(5f7.2)') E, F, G    ! E, F и G       хранятся в памяти одинаково.
end program sect0
```

Замечание При инициализации многомерного массива через конструктор помним, что в ФОРТРАНе **конструктор массива** — всегда **одномерный массив**. Для преобразования последнего в многомерный удобна функция **reshape**, которая через своё имя возвращает массив требуемой аргументом **shape** формы, заполненной данными из исходного массива, который в приведённом примере является одномерным.

```
 11    12    13    14    15       //  Результат работы sect0:
 21    22    23    24    25       //
 31    32    33    34    35       // Обратим внимание, что содержимое
 41    42    43    44    45       // каждого столбца матрицы A
 51    52    53    54    55       // выводится в строку, т.е. в порядке
 61    62    63    64    65       // расположения элементов матрицы в
 71    72    73    74    75       // оперативной памяти. Поэтому на
1.00  2.00  3.00  4.00  5.00  // выводе видим транспонированную
1.00  2.00  3.00  4.00  5.00  // матрицу.
1.00  2.00  3.00  4.00  5.00
```

reshape — указывает форму массива и способ его заполнения.

6.2 Секция (сечение) массива.

Секцией массива (или его **сечением**) называют некоторое подмножество его элементов. Задаётся секция именем массива и списком индексов секции. Секция массива сама является массивом и может состоять как из последовательно расположенных элементов, так и из выборки элементов, которые в оперативной памяти не соседствуют друг с другом. Например,

```
real, dimension :: A(5,7)=(/((10*j+i,i=1,5),j=1,7)/), U(4), W(5)
A(4,:)               ! четвёртая строка матрицы A
A(:,K)               ! k-й столбец матрицы A.
                     !
A(N1:N2, K1:K2:2)    ! прямоугольная секция матрицы, состоящая из
                     ! строк с N1-й по N2-ю, и из столбцов с индексами
                     ! одинаковой четности, начиная с K1-го по K2-ой.
                     !   K1:K2:2 - пример индексного триплета.
                     !
U=(/4, 2, 1, 7/)     ! Смысловая нагрузка элементов вектора U -
                     ! возможные номера столбцов матрицы, для которой
                     ! хотим построить сечение.
A(4,U)               ! Например, для матрицы A оно состоит из элементов
                     ! четвёртой строки, находящихся в 4, 2, 1, и 7
                     ! столбцах (сечение представляет собой вектор).
                     !      U - пример векторного индекса.
```

Индексом секции массива могут быть:

1) скалярный индекс; **2) индексный триплет**; **3) векторный индекс**.

Скалярный индекс — обычный индекс массива. Если все индексы секции скалярны, то сама секция — просто элемент массива.

Индексный триплет – символическое обозначение последовательности индексов, задаваемой в общем случае тремя характеристиками:

```
[нижний_индекс_триплета] : [верхний_индекс_триплета] [: шаг_по_индексу]
```

1. *квадратные скобки* – не синтаксический элемент, а напоминание о том, что заключённое в них, если нужно, можно не указывать.
2. При записи секции массива **индексным триплетом** из всех элементов массива выбираются лишь те, индексы которых заключены в указанном диапазоне триплета при учёте шага по индексу.
3. Если шаг не указан, то он полагается равным единице.
4. Если нижний индекс триплета не указан, то полагается, что он равен нижнему индексу соответствующего экстента массива.

```
program test1; implicit none; integer a(-5:14), i
a=5                           ! Всем элементам массива присваивается 5
a(-3:13:2)=6                  ! Элементы секции с нечетными индексами
a(-2:12:2)=0                  ! 3, 5, 7, 9, 11, 13 переопределяются на 6,
write(*,'(20i3)') (i,i=-5,14) ! а элементы секции с четными индексами
write(*,'(20i3)') a           ! -2, 0, 2, 4, 6, 8, 10, 12 обнуляются.
end program test1
```

```
 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
  5  5  6  0  6  0  6  0  6  0  6  0  6  0  6  0  6  0  6  5
```

Векторный индекс — целочисленный вектор, значениями элементов которого служат индексы элементов массива, попадающих в секцию.

1. **Векторный индекс** в отличие от индексного триплета позволяет включить в секцию произвольное множество элементов.
2. Значения индексов в векторном индексе могут:
   (a) располагаться в любом порядке (см. ниже программу **vec0**);
   (b) повторяться, т.е. секция может содержать по несколько экземпляров одного и того же элемента (см. ниже программу **vec1**). В этом случае она не должна встречаться в левой части оператора присваивания
3. Размер секции массива равен нулю, если векторный индекс — массив нулевого размера.

```
program vec0
implicit none
real p(10), q(7,10); integer :: vi(3), vj(5)
integer i, j
vi=(/7,1,5/);    vj=(/2,4,6,8,10/)
p=1;              q=2;            p(vi)=3.0
write(*,'(19x,"Массив p после модификации по векторному индексу vi")')
write(*,'(" Индекс массива p:",  10i5)')(i,i=1,10)
write(*,'(" Содержимое     p: ",10f5.0)') (p(i),i=1,10)
write(*,'(" Индексы секции         :",3i5)') vi
write(*,'(" Содержимое секции p(vi): ",3f5.0)') p(vi)
q(2,vj)=4.0
write(*,'(19x,"Матрица q после модификации по векторному индексу vj")')
write(*,'(" Номер столбца:",  10(i5))')(i,i=1,10)
write(*,'(" Номер  строки")')
write(*,'(5x,i2,9x,10f5.0)') (i,(q(i,j),j=1,10),i=1,7);
q(vi,vj)=5.0
write(*,'(19x,"Матрица q после модификации по векторным индексам vi и vj")')
write(*,'(" Номер столбца:",  10(i5))')(i,i=1,10)
write(*,'(" Номер  строки")')
write(*,'(5x,i2,9x,10f5.0)') (i,(q(i,j),j=1,10),i=1,7);
end
```

```
                  Массив p после модификации по векторному индексу vi
 Индекс массива p:    1    2    3    4    5    6    7    8    9   10
 Содержимое     p:    3.   1.   1.   1.   3.   1.   3.   1.   1.   1.
 Индексы секции         :    7    1    5
 Содержимое секции p(vi):    3.   3.   3.
                  Матрица q после модификации по векторному индексу vj
 Номер столбца:    1    2    3    4    5    6    7    8    9   10
 Номер  строки
      1            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      2            2.   4.   2.   4.   2.   4.   2.   4.   2.   4.
      3            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      4            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      5            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      6            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      7            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
                  Матрица q после модификации по векторным индексам vi и vj
 Номер столбца:    1    2    3    4    5    6    7    8    9   10
 Номер  строки
      1            2.   5.   2.   5.   2.   5.   2.   5.   2.   5.
      2            2.   4.   2.   4.   2.   4.   2.   4.   2.   4.
      3            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      4            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      5            2.   5.   2.   5.   2.   5.   2.   5.   2.   5.
      6            2.   2.   2.   2.   2.   2.   2.   2.   2.   2.
      7            2.   5.   2.   5.   2.   5.   2.   5.   2.   5.
```

```
program vec1; implicit none; real p(10); integer :: vi(3)
integer i, j
vi=(/7,1,4/)
p=1;                    p(vi)=(/61,21,31/)
write(*,'(19x,"Массив p после модификации по векторному индексу vi=(/7,1,4/)")')
write(*,'(" Индекс массива p:",  10i5)')(i,i=1,10)
write(*,'(" Содержимое     p: ",10f5.0)') (p(i),i=1,10)
write(*,'(" Индексы секции         :",3i5)') vi
write(*,'(" Содержимое секции p(vi): ",3f5.0)') p(vi)
vi=(/7,1,7/);           p(vi)=(/61,21,77/)
write(*,'(19x,"Массив p после модификации по векторному индексу vi=(/7,1,7/)")')
write(*,'(" Индекс массива p:",  10i5)')(i,i=1,10)
write(*,'(" Содержимое     p: ",10f5.0)') (p(i),i=1,10)
write(*,'(" Индексы секции         :",3i5)') vi
write(*,'(" Содержимое секции p(vi): ",3f5.0)') p(vi)
p((/7,1,7/))=(/35,6,25/)
write(*,'(19x,"Массив p после модификации по векторному индексу (/7,1,7/)")')
write(*,'(" Индекс массива p:",  10i5)')(i,i=1,10)
write(*,'(" Содержимое     p: ",10f5.0)') (p(i),i=1,10)
write(*,'(" Индексы секции         :",3i5)') vi
write(*,'(" Содержимое секции p(vi): ",3f5.0)') p(vi)
end
```

```
                  Массив p после модификации по векторному индексу vi=(/7,1,4/)
 Индекс массива p:    1    2    3    4    5    6    7    8    9   10
 Содержимое     p:   21.   1.   1.  31.   1.   1.  61.   1.   1.   1.
 Индексы секции         :    7    1    4
 Содержимое секции p(vi):   61.  21.  31.
                  Массив p после модификации по векторному индексу vi=(/7,1,7/)
 Индекс массива p:    1    2    3    4    5    6    7    8    9   10
 Содержимое     p:   21.   1.   1.  31.   1.   1.  77.   1.   1.   1.
 Индексы секции         :    7    1    7
 Содержимое секции p(vi):   77.  21.  77.
                  Массив p после модификации по векторному индексу (/7,1,7/)
 Индекс массива p:    1    2    3    4    5    6    7    8    9   10
 Содержимое     p:    6.   1.   1.  31.   1.   1.  25.   1.   1.   1.
 Индексы секции         :    7    1    7
 Содержимое секции p(vi):   25.   6.  25.
```

Выше в 2b) отмечалось (см.также [2, 3]), что секция с повторяющимися значениями индексов не может встречаться в левой [2] (правой [3]) части оператора присваивания. Как видим, используемая версия **gfortran**а разрешает подобные присваивания, беря в качестве окончательного значения, то, которое соответствует последнему правому из повторяющихся в списке индексов. Другими словами, контроль ситуации возлагается на человека, а не на компилятор.

6.3 Индексный триплет (временные оценки)

1. **Шаг индексного триплета равен 1:**

```
program testop1; implicit none            ! Программа kk раз выполняет
integer, parameter :: n=1000, kk=1000000  ! заполнение массива из n
real(8) a(n)                              ! элементов
integer k,i; real t(2),  r0, r1, r2, r3, r4, r5, r10
call etime(t,r0); do k=1,kk
                    do i=1,n; a(i)=1;enddo;   ! обычным циклом
                  enddo
call etime(t,r1); do k=1,kk;    a=2;    enddo ! простым присваиванием
call etime(t,r2); do k=1,kk; a(1:n:1)=3;enddo ! на базе индексного триплета
call etime(t,r3); do k=1,kk;  a(1:n)=4; enddo !
call etime(t,r4); do k=1,kk; a=(/ (i,i=1,n) /)! через конструктор  массива
                  enddo
call etime(t,r5); write(*,1000)
write(*,'(a,i7,5(f10.3))') 'n=',n,r1-r0,r2-r1,r3-r2,r4-r3,r5-r4
 1000 format(1x,11x,'do i=1,n',5x,'a=1',3x,'a(1:n:1)=2',2x,&
&                   'a(1:n)=3',3x,'a=(/ (i=1,n) /)')
end program testop1
```

```
            do i=1,n     a=1   a(1:n:1)=2  a(1:n)=3   a=(/ (i=1,n) /)
n=   1000     6.993     4.928     4.936     4.978     7.737            -O0
n=   1000     0.597     0.596     0.597     0.597     1.386            -O1

n=   4000    28.980    19.952    20.069    19.957    31.148            -O0
n=   4000     2.354     2.354     2.347     2.340     5.459            -O1
```

2. **Шаг индексного триплета равен 2:**

Заполнение нечетных элементы одномерного массива единицами, а четных нулями в современном ФОРТРАНе можно осуществить так

```
a(1:n:2)=1;  a(2:n:2)=0
a(:n:2)=1;   a(2::2)=0
a(::2)=1;    a(:n:-2)=0
```

Ниже приведена программа, позволяющая оценить быстродействие подобного присваивания по сравнению с обычным оператором цикла и с вызовом соответствующей подпрограммы. После исходного текста приведены результаты двух её пропусков. Первый — исполнимый файл получался при ключе оптимизации **-O0**; второй — при **-O1**.

```
program testop2  ! Программа kk раз заполняет два сечения (с чётными и
implicit none    ! нечетными индексами) массива из n элементов значениями
integer, parameter :: n=20000, kk=100000  ! 3.7d0 и -8.4d0 соответственно.
real(8) a(n)
integer k,i; real t(2),  r0, r1, r2, r3, r4, r5
call etime(t,r0); do k=1,kk                          ! Обычный цикл:
                    do i=1,n,2; a(i)=-8.4d0; enddo;  ! нечётные индексы
                    do i=2,n,2; a(i)= 3.7d0; enddo   ! чётные индексы
                  enddo
call etime(t,r1); do k=1,kk; call init(a,n,1)        ! Обычный цикл
                             call init(a,n,2)        ! в подпрограмме.
                  enddo
call etime(t,r2); do k=1,kk; a(1:n:2)=-8.4d0 ! Индекс.триплет: нечётные
                             a(2:n:2)= 3.7d0 !        и чётные индексы
                  enddo
call etime(t,r3); do k=1,kk; a(:n:2)= -8.4d0         ! нечетные индексы
                             a(2::2)= 3.7d0          ! чётные   индексы
                  enddo
call etime(t,r4); do k=1,kk; a(::2)= -8.4d0          ! нечетные индексы
                             a(2::2)= 3.7d0          ! чётные   индексы
                  enddo
call etime(t,r5); write(*,1000); write(*,2000)
write(*,'(a,i7,6(1x,f10.2,1x))') 'n=',n,r1-r0,r2-r1,r3-r2,r4-r3,r5-r4
1000 format(12x,'do i=1,n,2',6x,'init',5x,'a(1:n:2)',5x,'a(:n:2)',5x,&
&                                                          'a(::2)')
2000 format(12x,'do i=2,n,2',6x,'init',5x,'a(2:n:2)',5x,'a(2::2)',5x,&
&                                                           'a(2::2)')
end program testop2
subroutine init(a,n,k) ! Подпрограмма init(a,n,k) заполняет при k=1
implicit none          ! элементы вектора a с нечётными индексами
real(8) a(n)           ! 1(2)n константой -8.4,
integer i,n,k          ! a при k=2 элементы с чётными индексами
real(8) t(2)           ! 2(2)n константой 3.7
data t / -8.4, 3.7 /
do i=k,n,2 ; a(i)=t(k); enddo
end
```

Результат при ключе компиляции **-O0**:

```
           do i=1,n,2      init     a(1:n:2)     a(:n:2)     a(::2)
           do i=2,n,2      init     a(2:n:2)     a(2::2)     a(2::2)
n=  20000       9.27       13.55       11.04       11.10       11.09
```

Результат при ключе компиляции **-O1**:

```
           do i=1,n,2      init     a(1:n:2)     a(:n:2)     a(::2)
           do i=2,n,2      init     a(2:n:2)     a(2::2)     a(2::2)
n=  20000       3.55        3.75        3.51        3.54        3.56
```

6.4 Примеры использования сечений

6.4.1 Изменение порядка следования элементов массива

```
program revers; implicit none; integer, parameter :: kmax=10000, nn=100001
integer :: k, w, n; real  t1, t2, t3, t4, t5, t6
integer :: p(kmax)
do k=1,kmax; p(k)=k; enddo
write(*,'("  k =",(10i5))') (k,k=1,kmax,1000)
write(*,'("p(k)=",(10i5),"  Время работы ")') (p(k),k=1,kmax,1000)
                  do n=1,nn
                    do k=1,kmax/2
                      w=p(k);  p(k)=p(kmax-k+1); p(kmax-k+1)=w
                    enddo
                  enddo
call cpu_time(t2)
write(*,'("p(k)=",(10i5)$)') (p(k),k=1,kmax,1000); write(*,'(f7.2)') t2-t1
call cpu_time(t3)
                  do n=1,nn; call rev(p,kmax); enddo
call cpu_time(t4)
write(*,'("p(k)=",(10i5)$)') (p(k),k=1,kmax,1000); write(*,'(f7.2)') t4-t3
call cpu_time(t5)
                  do n=1,nn; p=p(kmax:1:-1);   enddo
call cpu_time(t6)
write(*,'("p(k)=",(10i5)$)') (p(k),k=1,kmax,1000); write(*,'(f7.2)') t6-t5
end
subroutine rev(a,kk)
implicit none
integer kk,k,a(kk),w
do k=1,kk/2; w=a(k); a(k)=a(kk-k+1); a(kk-k+1)=w; enddo
end
```

Результат при ключе оптимизации **-O0**:

```
  k =    1 1001 2001 3001 4001 5001 6001 7001 8001 9001
p(k)=    1 1001 2001 3001 4001 5001 6001 7001 8001 9001  Время работы
p(k)=10000 9000 8000 7000 6000 5000 4000 3000 2000 1000   3.32
p(k)=    1 1001 2001 3001 4001 5001 6001 7001 8001 9001   6.48
p(k)=10000 9000 8000 7000 6000 5000 4000 3000 2000 1000   2.77
```

Результат при ключе оптимизации **-O1**:

```
  k =    1 1001 2001 3001 4001 5001 6001 7001 8001 9001
p(k)=    1 1001 2001 3001 4001 5001 6001 7001 8001 9001  Время работы
p(k)=10000 9000 8000 7000 6000 5000 4000 3000 2000 1000   0.76
p(k)=    1 1001 2001 3001 4001 5001 6001 7001 8001 9001   0.76
p(k)=10000 9000 8000 7000 6000 5000 4000 3000 2000 1000   2.04
```

Почему при оптимизации -O1 индексный дескриптор работает медленнее?

6.4.2 Заполнение матриц

Здесь демонстрируется, как просто, используя сечения можно в первые три столбца матрицы **x(3,6)** занести содержимое матрицы **p(3,3)**, а в оставшиеся три содержимое матрицы **q**

```
program form0
implicit none
integer x(3,6); integer p(3,3) /  1,  2,  3,  4,  5,  6,  7,  8,  9/
integer i, j;   integer q(3,3) / 10, 20, 30, 40, 50, 60, 70, 80, 90/
write(*,'(" p=",3i3/2("   ",3i3/))') p
write(*,'(" q=",3i3/2("   ",3i3/))') q;  x(1:3,1:3)=p;   x(1:3,4:6)=q
write(*,'(" x=",3i3/5("   ",3i3/)//)') x
write(*,'(" p=",3i3/2("   ",3i3/))') ((p(i,j),j=1,3),i=1,3)
write(*,'(" q=",3i3/2("   ",3i3/))') ((q(i,j),j=1,3),i=1,3)
write(*,'(" x=",6i3/5("   ",6i3/))') ((x(i,j),j=1,6),i=1,3)
end
```

Результат:

```
 p=  1  2  3   На первый взгляд интуитивно кажется, что матрицы p и q
     4  5  6   выведены верно. И это, действительно, так.
     7  8  9   ОДНАКО, ПОМНИМ, что ФОРТРАН при указании в операторах
               инициализации, ввода и вывода только имени матрицы
 q= 10 20 30   заполняет её, вводит и выводит её элементы по СТОЛБЦАМ.
    40 50 60   Поэтому все выведенные строки есть содержимое СТОЛБЦОВ
    70 80 90                                                МАТРИЦЫ!
               Сказанное, в частности, относится и матрице x, в которой
 x=  1  2  3   согласно описанию ТРИ строки и ШЕСТЬ столбцов.
     4  5  6   Так что, в контексте данной программы содержимое каждой
     7  8  9   строки любой из матриц находится в соответствующем
    10 20 30   выведенном СТОЛБЦЕ. Если же хотим вывести содержимое
    40 50 60   матрицы так, чтобы СТРОКИ ВЫВОДА оказались действительно
    70 80 90   СТРОКАМИ МАТРИЦЫ, то об этом необходимо позаботиться
               особо, например, указывая очерёдность вывода элементов
               в циклоподобном списке вывода:

 p=  1  4  7           write(*,*) ((p(i,j),j=1,3),i=1,3)
     2  5  8   При такой форме его записи индекс j будет меняться
     3  6  9   чаще индекса i, и поэтому в строку ВЫВОДА будет
               выводиться действительно строка МАССИВА.
 q= 10 40 70
    20 50 80            write(*,*) ((q(i,j),j=1,3),i=1,3)
    30 60 90

 x=  1  4  7 10 40 70   write(*,*) ((x(i,j),j=1,6),i=1,3)
     2  5  8 20 50 80
     3  6  9 30 60 90
```

В примере из программы **form0** заполнение массивов **p** и **q** выполнялось при описании их типа посредством добавления списка констант, заключенного между двух прямых слешей. При таком задании начальных значений элементов матрицы данные из списка, трактуются по умолчанию, как расположенные последовательно друг за другом значения элементов очередного столбца матрицы. Современный ФОРТРАН предоставляет программисту и альтернативный способ посредством встроенной функции **reshape**.

1. **reshape** — размещает содержимое элементов массива одной формы нужным образом по элементам массива другой формы.

2. Вызов функции **reshape** в общем случае имеет вид:

```
reshape(source, shape [, pad] [,order])
```

   reshape — не подпрограмма, а именно **функция**, которая возвращает через своё имя **массив**. Формальные аргументы:

   source — исходный массив, значения элементов которого должны попасть в массив требуемой параметром **shape** формы;

   shape — вектор, задающий форму массива-результата (очередной элемент **shape** должен хранить число элементов по очередному измерению массива-результата).

   pad — Если **source** по размеру меньше результата, то незатронутые элементы последнего заполняются содержимым **pad**.

   order — указывает последовательность заполнения элементов массива (результата) данными из массива **source**.

```
program  form1; implicit none
integer a(9) / 1, 2, 3, 4, 5, 6, 7, 8, 9 /,  b(3,3), c(9,4), i, j
write(*,'(" a=",9i3)') a
b=reshape(a,(/3,3/));               write(*,'(" b="3i3/2("   ",3i3/))') b
b=reshape(a,(/3,3/),order=(/1,2/)); write(*,'(" b="3i3/2("   ",3i3/))') b
b=reshape(a,(/3,3/),order=(/2,1/)); write(*,'(" b="3i3/2("   ",3i3/))') b
c=reshape(a,(/9,4/), pad=(/ -1 /)); write(*,'(" c="9i3/3("   ",9i3/))') c
c=reshape(a,(/9,4/),(/ -1,-2/));    write(*,'(" c="9i3/3("   ",9i3/))') c
c=reshape(pad=(/ -1,-2,-3,(i,i=-90,-80,1)/),source=a,shape=(/9,4/))
                                    write(*,'(" c="9i3/3("   ",9i3/))') c
c=reshape(a,(/9,4/),(/ -1,-2,-3,(i,i=-90,-80,1)/),(/2,1/))
                                    write(*,'(" c="9i3/3("   ",9i3/))') c
end
```

```
a=  1  2  3  4  5  6  7  8  9   Как видим, применение reshape в виде
                                reshape(a,(/3,3/)) привело к тому же
b=  1  2  3
    4  5  6                     результату, что и в программе form0
    7  8  9
                                integer p(3,3) /1,2,3,4,5,6,7,8,9/
b=  1  2  3
    4  5  6     Тот же результат и при  reshape(a,(/3,3/),order=(/1,2/))
    7  8  9
                Однако,                 reshape(a,(/3,3/),order=(/2,1/))
b=  1  4  7  <---= даёт. Здесь содержимое  order
    2  5  8     указывает, что при заполнении  b быстрее всего должен
    3  6  9     меняться индекс её второго измерения, т.е. номер столбца,
                так что 2 из A должна попасть во 2-й столбец 1-ой строки.

c=  1  2  3  4  5  6  7  8  9  В матрице C на 27 элементов больше чем в A.
   -1 -1 -1 -1 -1 -1 -1 -1 -1  Параметр pad=(/-1/) при вызове
   -1 -1 -1 -1 -1 -1 -1 -1 -1            c=reshape(a,(/4,9/),pad=(/ -1/))
   -1 -1 -1 -1 -1 -1 -1 -1 -1  требует, чтобы элементы C, в которые
                               ничего не попало из A, заполнились -1.
c=  1  2  3  4  5  6  7  8  9  Если хотим, чтобы при их заполнении
   -1 -2 -1 -2 -1 -2 -1 -2 -1  чередовались -1 и -2, то используем
   -2 -1 -2 -1 -2 -1 -2 -1 -2
   -1 -2 -1 -2 -1 -2 -1 -2 -1    c=reshape(a,(/4,9/),pad=(/-1,-2/))

c=  1  2  3  4  5  6  7  8  9  Если же при заполнении оставшихся нужна
   -1 -2 -3-90-89-88-87-86-85  некоторая особая цикличность, то можно
  -84-83-82-81-80 -1 -2 -3-90  использовать соответствующий циклоподобный
  -89-88-87-86-85-84-83-82-81  элемент в конструкторе массива.

c=  1  5  9-90-86-82 -2-88-84  Результат подключения к предыдущему вызову
    2  6 -1-89-85-81 -3-87-83  reshape ключевого параметра:
    3  7 -2-88-84-80-90-86-82                          order=(/2,1/).
    4  8 -3-87-83 -1-89-85-81
```

3. Как видно из текста программы **form1** при вызове **reshape** возможны различные способы задания её аргументов:

   1) **b=reshape(a,(/3,3/))** — заданы лишь два обязательных: исходный массив, поставщик данных и форма массива-результата.
   2) **b=reshape(a,(/3,3/),order=(/2,1/))**. **Ключевой аргумент order** указывает, что второй индекс при заполнении массива-результата должен меняться быстрее, а первый — медленнее.
   3) **c=reshape(a,(/9,4/), pad=(/ -1 /))** — через ключевой аргумент **pad** передаётся **“наполнитель”**;

4) **c=reshape(a,(/9,4/),(/ -1,-2/))** — все аргументы **позиционные** (трактовка каждого определяется его позицией).

5) При вызове

c=reshape(pad=(/-1,-2,-3,(i,i=-90,-80,1)/),source=a,shape=(/9,4/))

очерёдность **ключевых аргументов** не имеет значения. При задании "**наполнителя**" использован циклоподобный список.

6) **c=reshape(a,(/9,4/),(/-1,-2,-3, (i,i=-90,-80,1)/),(/2,1/))** при вызове нет ключевых слов: все аргументы — позиционные.

4. При вызове любой процедуры можно использовать **ключевые** имена аргументов. **Ключевое имя** — имя формального параметра, соединённое с именем фактического знаком операции присваивания.

```
program key; implicit none; integer a, b, r
interface
 function f(x,y); integer y,f; integer, optional :: x; end function f;
end interface                                       ! Результат
a=85; b=17; r=f( a , b ); write(*,*) r              !    5
            r=f(x=a,y=b); write(*,*) r              !    5
            r=f(y=b,x=a); write(*,*) r              !    5
            r=f(y=a,x=b); write(*,*) r              !    0
            r=f( a, y=a); write(*,*) r              !    1
            r=f(y=0);     write(*,*) r              !  333
!           r=f(x=a, b )  Позиционный аргумент недопустим после ключевого
end
function f(x,y); implicit none; integer f, y; integer, optional :: x
if (y==0) then; f=333      !                            optional
          else; f=x/y      ! сообщает компилятору, что аргумент x ---
endif                      !                            необязательный
end
```

Подробнее см. [2, 3]

5. Ешё раз: **reshape** — именно функция, которая возвращает через своё имя массив. Современный ФОРТРАН предоставляет программисту возможность описывать функции, обладающие таким свойством. Ниже приводится программа **testfunarr** с тремя вариантами реализации подобного описания:

1) внутренней функцией **vecpow0** (описана в главной программе);

2) внешней функцией **vecpow1** (описанной в отдельном файле);

3) модульной функцией **vecpow2** (описана в **module myvec**).

и соответствующие результаты её работы. Каждая функция решает одну и ту же задачу — по заданному целому **n** вычисляет вектор, первые **n** элементов которого содержат соответствующие степени двойки.

```
program testfunarr; use myvec; implicit none
interface
  function vecpow1(n); integer n;
                       real(8) vecpow1(n);
  end function vecpow1;
end interface
integer, parameter :: n=5; integer i
real(8) a(n), b(n), c(n)
a=vecpow0(n); b=vecpow1(n); c=vecpow2(n)
write(*,'(1x," i",13x,"a",14x,"b",13x,"c")')
write(*,'(i3,3f15.2)') (i,a(i),b(i),c(i),i=1,n)
contains
function vecpow0(n); implicit none
integer n, i
real(8) vecpow0(n); do i=1,n;  vecpow0(i)=2.0**i; enddo
end function vecpow0
end

function vecpow1(n)
implicit none
integer n, i
real(8) vecpow1(n); do i=1,n; vecpow1(i)=2.0**i; enddo
end

module myvec; implicit none; contains
function vecpow2(n)
integer n, i
real(8) vecpow2(n); do i=1,n; vecpow2(i)=2.0**i; enddo
end function vecpow2
end module myvec
```

```
  i             a             b             c       ! Результат работы
  1          2.00          2.00          2.00     ! программы testfunarr
  2          4.00          4.00          4.00
  3          8.00          8.00          8.00
  4         16.00         16.00         16.00
  5         32.00         32.00         32.00
```

6.4.3 Сечение в качестве параметра процедуры

Процедура поиска в каждой строке матрицы первого отрицательного элемента (пример из [3]).

1. Реализация через внутреннюю функцию (т.е. функцию, описание которой находится в теле вызывающей программы; не путать со встроенной функцией).

```
program negel0; implicit none
integer, parameter :: m=10, n=5; integer i
integer b(m,n)
read (*,'(10i4)') b; write(*,'(10i4)') b
do i=1,m
  write(*,'(i3,a,10i3)') i,'...',fneg1(b(i,:))
enddo
contains
integer function fneg1(a); implicit none
integer a(:), k
fneg1=0
do k=1,size(a)
  if (a(k)<0) then
               fneg1=a(k)
               return
  endif
enddo
end function fneg1
end
```

Результат:

```
   1 -11  21  31  41  51  61  71 -81  91  ! Обратите внимание, что при
   2  12  22 -32  42  52  62 -72  82  92  ! выводе матрицы b её строки
   3  13  23  33  43 -53  63 -73  83  93  ! оказываются в столбцах
   4  14  24  34 -44 -54  64  74  84 -94  ! выведенного, т.к. элементы
   5  15  25  35  45  55  65  75 -85 -95  ! ФОРТРАН-матрицы в
  1...  0                                 ! оперативной памяти
  2...-11                                 ! расположены по столбцам.
  3...  0
  4...-32
  5...-44
  6...-53
  7...  0
  8...-72
  9...-81
 10...-94
```

2. Реализация через внешнюю функцию с использованием сечений.

```
program negel1; implicit none            ! Функция с перенимающим
interface                                ! форму массивом должна в
integer function fneg1(a); integer a(:); ! вызывающей её программе
end function fneg1                       ! иметь ЯВНО заданный
end interface                            !               интерфейс.
integer, parameter :: m=10, n=5; integer i
integer b(m,n)                           ! Программе negel0 он не
read (*,'(10i4)') b; write(*,'(10i4)') b ! нужен, ибо fneg1  была
 do i=1,m                                ! внутренней, т.е. имела
                                         ! имела ЯВНЫЙ интерфейс.
  write(*,'(i3,a,10i3)') i,'...',fneg1(b(i,:))
enddo
end
integer function fneg1(a); implicit none
integer a(:), k
fneg1=0
do k=1,size(a)
  if (a(k)<0) then
               fneg1=a(k); return
  endif
enddo
end
```

3. Реализация через внешнюю функцию без использования сечений.

```
program negel2; implicit none
integer, parameter :: m=10, n=5; integer i, j, fneg2
integer a(m), b(m,n)
read (*,'(10i4)') b; write(*,'(10i4)') b
do i=1,m
  do j=1,m
    a(j)=b(i,j)
  enddo
  write(*,'(i3,a,10i3)') i,'...',fneg2(a,m)
enddo
end
function fneg2(a,m); implicit none  ! Забавный факт. Если имя fneg2
integer fneg2, m, a(m), k           ! убрать из списка integer и
fneg2=0                             ! integer поместить перед
do k=1,m                            ! function, то вывод для j=1
  if (a(k)<0) then                  ! осуществится *** а не нулём.
               fneg2=a(k)           ! Почему? Проверить!
               return
  endif
enddo
end
```

6.5 Выборочное присваивание (присваивание по маске)

Пусть требуется в сто раз увеличить все положительные элементы массива. ФОРТРАН-77 предоставлял только одну возможность:

```
program where0; implicit none
integer a(10) /-1,0,3,-5,9,-3,8,4,-9,0/
integer i
write(*,'(10i4)') a             !            Результат работы
do i=1,10                       !  -1   0   3  -5   9  -3   8   4  -9   0
  if (a(i).gt.0) a(i)=100*a(i)  !  -1   0 300  -5 900  -3 800 400  -9   0
enddo
write(*,'(10i4)') a
end
```

В современном ФОРТРАНе имеются и альтернативные более удобные возможности. Это — оператор **where** и конструкция **where**. Конструкция отличается от оператора своей **рамочной структурой**, что позволяет в качестве её выполнимого тела содержать несколько операторов (т.е., как иногда говорят, блоки операторов и конструкций), в то время как выполнимым телом оператора **where** может быть лишь один единственный оператор. Например,

```
program where1; implicit none
integer a(10) /-1,0,3,-5,9,-3,8,4,-9,0/
integer i                                  ! Здесь оператор  where
write(*,'(10i4)') a;                       ! увеличивает содержимое
where (a>0) a=100*a;                       ! положительных элементов
write(*,'(10i4)') a                        ! массива a в 100 раз.
where (a>0)
            a=a/2                          ! Здесь конструкция where
endwhere                                   ! уменьшает их вдвое.
write(*,'(10i4)') a
where (a>0)                                ! Здесь конструкция where
            a=a*2                          ! не только увеличивает
  elsewhere                                ! вдвое положительные, но
            a=a-1                          ! и на единицу  уменьшает
endwhere                                   ! отрицательные.
write(*,'(10i4)') a
end
```

Результат программы **where1**

```
  -1   0   3  -5   9  -3   8   4  -9   0
  -1   0 300  -5 900  -3 800 400  -9   0
  -1   0 150  -5 450  -3 400 200  -9   0
  -2  -1 300  -6 900  -4 800 400 -10  -1
```

Оператор **where** — присваивание по маске имеет вид ([2])

where (логическое_выражение) оператор_присваивания

- **логическое_выражение** — выражение логического типа (маска).
- **оператор_присваивания** — оператор присваивания, у которого левая часть есть **массив**.

Конструкция **where** (присваивание по маске) может иметь вид:

where (логическое_выражение)
блок_операторов
[elsewhere
блок_операторов]
endwhere

Здесь

1. Квадратные скобки — не синтаксический элемент, а лишь напоминание, что их содержимое, когда это удобно, можно не писать.
2. **Блок_операторов** — ряд операторов присваивания, у которых левые части являются массивами.
3. Форма **массива-маски** должна совпадать с формой массивов в левой части операторов присваивания.
4. **Присваивание по маске** начинается с вычисления логического выражения маски.
5. Выражение, входящее в правую часть **where** (или в тело конструкции), вычисляется лишь для тех элементов, у которых значение маски — **ИСТИНА**, и результат присваивается элементам массива, соответствующим условию истинности логического выражения.
6. Встречающаяся в правых частях функция может быть **поэлементной** и **непоэлементной**.

   Для **поэлементной** функции (НЕ ЯВЛЯЮЩЕЙСЯ фактическим аргументом **непоэлементной**), как и ожидается, будут вычисляться лишь элементы, соответствующие условию истинности маски;

   Для **непоэлементной** — будут вычислены значения всех элементов массива независимо от условия истинности маски. Например,

```
program where2; implicit none; real a(10) /-1,0,3,-5,9,-3,8,4,-9,0/
integer i
write(*,'(10f7.2)') a;    where (a>0)  a=sqrt(a)
write(*,'(10f7.2)') a
where (a>0)
           a=sqrt(a)
endwhere
write(*,'(10f7.2)') a
end
```

Результат программы **where2**

```
  -1.00   0.00   3.00  -5.00   9.00  -3.00   8.00   4.00  -9.00   0.00
  -1.00   0.00   1.73  -5.00   3.00  -3.00   2.83   2.00  -9.00   0.00
  -1.00   0.00   1.32  -5.00   1.73  -3.00   1.68   1.41  -9.00   0.00
```

```
program where3; implicit none           ! Здесь ПОЭЛЕМЕНТНАЯ функция sqrt
real a(10) /-1,0,3,-5,9,-3,8,4,-9,0/    ! является фактическим аргументом
write(*,'(10f7.2)') a;                  ! НЕПОЭЛЕМЕНТНОЙ функции SUM, на
   where (a>0)  a=a+sum(sqrt(a))        ! которую действие маски
write(*,'(10f7.2)') a                   ! не распространяется:
where (a>0)                             ! Поэтому сначала вычислится сумма
           a=a+sum(sqrt(a))             ! корней квадратных от всех
endwhere                                ! элементов массива a, которая
write(*,'(10f7.2)') a                   ! ввиду отрицательности некоторых
end                                     ! из них окажется равной NaN.
```

Результат программы **where3**

```
  -1.00   0.00   3.00  -5.00   9.00  -3.00   8.00   4.00  -9.00   0.00
  -1.00   0.00    NaN  -5.00    NaN  -3.00    NaN    NaN  -9.00   0.00
  -1.00   0.00    NaN  -5.00    NaN  -3.00    NaN    NaN  -9.00   0.00
```

окажется эквивалентен результату программы:

```
program where3
implicit none
real a(10) /-1,0,3,-5,9,-3,8,4,-9,0/, s
integer i
s=sum(sqrt(a))
write(*,*) ' s=', s
write(*,'(10f7.2)') a;    where (a>0)  a=a+s
write(*,'(10f7.2)') a
where (a>0)
           a=a+s
endwhere
write(*,'(10f7.2)') a
end
```

6.6 Оператор и конструкция forall

Полезное средство ФОРТРАНа-95 (не было в ФОРТРАНом-90) [2], нацеленное на множественное выборочное присваивание значений элементам массива. Пример

```
program forall0; implicit none; integer, parameter :: n=9
integer a(n), b(n,n)
integer i
a=(/ (100*i,i=1,n) /)      ! Массив A заполняется посредством конструктора
b=0
forall (i=1:n) b(i,i)=a(i) ! Оператор forall заполняет диагональ матрицы B
write(*,'(9i5)') b
end program forall0
```

Результат работы **forall0**:

```
  100    0    0    0    0    0    0    0    0   WHERE ориентирован на
    0  200    0    0    0    0    0    0    0   весь массив целиком.
    0    0  300    0    0    0    0    0    0   FORALL позволяет
    0    0    0  400    0    0    0    0    0   специфицировать более
    0    0    0    0  500    0    0    0    0   широкий класс секций
    0    0    0    0    0  600    0    0    0   массива.
    0    0    0    0    0    0  700    0    0   Здесь посредством forall
    0    0    0    0    0    0    0  800    0   заполнена ДИАГОНАЛЬ
    0    0    0    0    0    0    0    0  900             матрицы.
```

1. Оператор **forall**:

forall (заголовок) оператор_присваивания

2. Конструкция **forall**:

[имя] forall (заголовок)
тело конструкции
end forall [имя]

Заголовок имеет вид:

`триплет [,триплет] ... [, логическое выражение]`

Логическое выражение — скалярное логическое выражение типа (маска).

Триплет:

`имя\_индекса = нижний\_индекс:верхний\_индекст [: шаг\_по\_индексу]`

Индексы и шаг(может быть и отрицательным) — скалярные целого типа. Имя индекса может присутствовать в **маске**.

Схема выполнения forall:

1) по триплетам определяется набор допустимых комбинаций индексов.

2) для этого набора вычисляются значения маски;

3) для него же вычисляются выражения в правых частях операторов присваивания и в индексах левых частей , если значения маски есть „ИСТИНА”.

4) осуществляется присваивание вычисленных значений соответствующим элементам левой части;

На первый взгляд **forall** функционально похож на оператор цикла. Однако, последовательность действий у них разная. Обратимся к примеру:

```
program forall1; implicit none       ! В программе описаны две матрицы B и C,
integer, parameter :: n=5            ! с одинаковыми ненулевыми элементами по
integer i, a(n), b(n,n), c(n,n)      ! по главной диагонали, заполняемой из
a=(/ (100*i,i=1,n) /); b=0           ! вектора A так, как в предыдущей задаче.
forall (i=1:n) b(i,i)=a(i); c=b                    !  Здесь матрицы B и C
write(*,*) 'b=c:'; write(*,'(5("    ",5i5,/) )') b  !           одинаковы.
forall (i=2:n) b(i,i)=b(i-1,i-1)                   !  FORALL !!!
write(*,'(a,33x,a)') '  b:','Работа forall c B:'
write(*,'( 5("    ",5i5/) )') b
do i=2,n; c(i,i)=c(i-1,i-1); enddo                 !    DO   !!!
write(*,'(a,33x,a)') '  c:','Работа   do   c C:';
write(*,'( 5("    ",5i5/) )') c
end
```

```
 b=c:                                Результат работы forall1:
      100    0    0    0    0
        0  200    0    0    0
        0    0  300    0    0
        0    0    0  400    0
        0    0    0    0  500


  b:                                 Работа forall c B:
      100    0    0    0    0
        0  100    0    0    0        Здесь  все диагональные элементы
        0    0  200    0    0        выбираются и используются ДО ТОГО
        0    0    0  300    0        как их модифицированное значение
        0    0    0    0  400        СОХРАНЯЕТСЯ в оперативной памяти.

  c:                                 Работа   do   c C:
      100    0    0    0    0
        0  100    0    0    0        Итерации do происходят последовательно.
        0    0  100    0    0        СОХРАНЕНИЕ очередного диагонального
        0    0    0  100    0        элемента происходит сразу как только
        0    0    0    0  100        он вычислен
```

Пример работы оператора forall с маской:

```
program forall2a           ! Программа демонстрирует использование
implicit none              ! оператора forall с маской, обеспечивающей
integer, parameter :: n=5  ! обнуление элементов матрицы, лежащих выше
integer i, j; real A(n,n)  ! её главной диагонали.
A=reshape((/ ((10.0*i+j, j=1,n),i=1,n)/),(/n,n/))
write(*,'(" Вывод исходной A по умолчанию:"/5(30x,5f6.0/))') A
write(*,'(18x," по   строкам:"/5(30x,5f6.0/))') ((a(i,j),j=1,n),i=1,n)
write(*,'(18x," по  столбцам:"/5(30x,5f6.0/))') ((a(i,j),i=1,n),j=1,n)
forall (i=1:n,j=1:n, j>i) A(i,j)=0          !  forall !!!
write(*,'(" Result  A:"/5(5f6.0/))') A      ! Вывод результата
write(*,'(18x," по   строкам:"/5(30x,5f6.0/))') ((a(i,j),j=1,n),i=1,n)
end
```

```
 Вывод исходной A по умолчанию:
                                11.   12.   13.   14.   15.
                                21.   22.   23.   24.   25.
                                31.   32.   33.   34.   35.
                                41.   42.   43.   44.   45.
                                51.   52.   53.   54.   55.

                    по   строкам:
                                11.   21.   31.   41.   51.
                                12.   22.   32.   42.   52.
                                13.   23.   33.   43.   53.
                                14.   24.   34.   44.   54.
                                15.   25.   35.   45.   55.

                    по  столбцам:
                                11.   12.   13.   14.   15.
                                21.   22.   23.   24.   25.
                                31.   32.   33.   34.   35.
                                41.   42.   43.   44.   45.
                                51.   52.   53.   54.   55.

 Result  A:
   11.   12.   13.   14.   15.
    0.   22.   23.   24.   25.
    0.    0.   33.   34.   35.
    0.    0.    0.   44.   45.
    0.    0.    0.    0.   55.

                    по   строкам:
                                11.    0.    0.    0.    0.
                                12.   22.    0.    0.    0.
                                13.   23.   33.    0.    0.
                                14.   24.   34.   44.    0.
                                15.   25.   35.   45.   55.
```

Замечание:

- Поместив в список вывода результата лишь имя матрицы, получили расположение нулей не над главной диагональю, а под ней.
- Причина — вывод матрицы по столбцам (т.е. в первой строке вывода оказывается первый столбец матрицы и т.д).
- Для вывода матрицы по строкам необходим соответствующий оператор цикла, что и продемонстрировано.

Пример работы конструкции forall с маской:

```
program forall2b; implicit none    ! Программа использует конструкцию
integer, parameter :: n=5          ! forall с маской A(i)/=0
integer i; real a(n)               ! для того, чтобы элементы вектора A
a=(/ (i-3,i=1,n) /)                ! приняли значения равные обратной
write(*,'("source a:",5f6.0)') a   ! величине их первоначального значения.
forall (i=1:n, a(i)/=0);           ! Обратите внимание, что при
     a(i)=1/a(i)                   ! наличии в спецификаторе формата f6.0
end forall                         ! после точки цифры нуль, дробная
write(*,'("result a0:",5f6.0)') a  ! часть числа НЕ ВЫВОДИТСЯ. Так
write(*,'("result a1:",5f6.1)') a  ! что можно долго недоумевать:
end                                ! почему это 1/2.0 = 0 (или 1).
```

```
source a:   -2.   -1.    0.    1.    2.
result a0:    0.   -1.    0.    1.    0.
result a1:  -0.5  -1.0   0.0   1.0   0.5
```

Тело конструкции **forall** может содержать:

операторы: **присваивания**, **where**, **forall**.

Конструкция **forall** допускает вложенность (см. [2]).

Конструкция может быть снабжена именем, что бывает полезным при наличии вложенных конструкций, т.к. с необходимостью требует завершения каждой конструкции соответствующим ей именем — тем самым снижается вероятность её некорректного закрытия.

Вообще говоря, помимо операторов старого ФОРТРАНа современный ФОРТРАН предоставляет соответствующие конструкции. Каждая конструкция может иметь имя. Мы уже активно пользовались конструкциями **if-then-endif**, **do-enddo**, **select case-end selectf**, но у нас они были **неименованными** (подробнее см. [2]).

6.7 О чем узнали из шестой главы? (второй семестр)

1. Современный ФОРТРАН для работы с массивами (наряду с возможностями ФОРТРАНа-77) предоставляет альтернативные, которые позволяют существенно сократить размер исходных текстов.

2. Инициализировать массив можно через **конструктор массива**.

3. **Конструктор массива** — одномерный массив.

4. **Секция** (или **сечение**) массива обеспечивает доступ к той или иной группе его элементов.

5. **Секция** массива — массив. Её индексами могут быть:

   1) **Скалярный индекс** — индекс массива по ФОРТРАНу-77;
   2) **Индексный триплет** – символическое обозначение последовательности индексов, задаваемой в общем случае тремя признаками: **[нижний_индекс] : [верхний_индекс] [: шаг_по_индексу]** .
   3) **Векторный индекс** — целочисленный вектор, значения элементов которого есть индексы элементов, входящих в секцию.

6. Встроенная функция **reshape** возвращает через своё имя массив, в котором размещаются данные из элементов исходного массива **source** согласно указанной в массиве **shape** форме результата. Порядок размещения определяется содержимым массива **order**. Если размер **source** меньше размера **reshape**, то её неприсвоенные элементы можно заполнить содержимым аргумента **pad**.

7. При вызове **reshape** её фактические аргументы могут быть **позиционными** или **ключевыми** (в зависимости от способа записи).

8. Фактический аргумент считается **позиционным**, если присутствует при вызове функции **без** соответствующего **ключевого слова**. Пример вызова функции **reshape** с **позиционными** аргументами:

```
c=reshape(a,(/9,4/),(/-1,-2,-3,(i,i=-90,-80,1)/),(/2,1/))
```

9. В качестве **ключевого слова** всегда выступает имя **формального** аргумента. Так, при описании встроенной функции **reshape** её формальные аргументы имеют имена: **reshape(source, shape , pad, order)**

10. Фактический аргумент считается **ключевым**, если ему при вызове функции ему сопоставлено имя формального аргумента.

11. Аргументы **pad** и **order** функции **reshape** при её вызове могут отсутствовать (если допустимо), т.к. являются **необязательными**.

12. Для наделения формального аргумента функции свойством **необязательности** используем служебное слово **optional**.

13. **Секция** массива допустима в качестве фактического аргумента.

14. Современный ФОРТРАН предоставляет операторы **where** и **forall** и одноимённые им **конструкции**, реализующие возможность выборочного присваивания значений элементов массива **по маске**.

15. **Конструкция** отличается от оператора своей рамочной структурой и позволяет в качестве её выполняемого тела использовать блоки операторов и конструкций.

16. Функции могут быть **поэлементными** и **непоэлементными**. Поэлементная в качестве фактического аргумента может иметь либо скаляр, либо массив. Результатом её работы в первом случае будет скаляр; во втором — массив, значение каждого элемента которого получается применением функции к соответствующему элементу массива-аргумента. Примеры поэлементных функций: **sin**, **cos** и др.

17. Современный ФОРТРАН позволяет наделять свойством поэлементности и процедуры, описываемые программистом. Для этого при описании функции достаточно снабдить её префиксом **elemental**. Поэлементные функции позволяют экономить время при работе с массивами, если компилятор допускает распараллеливание процесса вычислений. (подробнее см. [2]).

18. **Не путаем** термин **поэлементная функция** с понятием функции, возвращающей через своё имя **массив**. При описании последней в её теле необходимо специфицировать массив с именем функции. В случае, если упомянутая функция — внешняя, то в программе, вызывающей её, необходимо явное описание соответствующего интерфейса (подробнее см. [2]).

6.8 Шестое домашнее задание (второй семестр).

1. Разработать функции **sxy**, **sxz**, **syz**, получающие соответственно в качестве результата матрицы **k**-го слоя параллельного грани трёхмерного динамического массива **a(nx,ny,nz)**. В качестве тестового примера ввести **nx=3**, **ny=4**, **nz=5** и **a=**

   1 2 3 4 5 6 7 8 9 10 11 12

   10 20 30 40 50 60 70 80 90 100 110 120

   100 200 300 400 500 600 700 800 900 1000 1100 1200

   1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000

   10000 20000 30000 40000 50000 60000 70000 80000 90000 100000 11000 12000

2. Написать функцию **myresh2(ord,shape,b)** (по сути — упрощённый аналог встроенной **reshape**), преобразующую вектор **b(s(1)*s(2))** в матрицу из **s(1)** строк и **s(2)** столбцов. Заполнение матрицы проводить согласно значениям элементов аргумента **ord(2)**:

   при **ord=(/1,2/)** вести заполнение по росту номера строки;

   при **ord=(/2,1/)** вести заполнение по росту номера столбца.

   В главной программе можно использовать **reshape**, но исключительно с целью подтверждения правильности работы **myresh2**.

3. Модифицировать **mtresh2** так, чтобы параметр **ord** можно было не писать при вызове функции, если **ord=(/1,2/)**.

4. Написать функцию **myresh3(ord,shape,b)** (по сути —упрощённый аналог встроенной **reshape**), которая выполняет работу **reshape** по преобразованию вектора **b(s(1)*s(2)*s(3)** в трёхмерный массив из **s(1)** строк, **s(2)** столбцов и **s(3)** листов. Заполнение матрицы проводить согласно значениям элементов аргумента **ord(3)**:

   при **ord=(/1,2,3/)** заполнение сначала идёт по росту номера строки при фиксированном столбце и листе; затем (при переходе к следующему столбцу) снова по росту номера строки; затем по росту номера листа. Продемонстрировать правильную работу **myresh3** для всех наборов **ord=(/1,3,2/)**, **(/2,1,3/)**, **(/2,3,1/)**, **(/3,1,2/)**, **(/3,2,1/)**. В главной программе можно использовать встроенную **reshape**, но исключительно с целью подтверждения правильности работы **myresh3**.

7 Функции ФОРТРАНА для работы с массивами

Здесь имеются ввиду функции **СОВРЕМЕННОГО** ФОРТРАНа.

7.1 Справочные функции

1. **allocated(array)** — возвращает **.true.** (стандартного логического типа по умолчанию), если массив **array**, описанный с атрибутом **allocatable** (размещаемый), к моменту вызова **allocated** уже размещён, т.е. если под **array** уже выделена память посредством оператора **allocate**; если же выделения памяти ещё не было, то функция **allocated** возвращает в вызвавшую её процедуру значение **.false.**.

```
program test_allocated; implicit none; integer :: n = 4; logical l
real(4), allocatable :: a(:)
l=allocated(a)
write(*,*) ' Если массив "a" НЕ РАЗМЕЩЁН, то allocated(a)=',l
if (allocated(a) .eqv. .false.) allocate(a(n))
write(*,*) ' Если массив "a"    РАЗМЕЩЁН, то allocated(a)=',allocated(a)
end program test_allocated
```

```
 Если массив "a" НЕ РАЗМЕЩЁН, то allocated(a)= F
 Если массив "a"    РАЗМЕЩЁН, то allocated(a)= T
```

2. **size(array [, dim])** — возвращает значение стандартного целого типа, которое равно количеству элементов массива **array**. При наличии необязательного аргумента **dim**, задающего желаемое измерение массива **array**, функция **size** возвращает количество элементов вдоль указанного **dim** измерения.

```
program test_size; implicit none
integer kb, k; real(4), allocatable :: a(:,:,:,:); real(8) b(2,4,5,7)
kb=size(b)
allocate(a(-3:-2,-4:-1,-5:4,0:7))
write(*,'(" size(b)=2*4*5*7=",i3,10x,"size(a)=2*4*10*8=",i3)') kb,size(a)
write(*,'(" size(b,",i1,")=",i2,16x," size(a,",i1,")=",i2)')&
                &(k, size(b,k), k, size(a,k),k=1,4)
end program test_size
```

```
 size(b)=2*4*5*7=280          size(a)=2*4*10*8=640
 size(b,1)= 2                 size(a,1)= 2
 size(b,2)= 4                 size(a,2)= 4
 size(b,3)= 5                 size(a,3)=10
 size(b,4)= 7                 size(a,4)= 8
```

3. **shape(source)** — возвращает вектор (стандартного целого типа), хранящий форму (конфигурацию) массива или скаляра **source**.

```
program test_shape; implicit none; real(8), allocatable :: a(:,:,:,:)
integer v(4)
real(8) scalar /1.7/
allocate(a(-3:-2,-4:-1,-5:4,0:7)); v=shape(a)
write(*,'(" v=",4i4)') v
write(*,*) ' shape(scalar)=',shape(scalar),'aaaa'
write(*,*) ' size(shape(scalar))=',size(shape(scalar))
write(*,*) ' size(shape(  1.7 ))=',size(shape(1.7))
! write(*,*) ' shape(1.7)=',shape(1.7)   ! возможна ошибка сегментирования
! write(*,*) ' shape(42)=',shape(42),'b' ! возможна ошибка сегментирования
write(*,*) ' shape(42)=',shape(42)
end program test_shape
```

```
 v=   2   4  10   8
  shape(scalar)=aaaa                ! Скаляр - одномерный массив ранга 0.
  size(shape(scalar))=           0  ! В нём нет элементов.
  size(shape(  1.7 ))=           0  ! Размер массива ранга нуль равен 0.
  shape(42)=
```

4. **lbound(array [, dim])** и **ubound(array [, dim])** — при отсутствии **dim** (номера измерения) возвращают вектор (стандартного целого типа), элементы которого содержат нижние и соответственно верхние границы всех измерений. В этом случае число элементов вектора-результата равно числу измерений (рангу) массива **array**. При наличии **dim** возвращается скаляр равный нижней (соответственно верхней) границе **dim**-го измерения массива **array**.

```
program test_lubound; implicit none
integer, parameter :: l1=-3, u1=-2, l2=-4, u2=-1, l3=-5, u3=4, l4=0, u4=7
real(8), allocatable :: a(:,:,:,:)
integer, allocatable :: lb(:), ub(:)
allocate(a(l1:u1,l2:u2,l3:u3,l4:u4)); allocate(lb(size(shape(a))))
                                      allocate(ub(size(shape(a))))
lb=lbound(a); ub=ubound(a)
write(*,'(" lb=lbound(a)=",4i4,"    lbound(a,dim=3)=",i4)') lb, lbound(a,3)
write(*,'(" ub=ubound(a)=",4i4,"    ubound(a,dim=3)=",i4)') ub, ubound(a,3)
write(*,'(" lbound((a))=",4i4)') lbound( (a) )       ! (a) - не просто имя,
write(*,'(" ubound((a))=",4i4)') ubound( (a) )       !       а выражение
write(*,'(" lbound(a(:,:,:,:))=",4i4        )') lbound(a(:,:,:,:))
write(*,'(" ubound(a(:,:,:,:))=",4i4        )') ubound(a(:,:,:,:))
write(*,'(" lbound(a(:,:,-5:3,:))=",4i4     )') lbound(a(:,:,-5:3,:))
```

```
write(*,'(" ubound(a(:,:,-5:3,:))=",4i4     )') ubound(a(:,:,-5:3,:))
write(*,'(" lbound(a(-3,-4,-5:3,0))=",4i4   )') lbound(a(-3,-4,-5:3,0))
write(*,'(" ubound(a(-3,-4,-5:3,0))=",4i4   )') ubound(a(-3,-4,-5:3,0))
write(*,'(" lbound(a(-3:,-4,-5:3,0))=",4i4  )') lbound(a(-3:,-4,-5:3,0))
write(*,'(" ubound(a(-3:,-4,-5:3,0))=",4i4  )') ubound(a(-3:,-4,-5:3,0))
write(*,'(" lbound(a(-3:,-4:,-5:3,0))=",4i4 )') lbound(a(-3:,-4:,-5:3,0))
write(*,'(" ubound(a(-3:,-4:,-5:3,0))=",4i4 )') ubound(a(-3:,-4:,-5:3,0))
write(*,'(" lbound(a(-3:,-4:,-5:3,0:))=",4i4)') lbound(a(-3:,-4:,-5:3,0:))
write(*,'(" ubound(a(-3:,-4:,-5:3,0:))=",4i4)') ubound(a(-3:,-4:,-5:3,0:))
write(*,'(" lbound(a(-3:-3,-4:-2,-5:3,0:0))=",4i4)')&
                                        & lbound(a(-3:-3,-4:-2,-5:3,0:0))
write(*,'(" ubound(a(-3:-3,-4:-2,-5:3,0:0))=",4i4)')&
                                        & ubound(a(-3:-3,-4:-2,-5:3,0:0))
end program test_lubound
```

```
 lb=lbound(a)=  -3  -4  -5   0    lbound(a,dim=3)=  -5
 ub=ubound(a)=  -2  -1   4   7    ubound(a,dim=3)=   4
 lbound((a))=   1   1   1   1
 ubound((a))=   2   4  10   8
 lbound(a(:,:,:,:))=   1   1   1   1
 ubound(a(:,:,:,:))=   2   4  10   8
 lbound(a(:,:,-5:3,:))=   1   1   1   1
 ubound(a(:,:,-5:3,:))=   2   4   9   8
 lbound(a(-3,-4,-5:3,0))=   1
 ubound(a(-3,-4,-5:3,0))=   9
 lbound(a(-3:,-4,-5:3,0))=   1   1
 ubound(a(-3:,-4,-5:3,0))=   2   9
 lbound(a(-3:,-4:,-5:3,0))=   1   1   1
 ubound(a(-3:,-4:,-5:3,0))=   2   4   9
 lbound(a(-3:,-4:,-5:3,0:))=   1   1   1   1
 ubound(a(-3:,-4:,-5:3,0:))=   2   4   9   8
 lbound(a(-3:-3,-4:-2,-5:3,0:0))=   1   1   1   1
 ubound(a(-3:-3,-4:-2,-5:3,0:0))=   1   3   9   1
```

Обратите внимание! Если у функций **lbound** и **ubound** аргументом служит не просто имя массива, а выражение, то в качестве нижней границы выводится всегда **единица**, а в качестве верхней — количество элементов массива, согласованного с рангом и протяжённостью экстентов выражения, поданного в качестве аргумента.

7.2 Функции редукции массивов

Термин **редукция** (*reduction* — снижение, сокращение, приведение) в данном контексте означает, что на вход к функции в качестве аргумента подаётся массив, а на выходе получается значение с меньшим числом элементов чем имеется в массиве-аргументе (см., например, [2, 7]).

1. **all(mask [, dim])** — возвращает **.true.** (стандартного логического типа по умолчанию), если все элементы логического массива **mask** по заданному измерению **dim** имеют значение **.true.**, в противном случае возвращается **.false.**. При отсутствии необязательного аргумента **dim** анализируется полностью весь массив.

```
program test_all; implicit none               ! Программа выясняет:
integer, parameter :: n=5                     ! у всех ли элементов
integer :: a(n)=(/ 13, 103, 43, -1003, 73 /)  ! векторов A и B
integer :: b(n)=(/ 13, 103, 40, -1003, 73 /)  ! младшая цифра - тройка.
logical maska(n), maskb(n)
write(*,'(" Массив   a:    mod(a,10)=",5i3)')  mod(a,10)
maska=mod(a,10) == 3; write(*,'(20x,"maska=",5l3)') maska
if (all(maska))  then; write(*,1001)
                 else; write(*,1002)
endif
write(*,'(" Массив |a|:  mod(|a|,10)=",5i3)')  mod(abs(a),10)
maska=mod(abs(a),10) == 3; write(*,'(20x,"maska=",5l3)') maska
if (all(maska))  write(*,1001)
write(*,'(" Массив |b|:  mod(|b|,10)=",5i3)')  mod(abs(b),10)
maskb=mod(abs(b),10) == 3; write(*,'(20x,"maskb=",5l3)') maskb
                    if (.not.all(maskb))  write(*,1002)
1001 format(13x,' Младшая цифра ВСЕХ элементов массива равна 3')
1002 format(13x,' В массиве есть элемент с младшей цифрой отличной от 3')
end program test_all
```

```
 Массив   a:    mod(a,10)=  3  3  3 -3  3
                     maska=  T  T  T  F  T
              В массиве есть элемент с младшей цифрой отличной от 3
 Массив |a|:  mod(|a|,10)=  3  3  3  3  3
                     maska=  T  T  T  T  T
              Младшая цифра ВСЕХ элементов массива равна 3
 Массив |b|:  mod(|b|,10)=  3  3  0  3  3
                     maskb=  T  T  F  T  T
              В массиве есть элемент с младшей цифрой отличной от 3
```

Обратить внимание, что встроенная функция **mod**, находящая остаток, в случае отрицательности первого аргумента и остаток получает **отрицательным**.

2. **any(mask [, dim])** — возвращает **.true.** (стандартного логического типа по умолчанию), если хотя бы один элемент логического массива **mask** по заданному измерению **dim** имеет значение **.true.**, в противном случае возвращается **.false.**. При отсутствии необязательного аргумента анализируется полностью весь массив.

```
program test_any; implicit none               ! Программа выясняет: есть ли
integer, parameter :: n=5                     ! среди элементов вектора
integer :: a(n)=(/ 13, 103, 40, 1003, 73 /)   ! элемент, младшая цифра
logical, allocatable :: maska(:)              ! которого отлична от тройки.
allocate(maska(n))
write(*,'(" Массив   a( 5 )   mod(a,10)=",5i3)')  mod(a,10)
maska=mod(a,10) /= 3; write(*,'(23x,"maska=",5l3)') maska
if (any(maska))  then;  write(*,1002)
                 else;  write(*,1001)
endif
deallocate(maska)
write(*,'(" Сечение  a(1:2)   mod(a(1:2),10)=",5i3)')  mod(a(1:2),10)
allocate(maska(2))
maska=mod(a(1:2),10) /= 3; write(*,'(28x,"maska=",5l3)') maska
if (any(maska))  then;  write(*,1002)
                 else;  write(*,1001)
endif
1001 format(18x,' В  a(1:2) НЕТ элементов с младшей цифрой отличной от 3')
1002 format(18x,' В массиве ЕСТЬ  элемент с младшей цифрой отличной от 3')
end program test_any
 Массив   a( 5 )   mod(a,10)=  3  3  0  3  3
                       maska=  F  F  T  F  F
                   В массиве ЕСТЬ  элемент с младшей цифрой отличной от 3
 Сечение  a(1:2)   mod(a(1:2),10)=  3  3
                            maska=  F  F
                   В  a(1:2) НЕТ элементов с младшей цифрой отличной от 3
```

Обратите внимание:

1) Если закоментировать **deallocate(maska)**, то компиляция пройдёт, но при запуске исполнимого файла будет выдано сообщение:

Fortran runtime error: Attempting to allocate already allocated array.

2) Поэтому при необходимости использовать в качестве маски массив с прежним именем, но другим размером (и/или иными атрибутами), предварительно удаляем старый размещённый объект и, если хотим, сопоставляем новому объекту старое имя.

3. **count(mask [, dim])** — возвращает количество элементов логического массива **mask**, имеющих значение **.true.** по заданному измерению **dim**. При отсутствии необязательного аргумента **dim** анализируется весь массив. Программа **test_count**, приводимая ниже, помогает понять, как функция **count** получает результат для трёхмерного массива **b(2,3,4)** при наличии номера измерения.

```
program test_count; implicit none         ! Программа выясняет: сколько
integer :: a(4)=(/ 13, 103, 40, 1005 /)   ! элементов массива имеет в
integer i, j, k                           ! младшем разряде цифру 3.
integer :: b(2,3,4)= reshape( (/ 1, 2, 3, 4, 5, 6, 73, 8,&
&   93,10,11,12,31,43,13,63,17,53,19,20,33,22,23,24/), shape=(/2,3,4/))
character(12) :: txt(2,4)= reshape( (/  '   Первый   ','     лист   ',&
& '   Второй   ','     лист   ', '   Третий   ','     лист   ',&
& '   Четвёртый','     лист   '/), shape=(/2,4/))
write(*,'(" count(mod(a,10))==",i3)') count(mod(a,10)==3) ! Проверка для
write(*,'(" count(mod(b,10))==",i3)') count(mod(b,10)==3) ! всего массива
write(*,'(" a:",5i5)') a
write(*,'(" Размер массива a есть  size(a)=",i3)')  size(a)
write(*,'(" Форма  массива a есть shape(a)=",i3)') shape(a)
write(*,'(" b:")'); write(*,'( (3i4,a/3i4,a/3x,10("-")))') &
&        (((b(i,j,k),j=1,3),txt(i,k),i=1,2),k=1,4)
write(*,'(" Размер массива b есть  size(b)=",i3)')  size(b)
write(*,'(" Форма  массива b есть shape(b)=",3i3)') shape(b)
write(*,'(" Размер листа массива b есть size(b(:,:,1)=",5i3)')&
&                                              size(b(:,:,1))
write(*,'(" Форма  листа массива b есть shape(b)=",2i3/)') shape(b(:,:,1))
write(*,'(19x," dim=1.   Анализ столбцов текущего листа"/)')
write(*,'(" count(mod(b,10)==3,dim=1)=",i3)') count(mod(b,10)==3,dim=1)
write(*,*)
write(*,'(19x," dim=2.   Анализ строк текущего листа"/)')
write(*,'(" count(mod(b,10)==3,dim=2)",i3)')  count(mod(b,10)==3,dim=2)
write(*,'(/" Размер левой грани массива b есть  size(b(:,1,:)=",i3)')&
&                                                 size(b(:,1,:))
write(*,'(" Форма  левой грани массива b есть shape(b(:,1,:)=",2i3)')&
&                                                 shape(b(:,1,:))
write(*,'(" Содержимое  левой грани  массива b:")')
write(*,'(33x,4i4)') ((b(i,1,k),k=1,4),i=1,2)
write(*,'(" Содержимое среднего слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,2,k),k=1,4),i=1,2)
write(*,'(" Содержимое  правой грани массива b:")')
write(*,'(33x,4i4)') ((b(i,3,k),k=1,4),i=1,2); write(*,*)
write(*,'(19x," dim=3.   Анализ строк нормального сечения листов"/)')
write(*,'(" count(mod(b,10)==3,dim=3)",i3)')  count(mod(b,10)==3,dim=3)
end program test_count
```

```
count(mod(a,10))==  2
count(mod(b,10))==  9
a:   13  103   40 1005
Размер массива a есть  size(a)=  4
Форма  массива a есть shape(a)=  4
b:
  1   3   5   Первый
  2   4   6     лист
  ----------
 73  93  11   Второй
  8  10  12     лист
  ----------
 31  13  17   Третий
 43  63  53     лист
  ----------
 19  33  23   Четвёртый
 20  22  24     лист
  ----------
Размер массива b есть  size(b)= 24
Форма  массива b есть shape(b)=  2  3  4
Размер листа массива b есть size(b(:,:,1)=  6
Форма  листа массива b есть shape(b)=  2  3

                  dim=1.          Анализ столбцов текущего листа
                                !   Лист  Столбец  Цифра 3
count(mod(b,10)==3,dim=1)=  0   !     1      1      нет
count(mod(b,10)==3,dim=1)=  1   !     1      2           1
count(mod(b,10)==3,dim=1)=  0   !     1      3      нет
count(mod(b,10)==3,dim=1)=  1   !     2      1           1
count(mod(b,10)==3,dim=1)=  1   !     2      2           1
count(mod(b,10)==3,dim=1)=  0   !     2      3      нет
count(mod(b,10)==3,dim=1)=  1   !     3      1      нет
count(mod(b,10)==3,dim=1)=  2   !     3      2           2
count(mod(b,10)==3,dim=1)=  1   !     3      3           1
count(mod(b,10)==3,dim=1)=  0   !     4      1      нет
count(mod(b,10)==3,dim=1)=  1   !     4      2           1
count(mod(b,10)==3,dim=1)=  1   !     4      3           1

                  dim=2.          Анализ строк текущего листа
                                !   Лист  Строка   Цифра 3
count(mod(b,10)==3,dim=2)  1    !     1      1           1
count(mod(b,10)==3,dim=2)  0    !     1      2      нет
count(mod(b,10)==3,dim=2)  2    !     2      1           2
count(mod(b,10)==3,dim=2)  0    !     2      2      нет
count(mod(b,10)==3,dim=2)  1    !     3      1           1
count(mod(b,10)==3,dim=2)  3    !     3      2           3
count(mod(b,10)==3,dim=2)  2    !     4      1           2
count(mod(b,10)==3,dim=2)  0    !     4      2      нет
```

```
Размер левой грани массива b есть  size(b(:,1,:))=  8
Форма  левой грани массива b есть shape(b(:,1,:))=  2  4
Содержимое  левой грани  массива  b:
                                    1  73  31  19
                                    2   8  43  20
Содержимое среднего слоя массива b:
                                    3  93  13  33
                                    4  10  63  22
Содержимое  правой грани массива b:
                                    5  11  17  23
                                    6  12  53  24
                   dim=3.   Анализ строк нормального сечения листов
                                 !   Слой  Строка   Цифра 3
count(mod(b,10)==3,dim=3)  1     !  Левый    1            1
count(mod(b,10)==3,dim=3)  1     !  Левый    2            1
count(mod(b,10)==3,dim=3)  4     !  Средний  1            4
count(mod(b,10)==3,dim=3)  1     !  Средний  2            1
count(mod(b,10)==3,dim=3)  1     !  Правый   1            1
count(mod(b,10)==3,dim=3)  1     !  Правый   2            1
```

Приведённый результат снабжён поясняющим текстом (справа от восклицательного знака).

Функция **count** в случае многомерного массива и задания через параметр **dim** номера конкретного измерения возвращает в качестве результата не одно число, а столько чисел, сколько раз протяжённость требуемого измерения укладывается в исходном массиве. Например, при **dim=1**, что соответствует расположению элементов ФОРТРАН-массива в памяти по столбцам, оператор

write(*,*) count(mod(b,10)==3,dim=1)

выведет **12** чисел, получив результат для каждого из двенадцати двуэлементных столбцов, образующих исходный массив **b(2,3,4)**. При **dim=2**, что соответствует ФОРТРАН-строке оператор

write(*,*) count(mod(b,10)==3,dim=2)

выведет **8** чисел, получив результат для каждой из восьми трёхэлементных строк. Наконец, при **dim=3**, что соответствует в нашем примере нормальному сечению четырёх листов, выведется **6** чисел, по одному для каждой из шести четырёхэлементных строк, нормальных фасаду исходного массива **b(2,3,4)**.

4. **maxval(array [, dim] [,mask])** и **minval(array [, dim] [,mask])** — возвращают соответственно максимальное и минимальное из значений элементов массива **array** (целого или вещественного типов). Если при обращении к функции указан массив-маска **mask**, то максимальное (минимальное) значение будет выбираться из тех элементов **array**, для которых соответствующие элементы **mask** имеют значение **.true.**. При указании номера измерения **dim** выбор будет происходить для каждой его составляющей.

```
program test_maxval                          ! Программа демонстрирует
 implicit none                               ! работу функции   maxval
integer i, j, k
integer :: b(2,3,4)= reshape( (/ 1, 2, 3, 4, 5, 6, 73, 8,&
&   93,10,11,12,31,43,13,63,17,53,19,20,33,22,23,24/), shape=(/2,3,4/))
logical mask(2,3,4)
character(12) :: txt(2,4)= reshape( (/  '   Первый   ','     лист   ',&
&        '   Второй   ','     лист   ', '   Третий   ','     лист   ',&
&        '   Четвёртый','     лист   '/), shape=(/2,4/))
write(*,'(" b:")')
write(*,'( (3i4,a/3i4,a/3x,10("-")))') &
&              (((b(i,j,k),j=1,3),txt(i,k),i=1,2),k=1,4)
write(*,'(" Размер массива b есть  size(b)=",i3)')  size(b)
write(*,'(" Форма  массива b есть shape(b)=",3i3,/)') shape(b)
write(*,'(1x,"Работа maxval без dim и  без mask"/)')
write(*,'(" maxval(b)=",i3/)') maxval(b)
mask=mod(b,10)==3                                    !  Задание маски.
write(*,'(1x,"Работа maxval с mask=(mod(b,10)==3), но без dim:"/)')
write(*,'(" maxval(b,mask)=",i3/)') maxval(b,mask=mask)
write(*,'(1x,"Работа maxval с dim=1, но без mask"/)')
write(*,'(" maxval(b,dim=1)=",i3)') maxval(b,1); write(*,*)
write(*,'(1x,"Работа maxval с dim=2, но без mask"/)')
write(*,'(" maxval(b,dim=2)=",i3)') maxval(b,2)
write(*,*)
write(*,'(" Размер левого слоя массива b есть  size(b(:,1,:)=",5i3)')&
&                                                  size(b(:,1,:))
write(*,'(" Форма  левого слоя массива b есть shape(b(:,1,:)=",5i3)')&
&                                                 shape(b(:,1,:))
write(*,'(" Содержимое  левого слоя  массива b:")')
write(*,'(33x,4i4)') ((b(i,1,k),k=1,4),i=1,2)
write(*,'(" Содержимое среднего слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,2,k),k=1,4),i=1,2)
write(*,'(" Содержимое  правого слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,3,k),k=1,4),i=1,2); write(*,*)
write(*,*)
write(*,'(1x,"Работа maxval с dim=3, но без mask"/)')
write(*,'(" maxval(b,dim=3)=",i3)') maxval(b,3)
```

```
write(*,'(" Размер листа массива b есть size(b(:,:,1)=",5i3)')&
&                                                      size(b(:,:,1))
write(*,'(" Форма  листа массива b есть shape(b)=",5i3)') shape(b(:,:,1))
write(*,*)
write(*,'(1x,"Работа maxval с dim=1 и с mask:"/)')
write(*,'(" maxval(b,1,mask)=",i3)') maxval(b,1,mask)
write(*,*)
write(*,'(1x,"Работа maxval с dim=2 и с mask:"/)')
write(*,'(" maxval(b,2,mask)=",i3)') maxval(b,2,mask)
write(*,*)
write(*,'(" Размер левого слоя массива b есть  size(b(:,1,:)=",5i3)')&
&                                                       size(b(:,1,:))
write(*,'(" Форма  левого слоя массива b есть shape(b(:,1,:)=",5i3)')&
&                                                      shape(b(:,1,:))
write(*,'(" Содержимое  левого слоя  массива b:")')
write(*,'(33x,4i4)') ((b(i,1,k),k=1,4),i=1,2)
write(*,'(" Содержимое среднего слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,2,k),k=1,4),i=1,2)
write(*,'(" Содержимое  правого слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,3,k),k=1,4),i=1,2); write(*,*)
write(*,'(1x,"Работа maxval с dim=3 и с mask:"/)')
write(*,'(" maxval(b,3,mask)=",i3)') maxval(b,3,mask)
write(*,*)
end program test_maxval
```

```
 b:                              !   Результат работы test_maxval
   1   3   5   Первый
   2   4   6     лист
   ----------
  73  93  11   Второй
   8  10  12     лист
   ----------
  31  13  17   Третий
  43  63  53     лист
   ----------
  19  33  23   4етвёртый
  20  22  24     лист
   ----------
 Размер массива b есть  size(b)= 24
 Форма  массива b есть shape(b)=  2  3  4

 Работа maxval без dim и  без mask

 maxval(b)= 93

 Работа maxval с mask=(mod(b,10)==3), но без dim:
 maxval(b,mask)= 93
```

```
Работа maxval с dim=1, но без mask

maxval(b,dim=1)=  2
maxval(b,dim=1)=  4
maxval(b,dim=1)=  6
maxval(b,dim=1)= 73
maxval(b,dim=1)= 93
maxval(b,dim=1)= 12
maxval(b,dim=1)= 43
maxval(b,dim=1)= 63
maxval(b,dim=1)= 53
maxval(b,dim=1)= 20
maxval(b,dim=1)= 33
maxval(b,dim=1)= 24

Работа maxval с dim=2, но без mask

maxval(b,dim=2)=  5
maxval(b,dim=2)=  6
maxval(b,dim=2)= 93
maxval(b,dim=2)= 12
maxval(b,dim=2)= 31
maxval(b,dim=2)= 63
maxval(b,dim=2)= 33
maxval(b,dim=2)= 24

Размер левого слоя массива b есть  size(b(:,1,:)=  8
Форма  левого слоя массива b есть shape(b(:,1,:)=  2  4
Содержимое  левого слоя  массива b:
                                  1  73  31  19
                                  2   8  43  20
Содержимое среднего слоя массива b:
                                  3  93  13  33
                                  4  10  63  22
Содержимое  правого слоя массива b:
                                  5  11  17  23
                                  6  12  53  24


Работа maxval с dim=3, но без mask

maxval(b,dim=3)= 73
maxval(b,dim=3)= 43
maxval(b,dim=3)= 93
maxval(b,dim=3)= 63
maxval(b,dim=3)= 23
maxval(b,dim=3)= 53
```

```
Размер листа массива b есть size(b(:,:,1)=  6
Форма  листа массива b есть shape(b)=  2  3

Работа maxval с dim=1 и с mask:

maxval(b,1,mask)=***   !  В первом и третьем столбцах первого листа
maxval(b,1,mask)=  3   !  нет чисел с тройкой в младшем разряде.
maxval(b,1,mask)=***   !  Поэтому maxval и не ищет в них максимальное.
maxval(b,1,mask)= 73
maxval(b,1,mask)= 93
maxval(b,1,mask)=***   !  То же в третьем столбце второго листа
maxval(b,1,mask)= 43   !
maxval(b,1,mask)= 63   !
maxval(b,1,mask)= 53   !
maxval(b,1,mask)=***   !  и первом столбце четвёртого
maxval(b,1,mask)= 33
maxval(b,1,mask)= 23

Работа maxval с dim=2 и с mask:

maxval(b,2,mask)=  3
maxval(b,2,mask)=***   !  То же во второй строке первого листа,
maxval(b,2,mask)= 93
maxval(b,2,mask)=***   !  во второй строке второго листа
maxval(b,2,mask)= 13
maxval(b,2,mask)= 63
maxval(b,2,mask)= 33
maxval(b,2,mask)=***   !  и во второй строке чктвёртого.

Размер левого слоя массива b есть  size(b(:,1,:)=  8
Форма  левого слоя массива b есть shape(b(:,1,:)=  2  4
Содержимое  левого слоя  массива b:
                                1  73  31  19
                                2   8  43  20
Содержимое среднего слоя массива b:
                                3  93  13  33
                                4  10  63  22
Содержимое  правого слоя массива b:
                                5  11  17  23
                                6  12  53  24
Работа maxval с dim=3 и с mask:

maxval(b,3,mask)= 73   !  В каждой строке третьего измерения
maxval(b,3,mask)= 43   !  встречается число с тройкой в
maxval(b,3,mask)= 93   !  разряде единиц. Поэтому при dim=3
maxval(b,3,mask)= 63   !  maxval среди них находит максимальное
maxval(b,3,mask)= 23   !
maxval(b,3,mask)= 53   !
```

```
b:                                      ! Результат работы предыдущей
  1   3   5   Первый                    ! программы после замены
  2   4   6     лист                    ! в её исходном тексте
  ----------                            ! имени  maxval  на  minval.
 73  93  11   Второй
  8  10  12     лист
  ----------
 31  13  17   Третий
 43  63  53     лист
  ----------
 19  33  23   Четвёртый
 20  22  24     лист
  ----------
Размер массива b есть  size(b)= 24
Форма  массива b есть shape(b)=  2  3  4

Работа minval без dim и  без mask
minval(b)=  1

Работа minval с mask=(mod(b,10)==3), но без dim:
minval(b,mask)=  3

Работа minval с dim=1, но без mask
minval(b,dim=1)=  1
minval(b,dim=1)=  3
minval(b,dim=1)=  5
minval(b,dim=1)=  8
minval(b,dim=1)= 10
minval(b,dim=1)= 11
minval(b,dim=1)= 31
minval(b,dim=1)= 13
minval(b,dim=1)= 17
minval(b,dim=1)= 19
minval(b,dim=1)= 22
minval(b,dim=1)= 23

Работа minval с dim=2, но без mask

minval(b,dim=2)=  1
minval(b,dim=2)=  2
minval(b,dim=2)= 11
minval(b,dim=2)=  8
minval(b,dim=2)= 13
minval(b,dim=2)= 43
minval(b,dim=2)= 19
minval(b,dim=2)= 20
```

```
Размер левого слоя массива b есть  size(b(:,1,:)=  8
Форма  левого слоя массива b есть shape(b(:,1,:)=  2  4
Содержимое  левого слоя  массива b:
                                  1  73  31  19
                                  2   8  43  20
Содержимое среднего слоя массива b:
                                  3  93  13  33
                                  4  10  63  22
Содержимое  правого слоя массива b:
                                  5  11  17  23
                                  6  12  53  24
Работа minval с dim=3, но без mask

minval(b,dim=3)=  1
minval(b,dim=3)=  2
minval(b,dim=3)=  3
minval(b,dim=3)=  4
minval(b,dim=3)=  5
minval(b,dim=3)=  6
Размер листа массива b есть size(b(:,:,1)=  6
Форма  листа массива b есть shape(b)=  2  3

Работа minval с dim=1 и с mask:

minval(b,1,mask)=***
minval(b,1,mask)=  3
minval(b,1,mask)=***
minval(b,1,mask)= 73
minval(b,1,mask)= 93
minval(b,1,mask)=***
minval(b,1,mask)= 43
minval(b,1,mask)= 13
minval(b,1,mask)= 53
minval(b,1,mask)=***
minval(b,1,mask)= 33
minval(b,1,mask)= 23

Работа minval с dim=2 и с mask:

minval(b,2,mask)=  3
minval(b,2,mask)=***
minval(b,2,mask)= 73
minval(b,2,mask)=***
minval(b,2,mask)= 13
minval(b,2,mask)= 43
minval(b,2,mask)= 23
minval(b,2,mask)=***
```

```
Размер левого слоя массива b есть  size(b(:,1,:)=  8
Форма  левого слоя массива b есть shape(b(:,1,:)=  2  4
Содержимое  левого слоя  массива b:
                                  1  73  31  19
                                  2   8  43  20
Содержимое среднего слоя массива b:
                                  3  93  13  33
                                  4  10  63  22
Содержимое  правого слоя массива b:
                                  5  11  17  23
                                  6  12  53  24
Работа minval с dim=3 и с mask:
minval(b,3,mask)= 73
minval(b,3,mask)= 43
minval(b,3,mask)=  3
minval(b,3,mask)= 63
minval(b,3,mask)= 23
minval(b,3,mask)= 53
```

Обратите внимание, что присвоить значение функций **maxval** и **minval** переменной целого типа можно лишь в том случае, если их результат не является массивом.

```
program test_maxmin; implicit none            ! Программа может
integer :: a(4)=(/ 13, 103, 40, 1005 /)       ! продемонстрировать, что
integer :: k, b(2,3,4)= reshape(&             ! значения maxval, minval
 & (/ 1, 2, 3, 4, 5, 6, 73, 8,&               ! присваивать целой
 & 93,10,11,12,31,43,13,63,17,&               ! переменной можно лишь в
 & 53,19,20,33,22,23,24/),shape=(/2,3,4/))    ! результат не является
k=maxval(a);   write(*,*) ' k=',k             ! массивом.
k=maxval(a,1); write(*,*) ' k=',k             !
k=maxval(b);   write(*,*) ' k=',k             !
k=maxval(b(:,1,1)); write(*,*) ' k=',k        !
write(*,*) 'shape(b(1,:,:))=' shape(b(1,:,:)) ! Так раскомментирование
write(*,*) 'shape(b(:,1,:))=' shape(b(:,1,:)) ! любой из трёх последних
write(*,*) 'shape(b(:,:,1))=' shape(b(:,:,1)) ! строк этой программы к
! k=maxval(b,1);                              ! приведёт выводу сообщения:
! k=maxval(b,2);                              ! ошибка   Incompatible
! k=maxval(b,3);                              ! ranks 0 and 2 in assignment
end program test_maxmin
```

Дело в том, что в случае трёхмерного массива **b** указание при вызове **maxval** номера измерения, например, **1** заставит функцию искать в качестве результата наибольшие значения по каждому из 12 столбцов матрицы размером $\mathbf{3} \times \mathbf{4}$.

5. **product(array [,dim], [,mask])** — при отсутствии необязательных аргументов возвращает произведение значений элементов массива (целого или вещественного типа). При наличии одного дополнительного аргумента **dim** (номера измерения) возвращает произведение элементов массива по указанному измерению. Аргумент **mask** обеспечивает возможный отбор перемножаемых элементов.

```
program test_product                          ! Программа демонстрирует
 implicit none                                ! работу функции   product.
integer i, j, k
real(4) :: b(2,3,4)= reshape( (/ 1, 2, 3, 4, 5, 6, 73, 8,&
&   93,10,11,12,31,43,13,63,17,53,19,20,33,22,23,24/), shape=(/2,3,4/))
logical mask(2,3,4)
character(12) :: txt(2,4)= reshape( (/  '   Первый   ','     лист   ',&
&        '   Второй   ','     лист   ', '   Третий   ','     лист   ',&
&        '   Четвёртый','     лист   '/), shape=(/2,4/))
write(*,'(" b:")')
write(*,'( (3f5.1,a/3f5.1,a/3x,10("-")))') &
&        (((b(i,j,k),j=1,3),txt(i,k),i=1,2),k=1,4)
write(*,'(" Размер массива b есть  size(b)=",i3)')  size(b)
write(*,'(" Форма  массива b есть shape(b)=",3i3,/)') shape(b)
write(*,'(1x,"Работа product без dim и  без mask"/)')
write(*,'(" product(b)=",e15.7/)') product(b)
mask=mod(int(b),10)==3                              !  Задание маски.
write(*,'(1x,"Работа product с mask=(mod(b,10)==3), но без dim:"/)')
write(*,'(" product(b,mask)=",e15.7/)') product(b,mask=mask)
write(*,'(1x,"Работа product с dim=1, но без mask"/)')
write(*,'(" product(b,dim=1)=",e15.7)') product(b,1)
write(*,*)
write(*,'(1x,"Работа product с dim=2, но без mask"/)')
write(*,'(" product(b,dim=2)=",e15.7)') product(b,2)
write(*,*)
write(*,'(" Размер левого слоя массива b есть  size(b(:,1,:)=",5i3)')&
&                                                 size(b(:,1,:))
write(*,'(" Форма  левого слоя массива b есть shape(b(:,1,:)=",5i3)')&
&                                                shape(b(:,1,:))
write(*,'(" Содержимое  левого слоя  массива b:")')
write(*,'(3x,4e15.7)') ((b(i,1,k),k=1,4),i=1,2)
write(*,'(" Содержимое среднего слоя массива b:")')
write(*,'(3x,4e15.7)') ((b(i,2,k),k=1,4),i=1,2)
write(*,'(" Содержимое  правого слоя массива b:")')
write(*,'(3x,4e15.7)') ((b(i,3,k),k=1,4),i=1,2); write(*,*)
write(*,*)
write(*,'(1x,"Работа product с dim=3, но без mask"/)')
write(*,'(" product(b,dim=3)=",e15.7)') product(b,3)
write(*,'(" Размер листа массива b есть size(b(:,:,1)=",5i3)')&
```

```
&                                                              size(b(:,:,1))
write(*,'(" Форма  листа массива b есть shape(b)=",5i3)') shape(b(:,:,1))
write(*,*)
write(*,'(1x,"Работа product c dim=1 и c mask:"/)')
write(*,'(" product(b,1,mask)=",e15.7)') product(b,1,mask)
write(*,*)
write(*,'(1x,"Работа product c dim=2 и c mask:"/)')
write(*,'(" product(b,2,mask)=",e15.7)') product(b,2,mask)
write(*,*)
write(*,'(" Размер левого слоя массива b есть  size(b(:,1,:)=",5i3)')&
&                                                               size(b(:,1,:))
write(*,'(" Форма  левого слоя массива b есть shape(b(:,1,:)=",5i3)')&
&                                                              shape(b(:,1,:))
write(*,'(" Содержимое  левого слоя  массива b:")')
write(*,'(33x,4f5.1)') ((b(i,1,k),k=1,4),i=1,2)
write(*,'(" Содержимое среднего слоя массива b:")')
write(*,'(33x,4f5.1)') ((b(i,2,k),k=1,4),i=1,2)
write(*,'(" Содержимое  правого слоя массива b:")')
write(*,'(33x,4f5.1)') ((b(i,3,k),k=1,4),i=1,2); write(*,*)
write(*,'(1x,"Работа product c dim=3 и c mask:"/)')
write(*,'(" product(b,3,mask)=",e15.7)') product(b,3,mask)
write(*,*)
end program test_product
```

```
 b:                                ! Результат работы test_product
  1.0  3.0  5.0   Первый
  2.0  4.0  6.0     лист
    ----------
 73.0 93.0 11.0   Второй
  8.0 10.0 12.0     лист
    ----------
 31.0 13.0 17.0   Третий
 43.0 63.0 53.0     лист
    ----------
 19.0 33.0 23.0   4етвёртый
 20.0 22.0 24.0     лист
    ----------
 Размер массива b есть  size(b)= 24
 Форма  массива b есть shape(b)=  2  3  4

 Работа product без dim и  без mask

 product(b)=  0.7732151E+28

 Работа product c mask=(mod(b,10)==3), но без dim:

 product(b,mask)=  0.2885340E+14
```

```
Работа product с dim=1, но без mask

product(b,dim=1)=  0.2000000E+01
product(b,dim=1)=  0.1200000E+02
product(b,dim=1)=  0.3000000E+02
product(b,dim=1)=  0.5840000E+03
product(b,dim=1)=  0.9300000E+03
product(b,dim=1)=  0.1320000E+03
product(b,dim=1)=  0.1333000E+04
product(b,dim=1)=  0.8190000E+03
product(b,dim=1)=  0.9010000E+03
product(b,dim=1)=  0.3800000E+03
product(b,dim=1)=  0.7260000E+03
product(b,dim=1)=  0.5520000E+03

Работа product с dim=2, но без mask

product(b,dim=2)=  0.1500000E+02
product(b,dim=2)=  0.4800000E+02
product(b,dim=2)=  0.7467900E+05
product(b,dim=2)=  0.9600000E+03
product(b,dim=2)=  0.6851000E+04
product(b,dim=2)=  0.1435770E+06
product(b,dim=2)=  0.1442100E+05
product(b,dim=2)=  0.1056000E+05

Размер левого слоя массива b есть  size(b(:,1,:)=  8
Форма  левого слоя массива b есть shape(b(:,1,:)=  2  4
Содержимое  левого слоя  массива b:
    0.1000000E+01  0.7300000E+02  0.3100000E+02  0.1900000E+02
    0.2000000E+01  0.8000000E+01  0.4300000E+02  0.2000000E+02
Содержимое среднего слоя массива b:
    0.3000000E+01  0.9300000E+02  0.1300000E+02  0.3300000E+02
    0.4000000E+01  0.1000000E+02  0.6300000E+02  0.2200000E+02
Содержимое  правого слоя массива b:
    0.5000000E+01  0.1100000E+02  0.1700000E+02  0.2300000E+02
    0.6000000E+01  0.1200000E+02  0.5300000E+02  0.2400000E+02


Работа product с dim=3, но без mask

product(b,dim=3)=  0.4299700E+05
product(b,dim=3)=  0.1376000E+05
product(b,dim=3)=  0.1196910E+06
product(b,dim=3)=  0.5544000E+05
product(b,dim=3)=  0.2150500E+05
product(b,dim=3)=  0.9158400E+05
```

```
Размер листа массива b есть size(b(:,:,1)=  6
Форма  листа массива b есть shape(b)=  2  3

Работа product с dim=1 и с mask:

product(b,1,mask)=  0.1000000E+01      ! <-- PRODUCT полагает,
product(b,1,mask)=  0.3000000E+01      !     что, если среди
product(b,1,mask)=  0.1000000E+01      ! <-- элементов НЕТ,
product(b,1,mask)=  0.7300000E+02      !     удовлетворяющих
product(b,1,mask)=  0.9300000E+02      !     условию маски,
product(b,1,mask)=  0.1000000E+01      ! <-- то искомое
product(b,1,mask)=  0.4300000E+02      !     произведение
product(b,1,mask)=  0.8190000E+03      !     равно
product(b,1,mask)=  0.5300000E+02      !
product(b,1,mask)=  0.1000000E+01      ! <-- Е Д И Н И Ц Е.
product(b,1,mask)=  0.3300000E+02
product(b,1,mask)=  0.2300000E+02

Работа product с dim=2 и с mask:

product(b,2,mask)=  0.3000000E+01      ! Аналогично и для dim=2:
product(b,2,mask)=  0.1000000E+01      ! <--
product(b,2,mask)=  0.6789000E+04      !
product(b,2,mask)=  0.1000000E+01      ! <--
product(b,2,mask)=  0.1300000E+02      !
product(b,2,mask)=  0.1435770E+06      !
product(b,2,mask)=  0.7590000E+03      !
product(b,2,mask)=  0.1000000E+01      ! <--

Размер левого слоя массива b есть  size(b(:,1,:)=  8
Форма  левого слоя массива b есть shape(b(:,1,:)=  2  4
Содержимое  левого слоя  массива b:
                                1.0 73.0 31.0 19.0
                                2.0  8.0 43.0 20.0
Содержимое среднего слоя массива b:
                                3.0 93.0 13.0 33.0
                                4.0 10.0 63.0 22.0
Содержимое  правого слоя массива b:
                                5.0 11.0 17.0 23.0
                                6.0 12.0 53.0 24.0
Работа product с dim=3 и с mask:

product(b,3,mask)=  0.7300000E+02      ! В случае dim=3
product(b,3,mask)=  0.4300000E+02      ! в каждой строке
product(b,3,mask)=  0.1196910E+06      ! исследуемой секции
product(b,3,mask)=  0.6300000E+02      ! есть хотя бы один
product(b,3,mask)=  0.2300000E+02      ! элемент с 3
product(b,3,mask)=  0.5300000E+02      ! в младшей цифре.
```

6. **sum(array [,dim], [,mask])** — при отсутствии необязательных аргументов возвращает сумму значений элементов массива (целого или вещественного типа). При наличии одного дополнительного аргумента **dim** (номера измерения) возвращает сумму элементов массива по указанному измерению. Аргумент **mask** обеспечивает возможный отбор суммируемых элементов.

```
program test_sum;  implicit none   !    Демонстрация работы функции sum.
integer i, j, k
integer :: b(2,3,4)= reshape( (/ 1, 2, 3, 4, 5, 6, 73, 8,&
&   93,10,11,12,31,43,13,63,17,53,19,20,33,22,23,24/), shape=(/2,3,4/))
logical mask(2,3,4)
character(12) :: txt(2,4)= reshape( (/  '   Первый   ','     лист   ',&
&        '   Второй   ','     лист   ', '   Третий   ','     лист   ',&
&        '   Четвёртый','     лист   '/), shape=(/2,4/))
write(*,'(" b:")'); write(*,'( (3i4,a/3i4,a/3x,10("-")))') &
&                           (((b(i,j,k),j=1,3),txt(i,k),i=1,2),k=1,4)
write(*,'(" Размер массива b есть  size(b)=",i3 )')  size(b)
write(*,'(" Форма  массива b есть shape(b)=",3i3)') shape(b)
write(*,'(" Работа sum без dim и без mask:",23("."),"  sum(b)=",i5)') sum(b)
mask=mod(int(b),10)==3                                  !<--= Задание маски.
write(*,'(" Работа sum без dim, но с mask=(mod(b,10)==3):...",&
&                           " sum(b,mask)=",i5)') sum(b,mask=mask)
write(*,'(" Работа sum с dim=1, но без mask:")')
write(*,'(" sum(b,dim=1)=",i5)') sum(b,1);                 write(*,*)
write(*,'(" Работа sum с dim=2, но без mask:")')
write(*,'(" sum(b,dim=2)=",i5)') sum(b,2);                 write(*,*)
write(*,'(" Размер левого слоя массива b есть  size(b(:,1,:)=",5i3)')&
&                                                  size(b(:,1,:))
write(*,'(" Форма  левого слоя массива b есть shape(b(:,1,:)=",5i3)')&
&                                                 shape(b(:,1,:))
write(*,'(" Содержимое  левого слоя  массива b:")')
write(*,'(33x,4i4)') ((b(i,1,k),k=1,4),i=1,2)
write(*,'(" Содержимое среднего слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,2,k),k=1,4),i=1,2)
write(*,'(" Содержимое  правого слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,3,k),k=1,4),i=1,2);         !   write(*,*)
write(*,'(" Работа sum с dim=3, но без mask:")')
write(*,'(" sum(b,dim=3)=",i5)') sum(b,3);
write(*,'(/" Размер листа массива b есть size(b(:,:,1)=",5i3)')&
&                                                 size(b(:,:,1))
write(*,'(" Форма  листа массива b есть shape(b)=",5i3)')&
&                                        shape(b(:,:,1)); write(*,*)
write(*,'(1x,"Работа sum с dim=1 и с mask:"/)')
```

```
write(*,'(" sum(b,1,mask)=",i5)') sum(b,1,mask);           write(*,*)
write(*,'(1x,"Работа sum с dim=2 и с mask:"/)')
write(*,'(" sum(b,2,mask)=",i5)') sum(b,2,mask);           write(*,*)
write(*,'(" Размер левого слоя массива b есть  size(b(:,1,:)=",5i3)')&
&                                                           size(b(:,1,:))
write(*,'(" Форма  левого слоя массива b есть shape(b(:,1,:)=",5i3)')&
&                                                          shape(b(:,1,:))
write(*,'(" Содержимое  левого слоя  массива b:")')
write(*,'(33x,4i4)') ((b(i,1,k),k=1,4),i=1,2)
write(*,'(" Содержимое среднего слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,2,k),k=1,4),i=1,2)
write(*,'(" Содержимое  правого слоя массива b:")')
write(*,'(33x,4i4)') ((b(i,3,k),k=1,4),i=1,2); write(*,*)
write(*,'(1x,"Работа sum с dim=3 и с mask:"/)')
write(*,'(" sum(b,3,mask)=",i5)') sum(b,3,mask)
write(*,*)
end program test_sum
```

```
 b:
   1   3   5   Первый
   2   4   6     лист
   ----------
  73  93  11   Второй
   8  10  12     лист
   ----------
  31  13  17   Третий
  43  63  53     лист
   ----------
  19  33  23   Четвёртый
  20  22  24     лист
   ----------
 Размер массива b есть  size(b)= 24
 Форма  массива b есть shape(b)=  2  3  4
 Работа sum без dim и без mask:..................... sum(b)=  589
 Работа sum без dim, но с mask=(mod(b,10)==3):... sum(b,mask)=  397
 Работа sum с dim=1, но без mask:
 sum(b,dim=1)=    3
 sum(b,dim=1)=    7
 sum(b,dim=1)=   11
 sum(b,dim=1)=   81
 sum(b,dim=1)=  103
 sum(b,dim=1)=   23
 sum(b,dim=1)=   74
 sum(b,dim=1)=   76
 sum(b,dim=1)=   70
 sum(b,dim=1)=   39
 sum(b,dim=1)=   55
 sum(b,dim=1)=   47
```

```
Работа sum с dim=2, но без mask:
sum(b,dim=2)=    9
sum(b,dim=2)=   12
sum(b,dim=2)=  177
sum(b,dim=2)=   30
sum(b,dim=2)=   61
sum(b,dim=2)=  159
sum(b,dim=2)=   75
sum(b,dim=2)=   66

Размер левого слоя массива b есть  size(b(:,1,:)=  8
Форма  левого слоя массива b есть shape(b(:,1,:)=  2  4
Содержимое  левого слоя  массива b:
                                1  73  31  19
                                2   8  43  20
Содержимое среднего слоя массива b:
                                3  93  13  33
                                4  10  63  22
Содержимое  правого слоя массива b:
                                5  11  17  23
                                6  12  53  24
Работа sum с dim=3, но без mask:
sum(b,dim=3)=  124
sum(b,dim=3)=   73
sum(b,dim=3)=  142
sum(b,dim=3)=   99
sum(b,dim=3)=   56
sum(b,dim=3)=   95

Размер листа массива b есть size(b(:,:,1)=  6
Форма  листа массива b есть shape(b)=  2  3

Работа sum с dim=1 и с mask:

sum(b,1,mask)=    0
sum(b,1,mask)=    3
sum(b,1,mask)=    0
sum(b,1,mask)=   73
sum(b,1,mask)=   93
sum(b,1,mask)=    0
sum(b,1,mask)=   43
sum(b,1,mask)=   76
sum(b,1,mask)=   53
sum(b,1,mask)=    0
sum(b,1,mask)=   33
sum(b,1,mask)=   23
```

```
Работа sum с dim=2 и с mask:

sum(b,2,mask)=    3
sum(b,2,mask)=    0
sum(b,2,mask)=  166
sum(b,2,mask)=    0
sum(b,2,mask)=   13
sum(b,2,mask)=  159
sum(b,2,mask)=   56
sum(b,2,mask)=    0

Размер левого слоя массива b есть  size(b(:,1,:))=  8
Форма  левого слоя массива b есть shape(b(:,1,:))=  2  4
Содержимое  левого слоя  массива b:
                                1  73  31  19
                                2   8  43  20
Содержимое среднего слоя массива b:
                                3  93  13  33
                                4  10  63  22
Содержимое  правого слоя массива b:
                                5  11  17  23
                                6  12  53  24

Работа sum с dim=3 и с mask:

sum(b,3,mask)=   73
sum(b,3,mask)=   43
sum(b,3,mask)=  142
sum(b,3,mask)=   63
sum(b,3,mask)=   23
sum(b,3,mask)=   53
```

7.3 Функции умножения векторов и матриц

1. **dot_product(vector_a,vector_b)** — скалярное произведение векторов:

```
program tdot; implicit none; integer :: ia(3)=(/1,2,3/), ib(3)=(/4,5,6/)
real    ::  a(3)=(/1.0,2.0,3.0/), b(3)=(/4.0,5.0,6.0/)
complex :: ca(3)=(/(1.0,0.0),(2.0,0.0),(3.0,0.0)/)
complex :: cb(3)=(/(4.0,0.0),(5.0,0.0),(6.0,0.0)/)
logical :: la(3)=(/.false.,.true.,.false./)
logical :: lb(3)=(/.false.,.true.,.false./)
logical :: lc(3)=(/.true.,.false.,.true./)
write(*,'(" integer: ia=",3g4.0)') ia
write(*,'("          ib=",3g4.0)') ib
write(*,'("          dot_product(ia,ib)=",i4)') dot_product(ia,ib)
write(*,'("    real: a=",3g15.7)') a
write(*,'("          b=",3g15.7)') b
write(*,'("          dot_product( a, b)=",e15.7)') dot_product( a, b)
write(*,'(" complex: ca=",6g7.2)') ca
write(*,'("          cb=",6g7.2)') cb
write(*,'("          dot_product(ca,cb)=",2g15.7)') dot_product(ca,cb)
write(*,'(" logical: la=",3g7.0,10x,"  la=",3g7.0)') la,la
write(*,'("          lb=",3g7.0,10x,"  lc=",3g7.0)') lb,lc
write(*,'("          dot_product(la,lb)=",g5.0,10x,&
&"  dot_product(la,lc)=",g5.0)') dot_product(la,lb), dot_product(la,lc)
write(*,'(" complex: ca=",6g7.2)') ca
write(*,'(" integer: ib=",3g7.0)') ib
write(*,'("          dot_product(ca,ib)=",2g15.7)') dot_product(ca,ib)
end
```

```
 integer: ia=   1   2   3
          ib=   4   5   6
          dot_product(ia,ib)=  32
    real: a=   1.000000       2.000000       3.000000
          b=   4.000000       5.000000       6.000000
          dot_product( a, b)=  0.3200000E+02
 complex: ca=1.0    0.0    2.0    0.0    3.0    0.0
          cb=4.0    0.0    5.0    0.0    6.0    0.0
          dot_product(ca,cb)=   32.00000       0.000000
 logical: la=      F      T      F            la=      F      T      F
          lb=      F      T      F            lc=      T      F      T
          dot_product(la,lb)=    T            dot_product(la,lc)=    F
 complex: ca=1.0    0.0    2.0    0.0    3.0    0.0
 integer: ib=      4      5      6
          dot_product(ca,ib)=   32.00000       0.000000
```

Для векторов логического типа **dot_product** получает результат дизъюнкции поэлементных конъюнкций.

2. **matmul(matrix_a,matrix_b)** — умножение или двух матриц, или матрицы на вектора, или вектора на матрицу:

$$\mathbf{rab(i,j)} = \sum_{\mathbf{l=1}}^{\mathbf{k}} \mathbf{a(i,l)} * \mathbf{b(l,j)}; \quad \mathbf{i = 1(1)n}; \quad \mathbf{j = 1(1)m}$$

$$\mathbf{rac(i)} = \sum_{\mathbf{j=1}}^{\mathbf{m}} \mathbf{a(i,j)} * \mathbf{c(j)}; \quad \mathbf{i = 1(1)n}$$

$$\mathbf{rca(j)} = \sum_{\mathbf{i=1}}^{\mathbf{k}} \mathbf{c(i)} * \mathbf{a(i,j)}; \quad \mathbf{j = 1(1)m}$$

```
program tmatmul; implicit none
integer :: a(2,3)=reshape((/1,2,3,4,5,6/), shape=(/2,3/)), i, j
integer :: b(3,2)=reshape((/1,2,3,4,5,6/), shape=(/3,2/)), rab(2,2)
integer :: c(2)=(/1,2/), d(3)=(/1,2,3/)
write(*,'(" a: ",3g3.0/4x,3g3.0)') ((a(i,j),j=1,3),i=1,2)
write(*,'(" b: ",2(2g3.0/4x)2g3.0))')      ((b(i,j),j=1,2),i=1,3)
rab=matmul(a,b)
write(*,'(" matmul(a,b)=",(2g3.0/13x,2g3.0))') matmul(a,b)
write(*,'("         rab=",2(2g3.0/13x))') ((rab(i,j),j=1,2),i=1,2)
write(*,'(" a: ",3g3.0/"    ",3g3.0)') ((a(i,j),j=1,3),i=1,2)
write(*,'(" d: ",3g3.0)') d
write(*,'(" matmul(a,d)=",2g3.0)') matmul(a,d)
write(*,'(/" c: ",2g3.0)') c
write(*,'(" a: ",3g3.0/"    ",3g3.0)') ((a(i,j),j=1,3),i=1,2)
write(*,'(" matmul(c,a)=",3g3.0))') matmul(c,a)
end
```

```
 a:   1  3  5        |  a:   1  3  5        |  c:   1  2
      2  4  6        |       2  4  6        |
                     |                      |
 b:   1  4           |  d:   1  2  3        |  a:   1  3  5
      2  5           |                      |       2  4  6
      3  6           |                      |
 matmul(a,b)= 22 28  |  matmul(a,d)= 22 28  |  matmul(c,a)=  5 11 17
              49 64  |                      |
         rab= 22 49  |                      |
              28 64  |                      |
```

Помним, что умножение матриц определяется лишь для согласованных матриц (т.е. число столбцов первой должно равняться числу строк второй). Элемент матрицы-произведения, стоящий на пересечении **i**-ой строки и **j**-ого столбца, равен сумме произведений элементов **i**-ой строки первой матрицы на соответствующие элементы **j**-ого столбца второй.

matmul *ЧУВСТВУЕТ* отсутствие согласованности матриц:

```
program tmat1; implicit none
integer :: a(2,3)=reshape((/1,2,3,4,5,6/), shape=(/2,3/))
integer :: b(2,3)=reshape((/1,2,3,4,5,6/), shape=(/2,3/))
integer, allocatable :: f(:,:), g(:,:), h(:,:), x(:,:)
integer ier, i, j, l, m, n; character(1) sn; character(20) sf
! write(*,*) matmul(a,b)
write(*,'("Работа matmul с динамическими массивами")')
l=3; m=4; n=5
allocate(f(l,m),stat=ier); if (ier.ne.0) write(*,*) 'Не размещается f'
allocate(g(m,n),stat=ier); if (ier.ne.0) write(*,*) 'Не размещается g'
allocate(h(l,n),stat=ier); if (ier.ne.0) write(*,*) 'Не размещается h'
allocate(x(n,m),stat=ier); if (ier.ne.0) write(*,*) 'Не размещается h'
do i=1,l; f(i,:)=(/(   i+j,j=1,m)/); enddo
do i=1,m; g(i,:)=(/(10*i+j,j=1,n)/); enddo;      h=matmul(f,g)
write(*,'(a,i1,a,4i6)') ('f(',i,')=',f(i,:),i=1,l); write(*,*)
write(*,'(a,i1,a,5i6)') ('g(',i,')=',g(i,:),i=1,m); write(*,*)
write(*,'(a,i1,a,5i6)') ('h(',i,')=',h(i,:),i=1,l); write(*,*)
write(sn,'(i1)') n; sf='('//sn//'(i6)'//')'; write(*,sf)  (h(i,:),i=1,l)
h=matmul(g,f)
end
```

Так, раскомментировав в **tmat1** строку **!write(*,*) matmul(a,b)**, в случае **несогласованных** статических массивов **a** и **b** получим ещё на шаге компиляции сообщение:

```
write(*,*) matmul(a,b)
                  1
ошибка: Different shape on dimension 2 for argument 'matrix_a'
and dimension 1 for argument 'matrix_b' at (1) for intrinsic matmul
```

А в случае **несогласованных** размещаемых массивов при вызове **matmul(g,f)** (см. предпоследний оператор программы) получим соответствующее сообщение и на шаге выполнения:

```
Fortran runtime error: dimension of array B incorrect in MATMUL intrinsic
```

Полезно ещё раз обратить внимание на возможность использования:

1) сечений для построкового вывода матриц;

2) строковой переменной (в **tmat1** переменная **sf**) для оргранииации динамического повторителя (в данном случае для вывода нужного количества столбцов матрицы) при формировке формата вывода.

matmul и **dot_product** осуществляют лишь формальное приведение числовых типов друг к другу, т.е. данное типа **real(4)**, содержащее лишь 7-8 десятичных цифр, после преобразования в тип **real(8)** будет переписано по формату **real(8)**, но из шестнадцати соответствующих цифр результата, верными будут лишь 7-8 старших.

```
program tmat2; implicit none; real(4), parameter :: c13=1.0/3.0
real(8) :: a(2,2), c(2,2); real(4) :: b(2,2); integer i, j
a=1d0; b=c13; c=matmul(a,b)
write(*,'(14x,"a",18x,"b",15x,"matmul(a,b)")');
write(*,'(d25.16,e15.7,d25.16)') ((a(i,j),b(i,j),c(i,j),j=1,2),i=1,2)
end
```

```
              a                   b               matmul(a,b)
   0.1000000000000000D+01  0.3333333E+00   0.6666666865348816D+00
   0.1000000000000000D+01  0.3333333E+00   0.6666666865348816D+00
   0.1000000000000000D+01  0.3333333E+00   0.6666666865348816D+00
   0.1000000000000000D+01  0.3333333E+00   0.6666666865348816D+00
```

В [2] написано, что **matmul** при работе с матрицами и векторами типа **logical** выполняет операцию логического умножения. Точнее, результатом оказывается матрица элементы которой находятся согласно правилу перемножения числовых матриц, в котором сложение заменено дизъюнкцией, а умножение — конъюнкцией.

```
program tmat3; implicit none
logical :: a(2,2)=reshape((/.true.,.true.,.false.,.false./),(/2,2/))
logical :: b(2,2)=reshape((/.true.,.false.,.true.,.false./),(/2,2/))
logical c(2,2), d(2,2),e(2,2); integer i, j
write(*,*) ' a=',a; write(*,*) ' b=',b; d=matmul(a,b); e=a.and.b
write(*,'(1x,"i",1x,"j",2x"a",2x,"b",2x,"c",2x,"a.and.b",3x,&
                                                  &"matmul(a,b)")')
do j=1,2; do i=1,2;  c(i,j)= a(i,j).and.b(i,j)
                     write(*,'(2g2.0,3g3.0,3x,g3.0,g10.0)')&
&                      i,j,a(i,j), b(i,j), c(i,j),e(i,j),d(i,j)
enddo; enddo
end
```

```
  a= T T F F
  b= T F T F
 i j  a  b  c  a.and.b   matmul(a,b)
 1 1  T  T  T     T         T
 2 1  T  F  F     F         T
 1 2  F  T  F     F         T
 2 2  F  F  F     F         T
 T T T T
```

О старом ФОРТРАНе последнее слово.

В старых версиях ФОРТРАНа допускались только статические с **целочисленными константами** в качестве граничных индексов.

Однако во многих задачах размеры матриц естественнее было вводить или вычислять. Возникало противоречие между возможностями описания матриц и требованиями эксплуатации программы. Для его разрешения при описании матрицы в главной программе указывали максимально возможные (в разумных пределах) количества строк и столбцов, которые могли потребоваться в задаче, размещая матрицу с нужными размерами в левом верхнем углу описанной. Например:

```
program tmat4; implicit none; integer, parameter :: ndim=5
integer a(ndim,ndim), b(ndim,ndim), c(ndim,ndim), d(ndim,ndim)
integer l, m, n, i, j
l=2; m=3; n=4            ! Для краткости моделируем расчёт l,m и n
a=-1; b=-2; c=-3; d=-4   ! Для уяснения результата засеем массивы
do i=1,l; do j=1,m; a(i,j)=i+j; enddo; enddo
do i=1,m; do j=1,n; b(i,j)=i*j; enddo; enddo
do i=1,ndim; write(*,'(" a(",i2,",:)=",50i3)') i,(a(i,j),j=1,ndim); enddo
do i=1,ndim; write(*,'(" b(",i2,",:)=",50i3)') i,(b(i,j),j=1,ndim); enddo
call matmul1(a,b,c,ndim,l,m,n)
do i=1,ndim; write(*,'(" c(",i2,",:)=",50i5)') i,(c(i,j),j=1,ndim); enddo
call matmul2(a,b,d,l,m,n)
do i=1,ndim; write(*,'(" d(",i2,",:)=",50i5)') i,(d(i,j),j=1,ndim); enddo
end
```

В **tmat4** принято, что могут потребоваться матрицы размером не больше чем из пяти строк и пяти столбцов. Процедура умножения матриц (назовём её **matmul1**) могла иметь вид:

```
subroutine matmul1(a,b,c,ndim,l,m,n); implicit none
integer ndim, l, m, n, i, j, k, s; integer a(ndim,m),b(ndim,n),c(ndim,n)
do i=1,l; do k=1,n; s=0; do j=1,m; s=s+a(i,j)*b(j,k); enddo;   c(i,k)=s
enddo; enddo
end
```

В ней число строк матриц, служащих формальными аргументами равно заявленному **ndim**, что принципиально важно. Часто встречалась ситуация, когда программист при описании **формального аргумента** (матрицы) указывал в качестве её строковой размерности значение переменной, хранящей не заявленное в главной программе, а **нужное** по *проблемному разумению* число строк, т.е. так

```
subroutine matmul2(a,b,c,l,m,n); implicit none
integer ndim, l, m, n, i, j, k, s;    integer a(l,m),b(m,n), c(l,n)
do i=1,l; do k=1,n; s=0; do j=1,m; s=s+a(i,j)*b(j,k); enddo; c(i,k)=s
enddo; enddo
end
```

забывая, что число строк матрицы, описанной в главной программе, равно ПЯТИ. В итоге программист обманывал и подпрограмму, и себя. Уясним результат работы **test4** при **l=2**, **m=3** и **n=4**:

```
 a( 1,:)=  2  3  4 -1 -1 При работе matmul1 подпрограмма "считает",
 a( 2,:)=  3  4  5 -1 -1 что a(1,2), b(1,2), c(1,2) следуют
 a( 3,:)= -1 -1 -1 -1 -1 за  a(5,1), b(5,1) и c(5,1) соответственно,т.е
 a( 4,:)= -1 -1 -1 -1 -1 так как определено главной программой.
 a( 5,:)= -1 -1 -1 -1 -1 Поэтому результат расчёта, например, c(1,1)
                         по формуле:
 b( 1,:)=  1  2  3  4 -2     a(1,1)*b(1,1)+a(1,2)*b(2,1)+a(1,3)*b(3,1)=
 b( 2,:)=  2  4  6  8 -2          2*1 + 3*2 + 4*3 = 2+6+12= 20  ВЕРЕН.
 b( 3,:)=  3  6  9 12 -2
 b( 4,:)= -2 -2 -2 -2 -2  MATMUL2 по описаниям a(l,m), b(m,n), c(l,n),
 b( 5,:)= -2 -2 -2 -2 -2  что при вызове даёт  a(2,3), b(3,4), c(2,4)
                              "полагает", что a(1,2), b(1,2), c(1,2)
 c( 1,:)=  20  40  60  80  -3  следуют за     a(2,1), b(3,1), c(2,1),
 c( 2,:)=  26  52  78 104  -3  что противоречит определению главной
 c( 3,:)=  -3  -3  -3  -3  -3  программы. Поэтому подпрограмма,
 c( 4,:)=  -3  -3  -3  -3  -3  беря элемент a(1,2), с точки зрения
 c( 5,:)=  -3  -3  -3  -3  -3  главной программы берёт элемент a(3,1)=-1
                               Соответственно элемент, именуемый
 d( 1,:)=  -3   0  -4  -4  -4  подпрограммой как a(1,3), оказывается
 d( 2,:)=  10 -13  -4  -4  -4  для главной a(4,1). Так что для расчёта
 d( 3,:)=  -4   9  -4  -4  -4  c(1,1) берутся вовсе не те элементы:
 d( 4,:)=   2  -4  -4  -4  -4
 d( 5,:)=   4  -4  -4  -4  -4
MATMUL2 "думает" об элементах a(1,1)*b(1,1)+a(1,2)*b(2,1)+a(1,3)*b(3,1)=
а на самом деле берёт         a(1,1)*b(1,1)+a(3,1)*b(2,1)+a(4,1)*b(3,1)=
                                    = 2*1 + (-1*2)+ (-1*3)=2-2-3=2-5=-3
```

В ФОРТРАНе аргументы процедур по умолчанию передаются по адресу. Поэтому в качесве адреса начального элемента матрицы a(1:2,1:3) будет передан адрес начального элемента матрицы a(1:5,1:5), описанной в главной программе. Так как ФОРТРАН располагает элементы матрицы в оперативной памяти непосредственно друг за другом по столбцам, то третий и четвёртый элементы первого столбца матрицы a(1:5,1:5) с точки зрения MATMUL2 ,полагающей согласно описанию формальных аргументов, что она имеет дело с матрицей a(1:2,1:3), окажутся двумя последними элементами её первой строки.

Выводы:

1. Размещаемые массивы современного ФОРТРАНа позволяют просто решать те задачи, для которых на старом ФОРТРАНе приходилось использовать некие искусственные приёмы.

2. При незначительной модификации старых программ (для сохранения единого стиля исходного текста) приходится в качестве дополнительного аргумента процедуры передавать наряду с матрицей и её заявленный строковый размер.

3. Современный ФОРТРАН позволяет обойтись без упомянутой передачи строкового размера за счёт указания о заимствовании формальным аргументом, являющимся матрицей, формы фактического, т.е. список формальных аргументов процедуры можно уменьшить (меньше аргументов — меньше вероятность ошибки):

```
program tmat5; implicit none
interface; subroutine matmul3(a,b,c,l,m,n);
           integer l,m,n,a(:,:),b(:,:),c(:,:); end subroutine matmul3
end interface
integer, parameter :: ndim=5
integer a(ndim,ndim), b(ndim,ndim), c(ndim,ndim), d(ndim,ndim)
integer l, m, n, i, j; l=2; m=3; n=4  ! Моделируем расчёт l, m и n.
a=-1; b=-2; c=-3; d=-4                ! Инициализация массивов.
do i=1,l; do j=1,m; a(i,j)=i+j; enddo; enddo
do i=1,m; do j=1,n; b(i,j)=i*j; enddo; enddo
do i=1,l; write(*,'(" a(",i2,",:)=",5i3)') i,(a(i,j),j=1,m); enddo
do i=1,m; write(*,'(" b(",i2,",:)=",5i3)') i,(b(i,j),j=1,n); enddo
call matmul3(a,b,c,l,m,n)
do i=1,l; write(*,'(" c(",i2,",:)=",5i5)') i,(c(i,j),j=1,n); enddo
end

subroutine matmul3(a,b,c,l,m,n); implicit none; integer ndim,l,m,n,i,j,k,s
integer a(:,:), b(:,:), c(:,:)
do i=1,l; do k=1,n; s=0; do j=1,m; s=s+a(i,j)*b(j,k); enddo; c(i,k)=s
enddo; enddo
end
```

```
 a( 1,:)=  2  3  4
 a( 2,:)=  3  4  5
 b( 1,:)=  1  2  3  4
 b( 2,:)=  2  4  6  8
 b( 3,:)=  3  6  9 12
 c( 1,:)=   20   40   60   80
 c( 2,:)=   26   52   78  104
```

7.4 Транспонирование матриц

Функция **transpose(matrix)** транспонирует матрицу-аргумент, возвращая транспонированную матрицу через имя **transpose**.

```
                                                ! Результат работы trans:
program trans                                   !      a:
implicit none                                   !     1 3 5
integer a(2,3), b(3,2), i, j                    !     2 4 6
a=reshape((/ 1,2,3,4,5,6 /),(/2,3/))            !      b:
b=transpose(a)                                  !     1 2
write(*,*) ' a:'; write(*,'(3i2)') (a(i,:),i=1,2) !     3 4
write(*,*) ' b:'; write(*,'(2i2)') (b(i,:),i=1,3) !     5 6
end
```

7.5 Функция слияния массивов

Функция **merge(tsource,fsource,mask)** — поэлементная функция трёх аргументов — массивов одинаковой формы, из которых первые два одинакового типа, а третий — логического. **merge** возвращает через своё имя массив той же формы, в котором элементы первого аргумента заменены элементами второго аргумента в том и только в том случае, если соответствующие элементы маски имеют значение **.false.**.

Имя **merge** иногда именует процедуру слияния двух упорядоченных массивов в один упорядоченный. К упомянутой сортировке встроенная **merge** ФОРТРАНа никакого отношения не имеет. Правильнее её назвать функцией **совмещения элементов двух массивов**.

```
program tmerge                                  ! tmerg посредством вызова
implicit none                                   ! merge получает матрицу C,
integer a(2,3), b(2,3), c(2,3),  i              ! элементы которой равны
logical l(2,3)                                  ! элементам матриц A или B,
a=reshape((/ 1, 2,  3, 4, 5,  6/),(/2,3/));    ! в зависимости от того
b=reshape((/ 0, 8, -1, 4, 7, -5/),(/2,3/));    ! больше ли элементы матрицы
l=a>b                                           ! A соответствующих элементов
c=merge(a,b,l)                                  ! матрицы B или меньше.
write(*,'(5x,"a",11x,"b",11x,"l",11x,"c")')
write(*,'(3i3,3x,3i3,3x,3l3,3x,3i3)') (a(i,:),b(i,:),l(i,:),c(i,:),i=1,2)
end
```

```
    a          b          l          c
  1  3  5    0 -1  7    T  T  F    1  3  7
  2  4  6    8  4 -5    F  F  T    8  4  6
```

7.6 Функции упаковки и распаковки массивов

Функция **pack(array,mask,[vector])** возвращает вектор, собранный из тех и только тех элементов массива **array**, для которых значения маски **mask** есть **true**.

array — массив любой формы и типа. **mask** — скаляр или логический массив той же формы, что и **array**. **vector** — необязательный аргумент (вектор, элементы которого доопределяют хвостовые элементы массива-результата). Размер **vector** не меньше числа элементов **mask** со значением **.true.**.

```
program tpack1; implicit none; integer a(3,3),r1(2),r2(7)
a=0; a(2,2)=1; a(3,3)=2;
r1=pack(a,a/=0); write(*,'(a,2i3)') ' r1=',r1
r2=pack(a,a/=0,(/10,20,30,40,50,60,70/)); write(*,'(a,7i3)') ' r2=',r2
end
```

```
 r1=  1  2
 r2=  1  2 30 40 50 60 70
```

Функция **unpack(vector,mask,field)** возвращает массив, полученный расфасовкой вектора **vector** в соответствии с содержимым логического массива **mask** той же формы, что и **unpack**.

Именно, содержимое очередного элемента вектора **vector** присваивается тому и только тому элементу **unpack**, если соответствующий элемент **mask** имеет значение **.true.**. Если элемент **mask** имеет значение **.false.**, то соответствующему элементу **unpack** присваиваивается либо значение скаляра **field**, либо значение соответствующего элемента массива **field** той же формы, что и **mask**.

```
program tunpack; implicit none
logical mask(2,3); integer :: matr(2,3), vec(3)=((/3,4,5/))
mask=reshape((/.true.,.false.,.false., .true., .true., .false./),(/2,3/))
write(*,'(" mask(1,:)=",3l3/" mask(2,:)=",3l3)') mask(1,:), mask(2,:)
matr=unpack(vec,mask,77)
write(*,'(" matr(1,:)=",3i3/" matr(2,:)=",3i3)') matr(1,:), matr(2,:)
end
```

```
 mask(1,:)=  T  F  T  ! Под термином "упаковка" часто понимают сокращение
 mask(2,:)=  F  T  F  ! объёма информации без её потери. PACK "трактует"
 matr(1,:)=  3 77  5  ! термин "упаковка" более широко, допуская априори
 matr(2,:)= 77  4 77  ! и возможную потерю. По сути выполняемой работы
                      ! к функциям PACK и UNPACK на русском языке более
                      ! подходят термины "фильтрация" и "расфасовка".
```

7.7 Сборка массива через добавление измерения

Функция **spread(source, dim, ncopies)** добавляет **ncopies** копий исходного массива **source** вдоль заданного измерения **dim** массива, получаемого **spread**.

```
program tspread; implicit none
integer m(3) /1,2,3/, m2 (4,3), m3(4,3,2), i, j, k
m2=spread(m,1,4)
write(*,*) ' m=',m
write(*,'(" m2:")'); write(*,'(3i3)') ((m2(i,j),j=1,3),i=1,4)
write(*,'("m3: к матрице m2 добавились ещё две такие же,")')
write(*,'("    сформировав второй и третий листы книги m3:")')
m3=spread(m2,3,2)
m3(:,:,2)=10*m3(:,:,2) ! меняем содержимое второго и третьего листа,
m3(:,:,3)=20*m3(:,:,3) ! чтобы убедиться, что выводим действительно
                       ! разные листы, а не 3 копии первого.
do k=1,3; write(*,'(i3,a$)') k,"-й лист"; enddo; write(*,*)
write(*,'(3i3," ",3i3," ",3i3$)') (((m3(i,j,k),j=1,3),k=1,3),i=1,4)
write(*,*)
end
  m=           1           2           3
 m2:
  1  2  3
  1  2  3
  1  2  3
  1  2  3
m3: к матрице m2 добавились ещё две такие же,
    сформировав второй и третий листы книги m3:
  1-й лист  2-й лист  3-й лист
  1  2  3  10 20 30  20 40 60
  1  2  3  10 20 30  20 40 60
  1  2  3  10 20 30  20 40 60
  1  2  3  10 20 30  20 40 60
```

1. **m2=spread(m,1,4)** формирует матрицу из четырёх строк и трех столбцов, копируя в неё построчно исходный **m** четыре раза.

2. **m3=spread(m2,3,4)** формирует из трёхмерный массив (книгу, состоящую из трёх листов), копируя в неё полистно матрицу **m2**.

3. При указании номера измерения, не согласованного с формой копируемого исходного массива, получаем сообщение об ошибке:

```
m3=spread(m2,1,2)
 1
 Error: different shape for Array assignment at (1) on dimension 1 (4/2)
```

7.8 Функции сдвига массива

Функция **cshift (array, shift [, dim])** — циклический сдвиг.
Функция **eoshift(array, shift [, boundary] [,dim])** — нециклический.
Здесь:

1. **array** — массив, для которого необходим сдвиг позиций.
2. **shift** — аргумент, указывающий величину сдвига. В простейшем случае это — скаляр, так как величина подвижки позиций элементов одномерного массива всегда одно целое число. Если же сдвигаемый массив, например, матрица, то **shift** должен быть вектором с количеством элементов равным либо числу строк, либо числу столбцов сдвигаемого. В этом случае каждый элемент **shift** указывает величину сдвига по каждому из столбцов или строк соответственно. Для для многомерного сдвигаемого массива аргумент **shift** — массив с размерностью на единицу меньшей размерности сдвигаемого.
3. **boundary** — скаляр (или массив с размерностью на единицу меньшей размерности **array**), который указывает, на какие значения должны замениться элементы с позициями вне допустимого диапазона.
4. **dim** — скаляр целого типа, задающий номер измерения, по которому должна произойти подвижка.

```
program tshift; implicit none; integer :: i, m(4)=(/(i,i=1,4)/), k(4)
integer  j, n(4,4) /1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16/, n1(4,4)
write(*,'(a,4i5)') '  вектор m до смещения: ',m
write(*,'("Циклический cshift-сдвиг позиций вектора m: ")')
do i=-4,4; k=cshift(m,i,1); write (*,'(a,i2,4i5)') "i=",i,k; enddo
write(*,'("Нециклический eoshift-сдвиг позиций вектора m: ")')
do i=-4,4; k=eoshift(m,i,0,1); write (*,'(a,i2,4i5)') "i=",i,k; enddo
write(*,'(" Матрица n   до  смещения:")');
write(*,'(4i4)') ((n(i,j),j=1,4),i=1,4)
n1=eoshift(array=n,shift=(/0,-1,-2,-3/),boundary=1, dim=2)
write(*,'(" Матрица n после eoshift-смещения по ВТОРОМУ измерению:")')
write(*,'(4i4)') ((n1(i,j),j=1,4),i=1,4)
n1=eoshift(array=n,shift=(/0,-1,-2,-3/),boundary=1, dim=1)
write(*,'(" Матрица n после eoshift-смещения по ПЕРВОМУ измерению:")')
write(*,'(4i4)') ((n1(i,j),j=1,4),i=1,4)
end
```

```
  вектор m до смещения:     1    2    3    4
Циклический cshift-сдвиг позиций вектора m:
i=-4    1    2    3    4
i=-3    2    3    4    1
i=-2    3    4    1    2
i=-1    4    1    2    3
i= 0    1    2    3    4
i= 1    2    3    4    1
i= 2    3    4    1    2
i= 3    4    1    2    3
i= 4    1    2    3    4
Нециклический eoshift-сдвиг позиций вектора m:
i=-4    0    0    0    0
i=-3    0    0    0    1
i=-2    0    0    1    2
i=-1    0    1    2    3
i= 0    1    2    3    4
i= 1    2    3    4    0
i= 2    3    4    0    0
i= 3    4    0    0    0
i= 4    0    0    0    0
 Матрица n   до  смещения:
   1   5   9  13
   2   6  10  14
   3   7  11  15
   4   8  12  16
 Матрица n после eoshift-смещения по ВТОРОМУ измерению:
   1   5   9  13
   1   2   6  10
   1   1   3   7
   1   1   1   4
 Матрица n после eoshift-смещения по ПЕРВОМУ измерению:
   1   1   1   1
   2   5   1   1
   3   6   9   1
   4   7  10  13
```

1. При сдвиге матрицы по ВТОРОМУ измерению смещается ВПРАВО содержимое первой строки на **нуль** позиций, содержимое второй — на **одну** позицию, третьей — на две, и четвёртой — на 3 позиции.

2. При сдвиге матрицы по ПЕРВОМУ измерению смещается ВНИЗ содержимое первого столбца на **нуль** позиций, содержимое второго — на **одну** позицию, третьего — на две, и четвёртого — на 3 позиции.

3. В соответствии со значением аргумента **boundary** на пустые места, освободившееся после сдвига, помещается **единица**.

7.9 Функции определения положения в массиве

Функция **maxloc (array [, dim] [,mask])** — возвращает вектор с количеством элементов равным рангу массива **array**. При отсутствии необязательных аргументов **dim** и **mask** значение очередного элемента возвращаемого вектора равно индексу максимального элемента **array** по соответствующему измерению.
Функция **minloc (array)** — возвращает вектор с индексами минимального элемента **array** по каждому измерению.

При наличии **dim** ищется наибольший (наименьший) элемент вдоль заданного **dim** измерения. Наличие логического массива-маски **mask** позволяет находить искомое лишь среди тех элементов **array**, для которых значения соответствующих элементов **mask** равны **true**.

```
program tmaxloc; implicit none
integer  i, m(10) / 20, 13, -5, -8, 10, 7, 3, 26, -17, -1 /
integer  j, n(3,4) / 20, 13, -5, -8, 10, 7, 3, 26, -17, -1, 0, 5 /

write(*,'(a,10i4)') " Исходный вектор m=",m
write(*,*) ' Форма maxloc(m): shape(maxloc(m))=',shape(maxloc(m))
write(*,'(a,i4)') "maxloc(m)=",maxloc(m)
write(*,'(a,i4)') "minloc(m)=",minloc(m)
write(*,'(a,i4)') 'maxloc(m,mask=m<0)=', maxloc(m,mask=m<0)
write(*,'(a,i4)') 'mixloc(m,mask=m>0)=', minloc(m,mask=m>0)

write(*,'("Исходная матрица n(3,4):")')
write(*,*) ' Форма maxloc(n): shape(maxloc(n))=',shape(maxloc(n))
write(*,'(4i4)') ((n(i,j),j=1,4),i=1,3)
write(*,*) "         maxloc(n)=",maxloc(n)
write(*,*) "         minloc(n)=",minloc(n)
write(*,*) "     maxloc(n,n<0)=",maxloc(n,n<0)
write(*,*) "     minloc(n,n>0)=",minloc(n,n>0)
write(*,*) "    minloc(n,n>=0)=",minloc(n,n>=0)
write(*,*) ' Форма maxloc(n,1): shape(maxloc(n,1))=',shape(maxloc(n,1))
write(*,*) "       maxloc(n,1)=",maxloc(n,1)
write(*,*) "       minloc(n,1)=",minloc(n,1)
write(*,*) ' Форма maxloc(n,2): shape(maxloc(n,2))=',shape(maxloc(n,2))
write(*,*) "       maxloc(n,2)=",maxloc(n,2)
write(*,*) "       minloc(n,2)=",minloc(n,2)
write(*,*) "   maxloc(n,1,n<0)=",maxloc(n,1,n<0)
write(*,*) "   minloc(n,1,n<0)=",minloc(n,1,n<0)
write(*,*) "   maxloc(n,2,n<0)=",maxloc(n,2,n<0)
write(*,*) "   minloc(n,2,n<0)=",minloc(n,2,n<0)
end
```

```
 Исходный вектор m=  20  13  -5  -8  10   7   3  26 -17  -1
  Форма maxloc(m): shape(maxloc(m))=           1
maxloc(m)=   8
minloc(m)=   9
maxloc(m,mask=m<0)=  10
mixloc(m,mask=m>0)=   7
Исходная матрица n(3,4):
  Форма maxloc(n): shape(maxloc(n))=           2
  20  -8   3  -1
  13  10  26   0
  -5   7 -17   5
         maxloc(n)=           2           3
         minloc(n)=           3           3
     maxloc(n,n<0)=           1           4
     minloc(n,n>0)=           1           3
    minloc(n,n>=0)=           2           4
  Форма maxloc(n,1): shape(maxloc(n,1))=           4
       maxloc(n,1)=           1           2           2           3
       minloc(n,1)=           3           1           3           1
  Форма maxloc(n,2): shape(maxloc(n,2))=           3
       maxloc(n,2)=           1           3           2
       minloc(n,2)=           2           4           3
   maxloc(n,1,n<0)=           3           1           3           1
   minloc(n,1,n<0)=           3           1           3           1
   maxloc(n,2,n<0)=           4           0           1
   minloc(n,2,n<0)=           2           0           3
```

1. Результат **maxloc(m)** — вектор из одного элемента, а не скаляр. Запись **i=maxloc(m)** ошибочна (слева – скаляр; справа вектор).

2. Результат **maxloc(n)** — вектор из двух элементов: местоположение элемента матрицы определяется двумя индексами: строки и столбца (индекс строки – в первом элементе, индекс столбца – во втором).

3. Результат **maxloc(n,1)** тоже вектор, но уже из четырёх чисел, так как ищется максимальное для каждого из ЧЕТЫРЁХ столбцов.

4. **maxloc** под ПЕРВЫМ измерением *понимает* направление перехода от предыдущего столбца к очередному. после того, как для предыдущего был найден максимальный элемент.

5. **maxloc** под ВТОРЫМ измерением *понимает* направление перехода от предыдущей строки к следующей после нахождения максимального элемента. **maxloc(n,2)** есть вектор из трёх элементов, так как ищется максимальное для каждой строки, а их у матрицы **n** — три.

7.10 О чем узнали из седьмой главы? (второй семестр)

1. Современный ФОРТРАН предоставляет для работы с массивами набор функций, которых не было в ФОРТРАНе-77, что позволяет существенно сократить размер исходных текстов программ, обрабатывающих массивы, и уменьшить временные затраты.

2. К справочным функциям относятся:

   allocated(array), **size(array [, dim])**, **shape(source)**
   lbound(array [, dim]) и **ubound(array [, dim])**.

3. К функциям **редукции** (массивов) относятся:

   all(mask [, dim]), **any(mask [, dim])**, **count(mask [, dim])**, **maxval(array [, dim] [,mask])**, **minval(array [, dim] [,mask])**, **product(array [,dim], [,mask])**, **sum(array [,dim], [,mask])**.

   Термин **редукция** (reduction — снижение, понижение, уменьшение, сокращение, приведение) в данном контексте означает, что на вход к функции в качестве аргумента подаётся массив, а на выходе получается значение с меньшим числом элементов чем имеется в массиве-аргументе (см., например, [2, 7]).

4. Функция скалярного умножения даух векторов
   dot_product(vector_a,vector_b)

5. Функция умножения матриц **matmul(matrix_a,matrix_b)**.
   Умножение матриц определено лишь для согласованных матриц (т.е. число столбцов первой должно равняться числу строк второй).

6. **matmul** и **dot_product** осуществляют лишь формальное приведение числовых типов друг к другу, т.е. данное типа **real(4)** после преобразования в тип **real(8)** из шестнадцати десятичных цифр, будет иметь верными лишь старших 7.

7. Перемножение матриц логического типа происходит по правилу перемножения числовых (с заменой сложения дизъюнкцией, а умножения — конъюнкцией).

8. Размещаемые массивы современного ФОРТРАНа позволяют упростить решение задач, для которых на старом ФОРТРАНе приходилось использовать некие искусственные приёмы.

9. Если нужно располагать рабочие матрицы в левом верхнем углу матрицы большего размера, ОБЯЗАТЕЛЬНО включаем в список формальных аргументов соответствующих процедур переменную, хранящую строковый размер большей матрицы, заявленный в вызывающей программе.

10. Современный ФОРТРАН позволяет обойтись без подобного включения, так как способен посредством функций **lbound** и **ubound** определить граничные индексы фактических аргументов, если формальные перенимают их форму.

11. Функция **transpose(matrix)** транспонирует матрицу и возвращает результат (транспонированную матрицу) через своё имя.

12. Функция **merge(tsource,fsource,mask)** возвращает массив, значения элементов которого равны **tsource** или **fsource** в зависимости от истинности элементов массива **mask**.

13. Функция **pack(array,mask,[vector])** через имя **pack** возвращает вектор, собранный из тех и только тех элементов массива **array**, для которых значения маски **mask** есть **true**.

14. Функция **unpack(vector,mask,field)** возвращает через своё имя массив, полученный расфасовкой содержимого элементов вектора **vector** в соответствии с содержимым логического массива **mask** той же формы, что и **unpack**.

15. Функция **spread(source, dim, ncopies)** добавляет **ncopies** копий исходного массива **source** вдоль заданного измерения **dim** в массиве-результате **spread**.

16. **cshift (array, shift [, dim])** — реализует циклический сдвиг;
    eoshift(array, shift [, boundary] [,dim]) — нециклический.

17. Функции **maxloc** и **minloc** — возвращают вектор с индексами соответствующего элемента массива **array**.

7.11 Седьмое домашнее задание (второй семестр).

Задача N 1.

Написать программу, которая помогает выяснить:

Какие значения и когда вырабатываются встроенными функциями **lbound** и **ubound** при вызове их внутри подпрограммы, формальным аргументом которой является вектор, перенимающий форму фактического аргумента.

Задача N 2

Разработать функцию, которая для заданного положительного целого находит наибольшее из чисел, получаемых циклической перестановкой двоичного представления исходного целого.

Задача N3

Разработать функцию приведения матрицы системы линейных алгебраических уравнений с **n** неизвестными к эквивалентному треугольному виду.

Задачи N 4 — 8.

Разработать функции, моделирующие работу встроенных функций, указанных ниже, продемонстрировать их работу и оформить тестирующую программу так, чтобы она позволяла легко убедиться в совпадении их результатов с результатами встроенных. Разработанные функции поместить в модуль **myarray**.

4. Функция **my_dot_production** нахождения скалярного произведения двух векторов.
5. Функция **my_matmul** (перемножение матриц).
6. Функция **my_transpose** (транспонирование матрицы)
7. Функция **my_maxval** (поиск максимального элемента матрицы с наименьшими индексами) .
8. Функция **my_minloc** (поиск наибольших индексов минимального элемента матрицы)

8 Форматирование данных ввода-вывода

Изложение материала в основном по книгам [2] и [3].

В простых программах для ввода и вывода используются операторы

read(*,*) список_ввода и **write(*,*) список_вывода**

Первая * в этих операторах – синоним стандартного устройства ввода-вывода (по умолчанию это — экран). Вторая * — синоним управления вводом-выводом **под управлением списка** ввода-вывода, т.е. способ преобразования данных в машинное представление (или из машинного) определяется по типам элементов, указанных в списке ввода-вывода. Подобный способ удобен, когда результат нужен в понятной, но непритязательной форме, определяемой компилятором по умолчанию:

```
program frm1; implicit none                ! Ввод-вывод под управлением
integer n / 10 /                           ! списка ввода-вывода удобен
real x /1.7/, em /9e-28/, sunm /-1.99e+33/ ! удобен и тогда, когда
logical b(2) /.true., .false. /            ! заранее известно, что
character(1) abc(3) /'a','b','c'/          ! данные - это поток чисел,
integer matr(2,3) / 1, 2, 3, 4, 5, 6 /     ! разделённых пробелами,
write(*,*) '    n=', n                     ! причём нам неважно, сколько
write(*,*) '    b=', b                     ! чисел в каждой строке  и
write(*,*) '  abc=', abc                   ! каким количеством пробелов
write(*,*) '    x=', x                     ! одно число отделяется от
write(*,*) '   em=', em                    ! другого (важно наличие хотя
write(*,*) ' sunm=', sunm                  ! бы одного пробела для
 write(*,*) ' matr=', matr                 ! разделения чисел).
end program frm1
```

```
    n=          10     ! Здесь по умолчанию компилятор отвёл
    b= T F             ! на целое данное десять позиций;
  abc=abc              ! на булево значение - две; на однобайтовый;
    x=   1.7000000     ! символ - одну, а на вещественное значение
   em=  8.99999967E-28 ! - пятнадцать (включив недостающие пробелы).
 sunm=  1.99100001E+33
 matr=           1           2           3           4           5           6
```

Обычно данные выгоднее размещать на внешнем носителе, записывая в желательном формате. Выгода — большая наглядность и возможности при вводе размещать в невостребованной части строки поясняющий комментарий. Указание о месте расположения и формы записи осуществляется форматированием, управляемым **явным заданием формата**.

8.1 Явное задание формата

8.1.1 Примеры

Демонстрация некоторых возможностей задания формата:

```
program frm2
implicit none
integer n / 10 /
real    x / 1.7 /, em /9e-28/, sunm / -1.991e+33 /
logical b(2) /.true., .false. /             !
character(1) abc(3) /'a','b','c'/           !       Результат работы frm2
integer matr(2,3) / 1, 2, 3, 4, 5, 6 /      !
write(*,'(1x,5x,a,i2)')      '    n=', n    !          n=10
write(*,'(1x,5x,a,l1,3x,l1)')'    b=', b    !          b=T   F
write(*,'(1x,5x,a,3a4)')     '  abc=', abc  !        abc=   a   b   c
write(*,'(1x,19x,a,f5.2)')   '    x=', x    !                         x= 1.70
write(*,'(1x,5x,a,e9.2)')    '   em=', em   !         em= 0.90E-27
write(*,'(1x,5x,a,e10.3)')   ' sunm=', sunm !       sunm=-0.199E+34
write(*,'(1x,5x,a,6i2)')     ' matr=', matr !       matr= 1 2 3 4 5 6
end
```

В операторах **write** вместо второй звёздочки явно указан формат вывода. Синтаксически он обрамлён апострофами, т.е. представляет собой строковую константу.

1. В качестве первого элемента формата использовано сочетание двух символов **1x**, что на языке оператора **format** означает один пробел. **1x** — один из управляющих дескрипторов оператора **format**.

2. Дескрипторы отделяются друг от друга, т.е. строка **(1x,5x)** означает, что при выводе сначала выводится один пробел, а затем ещё пять. Вместо **(1x,5x)** можно обойтись и одним дескриптором **6x**.

3. В данной программе файл вывода обозначен ***** (первая звездочка в операторе **write**) и особо не описывается оператором **open** (его подробно не изучали, но использовали). Тем не менее полезно знать, что при открытии файлов ФОРТРАНа первый символ форматной строки может трактоваться как код управления кареткой печатающего устройства (в частности, первый **пробел** означает перевод строки и не печатается). Именно поэтому во многих старых ФОРТРАН-программах форматная строка начиналась с **1x**.

4. Дескриптор **a** используется для преобразования символьных данных. В данном случае одна буква **a** означает преобразование кодов всех символов, образующих строковое значение из списка вывода, в сами символы (т.е. всех символов строковой константы **’ n=’**, заключенных между апострофами).

5. Дескриптор **i2** (в **write**) используется для указания преобразования внутреннего двоичного представления целочисленного значения (в данном случае переменной **n**) в десятичное двузначное целое.

6. Дескрипторы **l1,3x,l1** указывает оператору **write** вывести два булева значения, разделённые тремя пробелами, и отводя под каждое только одно знакоместо.

7. Форматы **1x,5x,a,3a4** обеспечат вывод с седьмой позиции значения строковой константы **’ abc=’** и трёх элементов символьного массива, под каждый из которых хотим отвести четыре знакоместа.

8. Дескриптор **f5.2** при выводе вещественного значения переведёт в общепринятую запись числа, занимающую пять позиций с двумя десятичными цифрами после запятой.

9. Дескриптор **e9.2** (при выводе) запишет вещественное значение в форме с плавающей запятой (девять позиций на всю запись, с двузначной десятичной мантиссой и порядком числа).

10. Пример вывода через явное посредство оператора **format** с тем же результатом дан программой **frm3**:

```
program frm3; implicit none; integer n / 10 /
real    x / 1.7 /, em /9e-28/, sunm / -1.991e+33 /
logical b(2) /.true., .false. /
character(1) abc(3) /’a’,’b’,’c’/; integer matr(2,3) / 1, 2, 3, 4, 5, 6 /
write(*,100) ’    n=’, n
write(*,200) ’    b=’, b; write(*,300) ’  abc=’, abc
write(*,400) ’    x=’, x; write(*,500) ’   em=’, em
write(*,600) ’ sunm=’, sunm                          ! Целое число перед
write(*,700) ’ matr=’, matr                          ! словом format -
100 format(1x,5x,a,i2)                               ! метка, ссылаясь на
200 format(1x,5x,a,l1,3x,l1); 300 format(1x,5x,a,3a4)  ! которую, read или
400 format(1x,19x,a,f5.2);    500 format(1x,5x,a,e9.2) ! write "узнают" о
600 format(1x,5x,a,e10.3);    700 format(1x,5x,a,6i2)  ! нужных правилах
end                                                  ! форматирования
```

8.2 Способы явного задания формата

Явно указать формат ввода-вывода можно одним из двух способов:

- непосредственно в операторе ввода-вывода в виде константы или выражения символьного типа, что удобно при краткости их записи.
- в операторе **format**, указываемом оператору ввода-вывода посредством метки, что удобно, если запись формата громоздка, так как. оператор **format** можно расположить в любом месте программы, не загромождая подробностями форматирования её проблемную часть. Наиболее удобен оператор **format**, когда его используют несколько операторов ввода-вывода.

Спецификатор формата — выражение в скобках оператора **format**. **Спецификатор формата** состоит из списка **дескрипторов**:

```
метка format( список_дескрипторов )
```

Дескриптор – указывает правило, по которому данное преобразуется при вводе-выводе, и записывается символами служебного алфавита оператора **format**. Имеется три вида дескрипторов.

1. **Дескрипторы преобразования данных**:

Имя дескриптора	Тип данного	Примечание
I, **B**, **O**, **Z**	целый	можно
F,**E**, **EN**, **ES**, **D**	вещественный	использовать
L	логический	и заглавные,
A	символьный	и строчные
G	любой	буквы

2. **Управляющие дескрипторы: T**, **TL**, **TR**, **X**, **S**, **SP**, **SS**, **BN**, **BZ**,**P**.

3. **Дескриптор в виде символьной строки** удобен, когда нет необходимости под выводимый текст выделять дополнительную переменную. Например,

```
program frm4; implicit none
write(*,1001); 1001 format(' Так можно вывыести двойную кавычку ".')
write(*,1002); 1002 format(' Так можно вывыести апостроф ''      .')
write(*,1003); 1003 format(" Так тоже можно вывыести апостроф '  .")
write(*,1004); 1004 format(' В старых версиях ФОРТРАНа нельзя было'/&
                &" качестве апострофа использовать двойную кавычку")
end
```

8.2.1 Форматы данных целого типа

Основные дескрипторы	**Iw** и **Iw.d**
Дополнительные	**Bw**, **Bw.d**, **Ow**, **Ow.d**, **Zw**, **Zw.d**,

1. Здесь **w** — количество позиций, отводимое для размещения данного целого типа, т.е. число знакомест под данное, включая и знак числа.
2. Предполагается, что данное при чтении или записи должно быть выравнено по правой границе предоставляемого ему поля.
3. Если число цифр меньше **w**, то в левые позиции пишутся пробелы.
4. Если же запись числа требует больше позиций чем предоставлено шириной поля, то всё поле заполняется значком *.
5. **Iw.d** позволяет наряду с **w** задать и **d** — минимальное число цифр, которые надо вывести, заполняя пустые старшие позиции поля данного незначащими нулями, что удобно при необходимости. При вводе **Iw.d** эквивалентено **Iw** и значение **d** игнорируется.
6. Дескрипторы **B**, **O** и **Z** задают форму представления данного в двоичной, восьмеричной и шестнадцатеричной системах счисления (трактовка **w** и **d** — та же).

```
program frm5; implicit none; integer, parameter :: j=5, m=1024,n=-1024
write(*,'(" j=",i2," j=",i3," j=",i4)') -j, -j, -j
write(*,'(" m=",i3)') m
write(*,'(" m10=",i4," m2=",b11," m8=",O4," m16=",z4)') m, m, m, m
write(*,'(" n10=",i5," n2=",b32," n8=",O11," n16=",z8)') n, n, n, n
write(*,'(" n=",i4," n=",i5)') n, n
write(*,'(" m=",i5.2)') m
write(*,'(" n=",i7.6)') n
write(*,'(" n=",i10.6)') n
end
```

```
 j=-5 j= -5 j=  -5
 m=***
 m10=1024 m2=10000000000 m8=2000 m16= 400
 n10=-1024 n2=11111111111111111111110000000000 n8=37777776000 n16=FFFFFC00
 n=**** n=-1024
 m= 1024
 n=-001024
 n=   -001024
```

8.2.2 Форматы данных вещественного типа

Дескриптор преобразования **real**-числа без порядка	**Fw.d**
Дескрипторы преобразования **real**-числа с порядком	**Ew.d**, **Dw.d**, **Ew.dEe**, **ENw.d**, **ESw.d**

1. Здесь **w** — ширина поля; **d** — число цифр после десятичной точки.

2. Дескриптор **F** позволяет представить данное вещественного типа в обычном виде: целая часть числа, точка, дробная часть числа.

3. Дескрипторы **E** и **D** позволяют преобразовывать данные, в записи которых указан порядок числа. До ФОРТРАНа-90 **E** предназначалось для для преобразования данных типа **real(4)**, а **D** — **real(8)**. Это их назначение сохраняется. Однако, современный ФОРТРАН обслуживает данные типа **real(8)** и через дескриптор **E**, так что дескриптор **D** в значительной мере становится архаичным.

4. **Внимание!** Есть принципиальное различие между дескриптором **D** и использованием литеры **D** при указании порядка числа в тексте программы. В случае дескриптора допускается замена литеры **D** на литеру **E**, но в случае записи числовой константы в исходном тексте подобная замена недопустима, т.к. литера **E** однозначно укажет на одинарную точность, а **D** — на удвоенную. Правда, в современном ФОРТРАНе можно записать число удвоенной точности и с использованием литеры **E**, поместив после записи порядка символы **_8**, указывающие количество байт, отводимых под константу.

5. Дескриптор **Ew.dEx** позволяет указать через константу **x** после второй буквы **E** количество позиций для записи порядка числа.

6. Дескриптор **EN** в отличие **E** при выводе приводит число к диапазону **[1,1000)** (кроме нуля), а десятичный порядок данного делает кратным **3** (**EN**gineering — инженерный формат).

7. Дескриптор **ES** в отличие **E** при выводе приводит число к диапазону **[1,10)** (исключая нуль), и подстраивает соответственно десятичный порядок (**Scientific** — научный формат).

Ниже приведена программа **frm6**, демонстрирующая вышесказанное о дескрипторах вывода вещественных чисел:

```
program frm6; implicit none
real(4), parameter :: a=-1.23456789, b=1.234567e+33
real(8), parameter :: c=-1.2345678901234567d-28
real(8) p , q
character(66) :: text(16)=(/' Нет места для знака числа',' Теперь есть',&
&' Младшая цифра в 12-ой позиции после знака =',' 7 позиций мало',&
&' Две цифры с порядком после точки.',' То же можно и так',&
&' 7 цифр с порядком после точки.',' Под порядок четыре цифры',&
&' Раньше лишь литера D задавала формат удвоенной точности',&
&' Теперь можно использовать и E',' Верны старшие 7-8 цифр',' Верны 16 цифр',&
&'  EN-вывод в диапазоне [1,1000) и','   порядком кратным 3',&
&'  E-вывод (сравнить с EN и последующим ES)','  ES-вывод в диапазоне [1,10)'/)
write(*,'(" a=",f4.3,12x,a)') a,text(1) ! Нет места для знака числа !
write(*,'(" a=",f6.3,10x,a)') a,text(2) ! Теперь есть
write(*,'(" a=",f12.7,4x,a)') a,text(3) ! Мл. цифра в 12-ой позиции после =
write(*,'(" b=",f7.3,9x, a)') b,text(4) ! В 7 позиций мало для числа с порядком
write(*,105) " b=",b, text(5)           ! Две цифры с порядком после запятой
write(*,106)       b, text(6)           ! То же можно и так
write(*,107)       b, text(7)           ! 7 цифр с порядком после запятой
write(*,108)       b, text(8)           ! Под порядок четыре цифры
write(*,109)       c, text(9)   ! Раньше формат удвоенной точности указывался D
write(*,110)       c, text(10)  ! В современном можно и E
p=1.234567890123456789e7        ! E и D в константах - не дескрипторы
q=1.234567890123456789d7        ! a определяют тип константы:
write(*,111)       p, text(11)  ! верны только старшие семь цифр
write(*,112)       q, text(12)  ! верны старшие 16
write(*,113)       p, text(13)  ! EN-нормализация к диапазону [1,1000) и
write(*,114)       c, text(14)  !      порядку кратному 3 (инженерная нотация)
write(*,115)       c, text(15)  ! E-вывод (сравните с предыдущим и последующим)
write(*,116)       c, text(16)  ! ES-нормализация к диапазону [1,10)
105 format(1x,a,e9.2,6x,a);        106 format(1x," b=",e9.2,6x,a);
107 format(1x," b=",e15.7,a);      108 format(1x," b=",e15.7e4,a)
109 format(1x," c=",d9.2,16x,a);   110 format(1x," c=",e25.16,a)
111 format(1x," p=",e25.16,a);     112 format(1x," q=",e25.16,a)
113 format(1x," q=",en12.2,12x,a); 114 format(1x," c=",en25.16,1x,a)
115 format(1x," c=",e25.16,a);     116 format(1x," c=",es25.16,a)
end
```

Заметим, что приведённый исходный текст **frm6** набирался при использовании однобайтовой кодировки **koi8r**. В случае кодировки **utf8** при пропуске иногда возникают проблемы (в особенности при использовании кириллицы). Именно, по умолчанию (в режиме свободного формата записи исходного текста) компилятор обычно отводит строку длиной 130 символов. Кодировка же кириллицы в **utf8** — двухбайтовая, так что (как только размер исходного текста в одной строке потребует более 130 символов) последует сообщение об ошибке. Исправить ситуацию можно

переустановив при запуске компилятора новый предел длины строки, например, посредством опции **-ffree-line-length-500** (или **-ffree-line-length-0**, или **-ffree-line-length-none**; см. **Приложение VII**).

Результаты работы frm6.

```
a=****             Нет места для знака числа
a=-1.235           Теперь есть
a=  -1.2345679     Младшая цифра в 12-ой позиции после знака =
b=*******          7 позиций мало
 b= 0.12E+34       Две цифры с порядком после точки.
 b= 0.12E+34       То же можно и так
 b=  0.1234567E+34 7 цифр с порядком после точки.
 b=0.1234567E+0034 Под порядок четыре цифры
 c=-0.12D-27       Раньше лишь литера D задавала формат удвоенной точности
 c=  -0.1234567890123457E-27 Теперь можно использовать и E
 p=   0.1234567900000000E+08 Верны старшие 7-8 цифр
 q=   0.1234567890123457E+08 Верны 16 цифр
 q=   12.35E+06              EN-вывод в диапазоне [1,1000) и
 c=-123.4567890123456748E-30    порядком кратным 3
 c=  -0.1234567890123457E-27  E-вывод (сравнить с EN и последующим ES)
 c=  -1.2345678901234567E-28  ES-вывод в диапазоне [1,10)
```

8.2.3 Форматы ввода данных вещественного типа

Ввод осуществляется только из поля указанного дескриптором. Поэтому в остальной части строки можно поместить комментарий, напоминающий о проблемном назначении вводимого параметра. Например, пусть главная программа должна вводить:

1. **a** и **b** — абсциссы концов отрезка, на котором ищется корень уравнения $\mathbf{f(x)=0}$ методом деления отрезка пополам;
2. **epsx** и **epsy** — абсолютные погрешности поиска корня по абсциссе и ординате (т.е. считаем, что корень найден, если длина уточняющего отрезка после очередного сужения оказалась меньше **epsx** и $|\mathbf{f}(\mathbf{x}_{\text{искомое}})| < \mathbf{epsy}$).
3. **p** и **q** — некоторые дополнительные параметры, от которых помимо аргумента **x** зависит левая часть уравнения $\mathbf{f(x) = exp(-px^2)-q*x}$

Разместим вводимые величины во вводимом файле **frm7.inp** по одной в каждой строчке в первых 15 позициях так:

```
      -1.300+00<--= a (левая граница промежутка с корнем)      файл input
       2.3     <--= b (правая  граница промежутка с корнем)
1.000-05       <--= a (левая граница промежутка с корнем)
     0.00001   <--= b (правая  граница промежутка с корнем)
  314159265    <--= p (коэффициент при показателе экспоненты)
  2.71828182+00<--= q (коэффициент перед линейным слагаемым)
```

Тогда ввод этих исходных данных можно осуществить программой

```
program frm7; implicit none
real(8) a, b, epsx, epsy, p, q
read (*,'(e15.7)') a, b, epsx, epsy, p, q
write(*,1000)      a, b, epsx, epsy, p, q
!
! Вызов функции, реализующей алгоритм метода деления пополам
!
1000 format(1x,'    a=',e25.16/1x,'    b=',e25.16/1x,' epsx=',e25.16/&
&           1x,' epsy=', e25.16/1x,'    p=',e25.16/1x,'    q=',e25.16)
end
```

Замечания:

1. Перед записью числа могут находится пробелы (лишь бы они и число не выходили за пределы ширины поля ввода).

2. Если поле ввода содержит десятичную точку, то данное в пределах ширины поля можно сдвигать как угодно: значение **d** (число цифр после запятой в дескриптере) игнорируется.

3. При отсутствии десятичной точки **d** младших цифр числа трактуются как его дробная часть.

4. После записи мантиссы числа может присутствовать его десятичный порядок в виде целой констаны со знаком перед которым может быть литера **E** или **D**.

5. Записывать вводимые параметры столь разношёрстно, как это сделано в **frm7.inp** вряд ли практично. Обычно удобна однообразная форма записи. Здесь же приведены возможности форматного ввода.

6. Иногда спрашивают: **“ Почему при указании формата ввода записан один дескриптор e15.7, а список ввода состоит из нескольких переменных?”** Используемый способ ценен тем, что обеспечивает чтение очередного значения из первых **15** позиций очередной строки файла, позволяя в оставшуюся часть предыдущей поместить поясняющий комментарий. Как только прочитывается очередное число и обнаруживается, что список форматов закончился, то следующее данное читается по прежнему формату, но из новой строки. Это — частный случай проявления более общего ФОРТРАН-правила, называемого **реверсией формата** (см., например, [3]).

7. Для ввода всех данных из одной строки можно, например, воспользоваться дескриптором **’(6(e15.7,3x))’**. Здесь шестёрка перед **(e15.7,3x)** — **повторитель формата**.

8. Широко востребовано как размещение одного числа в одной строки, так и нескольких чисел. Например, все данные по одной звезде из каталога звёзд обычно размещаются в одной строке файла с каталогом, а при решении расчётных задач, требующих ввода небольшого числа параметров, каждый из них удобно размещать в отдельной строке с соответствующим пояснением.

8.2.4 Форматы ввода-вывода данных комплексного типа

Перевод комлексных данных из внутреннего машинного двоичного представления в обычное десятичное и обратно выполняется посредством двух вещественных дескрипторов. Первый задаёт преобразование для вещественной части, второй — для мнимой (при этом первый не обязан совпадать со вторым). Например,

```
program frm8; implicit none; complex(8) a, b, c, v
real(8) vre, vim
read (*,'(d10.3,d10.3)') a, b; write(*,*) ' a=',a; write(*,*) ' b=',b
c=a+b;                                              write(*,*) ' c=',c
write(*,'(" a=",f5.2,"+",f5.2,"i")') a
write(*,'(" b=",f5.2,"+",f5.2,"i")') b
write(*,'(" c=",f5.2,"+",f5.2,"i")') c
read(*,'(f10.2,d10.3)') vre, vim
write(*,*) ' vre=',vre,' vim=',vim
v=dcmplx(vre , vim); write(*,'("v=dcmplx(vre , vim)=",2d25.16)') v
v= cmplx(vre,vim,4); write(*,'("v= cmplx(vre,vim,4)=",2d25.16)') v
v= cmplx(vre,vim,8); write(*,'("v= cmplx(vre,vim,8)=",2d25.16)') v
v= cmplx(vre , vim); write(*,'("v= cmplx(vre , vim)=",2d25.16)') v
end
```

```
    1.5      1.6       a                Файл frm8.inp
    2.3      0.3       b
    3.2      7.51      vre vim
```

```
  a= (  1.5000000000000000     ,  1.6000000000000001     )
  b= (  2.2999999999999998     , 0.29999999999999999     )
  c= (  3.7999999999999998     ,  1.9000000000000001     )
 a= 1.50+ 1.60i
 b= 2.30+ 0.30i
 c= 3.80+ 1.90i
  vre=   3.2000000000000002       vim=   7.5099999999999998
v=dcmplx(vre , vim)=   0.3200000000000000D+01   0.7510000000000000D+01
v= cmplx(vre,vim,4)=   0.3200000047683716D+01   0.7510000228881836D+01
v= cmplx(vre,vim,8)=   0.3200000000000000D+01   0.7510000000000000D+01
v= cmplx(vre , vim)=   0.3200000047683716D+01   0.7510000228881836D+01
```

Внимание! Вызов **cmplx** с двумя аргументами типа **real(8)** (но без третьего, параметра разновидности результата) *полагает*, что погрешности округления вещественной и мнимой частей результата соответствуют типу **complex(4)**, а не **complex(8)**, т.е. в используемой версии компилятора механизм перегрузки функций для **cmplx** не работает, или, другими словами, имя **cmplx** здесь специфическое, а не родовое.

8.2.5 Форматы ввода-вывода данных логического типа

1. Для форматного ввода-вывода логических данных используется дескриптор **Lw** (**w** — ширина поля).
2. При выводе в самую крайнюю правую позицию выводится **T** или **F**.
3. При вводе возможны необязательные пробелы и символы **T** или **F**.
4. Дескриптор **L7** допускает ввод значений **.true.** и **.false**.

```
program frm9; implicit none; logical p, q, true; integer i
write(*,*) ' Ввод под управлением списка ввода вводим T и F:';
read(*,*) p, q; write(*,*) ' p=',p, '   q=',q
write(*,*) ' Ввод по дескриптору L7 из первых 7 позиций двух строк:'
do i=1,3
  read(*,'(l7)') p, q; write(*,'(a,L7,a,L7)') ' p=',p, '   q=',q
enddo
write(*,*) ' Ввод по формату (L7,3x,L7) из первых 17 позиций одной строки:'
do i=1,3
  read(*,'(l7,3x,l7)') p, q; write(*,'(a,L7,a,L7)') ' p=',p, '   q=',q
enddo
end
```

```
t f               Файл frm9.inp с вводимыми данными.
.TRUE.            Программа frm9 вводит его через операцию перенаправления
f                 стандартного ввода < frm9.inp
 .FALSE.          Помещение данного текста, начиная с 18 позиции этого файла
  t               не мешает работе, так ввод осуществляется исключительно
      t           по формату (L7,3x,L7). Однако, если в последней строке
  f               сместить букву T на одну позицию вправо или букву F
t               f на одну позицию влево, или её же в 18-ю позицию, то
   t         f    пропуск программы выдаст сообщение
      t   f       At line 13 of file frm9.f95 (unit = 5, file = 'stdin')
                  Fortran runtime error: Bad value on logical read
```

Вывод программы **frm9**

```
 Ввод под управлением списка ввода вводим T и F:
 p= T    q= F
 Ввод по дескриптору L7 из первых 7 позиций двух строк:
p=      T   q=      F
p=      F   q=      T
p=      T   q=      F
 Ввод по формату (L7,3x,L7) из первых 17 позиций одной строки:
p=      T   q=      F
p=      T   q=      F
p=      T   q=      F
```

8.2.6 Формат ввода-вывода данных символьного типа

1. Для форматного ввода-вывода символьных данных используем дескриптор **A** или **Aw** (**w** — ширина поля).
2. **A** определяет **w** по фактической длине элемента ввода-вывода.
3. **Aw** определяет ширину поля ввода-вывода лишь **w** символами.
4. Если **w** $>$ длины данного, то считываются все символы из правых позиций; а при выводе место слева дополняется пробелами.
5. Если **w** $<$ длины символьного данного, то при вводе считывается **w** левых символов (оставшееся поле данного заполняется пробелами), а при выводе выводится лишь **w** левых символов.

```
program frm10; implicit none; character(11)  b
read(*,'(a)') b  ; write(*,*) 'ввод по формату   a: b=',b,b
read(*,'(a15)') b; write(*,*) 'ввод по формату a15: b=',b,b
read(*,'(a11)') b; write(*,*) 'ввод по формату a11: b=',b,b
read(*,'(a6)')  b; write(*,*) 'ввод по формату  a6: b=',b,b
write(*,*) 'ASCII-code пробела в b(7:7)=',iachar(b(7:7))
write(*,'("вывод по формату   a: b=",a,a)') b, b
write(*,*) 'ASCII-code пробела в b(11:11)=',iachar(b(11:11))
write(*,'("вывод по формату a15: b=",a15,a15)') b, b
write(*,*) 'ASCII-code символа b(1:1)=',iachar(b(1:1)),' Это буква ',b(1:1)
write(*,'("вывод по формату a11: b=",a11,a11)') b, b
write(*,'("вывод по формату  a6: b=", a6, a6)') b, b
write(*,'("вывод по формату  a4: b=", a4, a4)') b, b
end
```

```
astronomy                                              Файл frm10.inp.
astronomy      Текст "Файл frm10.inp" можно помещать в первую строку, т.к.
astronomy      формат A действует лишь на все элемены переменной b, а их
astronomy      только 11.
```

```
 ввод по формату   a: b=astronomy  astronomy  |  Вывод программы frm10:
 ввод по формату a15: b=onomy      onomy      |
 ввод по формату a11: b=astronomy  astronomy  |
 ввод по формату  a6: b=astron     astron     |
 ASCII-code пробела в b(7:7)=          32     |
вывод по формату   a: b=astron     astron     |
 ASCII-code пробела в b(11:11)=          32           |
вывод по формату a15: b=    astron         astron    <--= Здесь b(1:1)='a'.
 ASCII-code символа b(1:1)=           97  Это буква a |   Пробелы добавлены
вывод по формату a11: b=astron     astron             |        не в b,
вывод по формату  a6: b=astronastron                  |   а в поле вывода.
вывод по формату  a4: b=astrastr
```

8.2.7 G — дескриптор для данных любого встроенного типа

Дескриптор **Gw.d[Ex]** работает с данными любого встроенного типа:

```
program frm20; implicit none;      integer :: k=375
complex :: z=(1.2,3.4);            real(4) :: a=-1.234567, p=1.991e33
real(8) :: b=1.2345678901234567_8; real(8) :: c=1.2345678901234567
    logical :: t=.true.
character(45) :: s='Это длинная строка выводится по формату g25.0'
read(*,*); ! пропуск строки с номерами позиций
write(*,'(" Вывод по ")'); write(*,'("     g5.0 integer  k= ",g5.0)' ) k
                           write(*,'("    g13.7 real(4)  a= ",g12.7)') a
                           write(*,'("    g13.7 real(4)  a= ",g13.7)') a
                           write(*,'("    g15.7 real(4)  p= ",g15.7)') p
                           write(*,'("   g25.16 real(8)  b=",d25.16)') b
                           write(*,'("   g25.16 real(8)  c=",d25.16)') c
                           write(*,'("    g15.7 complex  z=", g15.7)') z
write(*,'("     g1.0 logical  t=",g1.0,"   .not.t=",g1.0)') t, .not. t
write(*,'(g45.0)') s;      write(*,'(" Ввод  по ")')
read(*,100) k; write(*,'(5x," g5.0 : данное в первых 5 позициях k=",g5.0)') k
read(*,100) k; write(*,'(11x,": данное в первых 5 позициях k=",g5.0)') k
read(*,100) k; write(*,'(11x,": данное в первых 5 позициях k=",g5.0)') k
read(*,100) k; write(*,'(11x,": мл. цифра  ВНЕ поля ввода  k=",g5.0)') k
read(*,100) k; write(*,'(11x,": две цифры  ВНЕ поля ввода  k=",g5.0)') k
read(*,100) k; write(*,'(11x,": три цифры  ВНЕ поля ввода  k=",g5.0)') k
read(*,101) a; write(*,'(5x," g9.6 : по g9.6   1.3   можно ввести,"$)')
               write(*,'(" НО НЕ ВЫВЕСТИ a=",g9.6)') a
read(*,101) a; write(*,'(11x,":",32x," и по g10.6    a=",g10.6)') a
read(*,101) a
write(*,'(11x,":",4x," 1.3   можно ВЫВЕСТИ,  начиная  с g11.6    a=",g11.6)') a
read(*,101) a; write(*,'(5x," g9.6 :-1.234e+33 можно ВЫВЕСТИ, "$)')
               write(*,'(" начиная  с g12.6    a=",g12.6)') a
         write(*,'(5x," g9.6 :-1.234e+33",18x," ВЫВОД по g13.6    a=",g13.6)') a
read(*,101) b
      write(*,'(5x," g9.6 : 1.234+33  ВЫВОД по  g25.16  real(8)  b=",g25.16)') b
read(*,101) c
      write(*,'(5x," g9.6 : 1.234+33  ВЫВОД по  g25.16  real(8)  c=",g25.16)') c
read(*,101) z
write(*,'(5x," g9.6 : ВЫВОД по  g15.7 complex  z=",g15.7,g15.7)') z
read(*,100) t
      write(*,'(5x," g9.6 :logical  t=",g1.0,"   .not.t=",g1.0)') t, .not. t
read(*,102) s; write(*,'(5x," g45.0: Так оно выведется по  g45.0",&
& /11x,": s=",g45.0)') s; write(*,'(5x,"      : Так по  g50.0 s=",g50.0)') s
                          write(*,'(5x,"      : Так по  g25.0 s=",g25.0)') s
100 format(g5.0)
101 format(g9.6)
102 format(g45.0)
end
```

```
1234567890<--=номера позиций              !  Содержимое файла frm20.inp
376      :  376 в позициях 123            ! Программа осуществляет ввод
 376     :  376 в позициях 234            ! через операцию перенаправления
  376    :  376 в позициях 345            ! стандартного ввода.
   376   :  376 в позициях 456
    376  :  376 в позициях 567
     376 :  376 в позициях 678
1.3      :  Наличие точки в пределах поля ввода позволяет программе
 1.3     :  выяснить значение данного независимо от дескриптора формата
  1.3    :  (лишь бы ширина его поля была достаточна)
-1.234+33:
1.234+33 :
1.234e+33:
    3.7  :  Re(z)
    5.2  :  Im(z)
t
Это строковое данное введено по формату g45.0.
 Вывод по                                          !  Результат работы
     g5.0 integer  k=   375                        !  программы  frm20
    g13.7 real(4)  a= ********                     !
    g13.7 real(4)  a= -1.234567
    g15.7 real(4)  p=   0.1991000E+34
   g25.16 real(8)  b=   0.1234567890123457D+01
   g25.16 real(8)  c=   0.1234567880630493D+01
    g15.7 complex  z=   1.200000
    g15.7 complex  z=   3.400000
     g1.0 logical  t=T   .not.t=F
Это длинная строка выводится по формату g25.0
 Ввод  по
      g5.0 : данное в первых 5 позициях k=  376
           : данное в первых 5 позициях k=  376
           : данное в первых 5 позициях k=  376
           : мл. цифра  ВНЕ поля ввода  k=   37
           : две цифры  ВНЕ поля ввода  k=    3
           : три цифры  ВНЕ поля ввода  k=    0
      g9.6 : по g9.6   1.3   можно ввести, НО НЕ ВЫВЕСТИ a=*****
           :                                и по g10.6    a=******
           :     1.3   можно ВЫВЕСТИ,  начиная  с g11.6    a=1.30000
      g9.6 :-1.234e+33 можно ВЫВЕСТИ,  начиная  с g12.6    a=-.123400E+34
      g9.6 :-1.234e+33                  ВЫВОД по g13.6    a=-0.123400E+34
      g9.6 : 1.234+33  ВЫВОД по  g25.16  real(8)  b=   0.1234000000000000E+34
      g9.6 : 1.234+33  ВЫВОД по  g25.16  real(8)  c=   0.1234000000000000E+34
      g9.6 : ВЫВОД по  g15.7 complex  z=   3.700000       5.200000
      g9.6 :logical  t=T   .not.t=F
      g45.0: Так оно выведется по  g45.0
           : s=Это строковое данное введено по формату g45.0
           : Так по  g50.0 s=     Это строковое данное введено по формату g45.0
           : Так по  g25.0 s=Это строковое данное введ
```

8.2.8 Формат ввода-вывода данных производного типа

Для форматного ввода-вывода производного типа можно использовать подходящую последовательность дескрипторов, каждый из которых нацелен на обработку соответствующего поля производного типа.

Пусть, например, программа использует производный тип **star**, полями которого служат номер звезды (поле **num**), её имя (поле **nam**, прямое восхождение (поле **ra**) и склонение (поле **dec**):

```
program frm11
implicit none
type star
  integer num
  character (len=20) nam
  real ra, dec
end type star
type (star) a
read (*,'(i5,a20,f6.2,f6.2)') a%num, a%nam, a%ra, a%dec
write(*,1000) a%num, a%nam, a%ra, a%dec
1000 format(i5,a20,e15.7,e15.7)
end
```

```
  333         name_mystar 3.141  0.75                    Файл frm11.inp.
```

Вывод программы **frm11**

```
  333         name_mystar  0.3141000E+01  0.7500000E+00
```

Вообще говоря, придумывание производных типов требует глубокого понимания сути задачи: грамотно придуманный производный тип сильно упрощает написание, отладку и эксплуатацию программы. Поэтому, как правило, не стоит в теле основной программы работать непосредственно с конкретными полями структуры.

Выгоднее создать и вызывать соответствующие процедуры, на вход к которым подаётся просто имя структурной переменной или ссылка на неё, поручая этим процедурам обработку соответствующих задаче полей. В частности, это относится и к процедурам ввода и вывода.

Использование отдельных полей производного типа в главной программе объективно можно оправдать лишь отладочным характером работы с ней или же какой-то сиюминутной выгодой, которая в конечном итоге должна быть спрятана в соответствующей процедуре или модуле.

8.3 Управляющие дескрипторы

Дескриптор	Назначение (кратко)
BN, **BZ**	Интерпретация пробелов
S, **SP**, **SS**	Управление выводом знака
T, **TL**, **TR**, **X**	Управление табуляцией
/ (косая черта)	Завершение передачи данных в текущую запись
: (двоеточие)	Прерывание управления форматом вывода
P	Масштабный множитель

8.3.1 BN и BZ — управление интерпретацией пробелов

BN (blank null) и **BZ** (blank zero) указывают, что пробелы поля ввода (кроме начального)

1. **BN** — игнорируются.
2. **BZ** — интерпретируются как нули.
3. При вводе **BN** и **BZ** влияют только лишь на дескрипторы числовых данных.
4. При выводе **BN** и **BZ** игнорируются.
5. **BN** и **BZ** действуют до конца списка дескрипторов формата или пока не встретится другой дескриптор **BN** или **BZ**

```
program frm13
integer k, l, m, n
read (*, 100) k,l,m,n; 100 format(   i5,i5,i5,i5); write(*,1100) k,l,m,n
read (*, 101) k,l,m,n; 101 format(bz,i5,i5,i5,i5); write(*,1100) k,l,m,n
read (*, 102) k,l,m,n; 102 format(bn,i5,i5,i5,i5); write(*,1100) k,l,m,n
read (*, 103) k,l,m,n;
103 format(bn,i5,bz,i5,i5,i5); write(*,1100) k,l,m,n
1100 format(4i5)
end
```

```
12     13  14  15                                  Файл ввода frm12.inp
12     13  14  15
12     13  14  15
12     13  14  15
```

Вывод программы **frm12**

```
   12   13   14   15
12000  130 1400   15
   12   13   14   15
   12  130 1400   15
```

8.3.2 S, SP, SS —управление выводом знака

1. При вводе **S**, **SP**, **SS** игнорируются.
2. При выводе **S**, **SP**, **SS** влияют только на дескрипторы **числовых** данных.
3. **S**, **SP**, **SS** действуют до конца списка дескрипторов формата или пока не встретится снова **S**, **SP** или **SS**.
4. **SP** указывает, что знак "+" следует выводить всюду, где возможно.
5. **SS** указывает, что знак "+" следует везде опускать.
6. **S** установка принятого в реализации режима вывода знака "+".
7. Отрицательные числа всегда выводятся со знаком.

Например,

```
program frm13
implicit none
integer, parameter :: k=5, m=-7
   real, parameter :: a=3.14, b=-7.28
 write(*,'(" k=",i3," m=",i3," a=",f7.3," b=",e15.7 )') k,m,a,b
 write(*,'(sp," k=",i3," m=",i3," a=",f7.3," b=",e15.7 )') k,m,a,b
 write(*,'(ss," k=",i3," m=",i3," a=",f7.3," b=",e15.7 )') k,m,a,b
 write(*,'(sp," k=",i3," m=",i3," a=",f7.3," b=",e15.7 )') k,m,a,b
 write(*,'(s," k=",i3," m=",i3," a=",f7.3," b=",e15.7 )') k,m,a,b
end
```

Вывод программы **frm13**

```
 k=  5 m= -7 a=  3.140 b= -0.7280000E+01
 k= +5 m= -7 a= +3.140 b= -0.7280000E+01
 k=  5 m= -7 a=  3.140 b= -0.7280000E+01
 k= +5 m= -7 a= +3.140 b= -0.7280000E+01
 k=  5 m= -7 a=  3.140 b= -0.7280000E+01
```

8.3.3 Tn, TRn, TLn и nX —управление табуляцией

Табуляция — задание позиции в записи

Целая константа без знака **n** в

1. **Tn** задаёт номер позиции, с которой надо начать передачу данных.
2. **TLn** задаёт на сколько позиций **влево** следует сместить текущую позицию во внешней записи.
3. Позиция левой границы записи полагается равной единице.
4. **TRn** задаёт на сколько позиций **вправо** следует сместить текущую позицию во внешней записи.
5. **nX** обычно используется для вставки пробелов в выводимую запись или указания количества позиций, которые надо пропустить до начала данного для очередного числового дескриптора. Эквивалентно **TRn**.
6. Ценность дескрипторов управления табуляцией в том, что они позволяют после ввода отдельных частей записи осуществить повторную их обработку причём возможно с другим дескриптором.

Например,

```
program frm14
implicit none
integer k,m,n
character(4) sk, sm, sn
real(4)  rk, rm, rn, rr
read(*,100) k,m,n, sk,sm,sn, rk,rm,rn, rr
 write(*,'("  k=",i3,"  m=",i3,"  n=",i3)') k,m,n
 write(*,'(" sk=",a4," sm=",a4," sn=",a4)') sk(1:4), sm(2:4), sn(3:4)
 write(*,'(" rk=",e15.7,"  rm=",e15.7,"   rn="e15.7)') rk, rm, rn
 write(*,'(" rr=",e10.3)') rr
100 format(3(i4,6x),T7,3(a4,6x),T1,3E10.1,TL7,f3.1)
end
```

```
 123.4e+05 678.9e-08 987.5e+11                    Файл ввода frm14.inp
```

Вывод программы **frm14**

```
  k=123  m=678  n=987
 sk=e+05 sm= -08 sn=  11
 rk=  0.1234000E+08  rm=  0.6789000E-05   rn=  0.9875000E+14
 rr= 0.750E+01
```

8.3.4 Дескриптор “дробная черта”

Дескриптор “/” означает конец передачи в текущую запись или из неё.

Варианты использования:

1. При выводе в файл **последовательного доступа** (пока мы только с такими файлами и имели дело) создаётся новая пустая запись, которая становится текущей, и пишущий элемент устанавливается на её начало (по сути дела это — просто перевод строки).

2. При вводе из файла **последовательного доступа** остаток текущей записи пропускается и читающий элемент устанавливается на начало следующей записи, которая становится текущей.

3. При работе с файлом **прямого доступа** (познакомимся позже) номер записи увеличивается на единицу и читающий (или пишущий) элемент устанавливается в начало записи с полученным номером.

4. Дескриптор “/” можно повторить несколько раз ////, что приведёт к появлению в файле вывода **последовательного доступа** четырёх пустых строк. Того же можно достичь, используя повторитель **4**/, правда, такое сочетание придётся выделять запятыми.

```
program frm15
implicit none
integer a(15), i
open (5,file='frm15.inp')
read (5,101) a
write(*,101) a
101 format( 4i3 / 5i3 / 2i3/4i3)
end
```

```
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15       ! Содержимое файла ввода
 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
```

```
  1  2  3  4                              ! Результат работы программы frm15:
 21 22 23 24 25
 41 42
 61 62 63 64
```

Обратить внимание, что корректный ввод и вывод возможен лишь в случае, когда сумма повторителей перед дескритпором формата **равна** заявленному числу элементов массива (ведь по имени массива формально выводится заявленное число его элементов).

8.3.5 Дескриптор “$”

Дескриптор “**$**” — признак продолжения текущей записи. Позволяет:

1. после вывода приглашения ко вводу набрать данное в строке приглашения.
2. добавить в хвост строки вывода предыдущего данного очередное если по каким-то причинам для этого неудобно использовать один оператор общего вывода.

```
program frm16; implicit none
real p, q
 write(*,100); read(*,*) p; write(*,’(" p=",e15.7)’) p
 write(*,101); read(*,*) q; write(*,’(" q=",e15.7)’) q
 write(*,’(" p=",f7.3,$)’) p
 write(*,’(" q=",f7.3)’) q
100 format(’input parameter:’); 101 format(’input parameter:’,$)
end
```

```
3.2
7.5
```

```
input parameter:                    !  Результат работы программы frm16:
p=  0.3200000E+01
input parameter:q=  0.7500000E+01
 p=  3.200 q=  7.500
```

В некоторых реализация ФОРТРАНа-95 наряду с **$** с той же целью может использоваться дескриптор **обратный слеш**.

8.3.6 Дескриптор “:”

Дескриптор “**:**” — при исчерпании списка вывода прерывает управление форматом

Дескриптор “**:**” игнорируется в случае ввода или при наличии в списке вывода элементов.

```
program frm17; implicit none; real ::  p=3.2, q=7.5
 write(*,100) p, q
 write(*,100) p
100 format(1x,’ p=’, f10.3,: 2x,’ q=’,e10.3)
end
```

Вывод программы **frm17**

```
  p=     3.200   q= 0.750E+01
  p=     3.200
```

8.3.7 Дескриптор kP (масштабный множитель)

Буква **k** в синтаксической конструкции **kP** — константа, обозначающая показатель степени десятки, а **P** — символическое имя конструкции, указывающее её назначение (power — показатель степени).

Дескриптор **kP** устанавливает значение масштабного множителя $\mathbf{10^k}$, (часто само **k** называют масштабным множителем) используя который вместо истинного значения некоторой величины (обозначим её, например, **A**) можно вывести или ввести значение $\mathbf{A \cdot 10^{-k}}$, что иногда удобно.

Например, вывести значения $\mathbf{9.283 \cdot 10^{-28}}$ и $\mathbf{1.991 \cdot 10^{+28}}$ в форме **F** (с фиксированной запятой) можно, но вряд ли это выгодно. Более практичны в данном случае формы **E** (или D) или даже **ES**.

Однако, если на выводе потребуется обширная таблица подобных величин, то запись порядка каждого её числа в виде **E-28** или **E+33** займёт почти столько же места сколько значащие цифры, что неудобно.

Для повышения наглядности числа можно домножить на некоторый масштабный множитель (например, на $\mathbf{10^{28}}$ или $\mathbf{10^{-28}}$ — увеличить или уменьшить масштаб чисел) и вывести их по дескриптору **F** с небольшим количеством значащих цифр Цена за полученную наглядность — замечание к таблице о том, что порядок приведённых значений отличается от истинного на величину порядка масштабного множителя.

Полезно знать, что сочетание управляющего дескриптора **kP** (масштабного множителя) с дескрипторами **F** и **E** приводит к принципиально разным результатам. Однако, прежде чем сформулировать окончательные рекомендации, полезно пропустить программу **frm18**. Её тело состоит из вызова трёх процедур: **tstPout1**, **tstPout2** и **tstPinp**, которые дают возможность посмотреть на результаты работы сочетания дескрипторов **kP**, **F** и **E** на примерах:

1. **вывода** чисел $\mathbf{0.9234 \cdot 10^{-27}}$ и $\mathbf{0.1991 \cdot 10^{29}}$ (процедура **tstPout1**).
2. **вывода** числа **1.234** (процедура **tstPout2**).
3. **ВВОДА** числа **5.678** (процедура **tstPinp**).
4. С целью удобства сопоставления исходных текстов подпрограмм и соответствующих результатов последние помещены непосредственно под исходным текстом

```
program frm18
implicit none
call tstPout1
call tstPout2
call tstPinp
end
```

```
subroutine tstPout1; implicit none; real(8) :: a=9.234d-28, s=1.991d+28
write(*,'(78("=")/T33,a/)') ' Работа tstPout1:'
write(*,'(" a=",f33.30$)') a;     write(*,'(5x,"Вывод по F33.30")');
write(*,'(" s=",f33.0$)') s;      write(*,'(5x,"Вывод по F33.0"/)');
write(*,'("Вывод    E11.4",9x,"D11.4",10x,"EN15.7",13x,"ES15.7")')
write(*,'("  a:",2x,E11.4,3x,d11.4,3x, en15.7,3x,es15.7)') a,a,a,a
write(*,'("  s:",2x,E11.4,3x,d11.4,3x, en15.7,3x,es15.7/)') s,s,s,s
write(*,'("Вывод по   28P,F11.4  a:", 28P,F11.4)') a
write(*,'("Вывод по  -28P,F11.4  s:",-28P,F11.4)') s
write(*,'(28x,"a",12x,"s")');
write(*,'("Вывод по   -1P,e15.7",-1P,e15.7,e15.7)') a,s
write(*,'("Вывод по   -2P,e15.7",-2P,e15.7,e15.7)') a,s
write(*,'("Вывод по   -3P,e15.7",-3p,e15.7,e15.7)') a,s
write(*,'("Вывод по   -4P,e15.7",-4p,e15.7,e15.7/)') a,s
write(*,'("Вывод по    1P,e15.7", 1P,e15.7,e15.7)') a,s
write(*,'("Вывод по    2P,e15.7", 2P,e15.7,e15.7)') a,s
write(*,'("Вывод по    3P,e15.7", 3p,e15.7,e15.7)') a,s
write(*,'("Вывод по    4P,e15.7", 4p,e15.7,e15.7)') a,s
end;
```

```
 ======================================================================
                                  Работа tstPout1:

 a= 0.000000000000000000000000000923     Вывод по F33.30
 s=   19909999999999998662017000000.     Вывод по F33.0

Вывод    E11.4         D11.4          EN15.7            ES15.7
  a:   0.9234E-27    0.9234D-27   923.4000000E-30     9.2340000E-28
  s:   0.1991E+29    0.1991D+29    19.9100000E+27     1.9910000E+28

Вывод по   28P,F11.4  a:     9.2340
Вывод по  -28P,F11.4  s:     1.9910
                             a           s
Вывод по   -1P,e15.7  0.0923400E-26  0.0199100E+30
Вывод по   -2P,e15.7  0.0092340E-25  0.0019910E+31
Вывод по   -3P,e15.7  0.0009234E-24  0.0001991E+32
Вывод по   -4P,e15.7  0.0000923E-23  0.0000199E+33

Вывод по    1P,e15.7  9.2340000E-28  1.9910000E+28
Вывод по    2P,e15.7  92.340000E-29  19.910000E+27
Вывод по    3P,e15.7  923.40000E-30  199.10000E+26
Вывод по    4P,e15.7  9234.0000E-31  1991.0000E+25
```

```
subroutine tstPout2
real(8) :: z=1.234_8; write(*,'(78("=")/T33,a)') ' Работа tstPout2:'
write(*,'(" Вывод")');
write(*,'(" числа   kP    F9.3      E11.3       D11.3"$)');
write(*,'("      EN11.3     ES11.3     G11.4")'); write(*,'(78("-"))')
write(*,1000) z,' 0P',z,z,z,z,z,z
write(*,1001) z,' 1P',z,z,z,z,z,z
write(*,1002) z,' 2P',z,z,z,z,z,z
write(*,1003) z,' 3P',z,z,z,z,z,z
write(*,1004) z,' 4P',z,z,z,z,z,z
write(*,2002) z,'-2P',z,z,z,z,z,z
write(*,2001) z,'-1P',z,z,z,z,z,z
1000 format(f6.3,2x,a,1x,0P,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
1001 format(f6.3,2x,a,1x,1P,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
1002 format(f6.3,2x,a,1x,2P,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
1003 format(f6.3,2x,a,1x,3P,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
1004 format(f6.3,2x,a,1x,4P,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
2002 format(f6.3,2x,a,1x,-2P,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
2001 format(f6.3,2x,a,1x,-1P,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
end

================================================================================
                                Работа tstPout2:
 Вывод
 числа   kP    F9.3      E11.3       D11.3      EN11.3     ES11.3     G11.4
--------------------------------------------------------------------------------
 1.234   0P     1.234   0.123E+01   0.123D+01   1.234E+00  1.234E+00   1.234
 1.234   1P    12.340   1.234E+00   1.234D+00   1.234E+00  1.234E+00   1.234
 1.234   2P   123.400   12.34E-01   12.34D-01   1.234E+00  1.234E+00   1.234
 1.234   3P  1234.000   123.4E-02   123.4D-02   1.234E+00  1.234E+00   1.234
 1.234   4P 12340.000   1234.E-03   1234.D-03   1.234E+00  1.234E+00   1.234
 1.234  -2P     0.012   0.001E+03   0.001D+03   1.234E+00  1.234E+00   1.234
 1.234  -1P     0.123   0.012E+02   0.012D+02   1.234E+00  1.234E+00   1.234
```

Видим, что

1) при сочетании с дескриптором **F** результат **вывода** действительно представляет произведение выводимого числа на $\mathbf{10^k}$;

2) при сочетании с форматами **E** и **D** масштабный множитель никак не влияет на величину выводимого числа, **HO** влияет на форму его представления — сдвигает десятичную точку при положительном **k** и зарабатывает дополнительные нули в мантиссе числа при отрицательном (иногда это может пригодиться);

3) форматы же **EN**, **ES** и **Gw.d** при выводе попросту игнорируют масштабный множитель.

```
subroutine tstPinp
real(8) x, z
 write(*,'(78("=")/T33,a)') ' Работа tstPinp:'
x=5.678_8
write(*,'("  Ввод")');
write(*,'(" числа   kP    F9.3      E11.3       D11.3"$)');
write(*,'("      EN11.3     ES11.3     G11.4")'); write(*,'(78("-"))')
read(*,'( 0p,f11.3)') z; write(*,2000) x,' 0P', z,z,z,z,z,z
read(*,'( 1p,f11.3)') z; write(*,2000) x,' 1P', z,z,z,z,z,z
read(*,'( 2p,f11.3)') z; write(*,2000) x,' 2P', z,z,z,z,z,z
read(*,'( 3p,f11.3)') z; write(*,2000) x,' 3P', z,z,z,z,z,z
read(*,'( 4p,f11.3)') z; write(*,2000) x,' 4P', z,z,z,z,z,z
read(*,'(-4p,f11.3)') z; write(*,2000) x,'-4P', z,z,z,z,z,z
read(*,'(-3p,f11.3)') z; write(*,2000) x,'-3P', z,z,z,z,z,z
read(*,'(-2p,f11.3)') z; write(*,2000) x,'-2P', z,z,z,z,z,z
read(*,'(-1p,f11.3)') z; write(*,2000) x,'-1P', z,z,z,z,z,z
                                                    ! Форматы вывода:
2000 format(f6.3,2x,a,1x,f9.3,1x,e11.3,1x,d11.3,1x,en11.3,es11.3,1x,g11.4)
end
```

```
================================================================================
                                Работа tstPinp:
  Ввод
 числа   kP    F9.3      E11.3       D11.3      EN11.3     ES11.3     G11.4
--------------------------------------------------------------------------------
 5.678   0P     5.678   0.568E+01   0.568D+01   5.678E+00  5.678E+00   5.678
 5.678   1P     0.568   0.568E+00   0.568D+00 567.800E-03  5.678E-01  0.5678
 5.678   2P     0.057   0.568E-01   0.568D-01  56.780E-03  5.678E-02  0.5678E-01
 5.678   3P     0.006   0.568E-02   0.568D-02   5.678E-03  5.678E-03  0.5678E-02
 5.678   4P     0.001   0.568E-03   0.568D-03 567.800E-06  5.678E-04  0.5678E-03
 5.678  -4P 56780.000   0.568E+05   0.568D+05  56.780E+03  5.678E+04  0.5678E+05
 5.678  -3P  5678.000   0.568E+04   0.568D+04   5.678E+03  5.678E+03   5678.
 5.678  -2P   567.800   0.568E+03   0.568D+03 567.800E+00  5.678E+02   567.8
 5.678  -1P    56.780   0.568E+02   0.568D+02  56.780E+00  5.678E+01   56.78
```

Выводы:

1. Прежде чем пользоваться дескриптором **kP** необходимо тщательно ознакомиться с нюансами работы его сочетания с нужным числовым дескриптором при вводе и выводе.
2. Без особой нужды масштабный множитель НЕ ИСПОЛЬЗУЕМ.
3. Дескриптор **масштабный множитель** может встретиться в чужих программах. Поэтому важно составить о нём своё собственное мнение. С этой целью полезно разобрать результаты, получаемые приведённой выше программой.

8.4 Ещё раз о вводе-выводе данных, управляемом списком

1. Ввод-вывод, управляемые списком, обслуживаются операторами

1	**read(*,*) список_ввода**	**write(*,*) список_вывода**
2	**read *, список_ввода**	**print *, список_вывода**

   В отличие от первых вторые изначально нацелены на работу с экраном. Поэтому им не нужно явно указывать устройство, с которым они должны работать. Так что смысловая нагрузка их единственной * — та же самая, что в первых двух у второй. Поэтому, если данные вводятся с экрана или через перенаправление стандартного ввода, то удобнее **read *, список_ввода** (для вывода **print *, список_вывода**).

2. Мы же с самого начала привыкали работать с первыми двумя, чтобы первая * напоминала об их более общем характере (именно, вместо первой * можно указывать и номер устройства ввода-вывода).

3. Управление вводом-выводом **под управлением списка** ввода (или вывода) выгодно тем, что оно не требует жёсткого закрепления данных за позициями записи в файле ввода (или вывода). Данные достаточно отделять друг от друга либо пробелами (хотя бы одним), либо запятыми. Например,

```
program frm21; implicit none; integer :: i, a(5)=(/1,2,3,4,5/), b(5)
                                  character(8) :: s1, s2, s3, s4, s5
read(*,*) s1, a, s2, b, s3, s4, s5
write(*,*)' s1=',s1; write(*,*)'  a=', a; write(*,*)' s2=',s2
write(*,*)'  b=', b; write(*,*)' s3=',s3; write(*,*)' s4=',s4
write(*,*)' s5=',s5
end
```

   Файл с входными данными программы frm21:

```
'first', -1, -2, ,-4, ,'second',10 11
,,,14,123456789  12345678,'   12345'
```

```
 s1=first                                                    ! Вывод
  a=          -1          -2           3          -4           5  !
 s2=second                                                   ! frm21.
  b=          10          11     4904176    50960944          14  !
 s3=12345678                                                 !
 s4=12345678                                                 !
 s5=   12345                                                 !
```

8.5 Ввод-вывод данных, управляемый NAMELIST-списком

Иногда при вводе (выводе) неудобно перечислять имена данных в списке ввода (вывода). Можно сопоставить списку имя через оператор **namelist** и указать это имя в операторе ввода (вывода).

```
program frm22; implicit none; real(8) a, b, y(5)
integer n, i
namelist /my_list/ a, b, n, y; read(*,my_list); write(*,my_list)
write(*,'(5f6.2)') y;
end
```

Пример 1 (содержимое файла с входными данными к программе frm22):

```
&my_list n=3, a=1.0,
b=5.0, y(1:3)=3*0.25d0, y(4:5)=2*7.1d0 /
```

Пример 2 (содержимое файла с входными данными к программе frm22):

```
&my_list
 a=1.0, n=3, b=5, y(1:3)=3*0.25d0, y(4:5)=2*7.1d0
$end
```

Пример 3 (содержимое файла с входными данными к программе frm22):

```
&my_list
 b=5.0, y(1:3)=3*0.25d0, n=3, a=1.0, y(4:5)=2*7.1d0
 &end
```

Файл с выходными данными примера 1

```
&MY_LIST
 A=  1.00000000000000     ,
 B=  5.00000000000000     ,
 N=           3,
 Y= 3*0.250000000000000      , 2*7.10000000000000       ,  /

  0.25  0.25  0.25  7.10  7.10
```

Порядок ввода значений через именованный список **my_list** безразличен. и даже допускает во вводимом файле указывать значения не всех переменных, входящих в него, а только нужных.

Завершение файла с данными, вводимыми по имени **namelist**-списка, **gfortran** допускает тремя способами:

1) **дробной чертой** "/" (что проще всего);

2) **значком доллара** "**$**", за которым следует **end**;

3) **значком амперсанда** "**&**", за которым следует **end**.

В качестве примеров файлов ввода приведены все три варианта.

9 Контрольная работа №2

Моделирование работы разностной машины Чарльза Бэббиджа.

9.1 Справочная информация

9.1.1 Исторический экскурс

В Европе конца XVIII начала XIX веков широкое распространение получили арифметические, тригонометрические, логарифмические, астрономические и навигационные таблицы (см., например, [19]). Последние были особенно важны для мореплавания. Вычислялись они вручную и содержали множество совершенно недопустимых ошибок и опечаток. Французский математик Гаспар Клэр Франсуа Риш маркиз де Прони́ (1755-1839) организовал расчёт морских таблиц особым образом. Он распределил вычислителей на три группы. В первую входили пять-шесть выдающихся математиков того времени (в частности, М. Лежандр), которые выбирали наиболее подходящие схемы расчёта и соответствующие формулы. Вторая группа (семь-восемь очень опытных и высококвалифицированных вычислителей) по этим формулам вычисляли требуемые таблицы с крупным шагом по аргументу. Третья группа (около 90 человек) — вычислители низкой квалификации. Их задача состояла в уплотнении таблиц, т.е. в заполнении интервалов между строками, вычисленными второй группой. При этом формулы, по которым третья группа вела расчёт, содержали арифметическую операцию только одного вида — операцию **сложения**.

Чарльз Бэббидж (1791-1871), английский математик, инженер и изобретатель, высоко оценил проект графа де Прони́ и предложил заменить третью группу вычислителей машиной. Машина, предложенная и построенная Бэббиджем, вычисляла значения многочлена (полинома) по способу разностей (отсюда и название машины — разностная).

9.1.2 Математическая основа алгоритма

Ознакомимся с математической идеей способа на примере расчёта таблицы значений функции $\mathbf{x^4}$. Для упрощения возможности устной проверки положим, что табулирование нужно провести для натуральных значений аргумента $\mathbf{x = 1, 2, 3, 4, \cdots}$ (для машины это не принципиально). Пусть такая таблица уже вычислена (см. колонки (1)-(2)):

(1)	(2)	(3)	(4)	(5)	(6)
1	1				
		15			
2	16		50		
		65		60	
3	81		110		24
		175		84	
4	256		194		24
		369		108	
5	625		302		24
		571		132	
6	1296		434		24
		1105		156	
7	2401		590.		
		1695			
8	4096	...			

Вычтем из каждого последующего значения столбца (2) предыдущее, записывая результаты (первую разность) в столбец (3). Повторив аналогичную операцию с первыми разностями, найдём вторые разности, помещая результат в столбец (4). Аналогично запишем третьи разности в столбце (5), и четвёртые в столбце (6). Видим, что четвёртые разности постоянны. Это не случайность, а следствие важной теоремы:

« Если функция есть многочлен (полином) n-ой степени, то в таблице с постоянным шагом по аргументу её n-е разности постоянны»

n-е разности с постоянным шагом по аргументу для полинома **n**-ой степени характеризуют поведение его **n**-ой производной, которая равна произведению $\mathbf{n! \cdot p_0}$, где $\mathbf{p_0}$ — коэффициент при старшей степени аргумента.

Случай А. Поместив разности **15, 50, 60, 24** (*нисходящая* диагональ) в строку с начальными значениями аргумента и функции, можно, используя только операцию сложения, получить содержимое второй строки:

$$\mathbf{1+15=16}, \mathbf{15+50=65}, \mathbf{50+60=110}, \mathbf{60+24=84},$$

т.е. значение $2^4 = 16$ и все необходимые для дальнейшего расчёта разности $65, 110, 84$ (последняя, четвёртая разность остаётся неизменной).

(1)	(2)	(3)	(4)	(5)	(6)
1	1	15	50	60	24
2	16	65	110	84	24
3	81	175	194	108	24
4	256	369	302	132	24
5	625	571	434	156	...
6	1296	1105	590	...	
7	2401	1695	...		
8	4096	...			

Вводимые параметры программы: **n** — порядок полинома; **a** и **b** — начальный и конечный аргументы таблицы; **m** — количество строк таблицы, которые хотим вычислить; **d(-1:n)** — полная начальная строка. Для её расчёта надо знать $n+1$ первых значений полинома.

Случай Б. Схема Бэббиджа позволяет разместить в начальной строке разностной таблицы разности, получающиеся не только по нисходящей диагонали, но и по соответствующей восходящей. Например, если в начальную строку таблицы поместить соответственно

$$7,\ 2401,\ 1695,\ 590,\ 156,\ 24,$$

($2401 = 7^4$), то получить значение $8^4 = 4096$ можно прибавив к **2041** значение первой разности **1695**. В этом случае расчёт разностей придётся вести в направлении убывания их номеров, в то время как в случае **A** расчёт должен был вестись в направлении возрастания их номеров.

(1)	(2)	(3)	(4)	(5)	(6)
7	2401	1695	590	156	24
8	4096	2465	770	180	24
9	6561	3439	974	204	24
10	10000	369	302	132	24

Разностная машина Бэббиджа была механической: для представления десятичных чисел использовались регистры, роль которых выполняли *счётные колёса*; каждой колонке, кроме (1), соответствовал свой регистр, которых было 7, и каждый хранил числа из 18 десятичных разрядов.

Вычислительная часть машины была связана с печатающим механизмом. Результат расчёта автоматически выгравировывался на медной пластинке, которая затем использовалась для получения нужного количества оттисков, причём процесс расчёта текущей строки таблицы совмещался по времени с выводом ранее вычисленной.

Строгое постоянство **n**-ых разностей выполняется только для полиномов **n**-ой степени. Однако, многие математические функции могут с высокой степенью точности приближаться полиномами. Именно поэтому важно знать алгоритмы расчёта полиномов. Теперь мы знаем два из них: схема Горнера (см. домашние задания первого и второго семестров) и разностная схема машины Бэббиджа.

При сдаче зачёта по задаче **№2** важно уметь правильно обосновать результаты пропуска программы при расчёте полинома x^4 (**mp=4, a=1.0, b=10.0**) для следующих значений **m**:

1. $m = 9$. Почему при m=9 схема Бэббиджа получает точный результат P(10.0)=10000.000?

2. $m = 7$. Почему при m=7 результат P(10)=9999.994 менее точен нежели при m=9, хотя количество операций сложения при m=9 больше чем при m=7?

3. $m = 72$. Почему при m=72 результат P(10)=10000.000 точен?

4. $m = 144$. Почему при m=144 результат P(10)=10000.001, т.е. абсолютная погрешность $\approx 10^{-3}$, хотя значения всех аргументов таблицы имеют нулевую погрешность округления, как и в случае m=72?

Вывод каждой строки разностной таблицы выгодно дополнить выводом соответствующих абсолютной и относительной погрешностей значений полинома, вычисленных по схкеме Бэббиджа, взяв в качестве точного результат, полученный по схеме Горнера. Выгода состоит в том, что при изменении **mp** нет нужды изменять соответственно число цифр мантиссы в значении полинома, так как качество точности расчёта легко проследить по значениям абсолютной и относительной погрешностей.

9.2 Возможные затрагиваемые темы программирования

1. Одномерный массив (основная). Предполагается, что после задания содержимого начальной строки разностной схемы, потребуется лишь одномерный массив (НЕ МАТРИЦА)

2. Двумерные массивы (дополнительная). Матрицу, например, можно использовать для автоматического расчёта начальной строки (вторая задача), хотя проще обойтись одномерным.

3. Линейный связный список. В третьей задаче вместо одномерного вектора требуется использовать связный список для хранения аргумента, значения полинома и соответствующих разностей.

4. Форматные дескрипторы **a**, **x** (или **t**), **f** (или **e**, или **g**) и организация наглядного форматного вывода разностной схемы.

5. Внутренний файл. Использование **character**-переменной в качестве **внутреннего файла** для моделирования динамического повторителя элементов форматной строки. Так, вывод строки разностной таблицы для полинома четвёртой степени требует **шесть** столбцов, что при задании формата вывода указывается, например, оператором **write(*,'(6e12.3)') (d(i),i=-1,4)**, где **6** — повторитель формата. Не всякий компилятор допускает в качестве повторителя целочисленную переменную. Можно обойтись некоторой целочисленной константой для повторителя, например, **1000**. Однако, формировку нужной форматной строки можно поручить программе. Пусть она автоматически вставляет в форматную строку нужное значение повторителя (полезное упражнение по ФОРТРАНу).

6. Явное указание интерфейса процедур. В современном ФОРТРАНе берём за правило всегда явно (в частности, модульно) описывать интерфейс своих пользовательских процедур, хотя, если не выходить за рамки старого ФОРТРАНа, то этого можно и не делать.

7. Модульное программирование. Обеспечиваем простой переход на иную разновидность типа **real**. Подключаем механизм перегрузки функций за счёт описания родовых имён модульных процедур, что позволит, например, вызывать по одному имени **initial** и функцию **initial_2** и функцию **initial_2l**.

9.3 Вариант 1

9.3.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема A**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

неразмещаемые массивы **p(0:nmax)** и **d(-1:nmax)** (**nmax=100**);

внешнюю подпрограмму **rdr_1(p,n)** ввода коэф. полинома;

внешнюю подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

внешнюю подпрограмму **initial_1(d,a,n)** ввода начальной строки;

внешнюю подпрограмму **Babbage(d,n)** расчёта очередной строки;

внешнюю ФУНКЦИЮ **Horner(p,x,n)** расчёта значения полинома.

9.3.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.3.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.3.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.4 Вариант 2

9.4.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема A**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

неразмещаемые массивы **p(0:nmax)** и **d(-1:nmax)** (**nmax=100**);

модульную подпрограмму **rdr_1(p,n)** ввода коэф. полинома;

модульную подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

модульную подпрограмму **initial_1(d,a,n)** ввода первой строки;

модульную подпрограмму **Babbage(d,n)** расчёта очередной строки.

модульную функцию **Horner(p,x,n)** расчёта значения полинома.

9.4.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.4.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.4.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.5 Вариант 3

9.5.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема A**).

Программа должна использовать:

удобный переход на любую возможную разновидность типа **real**;

размещаемые массивы **p(0:n)** и **d(-1:n)**;

внешнюю подпрограмму **rdr_1(p,n)** ввода коэф. полинома;

внешнюю подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

внешнюю подпрограмму **initial_1(d,a,n)** ввода первой строки;

внешнюю подпрограмму **Babbage(d,n)** расчёта очередной строки;

внешнюю ФУНКЦИЮ **Horner(p,x,n)** расчёта значения полинома.

9.5.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.5.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.5.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.6 Вариант 4

9.6.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема A**).

Программа должна использовать:

удобный переход на любую из разновидностей типа **real**.

размещаемые массивы **p(0:n)** и **d(-1:n)**.

модульную подпрограмму **rdr_1(p,n)** ввода коэф. полинома.

модульную подпрограмму **wrt_1(p,n)** вывода коэф. полинома.

модульную подпрограмму **initial_1(d,a,n)** ввода первой строки;

модульную подпрограмму **Babbage(d,n)** расчёта очередной строки;

модульную функцию **Horner(p,x,n)** расчёта значения полинома.

9.6.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.6.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.6.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.7 Вариант 5

9.7.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема A**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

размещаемые массивы **p(0:n)** и **d(-1:n)**;

внешнюю ФУНКЦИЮ rdrf_1(n) ввода коэф. полинома;

внешнюю подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

внешнюю ФУНКЦИЮ initial_1(a,n) ввода первой строки;

внешнюю подпрограмму **Babbage(d,n)** расчёта очередной строки;

внешнюю функцию **Horner(p,x,n)** расчёта значения полинома.

9.7.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.7.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.7.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.8 Вариант 6

9.8.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема A**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

размещаемые массивы **p(0:n)** и **d(-1:n)**;

модульную ФУНКЦИЮ rdrf_1(n) ввода коэф. полинома;

модульную подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

модульную ФУНКЦИЮ initial_1(a,n) ввода первой строки;

модульную подпрограмму **Babbage(d,n)** расчёта очередной строки;

модульную функцию **Horner(p,x,n)** расчёта значения полинома.

9.8.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.8.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.8.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.9 Вариант 7

9.9.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x) = \sum_{k=0}^{n} p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема Б**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

неразмещаемые массивы **p(0:nmax)** и **d(-1:nmax)** (**nmax=100**);

внешнюю подпрограмму **rdr_1(p,n)** ввода коэф. полинома;

внешнюю подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

внешнюю подпрограмму **initial_1(d,a,n)** ввода начальной строки;

внешнюю подпрограмму **Babbage(d,n)** расчёта очередной строки;

внешнюю ФУНКЦИЮ **Horner(p,x,n)** расчёта значения полинома.

9.9.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.9.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.9.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.10 Вариант 8

9.10.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема Б**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

неразмещаемые массивы **p(0:nmax)** и **d(-1:nmax)** (**nmax=100**);

модульную подпрограмму **rdr_1(p,n)** ввода коэф. полинома;

модульную подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

модульную подпрограмму **initial_1(d,a,n)** ввода первой строки;

модульную подпрограмму **Babbage(d,n)** расчёта очередной строки.

модульную функцию **Horner(p,x,n)** расчёта значения полинома.

9.10.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.10.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.10.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.11 Вариант 9

9.11.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема Б**).

Программа должна использовать:

удобный переход на любую возможную разновидность типа **real**;

размещаемые массивы **p(0:n)** и **d(-1:n)**;

внешнюю подпрограмму **rdr_1(p,n)** ввода коэф. полинома;

внешнюю подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

внешнюю подпрограмму **initial_1(d,a,n)** ввода первой строки;

внешнюю подпрограмму **Babbage(d,n)** расчёта очередной строки;

внешнюю ФУНКЦИЮ **Horner(p,x,n)** расчёта значения полинома.

9.11.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.11.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.11.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.12 Вариант 10

9.12.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема Б**).

Программа должна использовать:

удобный переход на любую из разновидностей типа **real**.

размещаемые массивы **p(0:n)** и **d(-1:n)**.

модульную подпрограмму **rdr_1(p,n)** ввода коэф. полинома.

модульную подпрограмму **wrt_1(p,n)** вывода коэф. полинома.

модульную подпрограмму **initial_1(d,a,n)** ввода первой строки;

модульную подпрограмму **Babbage(d,n)** расчёта очередной строки;

модульную функцию **Horner(p,x,n)** расчёта значения полинома.

9.12.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.12.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.12.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.13 Вариант 11

9.13.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема Б**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

размещаемые массивы **p(0:n)** и **d(-1:n)**;

внешнюю ФУНКЦИЮ rdrf_1(n) ввода коэф. полинома;

внешнюю подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

внешнюю ФУНКЦИЮ initial_1(a,n) ввода первой строки;

внешнюю подпрограмму **Babbage(d,n)** расчёта очередной строки;

внешнюю функцию **Horner(p,x,n)** расчёта значения полинома.

9.13.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.13.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.13.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

9.14 Вариант 12

9.14.1 Задача №1

Написать программу, которая моделирует работу разностной машины Чарльза Бэббиджа. Программа должна вводить:

a и **b** — начальный и конечный аргументы разностной таблицы;

m — число участков равномерной дискретизации **[a,b]**;

n — показатель степени вычисляемого полинома;

p(0:n) — коэффициенты полинома $\mathbf{P_n(x)} = \sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{p_k x^{n-k}}$;

d(-1:n) — начальная строка схемы Бэббиджа. d(-1)=a — аргумент, d(0)=$P_n(a)$. d(1:n) — первые n разностей (**схема Б**).

Программа должна использовать:

удобный переход на любую из возможных разновидностей типа **real**;

размещаемые массивы **p(0:n)** и **d(-1:n)**;

модульную ФУНКЦИЮ rdrf_1(n) ввода коэф. полинома;

модульную подпрограмму **wrt_1(p,n)** вывода коэф. полинома;

модульную ФУНКЦИЮ initial_1(a,n) ввода первой строки;

модульную подпрограмму **Babbage(d,n)** расчёта очередной строки;

модульную функцию **Horner(p,x,n)** расчёта значения полинома.

9.14.2 Задача №2

Заменить процедуру **initial_1** процедурой **initial_2**, которая должна вычислять (не используя матрицу) то, что вводит **initial_1**.

9.14.3 Задача №3

Заменить в задаче №2 вектора **p(0:n)** и **d(-1:n)** связными списками. Описать соответствующие типы данных и, работающие со списками, процедуры: **initial_2l**, **Babbage_l**, **Horner_l**.

9.14.4 Задача №4

В задачах №2 и №3 обеспечить вызов (в пределах одной программы) процедур с близкими по сути выполняемой работы именами по одному имени, например, **rdr_1** и **rdr_1l** по имени **rdr**, **wrt_1** и **wrt_1l** — по имени **wrt**; **initial_2** и **initial_2l** — по имени **initial**; **Babbage** и **Babbage_l** — по мени **Babbage**; **Horner** и **Horner_l** — по имени **Horner**.

10 Приложение I. Функция передачи типа transfer

Функция **transfer(source, mold [, size])** (функция передачи типа; см. [2]) или (одна из функций преобразования типа; см. [3]), а точнее функция, позволяющая *взглянуть* на данное типа первого позиционного аргумента *через очки* типа данного второго позиционного аргумента, не осуществляя формального преобразования типов. Другими словами, взглянуть на данное **source** так, как если бы оно было типа **mold**, не изменяя внутреннего машинного битового представления **source**.

В некоторых ситуациях очень полезная функция, по сути обобщающая действие оператора **equivalence**, который в современном ФОРТРАНе считается архаичным. Рассмотрим примеры.

1. Допустим, что у нас нет функций **achar** и **iachar**. Как в этом случае по коду символа получить сам символ и наоборот по символу получить его код? В старом ФОРТРАНе была возможна программа:

```
program transf0                     ! Практически, ничего не делая,
implicit none                       ! смотрим на одну и туже область
character(1) x                      ! памяти: то как на данное типа
integer(1)   k                      ! character(1), то как на данное
equivalence (x,k)                   ! integer(1). И функции achar и
x='z'; write(*,*) ' x=',x,'  k=',k  ! iachar остаются без работы,
k=48;  write(*,*) ' x=',x,'  k=',k  ! хотя в современном фортране это
                                          ! можно получить так:
x='z'; write(*,*) ' x=',x,'  k=',iachar(x)
k=48;  write(*,*) ' x=',achar(k),'  k=',k
                                                        ! Однако, есть
x='z'; k=transfer(x,k); write(*,*) ' x=',x,'  k=',k     ! и такой способ
k=48;  x=transfer(k,x); write(*,*) ' x=',x,'  k=',k
end
```

Её результат:

```
 x=z  k=  122
 x=0  k=   48
 x=z  k=         122
 x=0  k=   48
 x=z  k=  122
 x=0  k=   48
```

Кто теперь скажет, что **equivalence** — архаичен? Просто не нужно его использовать не по делу там, где легко ошибиться.

2. В ФОРТРАНе нет, как в СИ, беззнакового целого типа. Поэтому иногда результат работы на типе **integer(4)** выглядит как отрицательное число из-за происшедшего якобы переполения. Так программа **random_seed** начальной настройки **random_number** (генератора равномерно распределённых по промежутку [0,1) чисел) получает *затравочные целые* в массиве **get** типа **integer(4)**. При их выводе могут встретиться и отрицательные целые. Однако можно взглянуть на них *через очки* типа **integer(8)**.

```
program transf1                          ! Работая с transfer не следует
implicit none                            ! забывать одну её пакостную
integer(4), allocatable :: sget(:)       ! особенность, за которую её
integer(8) i8                            ! можно критиковать с тем же
integer(4) :: n, i, k(2)=(/0,0/)         ! успехом, с каким раньше
call random_seed(n)                      ! критиковали equivalence.
write(*,*) ' n=',n                       ! Именно, архиважно следить,
allocate(sget(n))                        ! чтобы первый аргумент занимал
call random_seed()                       ! память того же объёма, что и
call random_seed(get=sget)               ! результат.
write(*,'("В первый раз крупно повезло!!!")')
write(*,'(2x,"i",5x,"sget",9x,"i8")')
do i=1,n                                   ! САМЫЙ ОПАСНЫЙ СЛУЧАЙ:
  i8=transfer(sget(i),i8)                  ! Результат верен. Но здесь
  write(*,'(i3,i12, i12)') i,sget(i),i8    ! нам просто "крупно повезло",
enddo                                      ! так как,
write(*,'("Второе повторение того же даёт уже другой результат;")')
write(*,'(2x,"i",5x,"sget",9x,"i8")')
do i=1,n
  i8=transfer(sget(i),i8)
  write(*,'(i3,i12, i12)') i,sget(i),i8
enddo
end
```

Её результат:

```
  n=           8               !
В первый раз крупно повезло!!! !   Второе повтор --- результат иной
  i     sget         i8        !     i     sget         i8
  1   437395160   437395160    !     1   437395160  4732362456
  2  1404128605  1404128605    !     2  1404128605  5699095901
  3   572505362   572505362    !     3   572505362  4867472658
  4 -1187264075  3107703221    !     4 -1187264075  7402670517
  5   454383258   454383258    !     5   454383258  4749350554
  6   525702629   525702629    !     6   525702629  4820669925
  7   973594203   973594203    !     7   973594203  5268561499
  8  1758310677  1758310677    !     8  1758310677  6053277973
```

Дело в том, что, собираясь смотреть на тип **integer(4)** глазами **integer(8)**, мы должны подать типу **integer(8)** для обзора **восемь байт**, в то время как тип **integer(4)** предоставляет их только **четыре**. Поэтому, что может увидеть тип **integer(8)** на месте старших четырёх байтов своего поля видимости, можно только гадать. В первый раз нас, видимо, спас интеллект компилятора (или случайность). Во второй — *интеллект* подвёл (может ошибка в старой версии компилятора). Разъяснение ситуации дано, например, в [4]:

если **source** — последовательность из **n** бит $(b_1 b_2 \cdots b_n)$, а данное **mold** — последовательность из **m** бит $(s_1 s_2 \cdots s_m)$, то при

- **n=m** : $b_1 b_2 \cdots b_n$, что видели на примере программы **transf0**.
- **n<m** : $b_1 b_2 \cdots b_n s_1 s_2 \cdots s_{m-n}$. Биты $s_1 s_2 \cdots s_{m-n}$ не определены!
- **n>m** : $b_1 b_2 \cdots b_m$.

3. Как следует поступить? — Подать в качестве **source** не 4 байта, а 8, например, целочисленный массив из двух элементов: в первый элемент записать наше данное из **sget(i)**, а во второй — нули:

```
program transf2; implicit none; integer(4), allocatable :: sget(:)
integer(8) i8;   integer(4) :: n, i, k(2)
call random_seed(n)
write(*,*) ' n=',n
allocate(sget(n))
call random_seed(); call random_seed(get=sget)
write(*,'("Теперь и в первый раз,")')
write(*,'(2x,"i",5x,"sget",5x,"k(2)",5x,"k(1)",9x,"i8")')
do i=1,n
  k(1)=sget(i); k(2)=0_4
  i8=transfer(k,i8)
  write(*,'(i3,i12, i5, i12, i12)') i,sget(i), k(2), k(1), i8
enddo
write(*,'("и во второй --- результаты одинаково верны!")')
write(*,'(2x,"i",5x,"sget",5x,"k(2)",5x,"k(1)",9x,"i8")')
do i=1,n
  k(1)=sget(i); k(2)=0_4
  i8=transfer(k,i8)
  write(*,'(i3,i12, i5, i12, i12)') i,sget(i), k(2), k(1), i8
enddo
end
```

Теперь результаты правильны в обоих случаях.

Заметим, что обычно элемент массива с более старшим индексом расположен по бóльшему адресу нежели элемент с более младшим индексом. Поэтому значение **sget(i)** важно помещать именно в **k(1)**, а не в **k(2)**.

Заметим также, что при использовании оператора **equivalence** всё выглядело бы до смешного проще: не потребовалось тратить время на вызов **transfer** вообще, и вряд ли бы совершили *ляпу* из программы **transf1.f95**, поскольку бы сразу задумались, что надо поместить в старшие четыре байта результата:

```
program transf3; implicit none; integer(4), allocatable :: sget(:)
integer(8) i8;   integer(4) :: n, i, k(2)
equivalence (k(1),i8)
call random_seed(n); write(*,*) ' n=',n
allocate(sget(n))
call random_seed(); call random_seed(get=sget)
write(*,'("Теперь и в первый раз,")')
write(*,'(2x,"i",5x,"sget",5x,"k(2)",5x,"k(1)",9x,"i8")')
do i=1,n
  k(1)=sget(i); k(2)=0_4
  write(*,'(i3,i12, i5, i12, i12)') i,sget(i), k(2), k(1), i8
enddo
write(*,'("и во второй --- результаты одинаково верны!")')
write(*,'(2x,"i",5x,"sget",5x,"k(2)",5x,"k(1)",9x,"i8")')
do i=1,n
  k(1)=sget(i); k(2)=0_4
  write(*,'(i3,i12, i5, i12, i12)') i,sget(i), k(2), k(1), i8
enddo
end
```

```
 n=          12
 Теперь и в первый, и во второй  раз --- результаты одинаково верны!
 i     sget     k(2)     k(1)         i8    !  k(2)     k(1)         i8
 1   287027030    0   287027030   287027030 !  0   287027030   287027030
 2  -719361131    0  -719361131  3575606165 !  0  -719361131  3575606165
 3   574274270    0   574274270   574274270 !  0   574274270   574274270
 4   292048305    0   292048305   292048305 !  0   292048305   292048305
 5   185733336    0   185733336   185733336 !  0   185733336   185733336
 6 -1598963619    0 -1598963619  2696003677 !  0 -1598963619  2696003677
 7   572469522    0   572469522   572469522 !  0   572469522   572469522
 8  1446716853    0  1446716853  1446716853 !  0  1446716853  1446716853
 9   437591706    0   437591706   437591706 !  0   437591706   437591706
10  1398099429    0  1398099429  1398099429 !  0  1398099429  1398099429
11   570932571    0   570932571   570932571 !  0   570932571   570932571
12 -1177695979    0 -1177695979  3117271317 !  0 -1177695979  3117271317
```

4. Приведём ещё пример, поясняющий суть дела при работе с **transfer**

```
program transf4; implicit none; integer(4) :: i, k(2); integer(8) i8
i=-huge(1_4)-1; write(*,'(" 0)  i=",i12,"...",b32)') i,i
i8=transfer(i,i8); write(*,*) '1) i8=',i8    ! Почему
i8=transfer(i,i8); write(*,*) '2) i8=',i8    !   разные
i8=transfer(i,i8); write(*,*) '3) i8=',i8    !     результаты?
i8=transfer(i,i8); write(*,*) '4) i8=',i8    !
k(1)=-huge(1_4)-1; k(2)=0_4                  ! <--= Безопасный
i8=transfer(k,i8); write(*,*) ' i8=',i8      !           вариант.
write(*,'("  i",9x,"k(2)",8x,"k(1)",8x,"i8")')
do i=-2,2; k(1)=i; i8=transfer(k,i8)
  write(*,'(i3,i12,i12, i20)') i,k(2),k(1),i8
enddo
end
```

```
 0)  i= -2147483648...10000000000000000000000000000000
 1) i8=      139820512837632
 2) i8=           2147483648
 3) i8=           2147483648
 4) i8=      140735340871680
  i8=           2147483648
  i         k(2)        k(1)        i8
 -2           0          -2          4294967294
 -1           0          -1          4294967295
  0           0           0                   0
  1           0           1                   1
  2           0           2                   2
```

Опять странно, что, выполняя одно и то же, получаем разные результаты Либо это — накладка древнего компилятора, либо — намёк на необходимость более ответственно задавать аргумент **source**.

5. Результат функции **transfer** либо скаляр (если **mold** — скаляр), либо вектор (если **mold** — массив любой формы и размера).

```
program transf5; implicit none; integer i, j, k
integer :: a(4,4), b(16), d(6), e(18)
k=1; do i=1,4; do j=1,4; a(i,j)=k; k=k+1; enddo; enddo
b=33; b=transfer(source=a,mold=b)       ! SIZE не указан: число элементов
      write(*,'(" b:",16i3)') b         !     в a и b одинаково (4*4=16).
d=55; d=transfer(source=a,mold=d)       !     Число элементов в a больше
      write(*,'(" d:",25i3)') d         !     числа элементов в d (16>6).
e=77; e=transfer(source=a,mold=e)       !     Число элементов в a меньше
      write(*,'(" e:",25i3)') e         !     числа элементов в e (16<18)
e=77; e(1:6)=transfer(source=a,mold=e,size=6) ! SIZE=6 УКАЗАН и задаёт
write(*,'(" e(1:6):",25i3)') e                ! размер вектора результата
end
```

Если **source** - массив любой формы и размера, то результат — всегда одномерный массив, у которого заполняются либо все элементы, либо столько, сколько их в **mold** — в зависимости от того, чей заявленный размер больше и отсутствия при вызове **transfer** аргумента **size**.

```
 b:  1  5  9 13  2  6 10 14  3  7 11 15  4  8 12 16
 d:  1  5  9 13  2  6
 e:  1  5  9 13  2  6 10 14  3  7 11 15  4  8 12 16 77 77
 e(1:6):  1  5  9 13  2  6 77 77 77 77 77 77 77 77 77 77 77 77
```

Наличие же **size**-аргумента с необходимостью требует, чтобы заявленный размер вектора-результата был равен **size**. Например, если бы в третьей строке снизу программы **transf5** вместо **e(1:6)**= написали бы **e**=, то получили бы сообщение об ошибке:

```
transf5.f95:10.6:

e=77; e=transfer(source=a,mold=e,size=6)
      1
      ошибка: Different shape for array assignment at (1) on dimension 1
                                                                  (18 and 6)
```

6. Ещё один пример работы **transfer** — получение из набора комплексных данных набора вещественных чисел, в котором Re и Im комплексных чередуются, а затем наоборт из одномерного массива типа **real** получить одномерный массив типа **complex**:

```
program transf6;
implicit none
complex(4) :: c(5)=(/(1.1,1.01),(1.2,1.02),(1.3,1.03),(1.4,1.04),&
                                                    & (1.5,1.05)/)
real(4)    r(10)
integer i
write(*,'(4x,"i",8x,"Re(c)",8x,"Im(c)"/(i5,e15.7,e15.7))') (i,c(i),i=1,5)
r=transfer(c,r)
write(*,'(4x,"i",8x,"r"/(i5,e15.7))') (i,r(i),i=1,10)
r(2:10:2)=-r(2:10:2)
c=transfer(r,c)
write(*,'(4x,"i",8x,"Re(c)",8x,"Im(c)"/(i5,e15.7,e15.7))') (i,c(i),i=1,5)
c(1:3)=transfer(r,c,3)
write(*,'(4x,"i",8x,"Re(c)",8x,"Im(c)"/(i5,e15.7,e15.7))') (i,c(i),i=1,3)
end
```

```
 i        Re(c)          Im(c)
 1  0.1100000E+01  0.1010000E+01
 2  0.1200000E+01  0.1020000E+01
 3  0.1300000E+01  0.1030000E+01
 4  0.1400000E+01  0.1040000E+01
 5  0.1500000E+01  0.1050000E+01
 i        r
 1  0.1100000E+01
 2  0.1010000E+01
 3  0.1200000E+01
 4  0.1020000E+01
 5  0.1300000E+01
 6  0.1030000E+01
 7  0.1400000E+01
 8  0.1040000E+01
 9  0.1500000E+01
10  0.1050000E+01
 i        Re(c)          Im(c)
 1  0.1100000E+01 -0.1010000E+01
 2  0.1200000E+01 -0.1020000E+01
 3  0.1300000E+01 -0.1030000E+01
 4  0.1400000E+01 -0.1040000E+01
 5  0.1500000E+01 -0.1050000E+01
 i        Re(c)          Im(c)
 1  0.1100000E+01 -0.1010000E+01
 2  0.1200000E+01 -0.1020000E+01
 3  0.1300000E+01 -0.1030000E+01
```

11 Приложение II. Азы GNU-make (часть 2)

Освоение возможностей утилиты **make** на основе примера, использовавшегося при начальном знакомстве с ней (см. пункты **4.3.1** и **4.3.2**) первого семестра. В **4.3.1** была рассмотрена ФОРТРАН-программа

```
program probtask;    ! Модель проблемной программы.  Файл probtask.f90
include 'probtask.hdr'               ! Подключение описаний.
open(unit=ninp,file='probtask.par')  ! Открытие файла  ввода.
open(unit=nres,file='result')        ! Открытие файла вывода.
read(ninp,100) n,a,b                 ! Ввод числа узлов и границ отрезка.
write(nres,110) n, a, b              ! Контрольная печать введенного.
h=(b-a)/(n-1)                        ! Расчет шага дробления [a,b].
write(nres,1000)                     ! Печать заголовка первой таблицы.
do i=1, n                            ! Для каждого из узлов
  x=a+(i-1)*h                        !   вычисляем текущий аргумент,
  y=fun(x)/2                         !   фиксируем результат fun(x),
  call subfun(x,z)                   !   получаем результат subfun
  write(nres,1001) i, x, y, z/4      !   вывод текущего результата
enddo                                !
close(nres)                          ! Закрытие файла результата.
  100 format(i10/d10.3/d10.3)        ! Список форматов ввода-вывода:
  110 format(1x,'#  Число узлов дискретизации (n)=',i4/&
&            1x,'# Левая граница промежутка  (a)=',d23.16/&
&            1x,'# Правая граница промежутка (b)=',d23.16)
 1000 format(//1x,'# N',15x,'x',21x,'fun(x)/2',19x,' z/4 ')
 1001 format(1x,i3,2x,d23.16,2x,d23.16,2x,d23.16)
end
```

Программа выполняет **n** точечное табулирование с равномерным шагом по аргументу из промежутка **[a,b]** функций **(2x-3)/2** и **(2x-3)/4**. Расчёт числителя оформлен двояко: функцией **fun** и подпрограммой **subfun**.

```
function fun(x) result(w)                ! Файл fun.f90
implicit none
real(8) w, x
w=2d0*x-3d0
end function fun
subroutine subfun(x,res)                 ! Файл subfun.f90
implicit none
real(8) x, res
res=2d0*x-3d0
end subroutine subfun
```

Здесь на простом примере напоминаются разные варианты описания алгоритма, поскольку первое знакомство с утилитой **make** проходило сразу после знакомства с темой **“Подпрограммы и функции”**.

Настоящее приложение просто использует эту простую задачу в качестве полигона, на котором будут продемонстрированы дополнительные возможности утилиты **make** для построения проекта компиляции и компоновки данной программы, которая в свою очередь моделирует достаточно широкий класс прикладных программ. Напомним существенные особенности рассматриваемого примера.

1. Функция и подпрограмма специально расположены в разных файлах, моделируя ситуацию компоновки многофайлового проекта.

2. В качестве расчётных формул специально выбраны *тривиальные* выражения именно для того, чтобы всё внимание было уделено в основном синтаксису описания функций и подпрограмм.

3. Даже по упомянутым тривиальным выражениям расчёт на ЭВМ при значениях **x** очень близких к **1.5** может дать результат весьма сомнительной точности (даже на типе **real(8)**).

4. В главной программе имеется инструкция **include ’probtask.hdr’**, которая информирует компилятор о необходимости включить в исходный текст программы **probtask** содержимое файла с именем **probtask.hdr**:

```
!============================================================
! Переменные главной программы            :   Файл probtask.hdr
!.........................................:...................
implicit none       ! Отмена правила умолчания ФОРТРАНа.
interface
function fun(x) result(w); real(8) x, w; end function fun
subroutine    subfun(x,w); real(8) x, w; end subroutine subfun
end interface
integer ninp / 5 /  ! Номер файла с вводимыми параметрами задачи.
integer nres / 6 /  ! --"--"--"-- с выводимымым результатом.
integer n           ! Количество узлов дискретизации промежутка.
integer i           ! Номер текущего узла.
real(8) a, b        ! Левая и правая абсциссы его границ.
real(8) h           ! Шаг его равномерной дискретизации.
real(8) x           ! Текущий аргумент
real(8) y           ! Хранитель значения, получаемого функцией.
real(8) z           ! -"--"--"--"--"--"--"--"- подпрограммой.
```

В данном случае таким содержимым служит раздел описания интерфейсов процедур и типов переменных, используемых в главной программе. Подобное (через файл-посредник) подключение описаний вряд ли целесообразно для короткой программы. Если переменных немного, то описание их структур и типов удобнее обозревать в явном виде перед разделом выполняемых операторов, нежели используя файл-посредник. Тем не менее подобная вставка текста может встретиться в программах, с которыми вам возможно придётся работать.

Заметим, что в **probtask.hdr** нет описания **real(8) fun**, указывающего тип значения, возвращаемого функцией (что было бы необходимо в старом ФОРТРАНе). Вместо этого в исходный текст файла включено описание **интерфейса** и функции **fun**, и подпрограммы **subfun**. Напомним, что наличие интерфейса позволит компилятору при компиляции главной программы проконтролировать соответствие количества и типов фактических аргументов процедур атрибутам их формальных аргументов.

Директива **include** ранее уже встречалась нам (см. пункты **3.1.1**, **3.1.6**, **4.3.1**). Напомним ещё раз, что директива **include** внедряет в место её расположения в текущем файле содержимое файла, имя которого указано ей в качестве параметра. Каким должно быть это содержимое – набор нескольких независимых программных единиц или же просто несколько ФОРТРАН-операторов – решаем мы.

При этом помним, что в случае подключения подобным способом процедур лишь создаётся иллюзия краткости исходного текста (ведь компилятор будет перекомпилировать все процедуры, входящие в файл, имя которого указано в инстркуции **include**). Поэтому временные затраты на компиляцию программы с подключаемыми через **include** процедурами, а так же на поиск и исправление синтасических неточностей, могут оказаться существенно более длительными чем при оптимальном подходе на основе использования координатора **make**.

Тем не менее уже наш пример позволит выявить некоторую неточность в подготовленном нами ранее (см. пункт **4.3.2**) **make**-файле.

11.1 Дальнейшие усовершенствования

1. Простые переменные утилиты make.
2. Операторы присваивания утилиты make.
3. Автоматические переменные утилиты make.
4. Уяснение выгоды от использования простых переменных.

11.1.1 Простые переменных утилиты make

Рассмотрим фрагмент **make**-файла (см. **4.3.2**), который координирует сборку и выполнение программы **probtask**, приведённой в начале приложения **2**.

```
     probtask : probtask.o subfun.o fun.o
[  TAB  ]       gfortran probtask.o subfun.o fun.o -o probtask
   probtask.o : probtask.f90
[  TAB  ]       gfortran -c probtask.f90
     subfun.o : subfun.f90
[  TAB  ]       gfortran -c subfun.f90
        fun.o : fun.f90
[  TAB  ]       gfortran -c fun.f90
       result : probtask probtask.par
[  TAB  ]       ./probtask
```

Исполнимый файл с именем **probtask** собирается из трех объектных **probtask.o**, **subfun.o** и **fun.o**, которые генерируются из трёх исходных файлов **probtask.f90**, **subfun.f90** и **fun.f90** посредством вызова компилятора с опцией **-c**.

Хотелось бы, чтобы запись **make**-файла выглядела наглядной, краткой и в меньшей мере подчёркивала бы зависимость от имени компилятора и от используемых им опций. Для этого команду вызова и нужные опции естественно запомнить в двух отдельных **make**-переменных (назовём их, например, **COMP** и **OPT**), содержимое которых будем извлекать по мере надобности.

Извлечь значение из простой переменной утилиты **make** позволяет конструкция **$(имя_переменной)**, а присвоить простой переменной значение (см., например, [13]) можно операторами **=** или **:=** С учётом сказанного предыдущий **make**-файл примет вид:

```
COMP=gfortran
 OPT=-c -O0
     probtask : probtask.o subfun.o fun.o
[  TAB  ]       $(COMP) probtask.o subfun.o fun.o -o probtask
   probtask.o : probtask.f90
[  TAB  ]       $(COMP) $(OPT) probtask.f90
     subfun.o : subfun.f90
[  TAB  ]       $(COMP) $(OPT) subfun.f90
        fun.o : fun.f90
[  TAB  ]       $(COMP) $(OPT) fun.f90
       result : probtask probtask.par
[  TAB  ]       ./probtask
```

11.1.2 Операторы присваивания утилиты make

1. Первый = – традиционный способ. При использовании = значение переменной вычисляется **в момент ее использования**.

2. Второй **:=** делает переменную подобной обычным переменным языков программирования ФОРТРАН, СИ или ПАСКАЛЬ, когда значение переменной вычисляется **в момент обработки оператора присваивания**.

3. Переменная может менять свой статус в соответствии с типом оператора присваивания, который применялся к ней последним. Оператор **:=** кажется более простым, правда, за счет сужения возможностей оператора =.

Помимо **make**-переменных с именами, придуманными нами, утилита **make** предоставляет так называемые **автоматические** переменные, которые позволяют существенно уменьшить количество досадных опечаток при наборе **make**-файла.

11.1.3 Автоматические переменные утилиты make

Автоматическая переменная – переменная с *особым* именем, которая **автоматически** принимает некоторое значение перед тем как выполняются команды достижения цели (команд может быть и несколько). Например, правило

```
     probtask : probtask.o subfun.o fun.o
[  TAB  ]       $(COMP) probtask.o subfun.o fun.o -o probtask
```

с использованием автоматических переменных **$^** и **$@** запишется так:

```
     probtask : probtask.o subfun.o fun.o
[  TAB  ]  $(COMP) $^  -o $@
```

- **$^** – имя автоматической переменной, содержащей список всех зависимостей цели, в данном случае это список **probtask.o subfun.o fun.o**, позволяя избежать перенабора списка в команде вызова компоновщика.
- **$@** содержит имя цели, точнее, имена всех целей, список которых находится левее **:** в заголовке рассматриваемого правила. **$@** выгодна, когда имя файла, получаемого в результате достижения цели, должно совпадать с её именем.

```
COMP=gfortran        #           Возможный пример make-файла с использованием
 OPT=-g -c -O0       #                     автоматических переменных $^ и $@.
      probtask : probtask.o subfun.o fun.o #
[  tab  ]       $(COMP)        $^ -o $@    # Подчеркнем, что $^ содержит
   probtask.o : probtask.for               # в записи команды список ВСЕХ
[  TAB  ]       $(COMP) $(OPT) $^          # зависимостей только для той цели,
     subfun.o : subfun.for                 # которая достигается.
[  TAB  ]       $(COMP) $(OPT) $^          # Поэтому в записи правил достижения
        fun.o : fun.for                    # целей компиляции обозначение $^
[  TAB  ]       $(COMP) $(OPT) $^          # относится к файлу необходимому
       result : probtask probtask.par      # для достижения соответствующей
[  TAB  ]       ./probtask                 # цели, т.е. к исходному коду.
```

Наряду с переменными **$^** и **$@** может оказаться полезной и автоматическая переменная **$<**, содержащая имя первой зависимости текущей цели. Заметим, что наш предыдущий **make**-файл обладает одним существенным недостатком:

в нем никак не отражена зависимость от файла probtask.hdr.

Поэтому возможная модификация последнего останется незамеченной при работе утилиты **make**. Действительно,

1. ```
   $ touch probtask.hdr                    # Изменили время модификации файла,
   $ make                                  # подключаемого через include,
   make: ‘probtask’ не требует обновления.# при наличии исполнимого кода.
                                           # Но make этого не заметила !!!
   ```

2. ```
   $ rm -f probtask      # Удалили исполнимый файл, оставив объектные и освежив
   $ touch probtask.hdr  # версию файла probtask.hdr. Видим, что перекомпиляции
   $ make                # probtask.f90, как хотелось бы, не происходит, т.е.
                                                  #    исполнимый код собран
   gfortran probtask.o subfun.o fun.o -o probtask #    из старых объектных.
   ```

3. Зависимость цели **probtask.o** от обоих файлов и **probtask.f90**, и **probtask.hdr** объективно обоснована. Однако, при этом инструкция **$(COMP) $∧** с необходимостью приведёт к компиляции не только исходного текста главной программы **probtask.f90** (что действительно нужно), но и содержимого файла **probtask.hdr** (что бессмысленно, так как оно не представляет собой независимую программную единицу; оно уже встроено в текст **probtask.f90** посредством инструкции **include ’probtask.hdr’**). Расширение последнего **hdr** просто придумано программистом и не входит в список известных системе по умолчанию расширений. Поэтому реакция компилятора может оказаться своеобразной:

   ```
   $ rm -f *.o  probtask; $ touch probtask.hdr
   $ make
   gfortran -c -g -O0 probtask.for probtask.hdr
   gfortran: probtask.hdr: linker input file unused because linking not done
   gfortran        probtask.o subfun.o fun.o -o probtask
   ```

   Если бы имя файла, подключаемого через **include** было, например, **probtask.f**, то реакция утилиты **make** предсказуема:

   ```
   $ make
   gfortran -c -g -O0 probtask.f90 probtask.f
   Error: Unexpected end of file in ’probtask.f’
   make: *** [probtask.o] Ошибка 1
   ```

   Дело в том, что исходный текст, подключаемый через **include**, не представляет собой в нашем случае **единицу компиляции**, синтаксически завершаемую служебным словом **end**. Он не должен компилироваться вне текста главной программы.

4. Наличие же автоматической переменной **$<**, как ссылки именно на первую зависимость выбранной цели, вместо **$^**, решает проблему:

```
COMP=gfortran
 OPT=-c -O0
     probtask : probtask.o subfun.o fun.o # Если независимая программная
  [ TAB ]       $(COMP)        $^  -o $@  # ФОРТРАН-единица использует
   probtask.o : probtask.for probtask.hdr # инструкцию INCLUDE, то имя
  [ TAB ]       $(COMP) $(OPT) $<         # файла, содержащего её, выгодно
     subfun.o : subfun.f90                # указатьпервой зависимостью и
  [ TAB ]       $(COMP) $(OPT) $^         # для ссылкина неё использовать
        fun.o : fun.f90                   # автоматическую переменную $< .
  [ TAB ]       $(COMP) $(OPT) $^
      result : probtask probtask.par
  [ TAB ]       ./probtask

$ rm *.o
$ rm probtask
$ make
gfortran -c -g -O0 probtask.f90
gfortran -c -g -O0 subfun.f90
gfortran -c -g -O0 fun.f90
gfortran        probtask.o subfun.o fun.o -o probtask
$ touch *.hdr
$ make
gfortran -c -g -O0 probtask.f90
gfortran        probtask.o subfun.o fun.o -o probtask
$ make result
./probtask
$ cat result
#  Число узлов дискретизации (n)=   5
# Левая граница промежутка  (a)= 0.1499999990000000D+01
# Правая граница промежутка (b)= 0.1500000000000000D+01

# N              x                     fun(x)/2                  z/4
  1  0.1499999990000000D+01 -0.9999999939225290D-08 -0.4999999969612645D-08
  2  0.1499999992500000D+01 -0.7499999954418968D-08 -0.3749999977209484D-08
  3  0.1499999995000000D+01 -0.4999999969612645D-08 -0.2499999984806323D-08
  4  0.1499999997500000D+01 -0.2499999984806323D-08 -0.1249999992403161D-08
  5  0.1500000000000000D+01  0.0000000000000000D+00  0.0000000000000000D+00
```

После пропуска кода **probtask** получена таблица искомых функций с шагом $\mathbf{2.5 \cdot 10^{-9}}$ на промежутке $[\mathbf{1.49999999}, \mathbf{1.5}]$. Видно, что для аргумента, отличающегося от **1.5** на величину шага его дробления, результат верен лишь в пределах восьми старших десятичных значащих цифр, хотя расчёт ведется с шестнадцатью.

5. Напомним, что для получения результата верного в пределах 15–16-ти значащих цифр (без перехода на многозначную арифметику) достаточно исключить из расчётной формулы вычитание двух почти равных чисел в окрестности точки **x=1.5**. Так, если в формуле **2*x-3** сделать замену переменной $\mathbf{x=t+1.5}$, используя **t** в качестве нового аргумента, то формула сведётся к выражению, в котором опасное вычитание уже выполнено нами аналитически:

$$\mathbf{2*x-3=2(t+1.5)-3=2*t+3-3=2*t} \quad ,$$

а диапазон изменения переменной **t**, соответствующий диапазону изменения $\mathbf{x} \in [\mathbf{a}, \mathbf{b}]$ окажется $\mathbf{t} \in [\mathbf{a-1.5}, \mathbf{b-1.5}]$. При этом для задания значений границ изменения **t** нет нужды находить разности **a-1.5** и **b-1.5** посредством машинного вычитания. Их можно, вычислив самим один раз, просто ввести (ведь при $\mathbf{x} \in [\mathbf{1.49999999}, \mathbf{1.5}]$ нетрудно сообразить, что $\mathbf{t} \in [-\mathbf{10^{-9}}, \mathbf{0}]$). Модифицируя в соответствии со сказанным исходные тексты главной программы, функции, подпрограммы и вводимых данных

```
program probtask   ! Модифицированная программа.  Файл probtask.f90
include 'probtask.hdr'
open(unit=ninp,file='probtask.par')
open(unit=nres,file='result',status='replace')
read(ninp,100) n,a,b               ! Ввод числа узлов и границ.
write(nres,110) n, a, b            ! Контрольная печать введенного.
h=(b-a)/(n-1)                      ! Расчет шага дробления [a,b].
write(nres,1000)                   ! Печать заголовка таблицы.
do i=1, n                          ! Для каждого из узлов:
  t=a+(i-1)*h                      !     новый текущий аргумент
  x=1.5d0+t                        !     старый --"--"--"--"--
  y=funt(t)/2                      !     результат через funt(t),
  call subfunt(t,z)                !     --"--"--"--"--  subfunt
  write(nres,1001) i, t, x, y, z/4 !     вывод текущего результата
enddo
close(nres)
  100 format(i10/d10.3/d10.3)      ! Список форматов ввода-вывода:
  110 format(1x,'#  Число узлов дискретизации (n)=',i4/&
&            1x,'# Левая граница промежутка  (a)=',d23.16/&
&            1x,'# Правая граница промежутка (b)=',d23.16)
 1000 format(//1x,'# N',5x,'t',12x,'x',13x,'funt(t)/2',14x,' z/4 ')
 1001 format(1x,i3,d10.2,f14.10,1x,d21.14,1x,d21.14)
end program probtask
```

```
                  !                              Файл probtask.hdr
implicit none       ! Отмена правила умолчания ФОРТРАНа.
interface
function funt(x) result(w); real(8) x, w; end function funt
subroutine     subfun(x,w); real(8) x, w; end subroutine subfun
end interface
integer ninp / 5 /  ! Номер файла с вводимыми параметрами задачи.
integer nres / 6 /  ! --"--"--"-- с выводимым результатом.
integer n           ! Количество узлов дискретизации промежутка.
integer i           ! Номер текущего узла.
real(8) a, b        ! Левая и правая абсциссы его границ.
real(8) h           ! Шаг его равномерной дискретизации.
real(8) t, x        ! Текущие аргументы
real(8) y           ! Хранитель значения, получаемого функцией.
real(8) z           ! -"--"--"--"--"--"--"--"- подпрограммой.


function funt(t) result(w) ! Функция funt(t) вычисляет значение
implicit none    ! функции 2*t, совпадающее со значением функции
real(8) w, t     ! 2*x-3, если в последней сделать замену
w=2d0*t          ! t=x-1.5, искдючая вычитание почти равных чисел.
end function funt


subroutine subfunt(t,res) ! Подпрограмма subfunt(t,res) вычисляет
implicit none             ! значение того же выражения, что и
real(8) t, res            ! и функция funt(t) (и аналогично funt).
res=2d0*t
end subroutine subfunt


        5<--= (n) Число узлов дискретизации.    :   Файл probtask.par
-1.0000-08<--= (a) Левая абсцисса промежутка t   :   -----------------
 0.0000+00<--= (b) Правая граница промежутка t   :...................
```

и, используя **make**-файл:

```
COMP=gfortran
 OPT=-c -O0
     probtask  : probtask.o subfunt.o funt.o  # Цель subfunt.o зависит
  [  TAB  ]      $(COMP)        $^ -o $@      # лишь от одного  файла,
   probtask.o  : probtask.f90 probtask.hdr    # который не содержит
  [  TAB  ]      $(COMP) $(OPT) $<           # инструкцию INCLUDE.
     subfunt.o : subfunt.f90                  # Поэтому аргументом
  [  TAB  ]      $(COMP) $(OPT) $^           # команды компиляции
        funt.o : funt.f90                     # можно использовать
  [  TAB  ]      $(COMP) $(OPT) $<           # любую из автоматических
        result : probtask probtask.par       # переменных: $^  и   $< .
  [  TAB  ]      ./probtask                  # (аналогично и цель funt.o)
```

получим окончательный результат:

```
#  Число узлов дискретизации (n)=   5
# Левая граница промежутка  (a)=-0.1000000000000000D-07
# Правая граница промежутка (b)= 0.0000000000000000D+00


# N     t            x             funt(t)/2              z/4
  1 -0.10D-07  1.4999999900 -0.10000000000000D-07 -0.50000000000000D-08
  2 -0.75D-08  1.4999999925 -0.75000000000000D-08 -0.37500000000000D-08
  3 -0.50D-08  1.4999999950 -0.50000000000000D-08 -0.25000000000000D-08
  4 -0.25D-08  1.4999999975 -0.25000000000000D-08 -0.12500000000000D-08
  5  0.00D+00  1.5000000000  0.00000000000000D+00  0.00000000000000D+00
```

11.1.4 Уяснение выгоды простых make-переменных

Несмотря на упрощение **make**-файла при использовании переменных **$^**, **$<** и **$@** имена исходных и объектных модулей рассредоточены по всему тексту **make**-файла, что неудобно при изменении или добавлении имён. Удобно, чтобы подобная модификация не касалась содержательной части **make**-файла, а была сосредоточена в каком-то одном его фрагменте.

Желаемое достигается помещением имени (или части имени, если нужно) каждого файла с исходными модулями в соответствующую простую **make**-переменную. Тогда целевая часть **make**-файла формально выразится через имена простых **make**-переменных. Тем самым все изменения коснутся правых частей лишь нескольких операторов присваивания, сгруппированных в начальном фрагменте **make**-файла. Например, наш предыдущий проект собирался из четырёх файлов:

probtask.for, **probtask.hdr**, **funt.for** и **subfunt.for**

Поместим:

1. Слово **probtask** — в **make**-переменную, например, с именем **MAIN**, что напомнит о статусе главной программы и имени главной цели.
2. Сочетание **.hdr** — в **make**-переменную **INCL**.
3. Имена процедур **funt** и **subfunt** — в переменные **F1** и **F2**.

В **make**-переменных **MAIN**, **F1**, **F2** выгодно хранить имена файлов без расширений (для генерации нужных имён расширения можно добавлять в виде суффикса: например, **$(MAIN)**, **$(MAIN).f90**, **$(MAIN).hdr**,

$(MAIN).o, **$(MAIN).par**, и т.д.). Таким образом предыдущий **make**-файл преобразуется к виду:

```
COMP:=gfortran
 OPT:=-c -Wall -O0
MAIN:=probtask
INCL:=.hdr
  F1:=funt
  F2:=subfunt
   $(MAIN)     : $(MAIN).o $(f1).o $(f2).o
  [  TAB  ]      $(COMP) $^ -o $@
   $(MAIN).o   : $(MAIN).for $(MAIN)$(INCL)
  [  TAB  ]      $(COMP) $(OPT) $< -o $@
     $(f1).o   : $(f1).f90
  [  TAB  ]      $(COMP) $(OPT) $^ -o $@
     $(f2).o   : $(f2).f90
  [  TAB  ]      $(COMP) $(OPT) $< -o $@
RESULT          : $(MAIN) $(MAIN).par
  [  TAB  ]      ./$<
```

Полученный **make**-файл практически пригоден для сборки любого проекта, компонуемого из трёх объектных модулей и одной **include**-вставки. Для перестройки данного **make**-файла на задачу, описываемую исходниками с другими именами (например, prog.for, prog.hd, prog.par, s1.for, s2.for) достаточно соответственно изменить в нём правые части операторов присваивания в 3-й, 4-й, 5-й и 6-й строках (**main:=prog**, **incl:=.hd**, **f1:=s1**, **f2:=s2**).

При наличии большого количества объектных файлов целесообразно завести простую **make**-переменную, например, с именем **OBJ**, которая будет содержать весь их список. С нею запись правила получения исполнимого кода окажется гораздо более компактной и наглядной:

```
 OBJ=$(MAIN).o $(f1).o\# Значок "\" (обратный слэш) --- признак
               $(f2).o#  продолжения текущего оператора в следующей строке.
   $(MAIN)   : $(OBJ)
  [  TAB  ]    $(COMP) $^ -o $@
```

«Комбинирование» правил с одинаковой целью. В предыдущем **make**-файле запись зависимостей правила получения объектного кода главной программы можно оформить чуть иначе. Например, так:

```
   $(MAIN).o : $(MAIN).hdr        # т.е. ОДНУ ЦЕЛЬ можно указать
   $(MAIN).o : $(MAIN).f90        # несколько раз, определяя в качестве
  [  TAB  ]    $(COMP) $(OPT) $<  # нужных те из зависимостей, которые
                                  # по каким-то причинам неудобны при
```

```
или так                              # однократном описании цели: утилита
                                     # make допускает"комбинирование" правил
   $(MAIN).o : $(MAIN).f90           # с одинаковой целью. Оба приведенные
  [  TAB  ]    $(COMP) $(OPT) $<    # варианта эквивалентны строке:
   $(MAIN).o : $(MAIN).hdr           #  $(MAIN).o : $(MAIN).f90 $(MAIN).hdr
                                     #    [ TAB ]  : $(COMP) $(OPT) $<
```

Конечно, здесь подобное комбинирование кажется на первый взгляд вычурным. Оно выгодно при **автоматической генерации имён целей**, когда некоторые из них нужно уточнить дополнительными зависимостями (пока этой возможностью утилиты **make** мы ещё не пользовались, зато теперь узнали про неё).

Легко понять, что, если из сорока целей только две требуют **include**-подключения, то странно указывать необходимость такого подключения в списке зависимостей остальных тридцати восьми.

Вот тут-то нас и выручает *«комбинирование»*: автоматическая генерация всех объектных целей использует лишь те зависимости, которые можно обозначить общим для них способом, а упомянутые выше две цели, зависящие от **include**-подключений, уточняем двумя, соответствующими только этим целям, дополнительными зависимостями, аналогично приведенным выше фрагментам **make**-файла.

11.2 Автоматическая генерация списка объектных файлов

Напомним, что

1) автоматические переменные позволили **указать неявно** имена объектных и исходных файлов;

2) простые переменные позволили практически исключить из заголовков правил оригинальные имена исходных модулей;

3) однако для получения каждого объектного модуля в **make**-файле имеется отдельное правило, так что чем больше исходных файлов, тем больше объём и самого **make**-файла, что вряд ли удобно.

Ранее отмечалось, что утилита **make** имеет средство, позволяющее ни разу явно не указать ни одного конкретного имени объектного файла и, тем не менее, полностью скомпоновать загрузочный код, то есть выполнить главную цель. Укажем плохое и приемлемое решения.

11.2.1 Плохое решение

Очевидная замена в предыдущем **make**-файле содержимого переменной **OBJ** с **$(MAIN).o $(F1).o $(F2).o** на ***.o** не выдерживает критики. Действительно, пусть в текущей директории имеются исходные файлы

probtask.f90, **probtask.hdr**, **funt.f90**, **subfunt.f90**

и объектные **probtask.o**, **funt.o**, **subfunt.o**. Используя их можно построить исполнимый код упомянутым выше порочным способом:

```
  COMP:=gfortran
   OPT:=-c -Wall -O0
  MAIN:=probtask
  INCL:=.hdr
    F1:=funt
    F2:=subfunt
   OBJ:=*.o# <--=       Порочный способ усовершенствования !!!
    $(MAIN)    : $(OBJ)
[  TAB  ]        $(COMP) $^ -o $@
  $(MAIN).o    : $(MAIN).hdr
  $(MAIN).o    : $(MAIN).f90
[  TAB  ]        $(COMP) $(OPT) $<
    $(F1).o    : $(F1).f90
[  TAB  ]        $(COMP) $(OPT) $^
    $(F2).o    : $(F2).f90
[  TAB  ]        $(COMP) $(OPT) $<
        RESULT : $(MAIN) $(MAIN).par
[  TAB  ]        ./$<
```

Протестируем работу этого **make**-файла:

```
$ touch *.f90                    # Изменили время модификации файлов *.f90
$ make                           # и
gfortran -c -g -O0 funt.f90      # убедились, что make-файл вроде бы
gfortran -c -g -O0 probtask.f90  # работоспособен
gfortran -c -g -O0 subfunt.f90
gfortran funt.o probtask.o subfunt.o -o probtask
$ touch *.hdr                    # Аналогичная картина и при
$ make                           # модификации файлов *.hdr
gfortran -c -g -O0 probtask.for
gfortran funt.o probtask.o subfunt.o -o probtask
$                                # Однако удаление объектных
$ rm *.o                         # файлов приводит к сообщению:
$ make
make: *** Нет правила для сборки цели ‘*.o’, требуемой для ‘probtask’.
                                                          Останов.
```

Поскольку в текущей директории не оказалось ни одного объектного файла, то утилита и информирует нас об этом, причём весьма своеобразно: полагая, что имя цели должно быть ***.o**. Помним, что шаблон «*» в имени файла означает любую строку символов (лишь бы она не начиналась с точки), в частности, и имена объектных кодов, получающиеся после компиляции соответствующих исходных. Поэтому при наличии в текущей директории объектных кодов шаблон ***.o** обеспечит их сборку. Однако при их отсутствии оболочка не находит в директории объектов для расширения шаблона, о чём и сообщает упомянутым образом.

Заметим, что данная ситуация – отсутствие всех объектников – наиболее благоприятна для нас: увидев сообщение утилиты, мы сразу начнём проверять их наличие, то есть поведём поиск причины отсутствия объектных кодов по правильному пути. Если же, например, файл **subfunt.o** оказался стертым случайно, а остальные объектные присутствуют, то запуск тестируемого **make**-файла даст

```
$ make                                          # Породить фразу
gfortran funt.o probtask.o -o probtaskн         # " undefined    | неопределено
probtask.o(.text+0x21f): In function ‘MAIN__’: # reference      | обращение
: undefined reference to ‘subfunt_’             # to ’subfunt_’" | к subfunt
collect2: ld returned 1 exit status             # могут две причины:
make: *** [probtask] Ошибка 1
```

1) либо случайно стёрт объектный файл подпрограммы **subfunt**;

2) либо имя подпрограммы, по которому к ней обращаемся, не совпадает с именем подпрограммы, которая описана в файле **subfunt.f90**.

Поиск причины в худшем случае придётся вести дважды, теряя время.

Вывод:

при компоновке исполнимого кода НЕ ИМЕНУЕМ объектные коды посредством шаблона *.o, который для командной оболочки служит лишь приказом получить список имён объектных файлов, реально имеющихся в директории, но не активирует механизм их получения.

11.2.2 Приемлемое решение

Заметим, что последнее *плохое «усовершенствование»* упрощает запись сборки исполнимого файла лишь при наличии всех нужных объектных, но никак не уменьшает ни количества правил, ни конкретных имён исходных файлов в списках зависимостей.

Нам же надо построить список объектных целей, который находится автоматически по списку известных исходных файлов. Таким образом, здесь можно поставить следующие две задачи.

1. Получить список исходных файлов, являющихся единицами компиляции (файлы с главной программой, процедурами, возможно с ФОРТРАН-модулями), которые необходимыми для решения задачи, т.е. файлы, например, с расширением **f90**. Сделать это можно, используя функцию **wildcard**, встроенную в утилиту **make**.

2. Преобразовать найденный список имён исходных файлов в соответствующий список имён объектных (т.е. заменить в каждом из найденных имён расширение **f90** на расширение **o**). Эту замену можно провести посредством функции **patsubst**, также встроенную в утилиту **make**.

Функция wildcard. По переданному ей шаблону (или шаблонам) находит список имён соответствующих файлов. В качестве шаблона используем известное файловой системе сочетание ***.f90** (или ***.f**, или ***.f95**), так что запуск **make**-файла

```
   PATTERN:=*.f90
   SOURCE :=$(wildcard $(PATTERN))
   NAME :
[  TAB  ] @echo $(SOURCE)
```

выведет на экран список файлов текущей директории, соответствующий шаблону ***.f90**,

```
$ make
funt.for probtask.for subfunt.for
```

Напомним, что значок @ перед командой **echo** гасит вывод текста команды, т.е. при отсутствии значка @ получили бы

```
$ make
echo funt.for probtask.for subfunt.for
funt.for probtask.for subfunt.for
```

Если переменная **PATTERN** содержит строку ***.for *.f95 *.f90** (с тремя шаблонами), а функция **subfunt** размещена в файле **subfunt.f90**, главная программа — в файле **probtask.f95**, а функция **funt** — в файле **funt.for**, хотя подобный разнобой в расширениях вряд ли можно признать удачной альтернативой, то, тем не менее, активация **make**-файла

```
   PATTERN:=*.f *.f95 *.f90
   SOURCE :=$(wildcard $(wild))
   NAME   :
 [  TAB ]   @echo $(SOURCE)
```

получит в переменной **SOURCE** список всех файлов текущей директории с указанными шаблонами:

```
make
subfunt.f90 funt.f probtask.f95
```

Функция patsubst (аббревиатура от *pattern substitution* — замена одного шаблона в списке имён другим) в соответствии с переданными ей

1) исходным шаблоном (например, **%.f90**);
2) новым желательным вариантом шаблона (например, **%.o**);
3) символьной строкой, состоящей из слов, отделяемых друг от друга не менее чем одним пробелом (например, списком имён исходных ФОРТРАН-файлов с расширением **f90**)

заменяет в каждом слове исходный шаблон (если он подходит к слову) на желаемый выриант шаблона. Здесь символ % – особый шаблон, означающий любое количество произвольных символов. Например, если в текущей директории имеются файлы с именами **funt.f90**, **probtask.f90**, **subfunt.f90** и **probtask.hdr**, то при активации **make**-файла

```
   PATTERN:=*.f90
   SOURCE :=$(wildcard $(PATTERN))
   OBJ:=$(patsubst %.f90, %.o, $(SOURCE))
   NAMES  :
  [  TAB  ]   @echo $(SOURCE)
   NAMEO :
  [  TAB  ]   @echo $(OBJ)
```

переменные **SOURCE** и **OBJ** получат следующие значения:

```
$ make NAMES
funt.f90 probtask.f90 subfunt.f90
$
$ make NAMEO
funt.o probtask.o subfunt.o
```

Таким образом, посредством функции **wildcard** можно составить список всех исходных файлов с единицами компиляции, хранящимися в текущей директории, а посредством функции **patsubst** получить соответствующий им список имён всех необходимых объектных файлов. Так что, если переменная **MAIN** нацелена на хранение имени исполнимого файла, а переменная **OBJ** – хранит список всех нужных объектных, то фрагмент получения исполнимого кода попрежнему запишется двумя строками

```
$(MAIN) : $(OBJ)
$(COMP) $^ -o $@
```

Другими словами, функции **wildcard** и **patsubst** избавляют программиста от необходимости вручную явно набирать конкретные имена файлов, входящих в проект, что несомненно очень удобно.
Единственное, что пока вызывает нарекание – разбухание **make**-файла при увеличении количества объектных целей, ведь на каждую цель требуется в простейшем случае две строки: строка зависимостей и строка команды, посредством которой цель достигается. Упомянутый недостаток просто устраняется посредством использования шаблонов **%.f90** и **%.o** (см. выше вызов функции **patsubst**) в так называемых так называемых **шаблонных правилах**.

Шаблонные правила. IMPLICIT rules или PATTERN rules — средство, позволяющее автоматизировать процесс получения всех объектных правил, упростить **make**-файл и сделать его более универсальным [11, 12, 13]. Оказывается, нет нужды выписывать каждое объектное правило. Достаточно привлечь уже знакомые нам шаблоны **%.f90** и **%.o** для записи одного объектного правила, которое, способно (в целом) заменить все требующиеся.

Укажем в строке зависимостей вместо конкретного имени объектной цели её шаблонное обозначение **%.o** и вместо конкретного имени исходного файла его шаблонное обозначение **%.f90**. В частности, Правило

```
   %.o : %.f90
[  TAB  ]   gfortran -c $^
```

проинформирует утилиту о том, что каждый файл с шаблоном **%.o** зависит от соответствующего файла с шаблоном **%.f90**. При этом символ процента в имени зависимости правила заменяется текстом, соответствующим символу процента в имени цели. Таким образом, соответствующие объектные и исполнимый файлы можно получить, активируя **make**-файл:

```
 COMP:=gfortran
  OPT:=-c -Wall -O0
  PATTERN:=*.f90
  SOURCE :=$(wildcard $(PATTERN))
  OBJ    :=$(patsubst %.f90, %.o, $(SOURCE))
  MAIN   :=probtask
   $(MAIN) : $(OBJ)
[  TAB  ]    $(COMP) $^ -o $@
       %.o : %.f90
[  TAB  ]    $(COMP) $(OPT) $<
 $(MAIN).o : $(MAIN).hdr
    RESULT : $(MAIN) $(MAIN).par
[  TAB  ]    ./$^
```

сколько бы в текущей директории не находилось файлов с расширением **f90**. При этом передача компилятору имен файлов, от которых зависит исполнение команд достижения цели, происходит через автоматическую переменную **$^**, не требуя явного указания имен файлов.

11.2.3 Тестирование приемлемого make-файла

1. Пусть в текущей директории нет объектных файлов – только нужные исходные:

```
$ rm *.o
$ make
gfortran -c -Wall -O0 funt.f90        # перекомпилированы все исходники и
gfortran -c -Wall -O0 probtask.f90    # собран исполнимый код
gfortran -c -Wall -O0 subfunt.f90
gfortran funt.o probtask.o subfunt.o -o probtask
```

2. Предположим, что случайно стерли **funt.o**:

```
$ rm funt.o
$ make                           # перекомпилирован только funt.for
gfortran -c -Wall -O0 funt.f90  #    для получения нового funt.o
gfortran funt.o probtask.o subfunt.o -o probtask
```

3. Пусть внесено изменение в файл, подключаемый к главной программе посредством инструкции **include**:

```
$ touch *.hdr                       # главная программа перекомпилирована
$ make                              # отработали  комбинирование правил и
gfortran -c -wall -O0 probtask.f90 #                        переменная $<
gfortran funt.o probtask.o subfunt.o -o probtask
```

4. Если думаем, что что-то изменили, а на самом деле нет, то

```
$ make                                  # make напомнит об этом,
make: ‘probtask’ не требует обновления. # не делая лишней работы.
```

5. Допустим, что изменили содержимое файла **probtask.par** (хотим пропустить программу с иными исходными данными).,

```
$ touch *.par     # то make RESULT выполнит это, так как
$ make RESULT     # цель RESULT зависит от файла с исходными данными,
 ./probtask       # о чём "говорит" автоматическая переменная $^.
```

6. Если программа с имеющимися исходными данными уже была пропущена, но мы в этом сомневаемся и (на всякий случай) инициируем повторное выполнение программы:

```
$ make RESULT                          # то make не позволит зря
make: ‘RESULT’ не требует обновления. # транжирить время.
```

7. Если же изменили, например, **subfunt.f90** и, не пытаясь сначала достичь главную цель – получить исполнимый код, сразу потребовали провести расчёт:

```
$ touch subfunt.f90                    # make получит новые версии
$ make result                          # subfunt.o, исполнимого кода
gfortran -c -Wall -O0 subfunt.f90      #            и результата
gfortran funt.o probtask.o subfunt.o -o probtask
./probtask
```

Таким образом, последняя предложенная версия **make**-файла полностью решает возложенную на неё задачу:

1) не допускает необоснованной перекомпиляции исходных файлов, которые уже отлажены, и при сборке исполнимого кода использует соответствующие им и хранимые в текущей директории объектники;

2) обеспечивает за счёт *«комбинирования правил с одинаковой целью»* перекомпиляцию файла, содержащего **include**-вставку, если последняя была модифицирована;

3) позволяет практически неограниченно увеличивать количество исходных файлов, если возникнет такая необходимость, не изменяя по существу дела размера **make**-файла, за счет применения шаблонов ***.for**, **%.f90**, **%.o**, встроенных **make**-функций **wildcard** и **patsubst**, а также *шаблонного правила* с зависимостью **%.o : %.f90**.

11.3 Make-файл для программы, использующей модуль

В современном ФОРТРАНе наряду с внешними процедурами (функциями и подпрограммами) используется ещё один вид программной единицы, называемой **модулем**. Смысловая нагрузка последнего термина отлична от смысловой нагрузки термина **модуль**, который ранее использовался в словосочетаниях: *исходный модуль*, *объектный модуль* или *загрузочный модуль*.

В новой трактовке термин **модуль** означает программную единицу, обеспечивающую при её подключении к какой-нибудь процедуре доступ этой процедуры к описанным в **модуле** объектам (типам, переменным и процедурам). **Модуль** не вызывают (как, например, процедуру), но подключают, получая возможность пользоваться объектами, импортируемыми из **module**.

Программная единица типа **модуль** часто содержит, например, описания процедур, нацеленных на решение определённого класса задач. Так модуль численного интегрирования может содержать процедуры, реализующие различные схемы расчёта интеграла (формулы трапеций, прямоугольников, Симпсона, Гаусса и т.д.).

Возникает вопрос: как следует отражать в **make**-файле правила, касающиеся компиляции и подключения модуля?

В качестве примера используем соответствующую нашей задаче модификацию ранее рассмотренной программы. Именно: поместим функцию **funt** и подпрограмму **subfunt** не в отдельных файлах, а в модуле, например, с именем **myunit**

```
module myunit; implicit none
contains
function funt(t) result(w)
real(8) w, t
w=2d0*t
end function funt
subroutine subfunt(t,res)
real(8) t, res
res=2d0*t
end subroutine subfunt
end module myunit
```

Обратите внимание, что нет нужды в каждой из процедур, помещаемой в модуль, отменять действие правила умолчания — достаточно это сделать один раз в разделе описаний самого модуля.

Модуль — программная единица, т.е. от компилятора можно требовать получить соответствующий модулю объектный файл. Например,

```
$ gfortran myunit.f90 -c          # откомпилировали модуль
$ ls myunit.*                     # хотим посмотреть на имя результата
myunit.f90  myunit.mod  myunit.o
```

После компиляции главной программы или внешней процедуры в качестве результата получаем только соответствующий объектный файл. После компиляции модуля **myunit.f90** (помимо его объектного файла **myunit.o**) получается и второй файл с именем **myunit.mod**.
myunit.o потребуется утилите **make** при сборке исполнимого файла.
myunit.mod потребуется компилятору при компиляции тех программных единиц, которые этот модуль используют.

В нашем случае модуль использует главная программа:

```
program probtask      ! Модифицированная программа.  Файл probtask.f90
use myunit            ! Оператор подключения модуля.
include 'probtask.hdr'
open(unit=ninp,file='probtask.par')
open(unit=nres,file='result',status='replace')
read(ninp,100) n,a,b                ! Ввод числа узлов и границ отрезка
write(nres,110) n, a, b             ! Контрольная печать введенного.
h=(b-a)/(n-1)                       ! Расчет шага дробления [a,b].
write(nres,1000)                    ! Печать заголовка первой таблицы.
do i=1, n                           ! Для каждого из узлов:
  t=a+(i-1)*h; x=1.5d0+t; y=funt(t)/2! расчёт текущего результата
  call subfunt(t,z)
  write(nres,1001) i, t, x, y, z/4  ! и его вывод.
enddo
close(nres)
  100 format(i10/d10.3/d10.3)       ! Список форматов ввода-вывода:
  110 format(1x,'#  Число узлов дискретизации (n)=',i4/&
&            1x,'# Левая граница промежутка  (a)=',d23.16/&
&            1x,'# Правая граница промежутка (b)=',d23.16)
 1000 format(//1x,'# N',5x,'t',12x,'x',13x,'funt(t)/2',14x,' z/4 ')
 1001 format(1x,i3,d10.2,f14.10,1x,d21.14,1x,d21.14)
end program probtask
```

Если функции описаны в модуле (т.е. являются модульными, а не внешними), то их интерфейс описывать через оператор **interface** не нужно, поскольку действует модульный интерфейс. Поэтому при наличии оператора **use myunit** в главной программе явное описание интерфейса процедур, включённых в модуль, необходимо убрать из файла **probtask.hdr**:

```
                    !                           Файл probtask.hdr
implicit none       ! Отмена правила умолчания ФОРТРАНа.
integer ninp / 5 /  ! Номер файла с вводимыми параметрами задачи.
integer nres / 6 /  ! --"--"--"-- с выводимымым результатом.
integer n           ! Количество узлов дискретизации промежутка.
integer i           ! Номер текущего узла.
real(8) a, b        ! Левая и правая абсциссы его границ.
real(8) h           ! Шаг его равномерной дискретизации.
real(8) t, x        ! Текущие аргументы
real(8) y           ! Хранитель значения, получаемого функцией.
real(8) z           ! -"--"--"--"--"--"--"--"- подпрограммой.
```

11.3.1 Кустарный «ручной» пропуск задачи с модулем.

```
$ gfortran myunit.f90 -c                    # откомпилировали модуль
$ ls myunit.*
myunit.f90  myunit.mod  myunit.o
$ gfortran probtask.f90 -c                  # --"-- главную программу
$ gfortran probtask.o myunit.o -o probtask # получили исполнимый файл
$ ./probtask                                # активировали его
```

Понятно, что перед компиляцией главной программы файл **myunit.mod** уже должен иметься в наличии и быть доступным — ведь именно из него компилируемая программа узнаёт имена, импортируемые ею из модуля.

Если, например, удалим из текущей директории файл **myunit.mod** и попытаемся получить исполнимый файл **probtask**, то получим:

```
rm myunit.mod
gfortran probtask.f90 -o probtask
use myunit
    1
    Фатальная ошибка:  Can't open module file 'myunit.mod'
    for reading at (1): Нет такого файла или каталога
```

11.3.2 Попытка использования make-файла из 11.2.2

При **наличии** в текущей директории файлов **myunit.mod** и **myunit.o** работоспособен **make**-файл из пункта **11.2.2**. Например,

1. ```
   $ gfortran -c myunit.f90                 # откомпилировали файл с модулем
   $ ls myunit.*                            # убедились в появлении
    myunit.for  myunit.mod  myunit.o        #          myunit.mod и myunit.o
   $ make                                   # убедились в работоспособности
   gfortran probtask.o myunit.o -o probtask # make-файла
   ```
   ```

2. ```
   $ touch probtask.f90                # изменили время модификации главной
   $ make                              # программы и убеждаемся, что
   gfortran -c -Wall -O0 probtask.f90  #      make-файл из 11.2.2 работает!
   gfortran probtask.o myunit.o -o probtask
   ```

3. ```
   $ touch probtask.hdr                # изменили время модификации
   $ make                              # include-файла и убеждаемся, что
   gfortran -c -Wall -O0 probtask.f90  #      make-файл из 11.2.2 работает!
   gfortran probtask.o myunit.o -o probtask
   ```

4. ```
   $ touch myunit.f90                  # изменили время модификации модуля
   $ make                              # myunit.f90 и снова убеждаемся, что
   gfortran -c -Wall -O0 myunit.f90    #      make-файл из 11.2.2 работает!
   gfortran probtask.o myunit.o -o probtask
   ```

5. Если, в модуле "*закомментировать*", например, описание функции **funt**, то получим:

   ```
   $ make                              # и опять работа make-файла не вызывает
    gfortran -c -Wall -O0 myunit.f90   #  нареканий.      Ситуация выявлена им
    gfortran probtask.o myunit.o -o probtask        #               адекватно.
    probtask.o(.text+0x201): In function `MAIN__':
    probtask.for:13: undefined reference to `__myunit__funt'
    collect2: ld returned 1 exit status
    make: *** [probtask] Ошибка 1
   ```

6. После *раскомментирования* в модуле описания функции **funt** и запуске **make**-файла получаем:

   ```
   make
   gfortran -c -Wall -O0 myunit.f90
   gfortran probtask.o myunit.o -o probtask
   ```

7. Создаётся впечатление, что **make**-файл универсален. Тем не менее, можно привести примеры, когда при использовании единицы компиляции **myunit** тестируемый **make**-файл не обеспечит корректную сборку исполнимого файла. Например,

   - Имена из **make**-переменной **PATTERN**, могут расположиться в неудобном порядке, так что сначала, начнёт компилироваться, например, главная программа, использующая какой-то **module**, который ещё не откомпилирован (нет для него файла с расширением **.mod**).
   - При модификации главной программы можно случайно стереть какой-то из файлов с расширением **.mod** и не заметить этого.
   ```

11.3.3 1-ая попытка коррекции make-файла

```
#                                   Работает Makefile:
   COMP:=gfortran
    OPT:=-c -Wall -O0
PATTERN:=*.f90
 SOURCE:=$(wildcard $(PATTERN))
    OBJ:=$(patsubst %.f90,%.o,$(SOURCE))
   MAIN:=probtask
   $(MAIN) : $(OBJ)
[  TAB  ]    $(COMP) $^ -o $@
       %.o : %.f90
[  TAB  ]    $(COMP) $(OPT) $<
 $(MAIN).o : $(MAIN).hdr
     CLEAR :
[  TAB  ]    rm -f *.o *.mod $(MAIN)
    RESULT : $(MAIN) $(MAIN).par
[  TAB  ]    ./$<
```

Уничтожим в текущей директории файл **myunit.mod** и изменим программную единицу **probtask.f90**, использующую **myunit.mod** модуль:

```
rm myunit.mod                    # При отсутствии файла myunit.mod запуск
touch probtask.f90               # make-файла после модификации главной
$ make                           # программы заканчивается аварийно.
 gfortran -c -Wall -O0 probtask.f90
In file probtask.f90:2           # Автоматической перекомпиляции
                                 # модуля с восстановлением файла
      use myunit                 # myunit.mod НЕ происходит, а хотелось бы.
           1
 Fatal Error: Can't open module file 'myunit.mod' for reading at (1):
                                                 No such file or directory
  make: *** [probtask.o] Ошибка 1
```

Замечаем, что утилита в первую очередь пытается откомпилировать главную программу, хотя (по идее) сначала нужно откомпилировать модуль. Можно, конечно, изменить порядок имён исходников в переменной **SOURCE**, указав, что первой нужна компиляция файла **myunit.f90**. Однако, при этом программисту придётся брать на себя процесс сортировки имён исходных файлов в переменной **SOURCE**, что очень неудобно (особенно, если проект включает большое число модулей). Тем не менее, попробуем предложенный вариант.

Для лучшего уяснения ситуации включим в **make**-файл псевдоцель **echo**, которая при её вызове должна вывести содержимое переменных **PATTERN**, **SOURCE** и **OBJ**.

```
    COMP:=gfortran
     OPT:=-c -Wall -O0
 PATTERN:=*.f90
  SOURCE:=$(wildcard $(PATTERN)))
     OBJ:=$(patsubst %.f90,%.o, $(SOURCE))
    MAIN:=probtask
    $(MAIN)  : $(OBJ)
         $(COMP) $^ -o $@
         %.o : %.f90
        $(COMP) $(OPT) $<
  $(MAIN).o  :  $(MAIN).hdr
    RESULT : $(MAIN) $(MAIN).par
        ./$^
        cat RESULT
CLEAR :
 rm -f *.o *.mod $(MAIN)
echo:
 @echo $(PATTERN)
 @echo $(SOURCE)
 @echo $(OBJ)
```

```
$ make CLEAR                                # Повторим пропуск предыдущего
rm -f *.o *.mod probtask                    # make-файла, выведя после
make                                        # завершения его работы
gfortran -c -Wall -O0 probtask.f90          # содержимое переменных
probtask.f90:2.4:                           # PATTERN, SOURCE и OBJ.

use myunit                                  # Ошибка, естественно осталась
    1                                       # прежней.
    Фатальная ошибка:  Can't open module # Однако теперь после make echo
     file 'myunit.mod' for reading at       # видим, что имена в PATTERN и
    (1): Нет такого файла или каталога      # в SOURCE находятся в обратном
    make: *** [probtask.o] Ошибка 1         # порядке. Поэтому probtask.f90
make echo                                   # начинает компилироваться
myunit.f90 probtask.f90                     # раньше, что и приводит к
probtask.f90 myunit.f90                     # указанной ошибке.
probtask.o myunit.o
```

Поместим в переменную **SOURCE** список имён исходных файлов, получаемый **$(wildcard ...)**, но отсортированный в алфавитном порядке посредством команды

SOURCE:=$(sort $(wildcard $(PATTERN)))

т.е. теперь будет работать **make**-файл:

```
    COMP:=gfortran
     OPT:=-c -Wall -O0
 PATTERN:=*.f90
  SOURCE:=$(sort $(wildcard $(PATTERN)))
     OBJ:=$(patsubst %.f90,%.o, $(SOURCE))
    MAIN:=probtask
    $(MAIN)  : $(OBJ)
         $(COMP) $^ -o $@
         %.o : %.f90
        $(COMP) $(OPT) $<
  $(MAIN).o  :  $(MAIN).hdr
    RESULT : $(MAIN) $(MAIN).par
        ./$^
        cat RESULT
CLEAR :
 rm -f *.o *.mod $(MAIN)
echo:
 @echo $(PATTERN)
 @echo $(SOURCE)
 @echo $(OBJ)

# Результаты его пропуска таковы:
make CLEAR                                  # Теперь первым
rm -f *.o *.mod probtask                    # компилируется
make                                        # модуль, файл
gfortran -c -Wall -O0 myunit.f90            # myunit.mod
gfortran -c -Wall -O0 probtask.f90          # появляется первым.
gfortran myunit.o probtask.o -o probtask
make
make: ‘probtask’ не требует обновления.
make echo
myunit.f90 probtask.f90          # PATTERN
myunit.f90 probtask.f90          # SOURCE   !!!
myunit.o probtask.o              # OBJ
rm *.mod                                    # Однако, как
touch probtask.f90                          # только myuit.mod
make                                        # случайно  уничтожен
gfortran -c -Wall -O0 probtask.f90          # опять
probtask.f90:2.4:                           #       получаем:
use myunit
    1  Фатальная ошибка:  Can’t open module file ’myunit.mod’
       for reading at (1): Нет такого файла или каталога ...
```

Таким образом, подобная модификация **make**-файла не решит проблему в целом. Кроме того, очень неудобно вручную указывать граф зависимостей программных единиц от используемых модулей. Хотелось бы, чтобы **make** автоматически разобралась с порядком компилирования.

11.3.4 2-ая попытка коррекции make-файла

Откажемся от сортировки имён исходников (тем более, что располагать имена файлов в лексикографическом порядке вряд ли удобно). Новая коррекция заключается в указании зависимости цели **probtask.o** не только от **probtask.f90** и от **probtask.hdr**, но и от **myunit.mod**:

```
    COMP:=gfortran
     OPT:=-c -Wall -O0
 PATTERN:=*.f90
  SOURCE:=$(wildcard $(PATTERN))
     OBJ:=$(patsubst %.f90,%.o, $(SOURCE))
    MAIN:=probtask
    $(MAIN)  : $(OBJ)
         $(COMP) $^ -o $@
         %.o : %.f90
        $(COMP) $(OPT) $<
  $(MAIN).o  :  $(MAIN).hdr myunit.mod
    RESULT : $(MAIN) $(MAIN).par
        ./$^
        cat RESULT
CLEAR :
 rm -f *.o *.mod $(MAIN)
```

Правда, в этом случае, если в текущей директории не будет файла с именем **myunit.mod**, то при запуске **make**-файла получим:

```
make CLEAR
rm -f *.o *.mod probtask
make
make: *** Нет правила для сборки цели ‘myunit.mod’,
          требуемой для ‘probtask.o’.  Останов.
```

Это сообщение отлично от предыдущего. Предыдущее информировало о фатальной ошибке — об отсутствии файла **myunit.mod**. Сейчас его тоже нет. Но зато есть указание о том, что он должен быть (имя файла указано в зависимостях). Так что утилита **make** *полагает*, что программист просто забыл указать правило, по которому этот файл должен быть получен (нет имени с расширением **.mod** в качестве имени настоящей цели), и *напоминает* об этом. Поэтому добавим в строку **%.o : %.f90** слева от двоеточия шаблон **%.mod**. Тогда **make**-файл примет вид:

```
    COMP:=gfortran
     OPT:=-c -Wall -O0
 PATTERN:=*.f90
  SOURCE:=$(wildcard $(PATTERN))
     OBJ:=$(patsubst %.f90,%.o, $(SOURCE))
    MAIN:=probtask
    $(MAIN)  : $(OBJ)
[  TAB  ]      $(COMP) $^ -o $@
   %.mod %.o : %.f90                    # Обратим внимание, что цели %.mod
[  TAB  ]      $(COMP) $(OPT) $<        # %.o достигаются одной командой!
  $(MAIN).o  :  $(MAIN).hdr myunit.mod
      RESULT : $(MAIN) $(MAIN).par
[  TAB  ]      ./$^
[  TAB  ]      cat RESULT
       CLEAR :
[  TAB  ] rm -f *.o *.mod $(MAIN)

make CLEAR               # Протестируем этот Makefile:
rm -f *.o *.mod probtask # Уничтожили *.o *.mod и исполнимый файл.
make                                    # После запуска make
gfortran -c -Wall -O0 myunit.f90        # всё работает, т.е.
gfortran -c -Wall -O0 probtask.f90      # пока модификация ничего
gfortran probtask.o myunit.o -o probtask # не испортила.
make                                    # При повторном запуске make
make: 'probtask' не требует обновления.  # реакция утилиты адекватная.
touch myunit.f90                        # Моделируем модификацию модуля!!!
make                                    # Пока
gfortran -c -Wall -O0 myunit.f90        #      всё хорошо!
gfortran probtask.o myunit.o -o probtask
rm *.mod                                # Случайно удалили myunit.mod
make                                    # make  продолжает работу:
gfortran -c -Wall -O0 myunit.f90        # находит правило достижения
gfortran -c -Wall -O0 probtask.f90      # цели и корректно выполняет
gfortran probtask.o myunit.o -o probtask # её.  Кажется, что решение
make: 'probtask' не требует обновления.  # найдено. Однако, есть нюанс:
touch myunit.f90                        # Изменим myunit.f90 и
make                                    # запустим make ещё раз.
gfortran -c -Wall -O0 myunit.f90        #   Исполнимый файл получен.
gfortran probtask.o myunit.o -o probtask #   Снова запустим make
make                                    # И что же видим? Вместо
gfortran -c -Wall -O0 myunit.f90        # сообщения о ненужности
gfortran probtask.o myunit.o -o probtask # обновления получаем новую
                                        # перекомпиляцию! Непорядок!!!
```

Эффект обнаружен студентами астрономического отделения, а его причина и способ устранения, излагаемые ниже, сообщены Никифоровым И.И., сотрудником кафедры небесной механики и звёздной астрономии НИАИ имени В.В.Соболева.

11.3.5 Выяснение причины сбоя 2-ой коррекции

Для уяснения причины происходящего отследим времена модификации файлов **myunit.o** и **myunit.mod**. Повторим предыдущую схему, но при этом выведем упомянутые времёна. После **make CLEAR** и **ls -l** имеем:

```
make CLEAR
rm -f *.o *.mod probtask
make
gfortran -c -Wall -O0 myunit.f90
gfortran -c -Wall -O0 probtask.f90
gfortran probtask.o myunit.o -o probtask
make
make: ‘probtask’ не требует обновления.           # Всё, как и ожидалось.
ls -l
-rw-r--r--. 1 aw aw   196 фев 20 12:16 myunit.f90 # Видно, что время
-rw-rw-r--. 1 aw aw  1239 фев 20 12:24 myunit.mod # модификации  и
-rw-rw-r--. 1 aw aw  1416 фев 20 12:24 myunit.o   # myunit.mod и myunit.o
-rwxrwxr-x. 1 aw aw 13698 фев 20 12:24 probtask   # одно и то же
```

Изменим время модификации модуля (иммитация подправки модуля), повторим вызов **make** и отследим времена:

```
touch myunit.f90
make
gfortran -c -Wall -O0 myunit.f90          #<---= Это не вызывает возражения
gfortran probtask.o myunit.o -o probtask  #      --"--"--"--"--"--"--"--"--
ls -l                                               #  Время модификации
-rw-r--r--. 1 aw aw   196 фев 20 12:25 myunit.f90
-rw-rw-r--. 1 aw aw  1239 фев 20 12:24 myunit.mod  #  <---=  НЕ ИЗМЕНИЛОСЬ!
-rw-rw-r--. 1 aw aw  1416 фев 20 12:25 myunit.o    #  <---=  ИЗМЕНИЛОСЬ.
-rwxrwxr-x. 1 aw aw 13698 фев 20 12:25 probtask
make
gfortran -c -Wall -O0 myunit.f90
gfortran probtask.o myunit.o -o probtask
ls -l
-rw-rw-r--. 1 aw aw  1239 фев 20 12:24 myunit.mod  #  <---=  НЕ ИЗМЕНИЛОСЬ!
-rw-rw-r--. 1 aw aw  1416 фев 20 12:26 myunit.o    #  <---=  ИЗМЕНИЛОСЬ.
```

Почему модуль перекомпилируется, если его модификация более НЕ ПРОВОДИЛАСЬ?

Причина: разные времена модификации **myunit.mod** и **myunit.o**.

11.3.6 3-я попытка коррекции make-файла

Дополним команду компиляции шаблонного правила

%.o %.mod : %.f90

командой **touch $@**.

```
    COMP:=gfortran
     OPT:=-c -Wall -O0
 PATTERN:=*.f90
  SOURCE:=$(wildcard $(PATTERN))
     OBJ:=$(patsubst %.f90,%.o, $(SOURCE))
    MAIN:=probtask
    $(MAIN)  : $(OBJ)
[  TAB  ]      $(COMP) $^ -o $@
   %.mod %.o : %.f90
[  TAB  ]      $(COMP) $(OPT) $<
[  TAB  ]      touch $@                  # <--=  Коррекция !!!
  $(MAIN).o  :  $(MAIN).hdr myunit.mod
      RESULT : $(MAIN) $(MAIN).par
[  TAB  ]      ./$^
[  TAB  ]      cat RESULT
       CLEAR :
[  TAB  ] rm -f *.o *.mod $(MAIN)
```

```
make CLEAR                              # Результат 3-ей коррекции:
rm -f *.o *.mod probtask
make
gfortran -c -Wall -O0 myunit.f90
touch myunit.mod
gfortran -c -Wall -O0 probtask.f90
touch probtask.o
gfortran probtask.o myunit.o -o probtask
make                                     # Теперь реакция утилиты make
make: ‘probtask’ не требует обновления.  # соответствует ожидаемой.
ls -l
-rw-r--r--. 1 aw aw   196 фев 20 12:25 myunit.f90
-rw-rw-r--. 1 aw aw  1239 фев 20 13:04 myunit.mod
-rw-rw-r--. 1 aw aw  1416 фев 20 13:04 myunit.o
touch myunit.f90
$ make
gfortran -c -Wall -O0 myunit.f90
touch myunit.mod
gfortran -c -Wall -O0 probtask.f90
touch probtask.o
gfortran probtask.o myunit.o -o probtask
make
make: ‘probtask’ не требует обновления.
```

11.4 Информация к размышлению

1. Старый ФОРТРАН не имел единицы компиляции **module**, используя лишь **program**, **function** и **subroutine**. Сборка программного продукта прекрасно обеспечивалась утилитой **make**.

2. Современный ФОРТРАН использует (причём с несомненной выгодой) и программные единицы вида **module**, предназначенные для доступа к объектам, объявленных в них, тех программных единиц, в которых есть операторы подключения соответствующих **module**. Утилита **make** заслуженно востребована и при сборке программ с единицами компиляции вида **module**.

3. При компиляции внешней процедуры вида (**function**, **subroutine** или **program**) образуется один объектный файл (с расширением **.o**).

4. При компиляции **module** образуются два файла:

   (a) первый с расширением **.mod**. Через его посредство компилятор *узнаёт* имена объектов модуля, которые может импортировать программная единица, использующая модуль.

   (b) второй с расширением **.o** обычный объектный файл, который используется при сборке исполнимого файла.

5. Модификация процедур **function** и **subroutine** может быть двоякой: модификация тела или модификация заголовка.

6. Первая никак не затрагивает тело программы, вызывающей процедуру. Поэтому от перекомпиляции процедуры никак не зависит объектный файл программы, вызывающей процедуру.

7. Вторая же (модификация заголовка) естественно затронет тело вызывающей программы через оператор вызова процедуры, так как изменится интерфейс последней. В этом случае придётся перекомпилировать и программу, вызывающую процедуру.

8. Аналогичная ситуация возникает и при использовании модулей.

   Если модульный интерфейс прежний (т.е. изменения не касались интерфейсов процедур, типов или объектов), то после перекомпиляции модуля не нужно перекомпилировать процедуры, использующие этот модуль (ведь информация, хранимая в файле с расширением **.mod**, — прежняя).

   Объектный же файл модуля нужно пересоздать, так как изменения в исходнике тела модульной процедуры проводились. В этом случае время создания файла с расширением **.mod** при очередном запуске **make**-файла может оказаться гораздо более ранним по сравнению со временем модификации соответствующего объектного.

9. Условия необходимости модификации файла с расширением **mod** вырабатываются компилятором так, чтобы по мере возможности не перекомпилировать исходники, зависимость которых от модуля всё-таки позволяет эти исходники не перекомпилировать. Соответствующий пример уже приводился выше (см. пункт 3 раздела 11.3.4).

Выводы

- При написании **make**-файлов ФОРТРАН-программ, использующих единицы компиляции вида **module**, следует учитывать, что времена модификации файлов с расширениями ***.mod** и ***.o** для одного и того же модуля могут быть различны. Иногда это приводит (при повторных запусках **make**-файла) к излишним перекомпиляциям и самого модуля, и единиц компиляции, подсоединяющих его.

- Если последний факт неважен, то команду **touch $@** к шаблонному правилу можно не добавлять.

- Однако, если хотим (**для подстраховки**), чтобы после любой модификации модуля наряду с его компиляцией всегда происходила и перекомпиляция процедур или модулей, использующих его, то команду **touch $@** целесообразно добавить, обновляя время модификации соответствующего файла с расширением **.mod**, а значит и, требуя тем самым перекомпиляцию программных единиц, подсоединяющих данный модуль через оператор **use**.

11.4.1 О чем узнали из приложения N II ?

1. Простые переменные, автоматические переменные, шаблоны утилиты **make** и ее встроенные функции позволяют добиться наиболее краткой и универсальной формы записи **make**-файла, размер которого не зависит от количества исходных и объектных модулей используемых программных единиц.

2. Утилита **make** допускает две формы оператора присваивания = и **:=**. Первая обеспечивает заполнение переменной в момент ее использования; вторая – в момент работы оператора присваивания.

3. Имя простой **make**-переменной придумываем сами. Например, список имён исходных файлов с независимыми программными единицами удобно поместить в переменную с именем **SOURCE**, а список соответствующих объектных – в переменную **OBJECT**, имя компилятора – в переменную **COMP** и т.д.

4. Автоматические переменные утилиты **make**:

   - **$∧** – содержит список зависимостей конкретного правила;
   - **$@** – содержит имя цели конкретного правила;
   - **$<** – содержит для конкретного правила первую зависимость из всех указанных для него зависимостей.

5. Шаблон – это символ, которому сопоставляется целое семейство различных имен. Например, в среде оболочки удобно пользоваться шаблоном, обозначаемым символом *. Так выражение ***.o** толкуется оболочкой как любое имя файла с расширением **o**.

6. Наряду с общеупотребительными шаблонами утилита **make** допускает использование шаблона, обозначаемого значком %, который нацелен на замену набора символов в правой части заголовка правила текстом, обозначенным значком % в левой части, например:

```
%.o : %.f90
```

7. **wildcard** и **patsubst** – встроенные функции утилиты **make**:
   а) **wildcard** – генерирует список имён файлов в соответствие с указанным ей шаблонам:

```
PATTERN:=*.f90
SOURCE :=$(wildcard $(PATTERN))
```

   б) **patsubst** – нацелена на замену слов в исходной строке, удовлетворяющих заданному шаблону, новым вариантом слова, например:

```
OBJ    :=$(patsubst %.f90, %.o, $(SOURCE))
```

   Если в переменной **$(SOURCE)** встретятся имена файлов с расширением **.f90**, то **patsubst** в качестве результата получит строку, состоящую из имён соответствующих объектных файлов согласно требуемому шаблону **%.o**.

8. При компиляции программной единицы **module** генерируются два файла: один объектный файл с расширением **.o**, другой вспомогательный с расширением **.mod**. Первый необходим для сборки исполнимого файла. Второй — для компиляции программных единиц, подключающих модуль.

9. При использовании программных единиц **module** в правилах **make**-файла описывающих цели, зависящие от подключаемых модулей, следует указывать зависимость от соответствующих файлов с расширением **.mod**

12 Приложение III. О подсчете времени.

12.1 Утилита time.

В **UNIX**-среде есть утилита **time** , которая позволяет выяснить время, затраченное на выполнение команды, указанной в качестве аргумента этой утилиты. Пусть хотим узнать:

```
$ time date                      // "Сколько времени работает команда date?"
Срд Апр 22 14:55:51 MSD 2009     // "По мнению" команды time:
real    0m0.003s   //   1) астрономическое время работы равно 0,003 секунды;
user    0m0.001s   //   2) время центрального процессора, затраченное на
                   //      выполнение пользовательской части программы;
sys     0m0.002s   //   3) время нужное для выполнения системных функций.
```

Реальное время работы команды **date** команда **time** *оценила* в **0,003** секунды. Аналогично узнается время работы любой программы. Пусть, например, есть программа

```
program tsttime1; implicit none ! Пустое тело цикла повторяется n раз
integer i, n /1000000000/       ! (n задается  на этапе компиляции).
write(*,*) ' n=',n              ! Временные затраты tsttime1 по команде
do i=1,n; enddo                 ! time при n=1000000000, 2 000000000
end                             ! таковы:
```

```
$ gfortran tsttime1.f      #  Видно, что в данном  случае общее время
$ time ./a.out             #                       равно к сумме времен
  n=  1000000000                 n=  2000000000    вызова и исполнения.
  real    0m2.488s               real    0m4.896s
  user    0m2.484s               user    0m4.892s
  sys     0m0.004s               sys     0m0.004s
```

```
program tsttime2; implicit none    ! Если n вводить с экрана,
integer i, n                       ! то результат существенно ИНОЙ:
write(*,*) 'введи число повторов'  !  n : 1000000000    2000000000
read(*,*)n; write(*,*) ' n=',n     ! real    9.257s      12.654s
do i=1,n; enddo                    ! user    2.380s       4.710s
end                                ! sys     0.003s       0.003s
```

Видно, что время **real** складывается не только из времен (*user*) и (*sys*), но еще и из времени, которое пользователь затратил на придумывание вводимого значения **n**, на его набор и на нажатие клавиши **enter**. Функция **time** удобна, когда надо выяснить время работы своей программы от запуска до завершения. При необходимости замерить время, требуемое каким-то фрагментом программы следует пользоваться соответствующими конкретному языку программирования функциями.

12.2 C-функция clock() и макрос CLOCKS_PER_SEC.

Функция **clock()** находит значение типа **long int** (или *clock_t*) равное числу временных импульсов прошедших с момента запуска программы. **CLOCKS_PER_SEC** – число временных интервалов в секунду.
Для оценки временных затрат на работу фрагмента программы запомним в переменных (например, **t1** и **t2**) значения, полученные **clocks()** до и после завершения его работы соответственно, а перевод в секунды осуществим по формуле **((double)(t2-t1))/(CLOCKS_PER_SEC).** Оценим для примера время работы трех пустых циклов из программы:

```
#include <stdio.h>
#include <time.h>  // хранит прототип clock(), тип clock_t и CLOCK_PER_SEC.
int main()
{ long int n, i, t0 = clock(); clock_t t1, t2, t3, t4, t5, t6;
  double tcl_12, tcl_23, tcl_45, tcl_06;
  printf("t0=%7ld        CLOCKS_PER_SEC= %d\n", t0, CLOCKS_PER_SEC);
  n=1000000000; t1=clock(); for (i=0;i<=n;i++); t2=clock();
                            for (i=0;i<=n;i++); t3=clock();
  n=2000000000; t4=clock(); for (i=0;i<=n;i++); t5=clock();
  tcl_12=((double)(t2-t1))/CLOCKS_PER_SEC;
  tcl_23=((double)(t3-t2))/CLOCKS_PER_SEC;
  tcl_45=((double)(t5-t4))/CLOCKS_PER_SEC;
  printf("введи параметр\n"); scanf("%d",&i);
  t6=clock(); tcl_06=((double)(t6-t0))/CLOCKS_PER_SEC;
  printf("%15s   Число импульсов к моменту   :   Время исполнения\n"," ");
  printf("%15s    запуска      завершения    :\n"," ");
  printf(" Первого  цикла : %8ld        %ld     :   %f\n", t1,t2,tcl_12);
  printf(" Второго  цикла :  %ld        %ld     :   %f\n", t2,t3,tcl_23);
  printf(" Третьего цикла :  %ld        %ld     :   %f\n", t4,t5,tcl_45);
  printf(" Всего расчета  : %8ld        %ld     :   %f\n", t0,t6,tcl_06);
  return 0; }
```

```
$ gcc tsttime3.c                          // Сравним результаты её
$ time ./a.out                            // работы с результатами
t0=      0        CLOCKS_PER_SEC= 1000000  // функции time.
введи параметр
888               Число импульсов к моменту   :   Время исполнения
                   запуска      завершения    :
 Первого  цикла :        0        1580000     :   1.580000
 Второго  цикла :  1580000        3140000     :   1.560000
 Третьего цикла :  3140000        6260000     :   3.120000
 Всего расчета  :        0        6260000     :   6.260000
real    0m12.903s   Реальное время зависит от продолжительности, например,
user    0m6.265s    раздумий над значением вводимого данного (для этого и
sys     0m0.004s    включен scanf). Ясно видно, что функция clock() отмеряет
                    не реальное время, а исключительно процессорное.
```

12.3 ФОРТРАН-подпрограмма CPU_TIME.

CPU_TIME – встроенная подпрограмма, возвращающая через свой аргумент типа **real** значение процессорного времени в секундах, прошедшее к моменту ее вызова от начала запуска программы.

```
program tsttime4; implicit none; real t0, t1, t2, t3, t4, t5, t6
integer i, n
call cpu_time(t0)
n=1000000000;                  call cpu_time(t1)
             do i=1,n; enddo; call cpu_time(t2)
             do i=1,n; enddo; call cpu_time(t3)
n=2000000000;                  call cpu_time(t4)
             do i=1,n; enddo; call cpu_time(t5)
write(*,1000)
write(*,1001) t1, t2, t2-t1
write(*,1002) t2, t3, t3-t2
write(*,1003) t4, t5, t5-t4
write(*,*) ' введи число.'
read(*,*) i; write(*,*) 'i=',i; call cpu_time(t6)
write(*,1004) t0, t6, t6-t0
 1000 format(1x,15x,'    Момент      Момент        Время '/&
&            1x,15x,'   запуска    завершения    исполнения')
 1001 format(1x,' Первого цикла  :', f7.3, 5x, f7.3, 7x, f7.3)
 1002 format(1x,' Второго  цикла :', f7.3, 5x, f7.3, 7x, f7.3)
 1003 format(1x,' Третьего цикла :', f7.3, 5x, f7.3, 7x, f7.3)
 1004 format(1x,' Всего расчета  :', f7.3, 5x, f7.3, 7x, f7.3)
end
```

```
$ gfortran tsttime4.f90
$ time ./a.out
                     Момент      Момент        Время
                    запуска    завершения    исполнения
  Первого цикла  :  0.001       2.620         2.618
  Второго  цикла :  2.620       5.193         2.574
  Третьего цикла :  5.193      10.327         5.134
  введи число.
 i=           5
  Всего расчета  :  0.001      10.327        10.326

real    0m23.974s
user    0m10.327s
sys     0m0.002s
```

В качестве синонима **CPU_TIME** иногда используется имя **SECOND**.

12.4 ФОРТРАН-подпрограмма DATA_AND_TIME.

Подпрограмма **DATE_AND_TIME([date] [,time] [,zone] [,values])** (см. [3, 4]) возвращает дату и время по встроенным системным часам. Здесь квадратные скобки – не синтаксическая конструкция ФОРТРАНа, а напоминание о **необязательности** заключенного в них аргумента. Первые три аргумента — текстовые скалярные переменные:

- **date** – в первых восьми символах дата в виде **CCYYMMDD**, где **CC** – век, **YY** – год, **MM** – месяц, **DD** – день.
- **time** – в первых десяти символах время в виде **HHMMSS.SSS**, где **HH** – часы, **MM** – минуты, **SS** – секунды, **SSS** – миллисекунды.
- **zone** – в первых пяти символах разность между местным временем и средним по Гринвичу в виде **SHHMM**, где **S** – знак (+ или -), **HH** – часы, **MM** – минуты.
- **values** – вектор целого типа (в в первых восьми элементах последовательность значений: год, месяц, день, разницу во времени по отношению ко Гринвичу, час, минуты, секунды, миллисекунды.

Подпрограмма **DATE_AND_TIME** сама определяет все свои аргументы. Применим её для решения задачи из предыдущего пункта.

```
program tsttime6; implicit none; real(8) timer, t0, t1, t2, t3, t4, t5, t6
integer i, n
t0=timer(); n=1000000000;                  t1=timer()
                          do i=1,n; enddo; t2=timer()
                          do i=1,n; enddo; t3=timer()
            n=2000000000;                  t4=timer()
                          do i=1,n; enddo; t5=timer()
write(*,1000)
write(*,1001) t1, t2, t2-t1; write(*,1002) t2, t3, t3-t2
write(*,1003) t4, t5, t5-t4
write(*,*) ' введи число '; read (*,*) i; write(*,*) ' i=',i
t6=timer(); write(*,1004) t0, t6, t6-t0
 1000 format(1x,19x,'    Момент       Момент        Время '/&
&            1x,19x,'   запуска     завершения    исполнения')
 1001 format(1x,' Первого цикла  :', 3x, f10.3, 3x, f10.3, 3x, f10.3)
 1002 format(1x,' Второго  цикла :', 3x, f10.3, 3x, f10.3, 3x, f10.3)
 1003 format(1x,' Третьего цикла :', 3x, f10.3, 3x, f10.3, 3x, f10.3)
 1004 format(1x,' Всего расчета  :', 3x, f10.3, 3x, f10.3, 3x, f10.3)
end
```

```
function timer()        ! Именно функция timer() содержит обращение
real(8)    :: timer     ! к DATA_AND_TIME.
integer(4) :: ival(8)
call date_and_time(values=ival)
timer=dble(ival(8))*0.001_8+dble(ival(7))+dble(ival(6))*60.0_8+&
&                                         dble(ival(5))*3600.0_8
end function timer
```

Компиляция этой программы в режиме оптимизации **-O1** и последующий пропуск исполнимого файла привели к следующему результату:

```
$ gfortran -O1 tsttime6.f     // Задание ключа оптимизации -O1 при компиляции.
$ time ./a.out                //
                          Момент       Момент        Время
                         запуска     завершения    исполнения
  Первого цикла  :    61883.424    61883.818        0.394
  Второго  цикла :    61883.818    61884.176        0.358
  Третьего цикла :    61884.176    61884.891        0.715
  введи число
  i=           5
  Всего расчета  :    61883.424    61910.404       26.980

real    0m26.983s
user    0m1.468s
sys     0m0.002s
```

Замечания:

1. До сих пор мы в ФОРТРАНе не встречали у подпрограмм аргументов, которые необязательно указывать при обращении и не знаем, как следует описывать формальные аргументы своих собственных подпрограмм, чтобы снабдить их аналогичным свойством (служебное слово **OPTIONAL**).

2. Кроме того, на протяжении данного курса мы при указании фактического аргумента никогда ещё не помещали **слева** от него через значок присваивания имя соответствующего формального аргумента (см. в функции **time()** вызов

   call date_and_time(values=ival)).

3. На примере вызова подпрограммы **DATE_AND_TIME**, узнали, что современный ФОРТРАН предоставляет упомянутые в замечаниях возможности. В дальнейшем познакомимся с ними на практике. Указанные возможности бывают иногда удобны и востребованы.

4. Возможность иметь необязательные аргументы полезна тогда, когда соответствующие формальные аргументы имеют значения, устанавливаемые по умолчанию, но при этом важно, чтобы они всё-таки были аргументами, а не локальными объектами процедуры, так как иногда при вызове процедуры нужны значения отличные от значений режима умолчания. В то же время, как правило, процедура работает в режиме умолчания этих аргументов, когда видеть их в списке аргументов вызова неудобно.

5. Возможность записи аргумента процедуры через значок присваивания (так называемая **ключевая** форма задания аргумента) тоже имеет некоторые преимущества по сравнению с привычной **позиционной** формой записи аргументов (т.е. когда позиция аргумента однозначно определяет его смысловую нагрузку). Например, **ключевая** форма (в отличие от позиционной) позволяет при вызове процедуры располагать аргументы в любом порядке. Цена — несколько более длинная запись списка аргументов.

6. При ключевой форме задания слева от знака присваивания пишется имя формального аргумента, а справа — имя фактического, что, собственно, и позволяет по имени формального аргумента распознать смысловую нагрузку фактического, не требуя соблюдения порядка записи аргументов (подробности см., например, в [3] пункт 16.4.4 **“Ключевые и необязательные параметры”**).

12.5 ФОРТРАН-подпрограмма SYSTEM_CLOCK

Подпрограмма **system_clock([count] [,count_rate] [, count_max])** возвращает через аргументы характеристики системного таймера:

1. **count** – текущее значение таймера увеличивается на единицу при каждом отсчете пока не достигнет значения **count_max** после чего начинается новый цикл отсчета времени (см., [3]).
2. **count_rate**: число отсчетов таймера в секунду.
3. **count_max** – максимальное значение таймера.

```
 program tsttime7; implicit none; integer  i, n, m, tr, tm
 integer t0, t1, t2, t3, t4, t5, t6
 call system_clock(count=t0, count_rate=tr, count_max=tm)
 write(*,*) 'count     =t0=', t0; write(*,*) 'count_rate=tr=', tr
 write(*,*) 'count_max =tm=', tm;  n=1000000000; m=2000000000
 call system_clock(count=t1); do i=1,n; enddo; call system_clock(count=t2)
                              do i=1,n; enddo; call system_clock(count=t3)
 call system_clock(count=t4); do i=1,m; enddo; call system_clock(count=t5)
 write(*,*) ' введи число '; read(*,*) i; call system_clock(t6)
 write(*,1000)
 write(*,1001) t1, t2, dble(t2-t1)/tr; write(*,1002) t2, t3, dble(t3-t2)/tr
 write(*,1003) t4, t5, dble(t5-t4)/tr; write(*,1004) t0, t6, dble(t6-t0)/tr
 1000 format(1x,15x,'    Момент      Момент        Время '/&
&             1x,15x,'   запуска    завершения    исполнения')
 1001 format(1x,' Первого цикла  :', i7, 5x, i7, 7x, f7.3)
 1002 format(1x,' Второго  цикла :', i7, 5x, i7, 7x, f7.3)
 1003 format(1x,' Третьего цикла :', i7, 5x, i7, 7x, f7.3)
 1004 format(1x,' Всего расчета  :', i7, 5x, i7, 7x, f7.3)
end program tsttime7
```

```
$ gfortran tsttime7.f95     ! Наглядно видно, ФОРТРАН-подпрограмма SYSTEM-CLOCK
$ time ./a.out              ! отмеряет отсчеты таймера реального времени:
 count     =t0=           0 !                 46850/1000 = 46.85сек (real),
 count_rate=tr=        1000 ! тогда как C-функция clock() измеряет отсчеты
 count_max =tm=  2147483647 ! времени работы процессора
  введи число               !.........................
777                                                    !
                    Момент      Момент        Время    !
                   запуска    завершения    исполнения !
  Первого цикла  :      0        5300         5.300    !
  Второго  цикла :   5300        9732         5.270    ! real    0m46.854s
  Третьего цикла :   9732       20352        10.620    ! user    0m20.326s
  Всего расчета  :      0       46850        46.850    ! sys     0m0.007s
```

12.6 GFORTRAN-подпрограмма ETIME.

Помимо встроенных процедур оценки временных затрат, упомянутых выше, компилятор **gfortran** имеет еще подпрограмму **etime(artime, result)**. Подпрограмма удобна для замера процессорного времени исполнения исследуемых фрагментов программ. Подпрограмма имеет два аргумента типа **real**, причем по структуре: первый – вектор из двух элементов, а второй – скалярная переменная, значение которой равно сумме значение элементов вектора. В первый элемент вектора подпрограмма помещает процессорное время, израсходованное командами пользователя с момента запуска исполнимого кода (т.е. то самое время, которому утилита **time** сопоставляет имя **user**). Во второй элемент вектора помещается время, затраченное системой. Второй аргумент хранит сумму времен **user**+**sys**.

```
program tsttime8; implicit none
integer(8) :: i, n
real, dimension(2) :: artime; real :: res0, res1, dt; real(8) s, di, z
call etime(artime, res0)
write(*,'(a,f7.3,a,f7.3,a,f7.3)') '   artime(1)=', artime(1),&
                                  '   artime(2)=', artime(2),&
                                  '        res0=', res0
n=1000000000; z=dble(n)
s=0; do i=1,n; di=dble(i); s=s+di*di; enddo  !  Имитация долгого счета.
call etime(artime, res1);  dt=res1-res0
write(*,'(a,f7.3,a,f7.3,a,f7.3)') '   artime(1)=', artime(1),&
                                  '   artime(2)=', artime(2),&
                                  '        res1=', res1
write(*,'(42x,a,f7.3)') '        dt=', dt
write(*,*) ' s=',s
write(*,*) ' s=',z*(z+1)*(2*z+1)/6
end program tsttime8

$ gfortran tsttime8.f95 -O0
$ time ./a.out
   artime(1)=  0.000   artime(2)=  0.002        res0=  0.002
   artime(1)=  6.488   artime(2)=  0.003        res1=  6.491
                                                  dt=  6.488
  s=   3.3333333383335519E+026
  s=   3.3333333383333334E+026
real   0m6.492s
user   0m6.488s
sys    0m0.003s
```

12.7 Чуть-чуть о профилировании.

Указанные выше способы замера времени обладают рядом существенных недостатков, хотя на первый взгляд, достаточно удобны при исследовании небольших программ.

1. Они требуют вкрапления в текст программы операторов вызова временны́х процедур и замеров, что неудобно, так как после уяснения всех вопросов потребуется удаление этих вызовов, не говоря уже об исправлении опечаток и поиском ошибок, допускаемых при этом.

2. В случае сложных проектов при наличии большого числа вложенных вызовов упомянутые встроенные временны́е функции вряд ли позволят оперативно обнаружить причину происходящего. Для решения подобных проблем существуют специальные утилиты, называемые **профилировщиками** (иногда *профайлерами*).

3. **Профилировщик** позволяет без внесения каких-либо изменений в исходные тексты проекта получить временны́е затраты различных частей программы и сообщить о них программисту, не смешивая временну́ю статистику с результатами работы программы.

Рассмотрим чуть изменённый и переписанный на ФОРТРАНе исходный текст СИ-программы из

https://www.ibm.com/developerworks/ru/library/l-gnuprof).

```
function a() result(g)     ! Функция a() моделирует долгоработающий
implicit none              !                        алгоритм
integer i, g
i=0; g=0
do while (i<100000)
  g=g+i; i=i+1
enddo
end function a
function b() result(g)     ! Функция b() моделирует алгоритм, работающий
implicit none              ! в четыре раза дольше алгоритма a().
integer i, g
i=0; g=0
do while (i<400000)
  g=g+i; i=i+1
enddo
end function b
```

```
program main; implicit none                 ! Главная программа
interface
integer function a() result(g); end function a
integer function b() result(g); end function b
end interface
integer i, N, p, q
write(*,*) 'введи N (число итераций)'       ! вводит N - количество
read(*,*) N                                 ! вызовов процедур
write(*,*) 'N=', N                          !     a() и (b)
if (N<1) then; write(*,*) 'Нет итераций!'   ! и
               stop 1                       !  в случае N>=1
  else                                      !  осуществляет их
      i=N;                                  !  вызов в теле
      do while (i/=0)                       !  цикла с предусловием.
        p=a();                              !
        q=b();                              ! В тексте программы НЕТ
        i=i-1                               ! вызова встроенных процедур,
      enddo                                 ! нацеленных на замеры длин
endif
write(*,*) 'p=',p                                       ! промежутков времени.
write(*,*) 'q=',q                                       ! промежутков времени.
stop 0
end program main
```

Для вызова **gnu**-профилировщика **gprof** необходимо:

1) провести компиляцию программы при включённой опции **-pg** компилятора **gfortran**: **gfortran -pg example1.f90**

2) осуществить активацию исполнимого файла:

```
./a.out                       # ./a.out, получая в качестве
 введи N (число итераций)     # результата (наряду с выводом,
 10000                        # запланированным программой,
  N=       10000              # и дополнительный файл   gmon.out
  p=   704982704              # в чём можно убедиться посредством
  q= -1604578624              # команды ls.
  STOP 0                      # В gmon.out внесена вся информация
ls                            # о временн'ых затратах, которую
a.out  example1.f90  gmon.out # можно извлечь, запустив  gprof.
```

3) осуществить запуск профилировщика:

gprof ./a.out gmon.out -p > flat.res.

В последней команде

- **./a.out** – исполнимый файл;
- **gmon.out** – соответствующий информационный файл для **gprof**
- **-p** – опция **gprof**, информирующая утилиту о том, что из всей информации мы хотим видеть лишь *простой профиль*.
- **flat.res** – имя файла, в который будет помещён *простой профиль*

flat.res содержит таблицу с некоторой статистикой временны́х замеров:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.96     12.12    12.12    10000     0.00     0.00  b_
 19.62     15.06     2.94    10000     0.00     0.00  a_
  0.00     15.06     0.00        1     0.00    15.06  MAIN__
```

В частности, можно видеть, что процедура **b()**, действительно, работает вчетверо медленнее нежели процедура **a** ($12.12/2.94 \sim 4$), так что 80% всего времени работы программы занимает процедура **b()** и только 20% процедура **a()**.

Смысловая нагрузка столбцов таблицы простого профиля.

% **time**	(процент времени) – Отношение временны́х затрат функции, имя которой находится в самом правом столбце простого профиля, ко всему времени работы исполнимого кода (в процентах);
cumulative secons	(накапливающиеся секунды) – общее время (в секундах), затраченное компьютером на работу данной функции, с добавлением времени, которое затрачено функциями, имена которых (в самом правом столбце) расположены выше имени данной;
self seconds	(собственные секунды) – время (в секундах), затраченных только данной функцией. Именно по нему и упорядочиваются (в порядке убывания времени) строки таблицы простого профиля

calls	— общее количество вызовов данной функции, т.е. сколько раз данная функция была вызвана. Если ни разу, или если невозможно определить (например, если функция не была откомпилирована с опцией **-pg**), то поле **calls** – пусто;
self ms/call	количество миллисекунд в среднем, затраченное данной функцией именно на её вызов (без учёта обращений к вызываемым ею функциям). Если функция откомпилирована без опции **-pg**, то данное поле пусто);
total ms/call	количество миллисекунд в среднем, затраченное данной функцией и её подпрограммами на вызов Если функция откомпилирована без опции **-pg**, то данное поле пусто);
name	— имя функции. Список имён функций в самом правом столбце после упорядочения по убыванию по времени **self seconds**.

Подробнее узнать об остальных возможностях утилиты **gprof** можно посредством **man gprof** или **info gprof**, или же через Интернет, в частности,например, по ссылке:

http://www.opennet.ru/docs/RUS/gprof/gprof-6.html

13 Приложение IV. Операторы ввода-вывода

Операторы ввода-вывода ФОРТРАНа имеют вид (см. [2]):

- **read(список_спецификаторов) [список_элементов_ввода]**
- **write(список_спецификаторов) [список_элементов_ввывода]**
- **read формат [,список_элементов_ввода]**
- **print формат [,список_элементов_вывода]**

Две последние формы записи операторов ввода-вывода обычно используются для выполнения операций ввода-вывода стандартными системными устройствами, под которыми понимаются устройства ввода с экрана и вывода на экран, хотя ничто не мешает осуществить переназначение этих устройств посредством **bash**-операций $<$ и $>$.

Напомним, что квадратные скобки в данном контексте не синтаксический элемент записи оператора, а лишь указание на возможность отсутствия элементов, помещённых в них. Например, вывод одного и того же текста, можно осуществить двояко:

```
write(*,*) 'Привет'  или  write(*,'("Привет")')
```

В первом случае оператор **write** имеет список элементов вывода (слово **Привет**); во втором — это слово входит в список спецификаторов. Какой из способов выбрать — решает программист.

Список_спецификаторов предназначается для управления режимом работы операторов ввода-вывода. Спецификаторы могут задаваться либо в позиционной, либо в ключевой форме. Например,

```
program rw1; implicit none; integer n, m, k
read(*,*) n          ! позиционная форма устройства ввода и формата
read(unit=*,fmt=*) m ! ключевая форма для устройства ввода и для формата
read(fmt=*,unit=*) k ! ключевая форма: не важен порядок спецификаторов
write(*,*) ' n=',n
write(unit=*,fmt=*) ' m=',m; write(fmt=*,unit=*) ' k=',k
end
```

Ключевые имена спецификаторов операторов ввода-вывода:

имя	Спецификатор
[unit=]	устройства ввода-вывода
[fmt=]	формата (только для форматных записей)
[nml=]	именованного списка
rec=	номера записи (только для файла прямого доступа)
iostat=	типа ошибки
err=	перехода по ошибке
end=	перехода по признаку окончания файла
advance=	продвижения
size=	числа введённых символов
eor=	перехода по признаку окончания записи

13.1 Спецификатор [unit=]u

u справа от знака равенства в **unit=u** при задании спецификатора в ключевой форме (или в позиционной — просто **u**) обозначает номер устройства, который можно задать:

- константой целого типа;
- выражением целого типа;
- именем внутреннего файла, в качестве которого используется переменная символьного типа. Внутренний файл полезен, например, когда возникает необходимость использовать в качестве повторителя оператора **format** имя упомянутой переменной (если, конечно, в арсенале средств оператора **format** нет соответствующей синтаксически встроенной возможности).
- символом * (*звёздочка*), который является синонимом стандартного устройства ввода-вывода, определяемого реализацией (обычно это экран). В **gfortran**е символу * по умолчанию сопоставляется номер **5** для устройства ввода и номер **6** для устройства вывода. Узнать, какие именно номера сопоставляются устройствам ввода-вывода, обозначаемым символом * можно, используя встроенный в gfortran модуль **ISO_FORTRAN_ENV**, в котором номерам этих устройств сопоставлены именованные константы:

INPUT_UNIT и OUTPUT_UNIT

```
program main
use ISO_FORTRAN_ENV                          !   Результат работы:
implicit none                                !
write(*,'(" INPUT_UNIT=",i3)')  INPUT_UNIT   !     INPUT_UNIT=  5
write(*,'("OUTPUT_UNIT=",i3)') OUTPUT_UNIT   !    OUTPUT_UNIT=  6
end program main
```

Программист, используя оператор **read(*,*)**, полагает, что данные вводятся с экрана. Однако, если в программе устройство под номером **5** явно переопределено посредством оператора **open** на файл текущей директории, то и обращение **read(*,*)** потребует для ввода наличия соответствующего файла на диске. Например,

```
program test_unit; implicit none;
integer i, j
read (*,*) i;                          write(*,*) ' i=',i
open(5,file='input' ); read (*,*) j; write(*,*) ' j=',j
end

88                      ! файл input

$ gfortran main.f95
$ ./a.out
7                       ! Семёрка введена с экрана,
  i=           7        ! а 88 из файла с именем input
  j=          88        ! несмотря на то, что оба ввода
./a.out > result        ! выполнены оператором read(*,*)
    5

  i=           5        ! файл result
  j=          88
```

Аналогично переопределяется и стандартный вывод при **write(*,*)**, если в программе есть отработал оператор **open(6,file=result)**.

13.2 Спецификатор формата [fmt=]f

f справа от знака равенства в **fmt=f** при задании спецификатора в ключевой форме (или в позиционной — просто **f**) обозначает

- либо метку оператора **format**;
- либо имя массива (или выражение символьного типа), содержимое которых должно синтаксически подходить под описание формата.
- либо символ * (*звёздочка*), который в данном случае означает, что ввод-вывод управляется списком ввода-вывода.

```
program test_fmt; implicit none; integer i; character(5) s; logical b
                                                  real r; complex c
character(26) str /'(i4,4x,a5,4x,l7,6x,3f10.3)'/
character(4)   arr(7) /'(i4,','4x,a','5,4x','l7,6','x,3f','10.3',')  '/
read (*,fmt= 100) i, s, b, r, c; write(*,fmt= 1000) i, s, b, r, c
read (*,fmt= str) i, s, b, r, c; write(*,1000) i, s, b, r, c
read (*,fmt= arr) i, s, b, r, c; write(*,1000) i, s, b, r, c
read (*,fmt=  * ) i, s, b, r, c; write(*,1000) i, s, b, r, c
 100  format(i4,4x,a5,4x,l7,6x,3f10.3)
! 1000 format(' i=',i5,/' s=',a5,/' b=',l1/' r=',e11.4/,&
! &           ' c=(',e10.3,',',e10.3,')')
 1000 format(' i=',i5,3x,'s=',a5,3x,'b=',l1,3x,'r=',e11.4,&
&            3x,'c=(',e10.3,',',e10.3,')')
end

1234    abcde    .FALSE.           3.142       5.6      -9.5   файл input
5678    fghij    .true.            2.718  56.0e-01 -0.95e1
9876    klmno    .false.           1.111  56.0e-1  -0.95e+1
5432    pqrst    .TRUE.            4.444 (0.56,-9.5)

 i= 1234   s=abcde   b=F   r= 0.3142E+01   c=( 0.560E+01,-0.950E+01)
 i= 5678   s=fghij   b=T   r= 0.2718E+01   c=( 0.560E+01,-0.950E+01)
 i= 9876   s=klmno   b=F   r= 0.1111E+01   c=( 0.560E+01,-0.950E+01)
 i= 5432   s=pqrst   b=T   r= 0.4444E+01   c=( 0.560E+00,-0.950E+01)
```

13.3 Спецификатор именованного списка [nml=]q

q справа от знака равенства в **nml=q** при задании спецификатора в ключевой форме (или в позиционной — просто **q**) обозначает имя списка объектов, заданное в операторе **namelist**.

```
program test_nml; implicit none; integer i; character(5) s; logical b
                                                  real r; complex c
namelist /my_list/ i, s, b, r, c; read (*,my_list)
                                  write(*,my_list)
                                  write(nml=my_list,unit=*)
end


&my_list
b=.false., i=5555, c=(6.7,-9.9), r=0.333333, s='uvwxy'
&end


&MY_LIST                                ! При отсутствии в операторах
 I=       5555,                         ! ввода-вывода имени NML (ключе-
 S="uvwxy",                             ! вого имени спецификатора име-
 B=F,                                   ! именованного списка) его
 R= 0.333332986    ,                    ! неключевое имя должно быть
 C=(  6.69999981    , -9.89999962    ), ! вторым в списке спецификаторов.
 /
&MY_LIST
 I=       5555,
 S="uvwxy",
 B=F,
 R= 0.333332986    ,
 C=(  6.69999981    , -9.89999962    ),
 /
```

13.4 Спецификатор номера записи rec=n

Спецификатор используется в операторах ввода-вывода файлов только **прямого доступа** и только в ключевой форме. **n** справа от знака равенства в **rec=n** — номер записи в файле **прямого доступа**.

До сих пор и в лекционных программах, и в задачах домашних заданий мы имели дело только с файлами последовательного доступа (т.е. прочесть нужную запись из такого файла можно было лишь посредством последовательного прохода всех предыдущих).

Доступ к записи файла **прямого доступа** происходит по её номеру, так что для чтения записи нет необходимости явно прочитывать все предшествующие.

Создадим текстовый файл с именем **input1.dat**, в каждой строке которого поместим текст, содержащий, например, номер строки, точку и текст из четырёх символов:

```
1.abcd                                                    ! файл input1.dat
2.efgh
3.ijkl
4.mnop
5.qrst
```

Рассмотрим программу:

```
program t_rec_read; implicit none; integer i; character(5) s
integer j
open(8,file='input1.dat',access='direct',recl=7,form='formatted')
do j=1,5
  read(8,rec=j,fmt='(i1,a5)') i, s; write(*,*) 'j=',j,' i=',i, 's=',s
enddo
end
```

В ней файл **input1.dat** посредством оператора **open** открыт как файл прямого доступа:

- Оператор

  open(8,file='input1.dat',access='direct',recl=7,form='formatted')

  подсоединяет файл с именем **input1.dat** к устройству под номером **8**. Для файлов последовательного доступа спецификатор **access** имеет значение **'sequential'** по умолчанию.

- **access='direct'** — устанавливаем режим прямого доступа к файлу.
- **recl=7** — сообщаем, что длина записи состоит из семи байт (в первый помещено однозначное целое, со второго по шестой помещены точка и четыре буквы). Когда, набрав очередную строку, нажимаем **enter**, то заносим после самой правой последней буквы **символ перевода строки** (в ASCII десятичный код **10**). Под этот однобайтовый символ тоже требуется место. Поэтому **recl=7**. Если же указать **recl=6**, то

```
$ gfortran mainr.f95
$ ./a.out
 j=           1  i=           1 s= abcd
 At line 7 of file mainr.f95 (unit = 8, file = 'input1.dat')
 Fortran runtime error: Bad value during integer read
```

  компиляция пройдёт успешно, но при запуске программы будет сообщено, что при чтении целого подаётся некорректное значение. Причина: подача в качестве однобайтового целого признака перевода строки.
- **form='formatted'** — указываем, что **form** (спецификатор форматности) принимает значение **'formatted'** (форматный — форма содержимого каждой записи файла определяется оператором **format**. Настройка умолчания **form** зависит от режима доступа:

access='direct' — **form='unformatted'**,

access='sequental' — **form='formatted'**.

Далее программа просто читает записи из файла под именем **input1.dat**, который через оператор **open** связан с устройством **8** в режиме прямого доступа при **recl=7** и форматном способе чтения. Вывод результата чтения имеет вид

```
 j=           1  i=           1 s= abcd
 j=           2  i=           2 s= efgh
 j=           3  i=           3 s= ijkl
 j=           4  i=           4 s= mnop
 j=           5  i=           5 s= qrst
```

Пример записи файла прямого доступа

```
program t_rec_write; implicit none; integer j, i, i0, i1;
character(5) s, s0, s1
open( 8,file='input1.dat',access='direct',recl=7,form='formatted')
open(10,file='input2.dat',access='direct',recl=7,form='formatted')
open(11,file='input3.dat',access='direct',recl=7,form='formatted')
do j=1,5
  read(8,rec=j,fmt='(i1,a5)') i, s                    ! читаем input1.dat
  write(10,rec=j,fmt='(i1,a5)')i,s                    ! пишем в input2.dat
  write(11,rec=j,fmt='(i1,a5,a1)')i,s,transfer(10,'a')! пишем в input3.dat
enddo
close(10); close(11)
open(10,file='input2.dat',access='direct',recl=7,form='formatted')
open(11,file='input3.dat',access='direct',recl=7,form='formatted')
do j=1,5
  read(10,rec=j,fmt='(i1,a5)') i0, s0                 ! читаем input2.dat
  read(11,rec=j,fmt='(i1,a5)') i1, s1                 ! читаем input3.dat
  write(*,'(a,i3,a,a,i3,a)') ' 10: ',i0, s0, '     11:',i1,s1
enddo
end program t_rec_write
```

Программа **t_rec_write** считывает данные файла прямого доступа **input1.dat** (из предыдущего примера), и переписывает их в два других файла прямого доступа **input2.dat** и **input3.dat**:

```
1.abcd 2.efgh 3.ijkl 4.mnop 5.qrst                ! файл input2.dat

1.abcd                                            ! файл input3.dat
2.efgh
3.ijkl
4.mnop
5.qrst
```

Для того, чтобы каждая запись в файле прямого доступа начиналась с новой строки необходимо предыдущую запись завершать символом, который соответствует нажатию клавиши **enter**. Символ при выводе в файл выглядит как пустое место (пробел), хотя таковым не является, т.е. не виден. Получить именно его можно посредством вызова встроенной функции **transfer(10,'a')**, которая, не изменяя физического представления первого аргумента (в данном случае типа integer), позволяет *посмотреть* на него через *очки* типа **char**.

13.5 Спецификатор типа ошибки iostat=ios

Спецификатор используется только в ключевой форме. **ios** справа от знака равенства в **iostat=ios** (Input Output Status) есть имя скалярной переменной пользовательской программы для приёма кода причины завершения, который может оказаться следующим:

- **0** *(нуль)*, если операция ввода-вывода завершилась успешно;
- **>0** *(целое большее нуля)*, если произошла ошибка;
- **<0** *(целое меньшее нуля)*: ошибок нет, но встречен признак окончания файла.

Конкретные числовые значения **ios** зависят от реализации.

Наряду с **iostat** имеются ещё и ключевые аргументы **err** и **end**, которые позволяют передать управление на метку программы при возникновении одного из двух последних частных случаев **iostat** соответственно. Обычно использование **err** и **end** выглядит проще, но требует постановки метки и ссылки на неё, что при опечатке чревато неприятностями.

Ошибка, контролируемая **iostat**, может иметь разные обличья и возникать по разным причинам. Рассмотрим некоторые примеры.

Пусть программа должна вводить сначала значения целочисленной переменной **n**, затем двух вещественных переменных **a** и **b**, далее переменной **l** булева типа, переменной **c** комплексного типа и, наконец, символьной переменной **w**.

```
program test_iostat; implicit none; integer ios, n;
real a, b; logical l; complex c; character w
read(*,*,iostat=ios) n, a, b, l, c, w; write(*,*) ' ios=',ios
write(*,*) ' n=',n, ' a=',a, '  b=',b
write(*,*) ' l=',l, '  c=',c, ' w=', w
select case(ios)
  case( 0 ); write(*,*) ' главная ветвь обработки'; stop 0
  case(:-1); write(*,*) ' достигнут признак окончания вводимого файла'
             stop 1
  case( 1:); write(*,*) ' ошибка ввода !!!'; stop 2
end select
end
```

Пусть **input** — файл с правильными входными данными:

```
5  0.51  1.23  .true. (90.4,-35.3) 'M'
```

Тогда в результате пропуска программы **test_iostat** получим:

```
ios=           0
n=           5  a=  0.509999990       b=   1.23000002
l= T   c= (  90.4000015    , -35.2999992    )  w=M
главная ветвь обработки
```

Как и указывалось, значение **ios** получилось равным нулю и оператор **select case** выбрал соответствующую ветвь обработки.

Если при подготовке файла с входными данными случайно вместо целого значения первого вводимого параметра **n** указали вещественное, то получили бы:

```
ios=        5010
n=           0  a=   0.00000000       b=   0.00000000
l= T   c= (  9397536.00    ,  4.59163468E-41)  w=
ошибка ввода !!!
```

Как видно, ни одна из вводимых преременных не приняла вводимого значения.

Таким образом, ключевой аргумент **iostat** позволяет выбрать нужную ветвь алгоритма, которая через посредство оператора **stop** может передать в переменную **$?** оболочки, код причины, по которой данный алгоритм завершил работу, что, в свою очередь, позволит оболочке адекватно выбрать соответствующую ветвь скрипта.

Использованием **iostat** с целью поиска ошибок ввода часто пренебрегают, так как без **iostat=ios** в качестве аргумента оператора **read** пропуск программы завершается аварийным сообщением вида:

```
At line 3 of file main.f95 (unit = 5, file = 'stdin')
Fortran runtime error: Bad integer for item 1 in list input
```

Подобное сообщение содержит более полезную для поиска ошибки информацию, чем фраза **ошибка ввода**, так как указывает и номер строки исходного текста и даже номер и тип параметра, по вине которого произошёл аварийный останов.

Поэтому аргумент **iostat=ios** (в случае **ios>0**) необходим тогда, когда при обнаружении ошибки выгодно не прекращать работу, а продолжать, выводя программно, например, имена файлов и строки с бракованными данными для последующего их анализа, но продолжая обработку корректных.

Отрицательное значение фактического аргумента **ios** при **iostat=ios** может служить индикатором выхода из бесконечного цикла, если оператор чтения встречает признак окончания файла.

Так, достаточно типична ситуация, когда программа при разных пропусках должна вводить разные файлы с неодинаковым количеством данных. Вряд ли нас устроит ручной пересчёт этих количеств для того, чтобы указывать при очередном запуске программы: *сколько данных она должна ввести.*

Естественно, поручить подобный подсчёт самой программе при вводе.

Кстати, такой подсчёт может, в частности, служить и дополнительным контролем ввода, если, всё-таки, заранее известно сколько строк должно быть введено.

Рассмотрим программу:

```
program test_iostat_1; implicit none; integer ios, k, m, n;
real a, b; logical l; complex c; character(1) q
k=0; m=0
write(*,'(1x,"  n",7x,"a",9x,"b",6x,"l",5x,&
&            "Re(c)",7x,"Im(c)",4x,"q",6x,"ios" )')
do
  k=k+1
  read(*,*,iostat=ios) n, a, b, l, c, q
  select case(ios)
    case( 0 ); m=m+1; call work
    case(:-1); write(*,*) ' Число корректных строк ........ (m)=',m; exit
    case( 1:); write(*,*) ' Ошибка в строке под номером ... (k)=',k
  end select
enddo
k=k-1
write(*,*) ' Число вводимых строк k=',k
stop 0
contains
subroutine work
write(*,'(i4,f10.3,f10.3,l5,2(f10.3,3x),a,3x,i5)')&
&                                        n, a, b, l, c, q, ios
end subroutine work
end
```

которая должна должна обрабатывать массу файлов, каждый из которых содержит разное число строк (по шесть параметров в строке).

Пусть файл **inp1** является одним из них.

```
 1       10.123       10.321   .false.     (54.567,     -801.765)   M
 2       20.123       20.321    .true.     (64.567,     -901.765)   A
3.       30.123       30.321    .true.     (74.567,     -701.765)   B
 4       40.123       40.321      al       (84.567,     -601.765)   C
 5       20           50.321       f       (64.567,     -901.765)   D
```

При этом известно, что среди данных могут быть сбойные значения. Обработка строки, введённой безошибочно, осуществляется внутренней процедурой **work**. Программа **test_iostat_1** вводит строки поочерёдно, анализирует их на предмет наличия ошибок, обрабатывает корректные строки, выводит номера некорректных и подсчитывает число всех введённых строк. Нас не должны смущать две звёздочки в операторе **read** — помним про операции перенаправления.

Файл с результатами обработки inp1:

```
 n       a         b      l     Re(c)       Im(c)    q      ier
 1    10.123    10.321    F    54.567     -801.765   M       0
 2    20.123    20.321    T    64.567     -901.765   A       0
Ошибка в строке под номером ... (k)=           3
Ошибка в строке под номером ... (k)=           4
 5    20.000    50.321    F    64.567     -901.765   D       0
Число корректных строк ........ (m)=           3
Число вводимых строк k=           5
```

13.6 Спецификаторы перехода err=l и end=l

Спецификаторы используется только в ключевой форме.

Спецификатор **err=l** — спецификатор перехода по ошибке.

Спецификатор **err=l** — спецификатор перехода по признаку окончания фйайла.

l — метка оператора, которому в соответствующем случае следует передать управление.

В качестве примера приведём программу **test_iostat_2**, которая делает то же самое, что и **test_iostat_1**, но используя при этом вместо **iostat** упомянутые спецификаторы перехода **err** и **end**.

```
program test_iostat_2; implicit none; integer k, m, n;
real a, b; logical l; complex c; character(1) q
k=0; m=0
write(*,'(1x,"  n",7x,"a",9x,"b",6x,"l",5x,&
&            "Re(c)",7x,"Im(c)",4x,"q")')
do
  k=k+1
  read(*,*,err=10,end=77) n, a, b, l, c, q; m=m+1; call work
   cycle
   77    write(*,*) ' Число корректных строк ........ (m)=',m; exit
   10    write(*,*) ' Ошибка в строке под номером ... (k)=',k
enddo
k=k-1
write(*,*) ' Число вводимых строк k=',k
stop 0
contains
subroutine work
write(*,'(i4,f10.3,f10.3,l5,2(f10.3,3x),a)') n, a, b, l, c, q
end subroutine work
end
```

Результат пропуска программы **test_iostat_2**:

```
   n       a         b      l     Re(c)       Im(c)    q
   1    10.123    10.321    F    54.567     -801.765   M
   2    20.123    20.321    T    64.567     -901.765   A
  Ошибка в строке под номером ... (k)=           3
  Ошибка в строке под номером ... (k)=           4
   5    20.000    50.321    F    64.567     -901.765   D
  Число корректных строк ........ (m)=           3
  Число вводимых строк k=           5
```

13.7 Спецификатор advance=e продвижения по файлу

Спецификатор **advance** используется только в ключевой форме.

e справа от **advance=e** (*advance — продвигаться*) — одно из двух скалярных значений символьного типа **'yes'** или **'no'**, которые при использовании **форматного** ввода-вывода задают один из двух способов подвижки указателя файла после чтения-записи данного:

- **yes** — к началу очередной записи (вне зависимости от того, сколько данных из текущей при однократном вызове оператора ввода-вывода было прочитано-записано);
- **no** — к началу очередного данного текущей записи.

При форматном вводе-выводе по умолчанию полагается **advance='yes'**. Другими словами, если имеется файл, состоящий из ряда записей, каждая из которых содержит какие-то числа, то при вызове **read** читающий элемент после чтения из **текущей записи** всегда нацеливается на чтение **следующей записи**. Подчеркнём ещё раз — не на чтение очередного данного **текущей записи**, а, именно, на чтение начального данного **следующей**.

Если же при вызове оператора **read** спецификатору **advance** установлено значение **'no'**, то читающий элемент после чтения какого-либо данного **текущей записи** будет нацелен на чтение **следующего данного** этой же **текущей записи**.

Обратимся к примеру. Пусть файл **input** содержит пять записей, причём в каждой по десять чисел типа **integer**, записанных по формату **10i5**:

```
  101  102  103  104  105  106  107  108  109  110
  111  112  113  114  115  116  117  118  119  120
  121  122  123  124  125  126  127  128  129  130
  131  132  133  134  135  136  137  138  139  140
  141  142  143  144  145  146  147  148  149  150
```

```
program test_advance; implicit none; integer i, j, k, a
character(3) q(2) /'yes','no '/
open (10,file='input')
do j=1,2
 write(*,'(" advance=",a3,3x," Номер ",3x,"Читаемое по "/&
&                      ,14x,"значения",3x," I4 значение")') q(j)
  do i=1,5
    if (j==2) then; k=i; else; k=10*(i-1)+1; endif

    read(10,'(i5)',advance=q(j)) a;  write(*,*) '      ',k,' ', a
  enddo
  rewind(10)
enddo
stop 0
end
```

Программа **test_advance** выводит результаты чтения файла **input** сначала в режиме **advance='yes'**, а затем (после переустановки указателя файла на его начало посредством оператора **rewind(10)**) — в режиме **advance='no'**.

```
       advance=yes       !      advance=no
=========================!=========================
   Номер    Читаемое по  !  Номер    Читаемое по
 значения    I4 значение ! значения    I4 значение
     1           101     !     1           101
    11           111     !     2           102
    21           121     !     3           103
    31           131     !     4           104
    41           141     !     5           105
```

Режим **advance='no'** выгоден, например, когда данные читаются из файла, содержимое которого — огромное количество однотипных чисел в одной длинной записи, которую накладно запоминать целиком в оперативной памяти (если вообще возможно). Аналогично, режим выгоден и при моделировании соответствующего файла вывода.

13.8 Спецификаторы eor=l и size=k для режима advance='no'

- Спецификатор **eor=l** служит для организации перехода по окончанию записи.

  l справа от знака равенства в **eor=l** есть метка оператора, которому передаётся управление при обнаружении окончания записи (*eor — End Of Record*).

- Спецификатор **size=k** служит для подсчёта **k** количества позиций отводимых под данное, которое читается или выводится.

- Оба спецификатора используются только в режиме **advance='no'**.

```
program test_advance_1; implicit none; integer i, j, k, ier, a
open (10,file='input'); open (20,file='result1',status='replace')
write(*,'(11x,"j",11x,"i",10x,"ier",9x,"a",12x,"k")')
j=1; i=0;
do;  i=i+1
  read (10,'(i5)',advance='no',eor=3, size=k,iostat=ier) a;
  if (ier<0) exit
  if (ier >0) then; write(*,'(a,i5,a)') 'error in', j,'-st line'
        write(20,'(a1)',advance='no') transfer(10_1,'a');  j=j+1
  endif
  if (ier==0) then; write(*,*) j,i,ier,a, k
                    write(20,'(i4)',advance='no') a
  endif
  cycle
3 continue; write(20,'(a1)',advance='no') transfer(10_1,'a'); j=j+1
            write(*,'(11x,"j",11x,"i",10x,"ier",9x,"a",12x,"k")')
enddo
write(*,'(31x,i5/a,i5,a)') ier,'Обработано ', i-2,' чисел'; close(20)
end
```

Программа **test_advance_1** демонстрирует пример работы с файлом **input**, каждая из записей (строк) которого содержит некоторое (возможно неодинаковое) количество чисел.

```
  101  102  103
  111  112  113  114  115  116  117
  121  122  123  124
  131  132  133  134  135  136  137  138  139  140
  141  142  143  144  145  146  147  148
```

Запись каждой строки завершалась нажатием клавиши **enter**, что помещало в качестве последнего символа строки невидимый символ, соответствующий **ASCII**-коду равному **10** (перевод строки).

Программа работает с тремя файлами:

- **input**, из которого оператором **read** в режиме **advance='no'** по формату **i5** через устройство **unit=10** происходит чтение чисел;
- **result1**, в который оператором **write** в режиме **advance='no'** через устройство **unit=20** по формату **i4** происходит перезапись чисел, вводимых из **input**;
- **result**, в который операторами

  write(*,'(11x,"j 11x,"i 10x,"ier 9x,"a 12x,"k")')
  write(*,'(31x,i5/a,i5,a)') ier,'Обработано ', i-2,' чисел'

  через стандартное устройство вывода осуществляется построчный вывод

  1. **j** — номера записи, в которой читаемое данное находится;
  2. **i** — сквозного (по всем записям) номера данного;
  3. **ier** — кода ошибки чтения текущего данного по спецификатору **iostat=ier** при чтении исходного файла **input**;
  4. **a** — самого читаемого данного;
  5. **k** — количества символов, из которых данное состоит.

  Перенаправление стандартного вывода в файл **result** осуществляется посредством операции $>$ при запуске исполнимого файла.

Как только ключ **eor** обнаруживает при чтении признак окончания текущей записи файла **input** программа

- передаёт управление оператору, помеченному меткой **3**;
- записывает через устройство **unit=20** в файл **result1** невидимый символ, соответствующий коду перевода строки;
- выводит через стандартное устройство вывода фразу о завершении текущей строки; вычисляет номер следующей и возвращается к первому оператору тела цикла для возможного чтения числа из очередной записи файла **input**.

Вывод программы в файлы result1 и result.

```
 101 102 103
 111 112 113 114 115 116 117
 121 122 123 124
 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148

          j           i          ier         a           k
          1           1           0         101         5
          1           2           0         102         5
          1           3           0         103         5
          j           i          ier         a           k
          2           5           0         111         5
          2           6           0         112         5
          2           7           0         113         5
          2           8           0         114         5
          2           9           0         115         5
          2          10           0         116         5
          2          11           0         117         5
          j           i          ier         a           k
          3          13           0         121         5
          3          14           0         122         5
          3          15           0         123         5
          3          16           0         124         5
          j           i          ier         a           k
          4          18           0         131         5
          4          19           0         132         5
          4          20           0         133         5
          4          21           0         134         5
          4          22           0         135         5
          4          23           0         136         5
          4          24           0         137         5
          4          25           0         138         5
          4          26           0         139         5
          4          27           0         140         5
          j           i          ier         a           k
          5          29           0         141         5
          5          30           0         142         5
          5          31           0         143         5
          5          32           0         144         5
          5          33           0         145         5
          5          34           0         146         5
          5          35           0         147         5
          5          36           0         148         5
          j           i          ier         a           k
                                  -1
Обработано    36 чисел
```

14 Приложение V. Простые поэлементные функции

Напоминание. Поэлементными называются функции, которые могут в качестве своего аргумента использовать как скаляр, так и массив. В последнем случае функция применяется к каждому элементу массива.

К простым числовым функциям отнесём:

1. **abs(a)** — абсолютное значение аргумента **a**.
2. **aimag(z)** — мнимая часть комплексного аргумента **z**.
3. **conjg(z)** — сопряжённое комплексному аргументу **z**.
4. **aint(a [, kind])** — усечение аргумента **a** до целого числа.
5. **anint(a [,kind])** — ближайшее к вещественному **a** целое число в форме вещественного.
6. **nint(a [,kind])** — ЦЕЛОЕ, ближайшее к вещественному аргументу.
7. **ceiling(a [,kind])** — наименьшее целое большее или равное аргументу.
8. **floor(a [,kind])** — наибольшее целое меньшее или равное аргументу.
9. **dim(x,y)** — разность **x-y**, если она положительна; иначе **нуль**.
10. **dprod(x,y)** — произведение вещественных аргументов
11. **max(a1, a2 [, a3, ...])** — максимальное значение
12. **min(a1, a2 [, a3, ...])** — минимальное значение
13. **mod(a,p)** — остаток от деления **a** на **p**.
14. **modulo(a,p)** — модифицированный остаток от деления **a** на **p**, когда за результат деления принимается наибольшее целое меньшее или равное аргументу.
15. **sign(a,b)** — абсолютное значение **a** со знаком **b**.

14.1 Примеры работы с функцией abs

```
program tsabs; implicit none
integer   , parameter :: i=-17; real(4), parameter :: a=-17.3
real(8)   , parameter :: abad=-17.3
real(8)   , parameter :: aa=-17.3_8              ! или -17.3d0
complex(4), parameter ::    c=(4.1,-3.1)
complex(8), parameter :: cbad=(4.1,-3.1)
complex(8), parameter ::   cc=(4.1d0,-3.1d0)     ! или 4.1_8, -3.1_8
integer(2), parameter :: ia(4)=(/-3,-4,-5,6/)
write(*,*) ' Функция abs(x): абсолютное значение аргумента.'
write(*,*) 'integer       i ==>    abs(i)=',abs(i)
write(*,*) 'real(4)       a ==>    abs(a)=',abs(a)
write(*,*) 'real(8)    abad ==> abs(abad)=',abs(abad)
write(*,*) 'real(8)      aa ==>   abs(aa)=',abs(aa)
write(*,*) 'complex(4)    c ==>    abs(c)=',abs(c)
write(*,*) 'complex(8) cbad ==> abs(cbad)=',abs(cbad)
write(*,*) 'complex(8)   cc ==>   abs(cc)=',abs(cc)
write(*,*) 'integer(2) ia(4)==>   abs(ia)=',abs(ia)
end program tsabs
```

```
 Функция abs(x): абсолютное значение аргумента.
integer       i ==>    abs(i)=          17
real(4)       a ==>    abs(a)=   17.30000
real(8)    abad ==> abs(abad)=   17.2999992370605
real(8)      aa ==>   abs(aa)=   17.3000000000000
complex(4)    c ==>    abs(c)=   5.140039
complex(8) cbad ==> abs(cbad)=   5.14003877677095
complex(8)   cc ==>   abs(cc)=   5.14003891035856
integer(2) ia(4)==>   abs(ia)=      3      4      5      6
```

- Обратите внимание, что, если при задании именованной константы (или переменной) типа **REAL(8)** её числовое значение будет записано в форме **F** (обычном виде с фиксированной запятой, например **4.1**), то определяемый 8-байтовый объект будет иметь лишь 7-8 верных десятичных цифр, а не 15-16, как хотелось бы.

  Форма **F** определяет **по умолчанию** число одинарной точности. Поэтому **4.1** вместе со своей погрешностью округления, соответствующей одинарной точности, будет лишь по форме преобразовано к виду удвоенной (т.е. отведётся большее число бит на порядок и мантиссу числа, добавив к имеющейся погрешности типа **real(4)** ещё и погрешность типа **real(8)**.

14.2 Примеры работы с функцией aimag

```
program ts_aimag; implicit none
complex(4),parameter ::    c=(4.1,-3.1)
complex(8),parameter :: cbad=(4.1,-3.1)
complex(8),parameter ::   cc=(4.1d0,-3.1d0)
complex(4),parameter :: car(3)=(/(1.2,-0.2),(1.3,0.3),(1.4,0.4)/)
write(*,*) ' Функция aimag(z): мнимая часть комплексного числа'
write(*,*) 'complex(4)     c ==>    aimag(c)=',aimag(c)
write(*,*) 'complex(8)  cbad ==> aimag(cbad)=',aimag(cbad)
write(*,*) 'complex(8)    cc ==>   aimag(cc)=',aimag(cc)
write(*,*) 'complex(4) car(3)==>   aimag(car)=',aimag(car)
end program ts_aimag
```

```
 Функция aimag(z): мнимая часть комплексного числа
complex(4)     c ==>    aimag(c)=  -3.100000
complex(8)  cbad ==> aimag(cbad)=  -3.09999990463257
complex(8)    cc ==>   aimag(cc)=  -3.10000000000000
complex(4) car(3)==>   aimag(car)= -0.2000000     -0.3000000     -0.4000000
```

Результат пропуска после **gfortran -fdefault-real-8 ts_aimag.f**:

```
 Функция aimag(z): мнимая часть комплексного числа
complex(4)     c ==>    aimag(c)=  -3.100000
complex(8)  cbad ==> aimag(cbad)=  -3.10000000000000
complex(8)    cc ==>   aimag(cc)=  -3.10000000000000
complex(4) car(3)==>   aimag(car)= -0.2000000     -0.3000000     -0.4000000
```

У ФОРТРАН-компиляторов разных фирм наименования опций могут различаться. Поэтому, спокойнее использовать ФОРТРАН-средства, управляющие ситуацией. Например, использовать для описания переменных операторы:

```
integer, parameter :: mp=4; real(mp) a, b, c; a=1.6_mp
```

Если же потребуется перевод многофайлового проекта на ячейки иной разрядности, то достаточно будет изменение константы **mp** осуществить лишь в модуле

```
module my_prec
implicit none
integer, parameter :: mp=4  ! 8, 10, 16
end module my_prec
```

который подсоединяется к нужным программным единицам оператором **use my_prec**.

14.3 Примеры работы с функцией conjg

```
      program ts_conjg; implicit none
      complex(4),parameter ::      c=(4.1,-3.1)
      complex(8),parameter ::   cbad=(4.1,-3.1)
      complex(8),parameter ::     cc=(4.1d0,-3.1d0)
      complex(4),parameter :: car(3)=(/(1.2,-0.2),(1.3,0.3),(1.4,0.4)/)
      complex(4) v(3)
      integer i
      write(*,*) ' Функция conjg(z): сопряжённое комплексному z'
      write(*,'(a,2e17.8)')    '   c ==> conjg(c)=',    conjg( c  )
      write(*,'(a/(2e25.17))') 'cbad ==> conjg(cbad)=', conjg(cbad)
      write(*,'(a/(2e25.17))') '  cc ==>   conjg(cc)=', conjg( cc )
      v=conjg(car)
      write(*,'(a/(i3,2e17.8))') 'complex(4) car(3)==>  v=conjg(car)=',
     >            (i,v(i),i=1,3)
      v(1)=(5,6)
      write(*,'(a/2e17.8))') ' conjg( v(1))=',conjg( v(1))
      write(*,'(a/2e17.8))') ' conjg((5,6))=',conjg((5,6))


      end program ts_conjg
```

```
  Функция conjg(z): сопряжённое комплексному z
   c ==> conjg(c)=   0.40999999E+01   0.30999999E+01
cbad ==> conjg(cbad)=
  0.40999999046325684E+01  0.30999999046325684E+01
  cc ==>   conjg(cc)=
  0.40999999999999996E+01  0.31000000000000001E+01
complex(4) car(3)==>  v=conjg(car)=
  1   0.12000000E+01   0.20000000E+00
  2   0.13000000E+01  -0.30000001E+00
  3   0.14000000E+01  -0.40000001E+00
 conjg( v(1))=
   0.50000000E+01  -0.60000000E+01
 conjg((5,6))=
   0.50000000E+01  -0.60000000E+01
```

14.4 Примеры работы с функцией aint

```
program ts_aint
implicit none
real(4), parameter ::       a=4.1
real(8), parameter ::    abad=4.1
real(8), parameter ::      aa=4.1_8
real(10), parameter ::      b=-5.9
write(*,*) ' Функция aint(a [,kind]): усечение до целого числа'
write(*,*) ' real(4)    a ==>         aint(a)=',aint(a)
write(*,*) ' real(8) abad ==>      aint(abad)=',aint(abad)
write(*,*) ' real(8)   aa ==>        aint(aa)=',aint(aa)
write(*,*) 'real(10)    b ==>         aint(b)=', aint(b)
write(*,*) 'real(8)    aa ==> aint(aa,kind=4)=', aint(aa,kind=4)
write(*,*) 'real(8)    aa ==>      aint(aa,4)=', aint(aa,4)
write(*,*) 'real(4)    aa ==>      aint(aa,8)=', aint(aa,8)
write(*,*) 'real(4)  aa ==> aint(a=aa,kind=8)=',aint(a=aa,kind=8)
write(*,*) 'real(4)  aa ==> aint(kind=8,a=aa)=',aint(kind=8,a=aa)
end program ts_aint
```

```
 Функция aint(a [,kind]): усечение до целого числа
 real(4)    a ==>         aint(a)=   4.0000000
 real(8) abad ==>      aint(abad)=   4.0000000000000000
 real(8)   aa ==>        aint(aa)=   4.0000000000000000
real(10)    b ==>         aint(b)=  -5.0000000000000000000
real(8)    aa ==> aint(aa,kind=4)=   4.0000000
real(8)    aa ==>      aint(aa,4)=   4.0000000
real(4)    aa ==>      aint(aa,8)=   4.0000000000000000
real(4)  aa ==> aint(a=aa,kind=8)=   4.0000000000000000
real(4)  aa ==> aint(kind=8,a=aa)=   4.0000000000000000
```

- Обратите внимание на то, что функция **aint** без второго параметра (необязательного, если он не нужен) получает результат того же типа, что и аргумент, т.е. соответствующего вещественного типа, о чём можно судить по количеству нулей мантиссы.

- **aint** работает с аргументами исключительно вещественного типа. Типы **integer** и **complex** приводят к сообщению об ошибке.

- **aint** производит именно усечение числа, путём отбрасывания его дробной части.

- Помним, что при вызове любой функции любой её аргумент можно задать как в ключевой форме (например,**aint(aa,kind=4)**), так и позиционной (например,**aint(aa,4)**).

14.5 Примеры работы с функцией anint

```
      program ts_anint
      implicit none
      real(4), parameter ::       a=4.1
      real(8), parameter ::       b=4.9
      real(8), parameter ::       c=-4.1_8
      real(10), parameter ::      d=-5.9_10
      write(*,*) ' Функция anint(a [,kind]): ближайшее к вещественному'
      write(*,*) '                    целое число в форме вещественного'
      write(*,*) ' real(4)    a=4.1     anint(a)=',anint(a)
      write(*,*) ' real(8)    b=4.9     anint(b)=',anint(b)
      write(*,*) ' real(8)    c=-4.1_8  anint(c)=',anint(c)
      write(*,*) 'real(10)    d=-5.9_10 anint(d)=',anint(d)
      end program ts_anint
```

```
 Функция anint(a [,kind]): ближайшее к вещественному
                   целое число в форме вещесьвенного
 real(4)    a=4.1     anint(a)=   4.0000000
 real(8)    b=4.9     anint(b)=   5.0000000000000000
 real(8)    c=-4.1_8  anint(c)=  -4.0000000000000000
real(10)    d=-5.9_10 anint(d)=  -6.0000000000000000000
```

14.6 Примеры работы с функцией nint

```
      program ts_nint
      implicit none
      real(4), parameter ::       a=4.1
      real(8), parameter ::       b=4.9
      real(8), parameter ::       c=-4.1_8
      real(10), parameter ::      d=-5.9_10
      write(*,*) ' Функция nint(a [,kind]): ЦЕЛОЕ, ближайшее к ',
     >           ' вещественному'
      write(*,*) ' real(4)    a=4.1     nint(a)=',nint(a)
      write(*,*) ' real(8)    b=4.9     nint(b)=',nint(b)
      write(*,*) ' real(8)    c=-4.1_8  nint(c)=',nint(c)
      write(*,*) 'real(10)    d=-5.9_10 nint(d)=',nint(d)
      end program ts_nint
```

```
 Функция nint(a [,kind]): ЦЕЛОЕ, ближайшее к  вещественному
 real(4)    a=4.1     nint(a)=           4
 real(8)    b=4.9     nint(b)=           5
 real(8)    c=-4.1_8  nint(c)=          -4
real(10)    d=-5.9_10 nint(d)=          -6
```

14.7 Примеры работы с функцией ceiling

```
    program ts_ceiling
    implicit none
    real(4), parameter ::       a=4.1
    real(8), parameter ::       b=4.9
    real(8), parameter ::       c=-4.1_8
    real(10), parameter ::      d=-5.9_10
    write(*,*) ' Функция ceiling(a [,kind]): наименьшее ЦЕЛОЕ большее'
    write(*,*) '                                 или равное аргументу'
    write(*,*) '                 ceiling(4.0)=',ceiling(4.0)
    write(*,*) ' real(4) a=4.1     ceiling(a)=',ceiling(a)
    write(*,*) ' real(8) b=4.9     ceiling(b)=',ceiling(b)
    write(*,*) ' real(8) c=-4.1_8  ceiling(c)=',ceiling(c)
    write(*,*) 'real(10) d=-5.9_10 ceiling(d)=',ceiling(d)
    end program ts_ceiling
```

```
 Функция ceiling(a [,kind]): наименьшее ЦЕЛОЕ большее
                                или равное аргументу
                 ceiling(4.0)=           4
 real(4) a=4.1     ceiling(a)=           5
 real(8) b=4.9     ceiling(b)=           5
 real(8) c=-4.1_8  ceiling(c)=          -4
real(10) d=-5.9_10 ceiling(d)=          -5
```

14.8 Примеры работы с функцией floor

```
    program ts_floor
    implicit none
    real(4), parameter ::       a=4.1
    real(8), parameter ::       b=4.9
    real(8), parameter ::       c=-4.1_8
    real(10), parameter ::      d=-5.9_10
    write(*,*) ' Функция floor(a [,kind]): наибольшее ЦЕЛОЕ меньшее'
    write(*,*) '                                  или равное аргументу'
    write(*,*) '                  floor(4.0)=',floor(4.0)
    write(*,*) ' real(4) a=4.1     floor(a)=',floor(a)
    write(*,*) ' real(8) b=4.9     floor(b)=',floor(b)
    write(*,*) ' real(8) c=-4.1_8  floor(c)=',floor(c)
    write(*,*) 'real(10) d=-5.9_10 floor(d)=',floor(d)
    end program ts_floor
```

```
 Функция floor(a [,kind]): наибольшее ЦЕЛОЕ меньшее
                               или равное аргументу
                 floor(4.0)=           4
 real(4) a=4.1     floor(a)=           4
 real(8) b=4.9     floor(b)=           4
 real(8) c=-4.1_8  floor(c)=          -5
real(10) d=-5.9_10 floor(d)=          -6
```

14.9 Примеры работы с функцией dim

```
      program ts_dim
      implicit none
      real(4)  a,  b,  c,  d
      complex(4) u, v
      write(*,*) ' Функция dim(x,y): разность x-y, если она'
      write(*,*) '                   ПОЛОЖИТЕЛЬНА;  иначе нуль'
      a=4.3; b=3.3;  write(*,'("dim(a,b)=",e15.7)') dim(a,b)
      c=3.3; d=4.3;  write(*,'("dim(c,d)=",e15.7)') dim(c,d)
      write(*,'("dim(4,1)=",i5)') dim(4,1)
      write(*,'("dim(1,4)=",i5)') dim(1,4)
      end program ts_dim
```

```
  Функция dim(x,y): разность x-y, если она
                    ПОЛОЖИТЕЛЬНА;  иначе нуль
dim(a,b)=  0.1000000E+01
dim(c,d)=  0.0000000E+00
dim(4,1)=    3
dim(1,4)=    0
```

14.10 Примеры работы с функцией dprod

```
program ts_prod1; implicit none
real(4)  a,  b
real(8)  a0, b0, a8, b8, aa, bb, cc, dd, ee, ff, gg
a=5.2;              write(*,*) '                 a=', a
a0=a;               write(*,*) '                a0=', a0
a8=real(a,kind=8); write(*,*) '                a8=', a8
b=2.3;              write(*,*) '                 b=', b
b0=b;               write(*,*) '                b0=', b0
b8=real(b,kind=8); write(*,*) '                b8=', b8
aa=5.2d0;           write(*,*) '                aa=', aa
bb=2.3d0;           write(*,*) '                bb=', bb
cc=aa*bb;           write(*,*) '        cc=aa * bb=', cc
dd=5.2*2.3;         write(*,*) '    dd=5.2 * 2.3  =', dd
ee=a*b;             write(*,*) '       ee= a * b  =', ee
ff=dprod(5.2,2.3); write(*,*) ' ff=dprod(5.2,2.3)=', ff
gg=dprod( a , b ); write(*,*) ' gg=dprod( a , b )=', gg
a=0.5;              write(*,*) '                 a=', a
b=3.5;              write(*,*) '                 b=', b
ee=a*b;             write(*,*) '       ee= a * b  =', ee
gg=dprod( a, b);   write(*,*) ' gg=dprod( a , b )=', gg
a=3.72382;      b=2.39265
aa=3.72382_8;  bb=2.39265_8; write(*,*) a*b, dprod(a,b), aa*bb
end
```

```
             a=   5.1999998
            a0=   5.1999998092651367
            a8=   5.1999998092651367
             b=   2.3000000
            b0=   2.2999999523162842
            b8=   2.2999999523162842
            aa=   5.2000000000000002
            bb=   2.2999999999999998
   cc=aa * bb=   11.959999999999999
dd=5.2 * 2.3  =   11.959999084472656
   ee= a * b  =   11.959999313354501
ff=dprod(5.2,2.3)=   11.959999313354501
gg=dprod( a , b )=   11.959999313354501
             a=  0.50000000
             b=   3.5000000
   ee= a * b  =   1.7500000000000000
gg=dprod( a , b )=   1.7500000000000000
 8.9097977       8.9097974404429010        8.9097979230000011
```

Возможен вопрос: Когда может быть выгодно получить произведение двух чисел типа **real(4)** в форме **real(8)** при сохранении погрешности округления типа **real(4)**?

14.11 Примеры работы с функциями max и min

```
      program ts_minmax
      implicit none
      integer a, b, c, aa(3), bb(4), cc(5)
      a=4; b=6; c=-3; write(*,*) ' max(a,b,c)=',max(a,b,c)
                      write(*,*) ' min(a,b,c)=',min(a,b,c)
      aa=(/1,2,3/);
      bb=(/-9,3,7,-10/)
      cc=(/0,1,2,3,4/)
      write(*,*) ' aa: ', aa; write(*,*) ' bb: ', bb
      write(*,*) ' cc: ', cc
      write(*,*) 'maxval(aa)=',maxval(aa)
      write(*,*) 'Наибольший из максимальных:'
      write(*,*) 'max(maxval(aa),maxval(bb),maxval(cc))=',
     >            max(maxval(aa),maxval(bb),maxval(cc))
      write(*,*) 'minval(aa)=',minval(aa)
      write(*,*) 'Наименьший из минимальных:'
      write(*,*) 'min(minval(aa),minval(bb),minval(cc))=',
     >            min(minval(aa),minval(bb),minval(cc) )
      write(*,*) 'Наибольший из минимальных:'
      write(*,*) 'max(minval(aa),mixval(bb),minval(cc))=',
     >            max(minval(aa),minval(bb),minval(cc) )
      write(*,*) 'Наименьший  из максимальных:'
      write(*,*) 'min(maxval(aa),maxval(bb),maxval(cc))=',
     >            min(maxval(aa),maxval(bb),maxval(cc) )
      end program minmax
```

```
 max(a,b,c)=           6
 min(a,b,c)=          -3
 aa:            1           2           3
 bb:           -9           3           7         -10
 cc:            0           1           2           3           4
maxval(aa)=           3
Наибольший из максимальных:
max(maxval(aa),maxval(bb),maxval(cc))=           7
minval(aa)=           1
Наименьший из минимальных:
min(minval(aa),minval(bb),minval(cc))=         -10
Наибольший из минимальных:
max(minval(aa),mixval(bb),minval(cc))=           1
Наименьший  из максимальных:
min(maxval(aa),maxval(bb),maxval(cc))=           3
```

14.12 Примеры работы функции mod

Функция **mod(a,p)** находит остаток от деления **a** на **p**, т.е.

mod(a,p)= a-int(a/p)*p

Например, при **аргументах a и p типа integer** получаем:

a	b	mod(a,p)= a-int(a/p)*p
17	3	17-int(17 / 3)* 3= 17 - 5 * 3 = 2
-17	3	-17-int((-17)/ 3)* 3=-17 -(-5)* 3 =-17+15=-2
17	-3	17-int(17 /(-3))*(-3)=17-(-5)*(-3)=17-15=2
-17	-3	-17-int((-17)/(-3))*(-3)=-17-5*(-3)=-17+15=-2

В случае **аргументов типа real** таблица выглядит так:

a	b	mod(a,p)= a-int(a/p)*p
17.5	5.5	17.5-int(17.5 / 5.5)* 5.5 = 17.5 - 3 * 5.5 = 1.0
-17.5	5.5	-17.5-int((-17.5)/ 5.5)* 5.5 =-17.5-(-3)* 5.5 =-17.5+16.5=-1.0
17.5	-5.5	17.5-int(17.5 /(-5.5))*(-5.5)=17.5-(-3)*(-5.5)=17.5-16.5=1.0
-17.5	-5.5	-17.5-int((-17.5)/(-5.5))*(-5.5)=-17.5-(-3)*(-5.5)=-17.5+16.5=-1.0

```
program ts_mod
implicit none
integer  a,  p
real     s,  t
a= 17; p= 3; write(*,'("mod(",i3,",",i2,")=",i5,i5)')&
                          & a,p,mod(a,p),a-int(a/p)*p
a=-17; p= 3; write(*,'("mod(",i3,",",i2,")=",i5,i5)')&
                          & a,p,mod(a,p),a-int(a/p)*p
a= 17; p=-3; write(*,'("mod(",i3,",",i2,")=",i5,i5)')&
                          & a,p,mod(a,p),a-int(a/p)*p
a=-17; p=-3; write(*,'("mod(",i3,",",i2,")=",i5,i5)')&
                          & a,p,mod(a,p),a-int(a/p)*p
!
s=  17.5; t= 5.5; write(*,*) 'mod(s,t)=', mod(s,t), s-int(s/t)*t
s= -17.5; t= 5.5; write(*,*) 'mod(s,t)=', mod(s,t), s-int(s/t)*t
s=  17.5; t=-5.5; write(*,*) 'mod(s,t)=', mod(s,t), s-int(s/t)*t
s= -17.5; t=-5.5; write(*,*) 'mod(s,t)=', mod(s,t), s-int(s/t)*t
end program ts_mod
```

```
mod( 17, 3)=    2    2
mod(-17, 3)=   -2   -2
mod( 17,-3)=    2    2
mod(-17,-3)=   -2   -2
 mod(s,t)=   1.0000000       1.0000000
 mod(s,t)=  -1.0000000      -1.0000000
 mod(s,t)=   1.0000000       1.0000000
 mod(s,t)=  -1.0000000      -1.0000000
```

14.13 Примеры работы функции modulo

Функция **modulo(a,p)** находит такой остаток от деления **a** на **p**, когда за результат деления принимается наибольшее целое меньшее или равное аргументу.

modulo(a,p)= a-floor(real(a)/real(p))*p

Часто функцию **modulo** называют функция **по модулю**.
Например, для **аргументов 17 и 3 типа integer** получаем:

a	b	modulo(a,p)= a-floor(real(a),real(p))*p
17	3	17 - floor(17.0 / 3.0) * 3 = 17 - 5 * 3 = 2
-17	3	-17 - floor(-17.0 / 3.0) * 3 =-17 -(-5)* 3 =-17+15=-2
17	-3	17 - floor(17.0 /(-3.0)) *(-3)= 17-(-5)*(-3)=17-15=2
-17	-3	-17 - floor((-17.0)/(-3.0)) *(-3)=-17-5*(-3)=-17+15=-2

```
program ts_modulo; implicit none; integer  a,  p; real     s,  t
a= 17; p= 3; write(*,*) 'modulo(a,p)=',modulo(a,p),a-floor(real(a)/real(p))*p
a=-17; p= 3; write(*,*) 'modulo(a,p)=',modulo(a,p),a-floor(real(a)/real(p))*p
a= 17; p=-3; write(*,*) 'modulo(a,p)=',modulo(a,p),a-floor(real(a)/real(p))*p
a=-17; p=-3; write(*,*) 'modulo(a,p)=',modulo(a,p),a-floor(real(a)/real(p))*p
s= 17.0; t= 3.0; write(*,*) 'modulo(s,t)=', modulo(s,t), s-floor(s/t)*t
s=-17.0; t= 3.0; write(*,*) 'modulo(s,t)=', modulo(s,t), s-floor(s/t)*t
s= 17.0; t=-3.0; write(*,*) 'modulo(s,t)=', modulo(s,t), s-floor(s/t)*t
s=-17.0; t=-3.0; write(*,*) 'modulo(s,t)=', modulo(s,t), s-floor(s/t)*t
write(*,*) 'modulo( 8, 5)=',modulo( 8, 5);
write(*,*) 'modulo(-8, 5)=',modulo(-8, 5)
write(*,*) 'modulo( 8,-5)=',modulo( 8,-5)
write(*,*) 'modulo(-8,-5)=',modulo(-8,-5)
write(*,*) 'modulo( 7.285, 2.35)=',modulo( 7.285, 2.35)
write(*,*) 'modulo(-7.285, 2.35)=',modulo(-7.285, 2.35)
write(*,*) 'modulo( 7.285,-2.35)=',modulo( 7.285,-2.35)
write(*,*) 'modulo(-7.285,-2.35)=',modulo(-7.285,-2.35)
end program ts_modulo
```

```
 modulo(a,p)=           2           2
 modulo(a,p)=           1           1
 modulo(a,p)=          -1          -1
 modulo(a,p)=          -2          -2
 modulo(s,t)=   2.0000000       2.0000000
 modulo(s,t)=   1.0000000       1.0000000
 modulo(s,t)=  -1.0000000      -1.0000000
 modulo(s,t)=  -2.0000000      -2.0000000
 modulo( 8, 5)=           3
 modulo(-8, 5)=           2
 modulo( 8,-5)=          -2
 modulo(-8,-5)=          -3
```

```
 modulo( 7.285, 2.35)=  0.23500013
 modulo(-7.285, 2.35)=   2.1149998
 modulo( 7.285,-2.35)=  -2.1149998
 modulo(-7.285,-2.35)= -0.23500013
```

14.14 Примеры работы функции sign

```
      program ts_sign
      implicit none
      integer j
      real a, b

      a=5.3
      do j=-2,2
        b=j
        write(*,'("sign(",f5.2,",",f5.2,")=",f5.2)') a, b,  sign(a,b)
      enddo
      a=-5.3
      do j=-2,2
        b=j
        write(*,'("sign(",f5.2,",",f5.2,")=",f5.2)') a, b,  sign(a,b)
      enddo


      end program ts_sign
```

```
sign( 5.30,-2.00)=-5.30
sign( 5.30,-1.00)=-5.30
sign( 5.30, 0.00)= 5.30
sign( 5.30, 1.00)= 5.30
sign( 5.30, 2.00)= 5.30
sign(-5.30,-2.00)=-5.30
sign(-5.30,-1.00)=-5.30
sign(-5.30, 0.00)= 5.30
sign(-5.30, 1.00)= 5.30
sign(-5.30, 2.00)= 5.30
```

15 Приложение VI. Функции запроса характеристик представления числовых данных

Функции запроса (см. [2]) — это запросы к окружению о характеристиках представления данных, зависящих от реализации. В качестве аргумента функции используют имена переменных или констант, однако результат работы зависит не от значения аргумента **x**, а от его типа и параметра разновидности типа. Обозначения и формулы, используемые в разделе **12.10.5** книги [2].

- **b** — основание системы счисления для вещественных чисел

  r — основание системы счисления для целых чисел;

  Любое из них можно получить встроенной фукцией **radix(x)**.

- **q** — количество значащих цифр в модельном представлении **целого**;

  p — количество значащих цифр в модельном представлении **вещественного**;

  Любое из них можно получить встроенной фукцией **digit(x)**.

- $\mathbf{b^{1-p}}$ — результат работы функции **epsilon(x)**

- $\mathbf{e_{max}}$ — максимальный порядок (показатель степени основания); находится встроенной функцией **maxexponent(x)**

  $\mathbf{e_{min}}$ — минимальный порядок (показатель степени основания); находится встроенной функцией **minexponent(x)**

- $\mathbf{r^q - 1}$ — наибольшее целое в модельном представлении **целого**;

  $\mathbf{(1 - b^{-p}) * b^{e_{max}}}$ — наибольшее вещественное в модельном представлении **вещественного**;

  Любое из них можно получить встроенной фукцией **huge(x)**.

  Наименьшее положительное число в модельном представлении вещественного данного можно получить функцией **tiny(x)**.

Десятичную точность модели и наибольший десятичный порядок данного той же разновидности типа, что и аргумент **x**, находятся посредством функций **precision(x)** и **range(x)** соответственно

Функция **bit_size(i)** находит число битов в модели данных той же разновидности целого типа, что и аргумент **i**.

15.1 Функция radix(x)

— возвращает значение типа **integer** равное основанию системы счисления, используемой для представления чисел того же типа, что и аргумент **x** (integer или real).

```
program test_radix; implicit none; real(4)  x4; real(8)  x8; real(10) xd
real(16) xh
write(*,*) 'По умолчанию для типа integer основание = radix( 1 )=',radix( 1 );
write(*,*) 'По умолчанию для типа  real   основание = radix(1.3)=',radix(1.3);
write(*,*) 'radix(x4)=',radix(x4),'  radix(1.3_4 )=',radix(1.3_4)
write(*,*) 'radix(x8)=',radix(x8),'  radix(1.3_8 )=',radix(1.3_8)
write(*,*) 'radix(xd)=',radix(xd),'  radix(1.3_10)=',radix(1.3_10)
write(*,*) 'radix(xh)=',radix(xh),'  radix(1.3_16)=',radix(1.3_16)
end
```

```
 По умолчанию для типа integer основание = radix( 1 )=           2
 По умолчанию для типа  real   основание = radix(1.3)=           2
 radix(x4)=           2   radix(1.3_4 )=           2
 radix(x8)=           2   radix(1.3_8 )=           2
 radix(xd)=           2   radix(1.3_10)=           2
 radix(xh)=           2   radix(1.3_16)=           2
```

15.2 Функция digits(x)

```
program test_digits; implicit none; character(14) sf
    integer i
       real r
   real(4) r4;    real(8) r8; real(10) rx;                  ! real(16) rh
integer(1) i1; integer(2) i2; integer(4) i4; integer(8) i8; ! integer(16) ih
write(*,*) ' По умолчанию:'
write(*,'(13x,a,i3)') '  digits( i )=', digits( i )
write(*,'(13x,a,i3)') '  digits( r )=', digits( r )
sf='(a,i3,3x,a,i3)'
write(*,'(/a)') ' INTEGER(...) при явном описании типа:'
write(*,sf) 'digits(i1)=',digits(i1),'digits( 1_1 )=',digits(1_1)
write(*,sf) 'digits(i2)=',digits(i2),'    digits( 1_2 )=',digits(1_2)
write(*,sf) 'digits(i4)=',digits(i4),'    digits( 1_4 )=',digits(1_4)
write(*,sf) 'digits(i8)=',digits(i8),'    digits( 1_8 )=',digits(1_8)
write(*,sf) 'digits(ih)=',digits(i8),'    digits( 1_16)=',digits(1_8)
write(*,'(/a)') '     REAL(...) при явном описании типа:'
write(*,sf) 'digits(r4)=',digits(r4),'   digits(1.3_4 )=',digits(1.3_4)
write(*,sf) 'digits(r8)=',digits(r8),'   digits(1.3_8 )=',digits(1.3_8)
write(*,sf) 'digits(rx)=',digits(rx),'   digits(1.3_10)=',digits(1.3_10)
!write(*,sf) 'digits(rh)=',digits(rh),'   digits(1.3_16)=',digits(1.3_16)
end

  По умолчанию:
              digits( i )= 31          ! Почему не 32 ?
              digits( r )= 24          ! Почему не 32 ?

 INTEGER(...) при явном описании типа:
digits(i1)=  7   digits( 1_1 )=  7      ! Почему не   8 ?
digits(i2)= 15       digits( 1_2 )= 15  ! Почему не  16 ?
digits(i4)= 31       digits( 1_4 )= 31  ! Почему не  32 ?
digits(i8)= 63       digits( 1_8 )= 63  ! Почему не  64 ?
digits(ih)=127       digits( 1_16)=127  ! Почему не 128.?

     REAL(...) при явном описании типа:
digits(r4)= 24      digits(1.3_4 )= 24  ! Почему не  32 ?
digits(r8)= 53      digits(1.3_8 )= 53  ! Почему не  64 ?
digits(rx)= 64      digits(1.3_10)= 64  ! Почему не  64 ?
digits(rh)=113      digits(1.3_16)=113  ! Почему не 128 ?
```

15.3 Функция epsilon(x)

— возвращает значение $\mathbf{b^{1-p}}$ (самая младшая степень двойки в пределах разрядной сетки соответствующей вещественной единицы):

1+epsilon(x)>1, но **1+epsilon(x)/2==1**.

Тип и параметр типа значения **epsilon(x)** такие же как у **x**.

```
program ts_epsilon; implicit none;
character(27), parameter :: sf(4)=(/'(i3,e26.8 ,i4,e26.8 ,l5,l8)',&
&                                   '(i3,e26.16,i4,e26.16,l5,l8)',&
&                                   '(i3,e26.19,i4,e26.19,l5,l8)',&
&                                   '(i3,e26.19,i4,e26.19,l5,l8)'/)
real( 4) r4,e4; real(8) r8,e8; real(10) rX,ex; real(16) rH,eH
integer p4, p8, pX, pH
p4= digits(r4);   p8= digits(r8);  pX= digits(rX); pH= digits(rH)
e4=epsilon(r4);   e8=epsilon(r8);  eX=epsilon(rX); eH=epsilon(rH)
write(*,*) 'k=4, 8, 10, 16 - параметр разновидности данного типа REAL'
write(*,'(" p=digits(r) - число битов для модели мантиссы REAL(k)")')
write(*,'(78("-"))')
write(*,'(2x,"k",12x,"epsilon",10x,"p",8x,"2.0_k**(1-p)",7x,&
&                                 "1+e/=1",2x,"1+e/2==1")')
write(*,'(78("-"))')
write(*,sf(1)) kind(e4), e4,p4, 2.0**(1-p4),    1+e4/=1, 1+e4/2==1
write(*,sf(2)) kind(e8), e8,p8, 2.0_8**(1-p8),  1+e4/=1, 1+e8/2==1
write(*,sf(3)) kind(eX), eX,pX, 2.0_10**(1-pX), 1+e4/=1, 1+eX/2==1
write(*,sf(4)) kind(eH), eH,pH, 2.0_10**(1-pH), 1+e4/=1, 1+eH/2==1
stop 0
end program ts_epsilon
```

```
k=4, 8, 10, 16 - ппараметр разновидности данного типа REAL
 p=digits(r) - число битов для модели мантиссы REAL(k)
------------------------------------------------------------------------------
  k            epsilon          p        2.0_k**(1-p)       1+e/=1  1+e/2==1
------------------------------------------------------------------------------
  4           0.11920929E-06  24           0.11920929E-06    T       T
  8    0.2220446049250313E-15  53    0.2220446049250313E-15    T       T
 10 0.1084202172485504434E-18  64 0.1084202172485504434E-18    T       T
 16 0.1925929944387235853E-33 113 0.1925929944387235853E-33    T       T
```

Так как результат работы **epsilon(x)** не зависит от величины аргумента, то не указывая явно параметр разновидности, можно, например, грубо оценить погрешность округления **x** по формуле **abs(x)*epsilon(x)**;

15.4 Функции maxexponent, minexponent

Функции **maxexponent(x)** и **minexponent(x)** находят соответственно $\mathbf{e_{max}}$ и $\mathbf{e_{min}}$ (максимальный и минимальный порядки в модели данного того же типа и его параметра разновидности, что и аргумент **x**). Порядок — показатель степени основания **b**, которое обычно равно 2.

```
program ts_maxminexp; implicit none
real(4) e, b; real(8) e8; real(10) eX; real(16) eH
integer(4) m, p;
m=maxexponent(e); write(*,'(" maxexponent(e)=",i5)') m
p=digits(e);         write(*,'("p=digits(e)=", i5  )') p
b= radix(e);         write(*,'("b= radix(e)=",e15.7)') b
e=2**m;      write(*,'(20x,a,e15.7)') 'e=2**m=',e
e=b**m;      write(*,'(16x,a,e15.7)') 'e=2.0_4**m=',e
e=b**(m-1);  write(*,'(12x,a,e15.7)')  'e=2.0_4**(m-1)=',e
e=2*(1-b**(-p))*b**(m-1);
write(*,'(2x,a,e15.7)') 'e=2*(1-b**(-p))*b**(m-1)=',e
write(*,'(15x,a,e15.7)') 'huge(1.0_4)=', huge(1.0_4)
m=minexponent(e); write(*,'(" minexponent(e)=",i5)') m
e=2**m;      write(*,'(20x,a,e15.7)') 'e=2**m=',e
e=b**m;      write(*,'(16x,a,e15.7)') 'e=2.0_4**m=',e
e=b**(m-1);  write(*,'(12x,a,e15.7)')  'e=2.0_4**(m-1)=',e
write(*,'(19x,a,e15.7)') 'tiny(e)=', tiny(e)
write(*,'("x",6x,"maxexponent(x)",3x,"minexponent(x)")')
write(*,'(a,2i15)') 'e8', maxexponent(e8), minexponent(e8)
write(*,'(a,2i15)') 'eX', maxexponent(eX), minexponent(eX)
write(*,'(a,2i15)') 'eH', maxexponent(eX), minexponent(eH)
end program ts_maxminexp
```

```
 maxexponent(e)=  128
p=digits(e)=   24
b= radix(e)=  0.2000000E+01
                    e=2**m=  0.0000000E+00
                e=2.0_4**m=       Infinity
            e=2.0_4**(m-1)=  0.1701412E+39
  e=2*(1-b**(-p))*b**(m-1)=  0.3402823E+39
              huge(1.0_4)=  0.3402823E+39
 minexponent(e)= -125
                    e=2**m=  0.0000000E+00
                e=2.0_4**m=  0.2350989E-37
            e=2.0_4**(m-1)=  0.1175494E-37
                 tiny(e)=  0.1175494E-37
x      maxexponent(x)   minexponent(x)
e8           1024          -1021
eX          16384         -16381
eH          16384         -16381
```

15.5 Функция tiny

Функция **tiny** находит наименьшее положительное число в модели данного того же типа и параметра разновидности, что и аргумент

Программа **ts_tiny** демонстрирует работу пользовательской функции **tiny_my**, которая, используя **minexponent(x)**, вычисляет то же, что и встроенная **tiny**.

```
module tinymy
implicit none
interface tiny_my
  module procedure tiny4, tiny8, tinyX, tinyH
end interface tiny_my
contains
function tiny4(x) result(w); real(4) x, w; w= 2.0**(minexponent(x)-1)
end function tiny4
function tiny8(x) result(w); real(8) x, w; w= 2.0_8**(minexponent(x)-1)
end function tiny8
function tinyX(x) result(w); real(10) x, w; w= 2.0_10**(minexponent(x)-1)
end function tinyX
function tinyH(x) result(w); real(16) x, w; w= 2.0_16**(minexponent(x)-1)
end function tinyH
end module tinymy

program ts_tiny
use tinymy
implicit none
integer(4) m, p
real(4) e, b
write(*,'(8x,"x",17x,"tiny(x)",14x,"tiny_my(x)")')
write(*,'(e15.7,2(e24.7,2x))')     0.0 ,   tiny(0.0_4), tiny_my(0.0_4)
write(*,'(e15.7,2(e25.16e3,2X))') -1._8,   tiny(-1._8), tiny_my(-1._8)
write(*,'(e15.7,2(e26.16e4,1x))')   3.7_10, tiny(3.7_10), tiny_my(3.7_10)
write(*,'(e15.7,2(e27.16e5))')   8e9_16, tiny(8e9_16), tiny_my(8e9_16)
stop 0
end program ts_tiny
```

```
        x                tiny(x)              tiny_my(x)
  0.0000000E+00           0.1175494E-37             0.1175494E-37
 -0.1000000E+01  0.2225073858507201E-307    0.2225073858507201E-307
  0.3700000E+01  0.3362103143112094E-4931   0.3362103143112094E-4931
  0.8000000E+10  0.3362103143112094E-04931  0.3362103143112094E-04931
```

15.6 Функция huge(x)

huge(x) — наибольшее число в модели того же типа и того же параметра разновидности, что и аргумент.

15.6.1 Семейство integer

Если для расчёта **huge(i)** (когда i — *целое*) воспользоваться непосредственно формулами

$$\mathbf{q = digits(i); h = 2^q - 1},$$

то можно обнаружить забавные эффекты конечнозначной целочисленной арифметики. Полезно проанализировать результаты работы

```
program ts_hugeint
implicit none
integer(1) i1
integer(2) i2
integer(4) i4, q1, h1, q2, h2, q4, h4, w4, q8, h8, w8
integer(8) i8, qo, ho
q1=digits(i1)
q2=digits(i2)
q4=digits(i4)
q8=digits(i8)
write(*,'("   q1=digits(i1)=",i4,";     2**",i2,"=",i11)') q1,q1,2**q1
write(*,'("   q2=digits(i2)=",i4,";     2**",i2,"=",i11)') q2,q2,2**q2
write(*,'("   q4=digits(i4)=",i4,";     2**",i2,"=",i11)') q4,q4,2**q4
write(*,'("   q8=digits(i8)=",i4,";     2**",i2,"=",i11)') q8,q8,2**q8
write(*,*)
write(*,'(" Почему расчёт 2**7 и 2**15 верен, а 2**31 и 2**63 ошибочен?")')
h1= 2**q1-1
h2= 2**q2-1
h4= 2**q4-1
h8= 2**q8-1
write(*,'("huge(i1)=",i10,"=2**q1-1=",i10)') huge(i1), h1
write(*,'("huge(i2)=",i10,"=2**q2-1=",i10)') huge(i2), h2
write(*,'("huge(i4)=",i10,"=2**q4-1=",i10)') huge(i4), h4
write(*,*)
write(*,'(a,i11,3x,a,i11,3x,a,i11)') '2**q4=',2**q4,&
          & '    2**q4-1=',2**q4-1,&
          &' 2*(2**(q4-1)-1)+1=',2*(2**(q4-1)-1)+1



write(*,'(" Почему, -2147483648-1=+2147483647?"/)')
```

```
write(*,'(70("="))')
w4=2**q4; write(*,'(a,b32.32)')  '      2**q4=',w4
          write(*,'(a,b32.32)')  'вычитаем  1=',1
          write(*,'(12x,32("-"))')
          write(*,'(a,b32.32)') '    2**q4-1=',h4
write(*,'(70(".."))')
w4=2**(q4-1); write(*,'(12x,a,b32.32)') '  2**(q4-1)=',w4
w4=w4-1;      write(*,'(12x,a,b32.32)') 'вычитаем  1=',1
              write(*,'(24x,32("-"))')
              write(*,'(9x,a,b32.32)') '   2**(q4-1)-1=',w4
w4=2*w4; write(*,'(10x,a,b32)') 'умножаем на 2=',2
         write(*,'(24x,32("-"))')
         write(*,'(4x,a,b32.32)')  '     2(2**(q4-1)-1)=',w4
         write(*,'(10x,a,b32.32)') 'прибавляем  1=',1
w4=w4+1; write(*,'(24x,32("-"))')
         write(*,'(2x,a,b32.32)') '   2(2**(q4-1)-1_1)+1=',w4
write(*,'(70("="))')
write(*,'("HUGE от целого аргумента")')
write(*,'(" kind(i)",3x,"digits(i)",11x,"huge(i)",9x,&
                      &"2*(2**(digits(i)-1)+1")')
write(*,'(4x,i1,i11,i28,i23)') kind(i1), digits(i1), huge(i1), h1
write(*,'(4x,i1,i11,i28,i23)') kind(i2), digits(i2), huge(i2), h2
write(*,'(4x,i1,i11,i28,i23)') kind(i4), digits(i4), huge(i4), h4
write(*,'(4x,i1,i11,i28,i23)') kind(i8), digits(i8), huge(i8), h8
write(*,'(70("="))')
write(*,*)
write(*,'(" Почему результаты расчёта одного и того же по формуле 2**q8-1"/&
&" и по huge(i8) различны, хотя, на первый взгляд, должны быть одинаковы?")')
qo=q8
write(*,*) ' q8=',q8
write(*,*) ' qo=',qo
write(*,'("2*(2**(q8-1)-1)+1=",i30)') 2*(2**(q8-1)-1)+1
write(*,'("2*(2**(qo-1)-1)+1=",i30)') 2*(2**(qo-1)-1)+1
write(*,'("2**qo=",b64.64)') 2**qo
write(*,'("2**qo=",i21)') 2**qo
end program ts_hugeint
```

Результаты работы ts_hugeint

```
   q1=digits(i1)=   7;     2** 7=        128
   q2=digits(i2)=  15;     2**15=      32768
   q4=digits(i4)=  31;     2**31=-2147483648
   q8=digits(i8)=  63;     2**63=          0

 Почему расчёт 2**7 и 2**15 верен, а 2**31 и 2**63 ошибочен?
huge(i1)=       127=2**q1-1=       127
huge(i2)=     32767=2**q2-1=     32767
huge(i4)=2147483647=2**q4-1=2147483647

2**q4=-2147483648       2**q4-1= 2147483647    2*(2**(q4-1)-1)+1= 2147483647
 Почему, -2147483648-1=+2147483647?

=====================================================================
      2**q4=10000000000000000000000000000000
вычитаем  1=00000000000000000000000000000001
           --------------------------------
    2**q4-1=01111111111111111111111111111111
.....................................................................
             2**(q4-1)=01000000000000000000000000000000
           вычитаем  1=00000000000000000000000000000001
                      --------------------------------
           2**(q4-1)-1=00111111111111111111111111111111
         умножаем на 2=                              10
                      --------------------------------
        2(2**(q4-1)-1)=01111111111111111111111111111110
         прибавляем  1=00000000000000000000000000000001
                      --------------------------------
     2(2**(q4-1)-1_1)+1=01111111111111111111111111111111
=====================================================================
HUGE от целого аргумента
 kind(i)   digits(i)           huge(i)         2*(2**(digits(i)-1)+1
    1          7                          127                    127
    2         15                        32767                  32767
    4         31                   2147483647             2147483647
    8         63         9223372036854775807                     -1
=====================================================================

 Почему результаты расчёта одного и того же по формуле 2**q8-1
 и по huge(i8) различны, хотя, на первый взгляд, должны быть одинаковы?
  q8=          63
  qo=                  63
2*(2**(q8-1)-1)+1=                            -1
2*(2**(qo-1)-1)+1=           9223372036854775807
2**qo=10000000000000000000000000000000000000000000000000000000000000000
2**qo= -9223372036854775808
```

Пояснения к результатам работы ts_hugeint.

1. Первые четыре выполняемых оператора получают количество двоичных цифр, соответствующих модельному представлению целых чисел. Напомним, что результат, возвращаемый функцией **digits** по умолчанию, имеет тип **integer(4)**.

2. Значения, получаемые по формулам **2**q1=128** и **2**q2=32768**, свободно размещаются в **integer(4)** и вычисляются верно (см. первые две строки файла-результата **ts_hugeint.res**).

3. Однако, **2**q4** — отрицательно (хотя абсолютная величина числа получена верно). Значение 2**31 для записи в четырёхбайтовую ячейку требует её самый старший бит (**31**-ый), который ФОРТРАН трактует как знак целого отрицательного числа, записанного в дополнительном коде. Поэтому неудивительно, что на типе **integer(4)** значение 2**31 равно -2147483648.

   Тем не менее формула **2**q4-1** даёт верный результат и для типа **integer(4)**, хотя пользуется при этом абсолютно неверным значением **2**31=-2147483648**. Почему так происходит, видно из вывода соответствующих операндов в двоичном виде.

4. **2**q8**: тип значений **2** и **q8** — **integer(4)**. Поэтому и тип результата — **integer(4)**. Однако, в отличие от случая **q4=31**, значение **q8=63**. Запись чисел всегда ориентирована на правую границу поля. Поэтому в поле, отводимое для типа **integer(4)** попадают лишь 32 младших двоичных разряда числа 2**63 (так что w8=0). Вычитая из четырёхбайтового нуля единицу, получаем -1 того же типа **integer(4)**. Умножая её на 2 получаем -2. И, наконец, прибавляя к -2 единицу, получаем -1.

   Для правильного представления результата операции **2**63** необходимо использовать тип **integer(8)**

15.6.2 Семейство real

В случае семейства **real** эффект переполнения при расчёте наибольшего числа возникает, если запрограммировать выражение $\mathbf{b^{e_{max}}}$ непосредственно. Для типа **real(4)** число $\mathbf{2.0^{128}} = +\mathbf{Infinity}$. Для программирования формулу $(\mathbf{1 - b^{-p}}) * \mathbf{b^{e_{max}}}$ удобно переписать в виде:

$$\mathbf{b} * (\mathbf{1 - b^{-p}}) * \mathbf{b^{e_{max}-1}}$$

```
program ts_hugereal; implicit none
real( 4) r4, h4; real( 8) r8, h8
real(10) rX, hX; real(16) rH, hH
integer emax
emax=maxexponent(r4)
write(*,'("emax=maxexponent(r4)=",i4)') emax
write(*,'("           2.0**emax=",e15.7)') 2.0**emax

h4=2*(1-2.0_4  ** (-digits(r4))) * 2.0_4  ** (maxexponent(r4)-1)
h8=2*(1-2.0_8  ** (-digits(r8))) * 2.0_8  ** (maxexponent(r8)-1)
hX=2*(1-2.0_10 ** (-digits(rX))) * 2.0_10 ** (maxexponent(rX)-1)
hH=2*(1-2.0_16 ** (-digits(rX))) * 2.0_16 ** (maxexponent(rH)-1)

write(*,'("m=kind(x)",2x,"digits(x)",10x,"huge(x)",7x,&
&"2*(1-2.0_m**(-digits(r)))*"/44x,"2.0_m**(maxexponent(r)-1)")')
write(*,'(i4,i12,e26.7,e26.7)') kind(r4), digits(r4), huge(r4), h4
write(*,'(i4,i12,e26.14e4,e26.14e4)') kind(r8), digits(r8), huge(r8), h8
write(*,'(i4,i12,e26.14e4,e26.14e4)') kind(rX), digits(rX), huge(rX), hX
write(*,'(i4,i12,e26.14e4,e26.14e4)') kind(rH), digits(rH), huge(rH), hH
end program ts_hugereal
```

```
emax=maxexponent(r4)= 128
           2.0**emax=       Infinity
m=kind(x)  digits(x)          huge(x)       2*(1-2.0_m**(-digits(r)))*
                                            2.0_m**(maxexponent(r)-1)
   4          24             0.3402823E+39             0.3402823E+39
   8          53    0.17976931348623E+0309    0.17976931348623E+0309
  10          64    0.11897314953572E+4933    0.11897314953572E+4933
  16         113    0.11897314953572E+4933    0.11897314953572E+4933
```

15.7 Функция precision(x)

— находит десятичную точность представления данного (количество десятичных цифр, которое соответствует модели его типа и параметра разновидности).

```
program ts_precision; implicit none
real(4)  x4
real(8)  x8
real(10) xX
!real(16) xH
write(*,*)
write(*,'(" PRECISION только для семейств REAL или COMPLEX:")')
write(*,*)
write(*,'(" precision( x4)=",i4)') precision( x4)
write(*,'(" precision( x8)=",i4)') precision( x8)
write(*,'(" precision( xX)=",i4)') precision( xX)
!write(*,'(" precision( xH)=",i4)') precision( xH)

write(*,*)
write(*,'(" precision( ( 3.0_4,  0.1_4))=",i4)') precision( (3.0_4, 0.1_4))
write(*,'(" precision( ( 3.0_8,  0.2_8))=",i4)') precision( (3.0_8, 0.2_8))
write(*,'(" precision( (3.0_10, 0.3_10))=",i4)') precision((3.0_10,0.3_10))
!write(*,'(" precision( (3.0_16, 0.3_16))=",i4)') precision((3.0_16,0.3_16))
end program ts_precision
```

```
 PRECISION только для семейств REAL или COMPLEX:

 precision( x4)=   6
 precision( x8)=  15
 precision( xX)=  18

 precision( ( 3.0_4,  0.1_4))=   6
 precision( ( 3.0_8,  0.2_8))=  15
 precision( (3.0_10, 0.3_10))=  18
```

15.8 Функция range(x)

— находит значение диапазона показателя степени десятки для модели данного того же типа и того же параметра типа, что и аргумент.

```
program ts_range; implicit none
integer(1) i1; integer(2) i2; integer(4) i4; integer(8) i8
   real(4) x4;     real(8) x8;    real(10) xX; complex y
write(*,'("RANGE для целых:")')
write(*,'("range(i)",10x,"huge(i)",10x,"int(lg(huge(real(i))))")')
write(*,'(i5,i25,i17)') range(i1), huge(i1), int(log10(real(huge(i1))))
write(*,'(i5,i25,i17)') range(i2), huge(i2), int(log10(real(huge(i2))))
write(*,'(i5,i25,i17)') range(i4), huge(i4), int(log10(real(huge(i4))))
write(*,'(i5,i25,i17)') range(i8), huge(i8), int(log10(real(huge(i8))))

write(*,'("RANGE для комплексного:",i3)') range(y)
write(*,'("RANGE для вещественных:")')

write(*,'("m=kind(x)",1x,"range(x)",4x,"huge(x)",2x,"[lg(huge(x))]",&
                          & 2x, "maxexponent(x)",2x,&
                          &"[maxexponent(x)*",/&
                         &66x,"lg(2.0_m)  ]")')

write(*,'(i5,i10,e15.3e2,i8,i17,i17)') kind(x4),range(x4), huge(x4),&
 & int(log10(huge(x4))), maxexponent(x4),int(maxexponent(x4)*log10(2.0_4))
write(*,'(i5,i10,e15.3e3,i8,i17,i17)') kind(x8),range(x8), huge(x8),&
 & int(log10(huge(x8))), maxexponent(x8),int(maxexponent(x8)*log10(2.0_8))
write(*,'(i5,i10,e15.3e4,i8,i17,i17)') kind(xX),range(xX), huge(xX),&
 & int(log10(huge(xX))), maxexponent(xX), int(maxexponent(xX)*log10(2.0_10))

write(*,'("m=kind(x)",1x,"range(x)",4x,"tiny(x)",2x,"[lg(tiny(x))]",&
                          & 2x, "minexponent(x)",2x,&
                          &"[minexponent(x)*",/&
                         &66x,"lg(2.0_m)  ]")')

write(*,'(i5,i10,e15.3e2,i8,i17,i17)') kind(x4),range(x4), tiny(x4),&
 & int(log10(tiny(x4))), minexponent(x4),int(minexponent(x4)*log10(2.0_4))
write(*,'(i5,i10,e15.3e3,i8,i17,i17)') kind(x8),range(x8), tiny(x8),&
 & int(log10(tiny(x8))), minexponent(x8),int(minexponent(x8)*log10(2.0_8))
write(*,'(i5,i10,e15.3e4,i8,i17,i17)') kind(xX),range(xX), tiny(xX),&
 & int(log10(tiny(xX))), minexponent(xX), int(minexponent(xX)*log10(2.0_10))
end
```

```
RANGE для целых:
range(i)          huge(i)          int(lg(huge(real(i))))
    2                      127                2
    4                    32767                4
    9               2147483647                9
   18      9223372036854775807               18
RANGE для комплексного: 37
RANGE для вещественных:
m=kind(x) range(x)    huge(x)  [lg(huge(x))]  maxexponent(x)  [maxexponent(x)*
                                                              lg(2.0_m)  ]
    4        37      0.340E+39      38              128              38
    8       307     0.180E+309     308             1024             308
   10      4931    0.119E+4933    4932            16384            4932
m=kind(x) range(x)    tiny(x)  [lg(tiny(x))]  minexponent(x)  [minexponent(x)*
                                                              lg(2.0_m)  ]
    4        37      0.118E-37     -37             -125             -37
    8       307     0.223E-307    -307            -1021            -307
   10      4931    0.336E-4931   -4931           -16381           -4931
```

15.9 Функция bit_size(i)

Функция **bit_size(i)** находит число битов в модели данных той же разновидности целого типа, что и аргумент **i**.

```
program ts_bit_size
implicit none
integer(1) i1; integer(2) i2; integer(4) i4; integer(8) i8
write(*,'("bit_size(i1)=",i4)') bit_size(i1)
write(*,'("bit_size(i2)=",i4)') bit_size(i2)
write(*,'("bit_size(i4)=",i4)') bit_size(i4)
write(*,'("bit_size(i8)=",i4)') bit_size(i8)
end
```

```
bit_size(i1)=         8
bit_size(i2)=        16
bit_size(i4)=        32
bit_size(i8)=        64
```

16 Приложение VII. Некоторые опции gfortran

16.1 Опции изменения правила умолчания

Напомним, что правило умолчания ФОРТРАНа полагает переменные, имена которых начинаются с букв **i**, **j**, **k**, **l**, **m**, и **n**, предназначенными для работы со значениями типа **integer**, а переменные, имена которых начинаются с любой другой буквы — со значениями типа **real**. Так что слова **integer** и **real** — те самые служебные слова, которые определяют тип переменной по умолчанию. Подчеркнём, ещё раз — не **real(4)** или **integer(4)**, а именно **integer** и **real**, т.е. без явного указания размера области памяти, отводимой под данные этих типов. При этом изначально считалось, что оба этих типа четырёхбайтовые по умолчанию.

Однако оказалось, что для задач расчётного характера в значительной мере могут быть востребованы типы **real(8)**, **real(10)**, **real(16)**. Возникла необходимость замены описания **real** на **real(8)**, **real(16)** или даже вообще на **real(mp)** (**mp** — именованная константа).

На первый взгляд наиболее практично — переопределить правило умолчания на новый тип данных, указав соответствующую опцию компилятора. Тогда обеспечивается 100%-ая гарантия такого переопределения во всех единицах компиляции, входящих в проект, при отсутствии каких-либо изменений в исходных текстах.

16.1.1 Опция -fdefault-real-8

Опция **-fdefault-real-8** переопределяет правило умолчания для типа **real** с *четырёхбайтового* на *восьмибайтовый*.

В таком случае описание **double precision** соответственно трактуется как *шестнадцатибайтовое*.

Рассмотрим, например, программу

```
program treal0; implicit none
real, parameter :: c=1.3;           real x, pi
                                    real(4) y;      real*4 z
double precision q, qa, qb;         real(8) t, s;   real(16) u
          write(*,*) '  c=',c; x=1.7
          write(*,*) '  x=',x
y=2.7;    write(*,*) '  y=',y
z=3.7;    write(*,*) '  z=',z
t=4.7;    write(*,*) '  t=',t
s=4.7_8;  write(*,*) '  s=',s
u=4.7_16; write(*,*) '  u=',u
pi=4*atan(1.0)
write(*,*) ' pi=',pi
write(*,*) '              sin(pi)=',sin(pi)
write(*,*) '              cos(pi)=',cos(pi)
write(*,*) 'sin(pi/4)-sqrt(2.0)/2=',sin(pi/4)-sqrt(2.0)/2
 q=pi
qa=4*atan(1.0_8);  qb=4*atan(1.0d0)
write(*,*) ' q=',q
write(*,'(a,e42.35,1x,a,e10.2)') ' qa=4*atan(1.0_8)=',qa,'sin(qa)=',sin(qa)
write(*,'(a,e42.35,1x,a,e10.2)') ' qb=4*atan(1.0d0)=',qb,'sin(qb)=',sin(qb)
end
```

Результат работы её исполнимого кода после **gfortran treal0.f95** и **./a.out** таков:

```
   c=   1.29999995
   x=   1.70000005
   y=   2.70000005
   z=   3.70000005            ! Если не оговорено особо, то по умолчанию
   t=   4.6999998092651367    ! запись 4.7 есть константа одинарной точности.
   s=   4.7000000000000002
   u=   4.7000000000000000000000000000000015
  pi=   3.14159274
               sin(pi)=  -8.74227766E-08
               cos(pi)=  -1.00000000
 sin(pi/4)-sqrt(2.0)/2=   0.00000000
  q=   3.1415927410125732
 qa=4*atan(1.0_8)= 0.31415926535897931159979634685441852E+01 sin(qa)=  0.12E-15
 qb=4*atan(1.0d0)= 0.31415926535897931159979634685441852E+01 sin(qb)=  0.12E-15
```

- Вывод результата происходит под управлением списка вывода (на что указывает вторая звёздочка в операторе **write**). Поэтому количество выводимых значащих цифр данных вещественных типов всегда соответствует разновидности типа, что позволяет по выводу результата проследить изменение трактовки компилятором правила умолчания ФОРТРАНа.

- Можно заметить, что хотя **t** описана *восьмибайтовой* (её значение, действительно, состоит из соответствующего разрядности числа цифр), тем не менее, в напечатанном значении к верным относится лишь старшая половина цифр.

- Причина — форма записи константы **4.7** есть форма записи четырёхбайтовой константы. Так что **4.7** перевелось в двоичную систему счисления с погрешностью округления порядка 10^{-7} и результат перевода с этой погрешностью был преобразован к формату двоичного представления типа **real(8)**.

- Аналогично, переменной **q** типа **double precision** (что при неизменённом правиле умолчания эквивалентно **real(8)**) присвоено значение четырёхбайтовой переменной **pi**, погрешность машинного представления которой по порядку величины такая же, как и у **4.7**. Поэтому, хоть мы и видим у **q** шестнадцать цифр, к верным следует отнести лишь старшие семь-восемь.

- Работа встроенных математических функций даёт ожидаемый для типа **real** результат: **pi** вычислено с одинарной (*четырёхбайтовой*) точностью; значение **sin(pi)** по сути дела равно погрешности машинного представления значения **pi**; $\cos(\mathbf{pi}) = \mathbf{-1.0}$ с погрешностью не хуже 10^{-8} и , вероятно, с не худшей погрешностью вычисляются $\mathbf{sin}(\mathbf{pi}/\mathbf{4})$ и $\mathbf{sqrt}(\mathbf{2.0})/\mathbf{2}$.

Однако, после **gfortran -fdefault-real-8 treal0.f95** и **./a.out** получим:

```
  c=   1.3000000000000000    <--= Работа опции
  x=   1.7000000000000000    <--= Работа опции
  y=   2.70000005                      Опция не отработала
  z=   3.70000005                      Опция не отработала
  t=   4.7000000000000002    <--= Работа опции
  s=   4.7000000000000002
  u=   4.7000000000000000000000000000000015
 pi=   3.1415926535897931
              sin(pi)=   1.2246467991473532E-016
              cos(pi)=  -1.0000000000000000
sin(pi/4)-sqrt(2.0)/2=  -1.1102230246251565E-016
 q=   3.1415926535897931159979634685441852
qa=4*atan(1.0_8)= 3.1415926535897931159979634685441852E+01 sin(qa)=  0.12E-15
qb=4*atan(1.0d0)= 3.1415926535897932384626433832795028E+01 sin(qb)=  0.87E-34
```

- Описания **real(4)** или **real*4** для опции **-fdefault-real-8** не эквивалентны **real** (они не являются описаниями по умолчанию: в них явно указано количество байт). Поэтому значения переменных **y** и **z** остались *четырёхбайтовыми*.

- На первый взгляд описатель **real** (разновидность типа по умолчанию) более лаконичен (по сравнению с **real(4)** или **real(8)**) и в то же время посредством **-fdefault-real-8** просто переводится в режим умолчания **real(8)**, причём опять-таки без какого-либо изменения исходного текста, что наиболее практично.

- Форма записи константы **4.7** не изменилась, но работа опции обеспечила погрешность округления, соответствующую типу **real(8)**.

- Тип **double precisin** переменной **q** после действия опции стал трактоваться как **real(16)** — по умолчанию вывелось 35 десятичных цифр, из которых верны лишь старшие 16: ведь *шестнадцатибайтовой* **q** присвоено значение *восьмибайтовой* **pi**.

- Для корректного присваивания переменной **qb** значения π используем оператор **qb=4*atan(1.0D0)**. Имя **atan** — родовое. Через него реализуется механизм перегрузки функций, подставляющий вместо **atan** соответствующее типу аргумента специфическое имя функции. Опция **-fdefault-real-8** сопоставляет константе **1.0** тип **real(8)**, а константе **1.0d0** — тип **real(16)**.

- В современном ФОРТРАНе **1.0d0** и **1.0_8** обозначают одну и ту же вещественную константу. Опция же **-fdefault-real-8** трактует синтаксис их записи по разному.
  В операторе **qa=4*atan(1.0_8)** явно указано, что аргумент типа **real(8)**. Поэтому для расчёта **arctg** вызывается функция со специфическим именем **datan**, получающая значение типа **real(8)**. Для выполнения расчёта с четверной — необходимо константу записать в виде **1.0d0**, т.е. именно в форме **D**

- **Вывод. -fdefault-real-8 обеспечивает переход на четверную точность при описаннии констант посредством их записи только в форме D, а при описании типа переменных — только посредством double precision.**

16.1.2 Опции -fdefault-real-8 и -fdefault-double-8

Пусть заранее известно, что работа программы должна будет проверяться как в режиме типа **real(8)**, так и в режиме типа **real(16)**. Для реализации режима **real(16)** (в предположении использования только опции **-fdefault-real-8**) описания констант и переменных придётся выполнять посредством **double precision**.

Получение же исполнимого кода в режиме **REAL(8)** (без внесения каких-либо изменений в исходный текст) достигается совместным указанием опций **-fdefault-real-8 -fdefault-double-8**. Рассмотрим пример:

```
program treal1; implicit none
double precision, parameter :: c=1.3d0
real, parameter :: d=1.4d0
double precision cc
real dd
write(*,*) '  c=',c;  write(*,*) '  d=',d
cc=c
dd=d
write(*,*) ' cc=',cc; write(*,*) ' dd=',dd
end

$ gfortran -fdefault-real-8 treal1.f95
$ ./a.out

   c=   1.300000000000000000000000000000000004
   d=   1.3999999999999999
  cc=   1.300000000000000000000000000000000004
  dd=   1.3999999999999999
```

- Как видим, описатель **double precision** привёл к генерация константы **c** и переменной **cc** четверной точности.

Результат же исполнимого кода **treal1** при совместном действии опций **-fdefault-real-8 -fdefault-double-8** имеет вид:

```
$ gfortran -fdefault-real-8  -fdefault-double-8 treal1.f95
$ ./a.out
   c=   1.3000000000000000
   d=   1.3999999999999999
  cc=   1.3000000000000000
  dd=   1.3999999999999999
```

16.2 Опции изменения явно указанных разновидностей

Во многих программах для описания типа вещественных переменных часто встречаться описатели с явно указанной разновидностью:

real*4, **real*8**, **real*16**, **real(4)**, **real(8)**, **real(10)**, **real(16)**

(описатели **real** или **double precision** задают разновидность типа неявно, т.е. по умолчанию, в зависимости от настройки компилятора, отводя обычно под тип **real** четыре байта).

Часто, однако, встречаются программы, в которых разновидность типа указана явно. Так что иногда возникает необходимость изменить явно указанную разновидность типа на другую, причём без явного изменения исходного текста. Похожая возможность для неявно заданных разновидностей **real** и **double precision** реализуется посредством опций **-fdefault-real-8** и **-fdefault-real-8 -fdefault-double-8** (см. 16.1.1 и 16.1.2).

Явно указанную разновидность типа **gfortran**изменяет посредством опций

-freal-4-real-10 -freal-4-real-16 -freal-4-real-8
-freal-8-real-10 -freal-8-real-16 -freal-8-real-4

Рассмотрим самую первую программу первого семестра, которая по значению аргумента $\mathbf{x = 10^{-7}}$, вычисляет значения переменных **y=1+x**, **z=y-1**, **t=z/x**, если все переменные описаны как **real(4)** или **real*4**.

```
program treal2
implicit none
real(4) x, y !; complex u
real*4  z, t
x=1e-7;    y=1+x;   z=y-1;   t=z/x !; u=(1.2,3.7)
write(*,*) ' x=',x
write(*,*) ' y=',y
write(*,*) ' z=',z
write(*,*) ' t=',t
!write(*,*) ' z=',u
end
```

1. Результат работы её исполнимого кода, полученного без указания опций:

```
$ gfortran  treal2.f95
$ ./a.out
  x=   1.00000001E-07
  y=   1.00000012
  z=   1.19209290E-07
  t=   1.19209290
```

2. Результат работы опции **-freal-4-real-8**:

```
$ gfortran -freal-4-real-8 treal2.f95
$ ./a.out
  x=   9.9999999999999995E-008
  y=   1.0000001000000001
  z=   1.0000000005838672E-007
  t=   1.0000000005838672
```

3. Результат работы опции **-freal-4-real-10**:

```
$ gfortran -freal-4-real-10 treal2.f95
$ ./a.out
  x=   9.99999999999999999985E-0008
  y=   1.00000009999999999995
  z=   9.99999999999482205859E-0008
  t=  0.99999999999482205859
```

4. Результат работы опции **-freal-4-real-16**:

```
$ gfortran -freal-4-real-16 treal2.f95
$ ./a.out
  x=   9.999999999999999999999999999999976E-0008
  y=   1.00000010000000000000000000000000002
  z=   1.00000000000000000000000000015020405E-0007
  t=   1.00000000000000000000000000015020405
```

5. Если в программе treal2 изменить описания типов на **real(8)** или **real*8**, то соответствующий исполнимый код, полученный без указания опций даст результат:

```
$ gfortran treal28.f95
$ ./a.out
  x=   1.0000000116860974E-007
  y=   1.0000001000000012
  z=   1.0000000116860974E-007
  t=   1.0000000000000000
```

6. Включение опции **-freal-8-real-4** приведёт к результату:

```
$ gfortran -freal-8-real-4 treal28.f95
$ ./a.out
  x=   1.00000001E-07
  y=   1.00000012
  z=   1.19209290E-07
  t=   1.19209290
```

7. Включение опции **-freal-8-real-10** приведёт к результату:

```
$ gfortran -freal-8-real-10 treal28.f95
$ ./a.out
  x=   1.0000000116860974230E-0007
  y=   1.0000010000000116861
  z=   1.0000000116860974230E-0007
  t=   1.0000000000000000000
```

8. Опция **-freal-8-real-16** получит:

```
$ gfortran -freal-8-real-16 treal28.f95
$ ./a.out
  x=   1.00000001168609742308035492897033691E-0007
  y=   1.00000010000000116860974230803549290
  z=   1.00000001168609742308035492897033691E-0007
  t=   1.00000000000000000000000000000000000
```

16.3 Опция -fimplicit-none

Заметим, что, вообще говоря необязательно отключать действие правила умолчания посредством ФОРТРАН-оператора **implicit none**. Того же можно достичь включением опции компилятора **gfortran**: **-fimplicit-none**, которая означает, что не допускается неявное описание типов, если только в тексте программы не встретится оператор **implicit** в явной форме.

Например, если из программы **treal2** исключить оператор **implicit none** и описание переменных **y** и **t** (допуская действие правила умолчания), но компиляцию программы провести с опцией **-fimplicit-none**, то получим

```
$ gfortran -fimplicit-none treal3.f95
treal3.f95:3.30:
x=1e-7;    y=1+x;   z=y-1;   t=z/x
                             1
Error: Symbol 't' at (1) has no IMPLICIT type
treal3.f95:3.12:
x=1e-7;    y=1+x;   z=y-1;   t=z/x
            1
Error: Symbol 'y' at (1) has no IMPLICIT type
aw@burst:~/lecture/semestr2/options_gfortran>
```

16.4 Опции -ffixed-line-length-N и -ffree-line-length-N

Опции служат для увеличения длины строки исходного текста, которая воспринимается компилятором (по умолчанию эта длина равна 72 или 132 однобайтовым символам). Конечно, никто специально не будет использовать в своей программе строки большей длины. Однако, при переходе с одного вида кодировки (**koi8r**) на другой (**utf8**) превышение допустимого предела может возникнуть неявно. В **utf8** коды букв кириллицы — двухбайтовые, а **koi8r** — однобайтовые. Поэтому после упомянутого перехода строка кириллицы, занимающая в **koi8r** только 40 байт, в **utf8** потребует **80**, хотя на экране будет выглядеть как 40-буквенная. Например, исходный текст ФОРТРАН-программы

```
      program tlength; implicit none; real dl /1.0/, a0
      a0=2*(1-sqrt(1-dl))/dl
      write(*,'("Точное значение нулевого момента функции)=",e15.7)') a0
      end
```

содержащей кириллицу выглядит на экране вполне безобидно, и в **koi8r**, компилируется без ошибок, но в кодировке **utf8** получим:

```
$ gfortran tlutf8.f
tlutf8.f:3.14:

      write(*,'("\xD0\xA2\xD0\xBE\xD1\x87\xD0\xBD\xD0\xBE\xD0\xB5 \xD0\xB7\xD0\x
BD\xD0\xB0\xD1\x87\xD0\xB5\xD0\xBD\xD0\xB8\xD0\xB5 \xD0\xBD\xD1\x83\xD0\xBB\xD0\
xB5\xD0\xB2\xD0\xBE\xD0\xB3\xD0\xBE \xD0\xBC\xD0\xBE\xD0\xBC\xD0\xB5
              1
ошибка: Syntax error in WRITE statement at (1)
```

- **-ffixed-line-length-N** устанавливает **N** — число байт, отводимое под часть строки, воспринимаемой компилятором, если исходный текст имеет фиксированный формат записи.
- **-ffree-line-length-N** делает то же для свободного формата.
- **N** можно задать равным **none**, что означает отсутствие какого-либо ограничения на длину строки (то же самое означает и **N=0**).

Результат активации **tlength**, полученной при **-ffixed-line-length-108**:

```
$ gfortran tlutf8.f -ffixed-line-length-108
$ ./a.out
Точное значение нулевого момента функции)=  0.2000000E+01
```

Список литературы

[1] Горелик А.М. 2018. Эволюция языка ФОРТРАН.

Устаревшие черты языка и средства для их замены // Препринты ИПМ им. М.В.Келдыша. 2018. №130. 13 с.

doi:10.20948/prepr-2018-130

URL:http://library.keldysh.ru/preprint.asp?id=2018-130

https://library.keldysh.ru/preprint.asp?id=2018-130

[2] Горелик А.М. 2006. Программирование на современном Фортране. – М.: Финансы и статистика, – 352 с.

[3] Бартеньев О.В. 1999. Фортран для студентов. – М.:"ДИАЛОГ – МИФИ – 400 с.

[4] Бартеньев О.В. 2000. Современный ФОРТРАН. "3-е изд., доп. и перераб. – М.; ДИАЛОГ – МИФИ, – 448 с.

[5] Рыжиков Ю.И. 2004. Современный ФОРТРАН: Учебник. – СПб.: КОРОНА принт, –288 с.

[6] Немнюгин М.А., Стесик О.Л. 2004. Современный ФОРТРАН: Самоучитель. – СПб.: БХВ-Петербург, 496 с.

[7] Немнюгин М.А., Стесик О.Л. 2008. ФОРТРАН в задачах и примерах многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 320 с.

[8] Касаткин А.И., Вальвачев А.Н. 1992. Профессиональное программирование на языке СИ: От Turbo C к Borland C++: Справ. пособие; Под общ. ред. А.И. Касаткина. – Мн.: Выш.шк., – 240 с.

[9] Шилдт Г. 2002. - Самоучитель C++: Пер. с англ. – 3-е изд. – СПб.: БХВ–Петербург, – 688 с.

[10] Клюшин Д.А. 2004. Полный курс C++. Профессиональная работа. – М.: Издательский дом “Вильямс” – 624 с. : ил.

[11] Игнатов В. 2000. Эффективное использование GNU Make.

[12] Richard M.Stallman, Roand McGrath

GNU Make Программа управления компиляцией.

GNU make Версия 3.79 Апрель 2000. перевод (С) Владимира Игнатова
http://linux.yaroslavl.ru/docs/prog/gnu_make_3.79_russian_manual.html

[13] Левин М. 2005. СИ++: Самоучитель / Максим Левин. – М.: ЗАО "Новый издательский дом", – 176с.

[14] Д.Кнут 2007. - Искусство программирования для ЭВМ т. 2. Получисленные алгоритмы (полиномиальная арифметика) 833 с.

[15] Гашков С.Б. 2014 Сложение однобитных чисел. Треугольник Паскаля, салфетка Серпинского и теорема Куммера. — Издательство Московского центра непрерывного математического образования, — 40с.

[16] Генри Уоррен, мл. 2003 Алгоритмические трюки для программистов. Издательский дом "Вильямс"; Москва; Санкт-Петербург; Киев; 285с. алгоритмы.

[17] Вирт Н., Алгоритмы + структуры данных = программы:

Пер. с англ. – М.: Москва, Мир, 1985. – 406 с., ил.

[18] Дмитриева М.В. Кубенский А.А. Элементы современного программирования: Учеб. пособие/Под ред. С.С. Лаврова. – СПб.: Издательство С.–Петербургского университета, 1991. – 272 с.

[19] Гутер Р.С., Полунов Ю.Л. От абака до компьютера. — 2-е изд., испр. и доп. — М.: Знание, 1981. — 208 с. + 32 с. вкл. (Библиотека «Знание»)

[20] www.vbstreets.ru/VB/Articles/66541.aspx

[21] «Нежданчики» языка ФОРТРАН.

https://habr.com/en/company/intel/blog/254235/