Ловягин Никита Юрьевич
Алимова Ольга Викторовна

Практикум на ЭВМ

Часть первая

Структурное и процедурно-модульное программирование

лабораторные работы и задачи

для студентов прикладных
математических специальностей

Санкт-Петербург
2020

Оглавление

Список лабораторных работ

Глава 1. Основы создания программ на языке Си

§1.1. Введение

Прежде чем начинать изучение языка Си следует ввести ряд ключевых понятий. В практическом курсе они вводятся неформально.

О языках программирования

Компьютеры "понимают" т.н. **машинные команды** или **коды**, представляющие собой числа. Эти команды исполняются **процессором** — устройством, осуществляющим управление компьютером, а также арифметические и логические вычисления. Числовые коды — язык, понятный компьютеру. Для облегчения понимания машинных команд человеком используется их буквенная запись. Наиболее близким к кодированию с помощью буквенного представления машинных команд является язык **ассемблера**.

Машинные коды и язык ассемблера используют простейшие синтаксические конструкции и команды, это языки программирования **низкого уровня**. Создание программы с их помощью возможно, но чаще всего сложно и неэффективно. Во-первых в силу элементарности инструкций для достижения "нетривиального" результата требуется написание их большого количества. Во-вторых отсутствие выразительных синтаксических конструкций, позволяющих, в частности, представить структуру программы более наглядно, более понятно человеку, является источником труднообнаружимых ошибок. Существуют и другие причины, в силу которых использовать машинные коды следует лишь там, где это действительно необходимо, например, при создании языков программирования и некоторых элементов операционных систем.

В противоположность языкам программирования низкого уровня разработаны **языки программирования высокого уровня**, лишенные ряда недостатков. Их использование позволяет

- сократить длину программы за счет включения команд, решающих стандартные задачи, решение которых требует десятки или сотни машинных команд;
- сократить количество ошибок и облегчить их поиск за счет использования выразительных синтаксических и семантических средств, приближающих язык к пониманию человеком и языку решаемых задач;
- создавать программы, не зависящие от машинных и системных конструкций, что позволяет исполнять их на различных компьютерах (без изменений или после некоторой адаптации).

.

Даже программу, написанную на ассемблере, а тем более написанную на языке программирования высокого уровня (ЯВУ), следует перевести в машинный код, то есть из **исходного кода** — текстового файла — получить **исполняемый файл** в машинном коде. Это осуществляется с помощью специальной программы, **компилятора**, являющимся разновидностью **транслятора**. В силу ряда обстоятельств в ходе компиляции создается не только исполняемый файл, но и **объектный модуль** — промежуточный формат, в некотором смысле близкий к исполняемому, но не являющийся таковым.

Считается, что "платой" за использование языков программирования высокого уровня является увеличение размера и уменьшение скорости работы программ. Однако в настоящее время это не совсем верно, т.к. перевод программы с языка высокого уровня в машинные коды включает в себя оптимизацию ("улучшение") кода и автоматизированный учет широкого спектра

возможностей и особенностей современных процессоров, который затруднителен при написании машинных кодов человеком. В частности язык программирования Си является инструментом создания одних из самых быстро работающих программ.

Следует отметить, что язык программирования и компилятор языка программирования — это разные вещи. Язык программирования представляет собой математическое описание структур и возможностей, теоретически по этому описанию каждый может создать компилятор, поддерживающий данный язык. Поэтому для одного и того же языка, в частности языка Си, разработано большое количество компиляторов.

На практике компиляторы различных производителей не являются тождественными между собой. Это относится не только к тому, что они могут быть разного качества — иметь разную скорость работы, по разному оптимизировать код, по разному сообщать об ошибках в исходном коде и т.п. Компиляторы могут расширять язык дополнительными возможностями, по разному обрабатываться некоторые ошибочные ситуации и т.п. Более того, сам язык Си не является 100% стабильным: его **стандарт** модифицировался несколько раз и не все компиляторы поддерживают все стандартны. В связи с этим, с одной стороны есть заинтересованность промышленного программирования использовать все возможности современных стандартов и компиляторов для повышения эффективности процесса создания программ, с другой — есть желание создавать переносимые программы, не привязанные к компилятору, модели компьютера, операционной системе и т.п., поэтому если нет уверенности в том, что программа всегда будет переводиться в машинный код с помощью определенного компилятора, следует использовать самый базовый вариант языка, который будет строго обрабатываться любыми компиляторами.

Программы, функции и библиотеки

Любая программа есть инструмент обработки некоторых **данных**, преобразования одних данных в другие. Процесс исполнения программы — обработка конкретных данных. Соответственно, для программы существуют **входные** и **выходные** данные, то есть те данные, которые необходимо обработать и те данные, которые получаются в результате обработки. Теоретически и входные, и выходные данные могут отсутствовать. В первом случае программа при каждом запуске будет выдавать одинаковый результат, фактически зависящий от данных, жестко включенных в сам код программы — пользователь не сможет как-либо модифицировать ее работу, но сам результат может быть осмысленным. Во втором случае программа не выполнит никакой полезной работы. Следует отметить, что для программы входные и выходные данные могут не только поступать с устройств ввода (клавиатура, мышь) и выводиться на устройства вывода (монитор, принтер) — то есть вводиться и получаться пользователем — а еще и передаваться в другие программы, записываться на носители информации (в файлы), и отправляться на другие устройства (в том числе другие компьютеры, например, по сети).

Одним из ключевых инструментов программирования вообще, и программирования на языках высокого уровня в частности, особенно на языке Си, является **функция**. Ее также можно назвать подпрограммой (в некоторых языках разделяют два вида подпрограмм — процедуры и функции — но в Си такого деления не предусмотрено, поэтому данные слова можно использовать как синонимы). Функция представляет собой программную реализацию некоторой задачи, малую программу, обрабатывающую входные данные и выдающую выходные некоторым конкретизированным образом. Например, функция может вычислять квадратный корень, собственно вводить и выводить данные особым образом, изменять состояние программы и системы, и т.д. и т.п. Входные и выходные данные функции могут быть как реально входными и выходными данными, вводимыми пользователем и выводимыми для нее, так и передаваемые в другие функции.

Готовые к использованию (т.е. уже написанные) функции объединяют в **библиотеки функ-**

ций, представляющие собой исполняемый файл особого рода. Библиотеки функций имеются у операционных систем и компиляторов языков программирования высокого уровня, а также поставляются как отдельный программный продукт или часть программного продукта. Стандартная библиотека языка программирования высокого уровня позволяет с помощью вызова функций языка, осуществляемого обычно за одну инструкцию программы, решить за "одно действие" задачу, требующую исполнения значительного объема машинных инструкций. В частности, функции ввода и вывода данных не являются элементарными машинными командами, в языке Си это функции стандартной библиотеки языка Си. По сути программа состоит из функций, вызывающих друг друга все глубже и глубже, пока в конце концов не встретится вызов машинных инструкций.

Кроме функций в языки программирования высокого уровня включаются различные выразительные средства, за использованием которых также "скрывается" вызов ряда машинных команд, но использование которых проще и понятнее человеку. Такие средства могут быть различны в языках программирования разных типов. В частности, язык Си является языком **структурного программирования**, поэтому в нем имеется ряд элементов, позволяющих правильно организовать структуру программы — порядок исполнения кода в зависимости от условий.

Подобные вещи в программировании объединяют под общим термином **абстракция**, подразумевающим уход от элементарных конструкций и машинной реализации к более сложным конструкциям и понятиям, близким к человеческому пониманию и терминологии решаемых прикладных задач.

Первая часть данного пособия посвящена основам программирования на языке Си и изучению структурного программирования, процедурного программирования (т.е. разделения программы на подпрограммы, на функции) и модульного программирования (т.е. разделение программы на части — модули, представляющие собой объединение функций, решающих определенный класс задач).

Hello, World! на языке Си

Традиционно изучение языка программирования начинается с рассмотрения программы, выводящей сообщение *Hello, World!* ("Привет, Мир!"). На языке Си данная программа может выглядеть следующим образом.

```
1 #include <stdio.h>
2
3 int main (void)
4 {
5     printf ("Hello,␣World!\n");
6
7     return 0;
8 }
```

Это далеко не единственный способ реализовать на языке Си вывод такого сообщения на экран, но именно данный вариант считается классическим и полезным для дальнейшего изучения. Не вдаваясь в фундаментальные детали, которые поясняются в теоретическом курсе, неформально и простым языком поясним каждую строчку данной программы.

Строка 1 содержит команду, позволяющую сделать возможным *вызов ряда функций стандартной библиотеки*, в том числе корректно использовать инструкцию строки 5.

`#include` — команда подключить нужный файл, файл `stdio.h` отвечает за функции ввода и вывода. Как правило такие инструкции должны следовать в начале программы.

Следует обратить внимание на то обстоятельство, что если опустить данную строку, то весьма вероятно, что программа будет работать и во многих случаях даже работать правильно, что создает ложное впечатление ненужности данной строки. Однако это далеко не всегда так, та *ошибочная программа, которая сработала правильно на одном компиляторе или при одном запуске, может не сработать на другом или при другом запуске*.

Строка 2 оставлена пустой в качестве разделителя: выше нее подключение библиотек, ниже — программа.

Строка 3 содержит создание *главной функции* программы. Это такая функция, которой передается управление при запуске программы. Суть строки в следующем: `int` — *тип данных* (о них речь пойдет ниже) выходного параметра функции, слово `main` — обязательное имя главной функции, `void` внутри круглых скобок — отсутствие входных данных функции.

Почему главная функция имеет возвращаемое значение, описывается в пояснении к строке 7.

Строки 4–8 содержат **тело** функции `main` — те команды, которые будут исполнены при вызове функции.

Строки 4 и 8 представляют собой **операторные скобки**. Вообще говоря, тело функции представляет собой набор инструкций, **операторов**, исполняемых последовательно, друг за другом. Операторные скобки в данном случае ограничивают тело функции, хотя формально они "превращают" набор операторов в один составной оператор, поэтому синтаксически тело функции есть один **составной оператор**. Упрощенно говоря, операторы внутри тела функции разделяются точкой с запятой.

Строка 5 содержит вызов функции `printf`, выводящей на экран строку. Выводимая строка является **аргументом** функции, аргументы функции записываются в круглых скобках. Строка — строковые данные — записываются в двойных кавычках. Особый набор символов '`\n`' представляет собой команду перевода строки, чтобы следующий вывод данных, в том числе системных, осуществлялся с новой строки, а не в продолжении данной. В Си вызов функции является самостоятельным оператором, точка с запятой завершает его.

Строка 6 оставлена пустой в качестве разделителя: выше нее вывод данных, ниже — служебное действие.

Строка 7 содержит оператор возврата значения из функции `main`. Как требует стандарт Си это должно быть целое число, что подчеркивается типом `int` возвращаемого значения в строке 3. Хотя на первый взгляд для функции `main` имеют смысл только те выходные данные, что передаются пользователю, никакая функция программы `main` не вызывает, это не совсем так. Дело в том, что у всякой программы есть **код возврата**, передаваемый в операционную систему. Именно этот код возврата передается в данной строке. Ноль означает, что программа завершена без ошибок. Ненулевое значение сообщало бы операционной системе (и, возможно, пользователю), что внутри программы что-то пошло не так, то есть возникла **ошибка времени исполнения программы** — не по причине того, что программа написана не правильно, а по причине того, что входные данные неверны или состояние компьютера не позволяет обработать их корректно: в данной программе, например, можно было бы провести проверку, успешно ли выведена строка на экран или произошел отказ оборудования: в первом случае возвращается 0, во втором — ненулевое значение, например 1.

Правилом хорошего тона программирования является указание в программе комментариев. Комментарии — участки текста программы, которые не являются исполняемыми, а представляют собой текст для автора и читателя программы. Например, де-факто обязательным является указание того, что представляет собой программа (название, цель создания, какую задачу и как программа решает и т.п.), а также кто ее автор. Комментарии в Си записываются между комбинациями символов '`/*`' и '`*/`'. Комментарии могут быть частью строки, в одну или в несколько строк.

```
/* Sample C Hello, World! programm
 * for C programming workshop course
 * (C) John J. Smith
 */

#include <stdio.h>

/* main types "Hello, World!" */

int main (void)
{
    printf ("Hello,␣World!\n");

    return 0;
}
```

Многострочные текстовые комментарии следует начинать с символа `*`, чтобы отличать из от "закомментированного кода" — удобного инструмента временного исключения некоторых строк кода из программы с целью поиска ошибок.

Отметим еще одно важное обстоятельство. Си является языком свободного переноса строк. Вообще говоря, только инструкцию `#include` критично записать в отдельной строке, поэтому такие программы как

```
1 #include <stdio.h>
2 int main (void){printf("Hello,␣World!\n");return 0;}
```

или

```
1 #include <stdio.h>
2 int
3 main
4 (void)
5 {printf
6 ("Hello,␣World!\n");
7 return 0;}
```

будут эквиваленты исходной, но сразу видно, что человеку их трудно читать, воспринимать, исправлять и создавать. В связи с этим следует придерживаться ряда правил форматирования и оформления исходных текстов программ.

- Каждый оператор (инструкцию) следует записывать в отдельную строку.
- Следует использовать пробелы и отступы, например, внутри операторных скобок следует делать отступ в 4 пробела.
- Следует использовать столбцы. В данном случае операторные скобки указаны друг под другом, хотя такой вариант

```
1 #include <stdio.h>
2
3 int main (void) {
4     printf("Hello,␣World!\n");
5
6     return 0;
7 }
```

тоже допустим.

- Следует разделять участки кода программы пустыми строками по смыслу (по аналогии с абзацами текста). В данном примере пустой строкой отделяется блок подключения файлов библиотек от кода функции, а внутри функции отделен блок вывода и оператор возврата значения.

Создание программ на ЯВУ

Процесс создания программ языках программирования высокого уровня может быть организован различными способами. Все они включают несколько этапов:

- написание исходного кода (текста) программы;
- попытки компиляции и исправление ошибок или предупреждений, выявляемых компилятором. **Ошибки компиляции** — ошибки в тексте программы, препятствующие ее "пониманию" компилятором. Они могут возникать по причине несоответствия текста программы синтаксическим правилам языка, использовании несуществующих (недоступных) функций или переменных, использовании несоответствующего типа данных или иным причинам, делающим невозможным перевод программы в машинный код. **Предупреждения компилятора** сообщают о потенциальных или реальных **ошибках программирования**, когда программа может быть переведена в машинный код, но может работать неверно. Конечно, ни один компилятор не сможет выявить все такие ошибки, как правило выявляются наиболее основные из них, связанные с семантически неправильным использованием языковых конструкций (примеры подобных ситуаций есть в пособии), а не с использованием неверных расчетных формул или неверного процесса обработки данных;
- тестирование (пробные запуски) и отладку, то есть нахождение и исправление смысловых ошибок, приводящих к неверному результату работы программы;
- документирование и публикация.

Минимальный набор инструментов программирование включает в себя написание и правку исходного кода программы с помощью любого текстового редактора (не путать с текстовым процессором, который использует файл в особом двоичном формате!), вызов программы-компилятора, программы-отладчика и самой программы посредством командной строки.

Существуют также специальные программы,т.н. **интегрированные среды разработки** (IDE, Integrated Desktop Environment), представляющие собой комбинацию средств текстового редактора, вызова компилятора и визуализации процесса отладки. В отличие от языков программирования, такие программы, как компиляторы, отладчики и интегрированные среды разработки быстроменяющимися продуктами, поэтому их детальное описание в настоящем пособии нецелесообразно. Укажем ключевые принципы, которыми следует руководствоваться при решении лабораторных работ.

1. В IDE следует создавать т.н. “проект” (может также использоваться название “решение” или иное). Недостаточно просто создать Си-файл! Если в начале обучения программы просты и могут состоять из одного файла, то в более сложных задачах создаются многофайловые программы, а IDE может обработать их только с помощью проектов. Кроме того, функционал IDE при работе с одним Си-файлом может быть ограничен.
2. В проект действительно следует включать хотя бы один файл исходного кода программы (как правило, это происходит автоматически). Это файл должен иметь расширение `.c`.
3. Для большинства задач курса следует использовать тип проекта “консольное приложение” (Console application). В базовом курсе изучается создание программ, работающих в текстовом режиме, в текстовом терминале, с консольным вводом и выводом, в противоположность более сложным графическим приложениям.
4. Как правило, IDE для каждого проекта создают две конфигурации компилятора — отладочную (Debug) и публикационную (Release). Они отличаются тем, то в первом случае возможен пошаговый анализ программы, а во втором созданная программа будет работать быстрее благодаря подключению оптимизации в настройках компилятора. Каждую из этих конфигураций следует использовать по назначению.
5. **Полезно проследить, чтобы в настройках компилятора были подключены все предупреждения. Язык Си допускает большую свободу синтаксиса, которая может привести к труднообнаружимым семантическим ошибкам. Значительная часть таких ошибок может быть выявлена с помощью предупреждений.**
6. Крайне желательно также настроить компилятор на строгое следование стандарту языка Си, C89 или C90, в некоторых случаях можно использовать C99 или более новый, но не все компиляторы его поддерживают.
7. Следует учесть, что IDE не будет перекомпилировать программу, если ее исходный код не был изменен. Это делается для ускорения процесса компиляции больших многофайловых проектов (т.н. “сборка”, Build) Однако, в некоторых случаях перекомпиляция необходима. Например, если программа не содержала синтаксических ошибок, но компилятор выдал предупреждения, то исполняемый файл — скорее всего неверно работающий — будет получен все равно. Если предупреждения не исправить сразу, то они исчезнут из “поля зрения” программиста, а при следующей попытке компиляции они не будут выведены вновь — ведь исполняемый файл уже получен и программа не будет перекомпилироваться. Обычно в IDE присутствуют возможности сборки проекта, учитывающие наличие изменений, и пересборки, проводящую полную компиляцию в любом случае.

   Поэтому крайне важно исправлять выданные предупреждения незамедлительно.

Примеры простейших программ

Рассмотрим следующую программу.

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2020
3   * Входные данные: два целых числа
4   * Выходные данные: сумма введенных чисел
5   */
6
7  #include <stdio.h>
8
9  int main (void)
```

```
10 {
11     int n1, n2, sum;
12
13     printf ("Введите␣два␣целых␣числа:␣");
14     scanf ("%d%d", &n1, &n2);
15
16     sum = n1 + n2;
17
18     printf ("Сумма␣введенных␣чисел␣равна␣%d\n", sum);
19
20     return 0;
21 }
```

Как видно из описания, программа запрашивает на вход два целых числа и выводит их сумму. Прежде всего следует обратить внимание на **строку 11**: в этой строке **объявляются переменные**. Объявление переменной сообщает компилятору, что такое-то имя (**идентификатор**) будет соответствовать переменной такого-то типа, то есть компилятор

- резервирует ячейку оперативной памяти необходимого размера, которую связывает с данным именем;
- при работе с этой ячейкой памяти выбирает машинные команды, кодирующие и обрабатывающие данные соответствующим типу образом.

Конкретно в этой строке сказано, что в программе заводятся три переменных — `n1`, `n2` и `sum`, в которых будут храниться целые числа: это определяется ключевым словом `int` перед именем переменной.

Переменную можно понимать как поименованную область (**ячейку**) памяти, способ обратиться к данным, находящимся в памяти, по понятному программисту, говорящему, имени, а не безликому и часто неизвестному в момент написания программы цифровому адресу. Переменная в программировании имеет и схожее с переменной в математике значение: это некоторый символ (слово), которому сопоставляется значение (данные), причем это значение может быть изменено в процессе работы программы.

Все данные, с которыми работает программа, написанная на языке высокого уровня, различаются по **типам**. С точки зрения хранения в памяти компьютера все данные представляют собой двоичные числа, от типа зависит, во-первых, размер одного экземпляра данных, во-вторых, способ их кодирования (формат их представления), в-третьих — способ их обработки. Например, представление и арифметические операции над целыми и вещественными числами производятся принципиально разным образом.

Строка 13 выводит сообщение на экран знакомым образом, **строка 14 вводит** значения переменных с клавиатуры. Здесь следует обратить внимание на три момента. Во-первых, ввод осуществляется с помощью функции `scanf`. Во-вторых, первым аргументом функции является особая строка, в которой указывается какие переменные и в каком формате вводятся. В данном случае вводятся 2 целочисленных переменных, поэтому для каждой из них указана комбинация `%d`. В-третьих, вторым и третьим аргументами являются вводимые переменные, но предваренные знаком `&`. Это связано с тем, что когда переменная используется без такого знака, то Си передает в функцию ее **значение**, а в данном случае функции `scanf` требуется не значение переменной, а адрес той ячейки памяти, куда нужно записать данные. Чтобы получить этот адрес используется знак `&`.

Строка 16 ответственна за собственно вычисление суммы. Здесь указывается, что в переменную `sum` следует записать результат вычисления арифметического выражения: знак `=` означает **присваивание**, то есть задание нового значения переменной, справа от знака при-

сваивания указывается присваиваемое значение, в данном случае сумма (+) значений двух переменных.

Строка 18 выводит результат на экран с помощью функции `printf`. Здесь внутри строки также используется комбинация `%d`, которая будет заменена на значение целочисленной переменной, являющейся очередным аргументом функции `printf`. Эта функция использует именно значение переменной, поэтому указание знака `&` не требуется.

Строки 12, **15**, **17** и **19** оставлены пустыми, чтобы визуально разделить участки кода, ответственные за разные процессы: объявление переменных, ввод данных, вычисление результата, вывод данных, завершение работы программы.

Строго говоря, без переменной `sum` и строки 16 можно было обойтись, программа с таким же функционалом имела бы вид

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2020
3   * Входные данные: два целых числа
4   * Выходные данные: сумма введенных чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     int n1, n2;
12
13     printf ("Введите␣два␣целых␣числа:␣");
14     scanf ("%d%d", &n1, &n2);
15
16     printf ("Сумма␣введенных␣чисел␣равна␣%d\n", n1 + n2);
17
18     return 0;
19 }
```

Такой подход укорачивает программу, снижает ее сложность как за счет количества строк, так и за счет количества переменных — хотя и не экономит память, как может показаться на первый взгляд, так как для хранения результата сложения все равно потребуется ячейка, и не улучшает быстродействие. С другой стороны, здесь вывод результата (часть, относящаяся к пользовательскому интерфейсу программы) и вычисления (рабочая часть) объединены, что, вообще говоря, является плохой практикой программирования, затрудняющей модификацию и сопровождение программ, о чем еще будет вестись речь в этом курсе.

Рассмотрим аналогичную программу, но работающую уже с вещественными числами.

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2020
3   * Входные данные: два вещественных числа
4   * Выходные данные: сумма введенных чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     double x1, x2, sum;
```

```
12
13      printf ("Введите␣два␣вещественных␣числа:␣");
14      scanf ("%lf%lf", &x1, &x2);
15
16      sum = x1 + x2;
17
18      printf ("Сумма␣введенных␣чисел␣равна␣%lf\n", sum);
19
20      return 0;
21  }
```

Программа претерпела ряд принципиальных изменений и одно изменение, связанное с удобством восприятия. В **строке 11** вместо `int` указано `double` — это объявление вещественной переменной. В **строках 14** и **18** вместо `%d` указывается `%lf`, что соответствует вещественному числу. Последнее связано с тем, что хотя Си предоставляет свободу выбора переменных программисту, имена должны быть удобными для восприятия. Кроме того, существуют некоторые традиции, согласно которым (как и в математике), переменные n, m и т.п. — целые, а x, y и т.п. — вещественные. В остальном код программы не отличается, поэтому для удобства чтения программы имена переменных заменены.

Рассмотрим другие арифметические действия.

```
1   /* Иллюстрация основных арифметических действий
2    * (C) Авторы пособия, 2020
3    * Входные данные: два целых числа
4    * Выходные данные: сумма, разность, произведение, отношение,
5    *                  неполное частное, остаток
6    */
7
8   #include <stdio.h>
9
10  int main ()
11  {
12      int n1, n2, sum, sub, mul, div, mod;
13      double frac;
14
15      printf ("Введите␣два␣целых␣числа:␣");
16      scanf ("%d%d", &n1, &n2);
17
18      sum = n1 + n2;
19      sub = n1 - n2;
20      mul = n1 * n2;
21      div = n1 / n2;
22      mod = n1 % n2;
23
24      frac = (double) n1 / n2;
25
26      printf ("Сумма␣введенных␣чисел␣равна␣%d\n"
27              "Разность␣введенных␣чисел␣равна␣%d\n"
28              "Произведение␣введенных␣чисел␣равно␣%d\n"
29              "Отношение␣введенных␣чисел␣равно␣%lf\n"
30              "Неполное␣частое␣введенных␣чисел␣равно␣%d\n"
31              "Остаток␣от␣деления␣чисел␣равен␣%d\n",
```

```
32              sum, sub, mul, frac, div, mod
33             );
34
35      return 0;
36  }
```

В **строках 18–22** проводятся арифметические действия над целыми числами: сложение (+), вычитание (-), умножение (*), нахождение неполного частного (/), нахождение остатка от деления (%). Для вещественных чисел арифметические операции выполняются аналогично, только вычисление остатка неприменимо. Однако получить вещественное число в результате прямого деления целых чисел невозможно. Вместо этого в **строке 24** сначала первое число делается вещественным, то есть *приводится к соответствующему типу*. Если хотя бы один из операндов арифметического действия — вещественное число, то и результат будет вещественным числом. В принципе, того же самого можно было достичь с помощью `frac = n1 / (double) n2;` или наивным образом `frac = 1.0 * n1 / n2;`.

На **строки 26–33** разорван вызов функции вывода результата на экран. На самом деле у этой функции по-прежнему первый параметр — единственная строка, она разделена на несколько только для удобства: следующие подряд строки Си объединяет в одну, остальные параметры — список выводимых значений. При этом в строках сделаны отступы для соблюдения правила столбцов, по этой же причине круглые скобки расположены одна под другой.

Константы, переменные, операции, выражения, операторы

Под **константами** в Си понимаются значения (данные), записанные непосредственно в код программы. Например, в строке

```
    printf ("1␣+␣2␣=␣%d", 1 + 2);
```

имеются 3 константы: число 1, число 2 и строка `"1 + 2 = %d"`. Заметим, что число 0 в строке

```
    return 0;
```

также является константой.

Переменные — поименованные ячейки памяти, хранящие некоторые данные, которые можно изменить в процессе работы программы.

```
    int n1, n2, sum;

    printf ("Введите␣два␣целых␣числа:␣");
        scanf ("%d%d", &n1, &n2);

    sum = n1 + n2;

    printf ("1␣+␣2␣=␣%d", sum);
```

Здесь используются целочисленные переменные `n1`, `n2` и `sum`. Изначально их значения не заданы: у `n1`, `n2` они вводятся с клавиатуры, значение `sum` вычисляется программой и задается с помощью присваивания.

Операции — функции особого рода. Они также представляют собой алгоритм, доступный по имени — **знаку операции** — у которого есть входные данные, называемые **операндами**, и выходные — значение операции.

Главное отличие функций и операций в том, что входные данные функций, называемые **аргументами** или **фактическими параметрами** записываются в скобках после имени функции

(в предыдущем примере это аргументы `scanf` и `printf`), а знак операции указывается между, перед или после операндов. В предыдущем примере операндами операции `+` являются переменные `n1` и `n2`.

Выражения — инструмент вычисления данных в программе. Всякая константа или переменная уже представляет собой выражение. Операция со своими операндами представляет собой выражение. Функция со своими аргументами представляет собой выражение. *В качестве аргументов функции и операндов операции могут выступать выражения.*

В строке

```
sum = n1 + n2;
```

имеется 5 выражений:

- `n1`, `n2` и `sum` — переменные;
- `n1 + n2` — операция сложения `+` со своими операндами `n1` и `n2` (числа, которые нужно сложить);
- `sum = n1 + n2` — операция присваивания `=` со своими операндами `sum` (переменная, куда нужно записать значение) и `n1 + n2` (выражение, значение которого нужно записать в указанную переменную); `sum = n1 + n2` — тоже выражение, его значение равно присвоенному в переменную `sum` значению, но здесь оно не используется.

Выражения в программировании по своей сути близки к математическим выражениям. На порядок вычисления выражений влияет **приоритет операций** и круглые скобки. Для арифметических операций приоритет естественен: умножение и деление выполняется раньше сложения и вычитания

```
int n1 = 2, n2 = 3, expr1, expr2;

expr1 = n1 + n2 * 2;
expr2 = (n1 + n2) * 2;
```

В переменную `expr1` будет записано 8, в переменную `expr2` — 10.

Присваивание имеет более низкий приоритет, чем арифметические операции, поэтому в переменные записывается значение именно того выражения, которое стоит справа от знака `=`.

Оператор — минимальная исполняемая инструкция языка. Функция представляет собой совокупность операторов, исполняемых последовательно, друг за другом. Условно можно считать, что операторы разделяются знаком точки с запятой `;`, хотя строго говоря это не так — на самом деле этот символ является составной частью оператора.

Синтаксически выражение еще не является оператором, но выражение, после которого поставлен знак точки с запятой `;` становится особым оператором-выражением: поэтому в наших примерах выражение присваивания и выражения вызовов функций `scanf` и `printf` становились исполняемыми инструкциями.

Другой оператор в наших примерах — `return`, после которого должно следовать выражение и точка с запятой. Он завершает работу функции и возвращает из нее значение указанного выражения.

Разумеется осмысленным оператором-вырежением является только такое выражение, у которого есть не только значение, но и которое выполняет некоторое действие, т.е. имеет **побочный эффект** — например, изменяет значение переменной, как вызов функции `scanf` или вызов операции присваивания, или выводит данные на экран, как вызов функции `printf`.

Поэтому пример оператора-выражения

```
n1 + n2;
```

синтаксически корректен с точки зрения языка, но содержит в себе явную семантическую ошибку — зачем вычислять выражение, которое ничего не меняет — ни в памяти компьютера, ни на устройствах ввода-вывода? Если компилятор настроен на соответствующие предупреждения, то он сообщит о том, что данное выражение не имеет побочных эффектов, хотя и создаст исполняемый код. Обратить на такое предупреждение важно, т.к. это позволит найти ошибку в коде программы — например, такая строка могла появиться из-за того, что программист забыл присвоить значение суммы переменной:

```
sum = n1 + n2;
```

Лабораторная работа №1

Знакомство с IDE языка Си.

Задание 1.1. *Добейтесь работоспособности выбранного IDE: запустите его на учебном компьютере, установите на домашнем компьютере (при наличии), установите компилятор и отладчик.*

Задание 1.2. *Создайте проект, разберитесь с настройками компилятора. Изучите процесс создания проекта.* При выполнении данной задачи следует иметь в виду, что каждый проект — исходный код программы. Следует использовать говорящие имена проектов, соответствующие названию программы, при этом название программы должно говорить само за себя, то есть из его названия должно прямо следовать, что программа делает.

Задание 1.3. *Реализуйте программу "Hello, World!", добейтесь, чтобы она компилировалась и исполнялась в отладочной и финальной конфигурациях. Изучите процесс компиляции, перекомпиляции и запуска программы средствами IDE. Изучите реакцию IDE на нулевой и ненулевой код возврата функции* `main`.

Задание 1.4. *Разберитесь, где в какой директории (папке) IDE создает файл проекта, где хранят исходные коды программы, где создаются объектные и исполняемые файлы.*

Задание 1.5. *Модифицируйте программу таким образом, чтобы сообщение "Привет, Мир!" выводилось на русском языке кириллическими символами.* При работе с некоторыми ОС, например, Windows, может возникнуть проблема: редактор IDE, скорее всего, будет работать в одной кодировке символов (Windows-1251), а консольный терминал — в другой (DOS-866), поэтому кириллические символы отобразятся некорректно. Необходимо либо настроить IDE на кодировку DOS-866, либо переключить терминал на кодировку Windows-1251 в начале запуска программы. Последнее делается с помощью добавления двух строк кода:

```
1 #include <stdio.h>
2 #include <locale.h>
3
4 int main()
5 {
6     setlocale(LC_CTYPE, "Russian");
7     printf("Привет,␣Мир!\n");
8     return 0;
9 }
```

2-я строка подключает необходимую библиотеку функций, 6-я — вызов функции по переключению настройки локали. Такой способ не универсален, т.к. использует системно-зависимые конструкции, и не совсем корректен, т.к. "подстраивает" систему под кодировку программы (например, это может привести к проблемам с другими программами, работающими в том же терминале, проблемой с кодировкой в заголовке терминала и т.д.), поэтому правильнее использовать соответствующую кодировку исходного кода программы.

Задание 1.6. *Запустите пример иллюстрации всех арифметических вычислений. Добейтесь его компиляции и исполнения в обоих конфигурациях. Запустите программу через отладчик. Изучите процесс пошагового исполнения программы. Изучите процесс наблюдения за изменением переменных в процессе отладки программы.*

§1.2. Некоторые тонкости

Типы данных

Основные типы данных в Си делятся на две группы: целочисленные и с плавающей точкой. Среди целочисленных выделяется символьный тип `char`, представляющий собой код одного символа. В памяти значение такого типа всегда занимает строго в 1 байт. В Си коды символов и сами символы с точки зрения типов данных неразличимы, различие определяется на стадии ввода-вывода.

Целочисленные типы различаются по длине и наличию или отсутствию знака. Базовый знаковый целочисленный тип обозначается `int`, беззнаковый — `unsigned int` или просто `unsigned`. Сколько именно байтов отводится под хранение данных этого типа в стандарте языка не конкретизируется, это зависит от архитектуры компьютера и компилятора. Типы другой длины — `short` (более короткий чем `int`, или равный ему), `long int` (более длинный, чем `int`, или равный ему) и (в новых стандартах языка) `long long int`. **Длина типа данных** — это количество байт, отводимое для представления данных этого типа. В случае с целыми числами это соответствует максимальному количеству двоичных разрядов (оно равно количеству байт, умноженному на 8, т.к. в одном байте как правило 8 бит), которое может быть использовано для записи числа. Таким образом, чем больше длина типа, тем бóльшее по значению целое число можно в нем представить.

Базовый тип для приближенного моделирования вещественных чисел — с помощью чисел с плавающей точкой — `double`, числа двойной точности. Также может использоваться одинарная точность `float` и повышенная точность — `long double`.

Подробнее о представлении данных речь пойдет в соответствующем параграфе.

Если нет причин использовать другие типы, то в качестве целого типа следует использовать `int`, в качестве вещественного — `double`, в качестве символьного — `char`.

Отдельного строкового типа нет, по сути строка представляет собой **массив** (набор) символов, условно данный тип можно обозначать как `char *` или `char []` или в некоторых случаях это `const char *`, подробное объяснение смысла данного обозначения выпадает за рамки данной темы.

Константы: жесткое кодирование данных

Один из способов задания данных в программу — указание их непосредственно в коде программы, жесткое кодирование. Такие данные в Си называются константами.

Второй способ — получение данных с помощью операций ввода с внешнего устройства.

В примерах данными, указанными непосредственно в коде программы, являются строковые константы, записываемые в двойных кавычках. Также можно указывать и числовые данные.

Например, программа вычисления суммы чисел могла бы выглядеть так

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2020
3   * Входные данные: нет
4   * Выходные данные: сумма двух чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     int n1 = 1, n2 = 2, sum;
12
13     sum = n1 + n2;
14
15     printf ("Сумма␣чисел␣равна␣%d\n", sum);
16
17     return 0;
18 }
```

или даже так

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2020
3   * Входные данные: нет
4   * Выходные данные: сумма двух чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     printf ("Сумма␣чисел␣равна␣%d\n", 1 + 2);
12
13     return 0;
14 }
```

В коде этих программ использованы 4 константы — строковая, числа 1 и 2, а также возвращаемое число 0.

В данном случае использование констант в качестве слагаемых, то есть основных входных (рабочих) данных программы не является удачной идеей: ясно, что редкая программа будет всегда применяться к одним и тем же данным, необходимо иметь возможность получать результат для разных входных данных, а если эти данные будут жестко прописаны в программу, то при смене данных программу придется изменять, что, как правило, как минимум неудобно для конечного пользователя. Однако, часто требуется явное задание вспомогательных данных в код программы (в виде констант) — это могут быть, например, выводимые строки, константы формул, возвращаемые значения и другие данные, по сути не являющимися входным для программы и алгоритма.

Тип данных констант определяется самими данными. Так, если данные содержат только цифры — то это целочисленные данные, при этом числа, начинающиеся с `0`, считаются записанными в восьмеричной системе счисления, с комбинации `0x` — в шестнадцатеричной, с любой

другой цифры — в десятичной. Например `12` — число 12, `-1` — число -1, `010` — число 8, `0xA` — число 10.

Если число содержит десятичную точку или порядок — комбинацию вида `e` / `E` и целое число (возможно со знаком) — то это считается числом с плавающей точкой. Запись вида m`E`p означает число $m \cdot 10^p$. Например, `1.0` — число 1, но не целое, `-1.5` — число -1.5, `1.2e-3` — число `0.0012`, `1.2E5` — число `120000` (вещественное), `1e0` — число 1, но тоже не целое.

Символьные данные записываются в одинарных кавычках (апострофах), например `'A'`, `'1'` — символ 1, но не целое число 1, код символа 1 может отличаться от числа 1 (и на практике это почти всегда так и есть).

Строковые данные записываются в кавычках, при этом строку данных можно разорвать на несколько заключенных в кавычки, компилятор их объединит. Это позволяет разбить строку данных на несколько строк программы. Например, следующие инструкции эквивалентны

```
printf ("Hello,␣World!\n");

printf ("Hello,␣"   "World!\n");

printf ("Hel" "lo,␣World!\n");

printf ("Hello,␣"
        "World!\n"
       );
```

Второй и третий пример являются примерами плохого кода, четвертый — достаточно хорошего, просто для столь коротких строк в таком разрыве нет необходимости. За исключением собственно содержимого строк и символов, где следует закрыть кавычку и апостроф до завершения строки программы, Си переносы строки приравнивает к пробелам, а множественные пробелы и переносы строки считает за один (это, конечно, не относится к пробелам внутри строковых констант).

Переменные: инициализация и присваивание значений

В процессе работы программы все данные хранятся в памяти. Данные, жестко записанные в код программы, хранятся вместе с кодом программы, для данных, вводимых с устройств ввода, и для хранения результатов вычислений, необходимо резервировать ячейки памяти. Делается это с помощью создания **переменных** — особых синтаксических объектов языка Си. Переменная характеризуется именем и типом данных, который хранится в соответствующей ячейке памяти.

Имя переменной выбирает программист исходя из того, что именно по смыслу хранится в данной переменой. В качестве имен переменных могут выступать последовательности букв латинского алфавита и цифр, но начинаться имя переменной должно с буквы. При этом буквы разного регистра считаются разными, поэтому `var`, `Var`, `VAR` и `vAr` — разные переменные. Компилятор автоматически определяет номер (адрес) ячейки памяти, где будут храниться данные задаваемой переменной. Чтобы это произошло необходимо переменную **объявить**. Делается это с помощью специальной инструкции, представляющей собой указание типа данных и перечисление имен объявляемых переменных через запятую. Завершатся объявление переменных одного типа точкой с запятой. Пример:

```
double x, y, z;

int i1, i2;
```

```
char c;
```

По стандарту Си объявление переменных может производится только в начале операторного блока, то есть сразу после открывающейся фигурной скобки, до списка операторов. Хотя объявления переменных похожи на операторы, строго говоря они таковыми не являются.

Перед использованием переменных их необходимо **инициализировать**, то есть задать значение. Не инициализированная переменная является источником **непредсказуемого** поведения, т.к. не гарантируется, что ее значение будет нулевое, во многих случаях это не так и на практике. *Очень важно следить, чтобы в программе не было использования значений не инициализированных переменных. Современные компиляторы имеют возможность подключить предупреждения о неинициализированных переменных, на него следует обязательно обращать внимание.*

Задание значения переменных осуществляется несколькими способами.

Первый — задать значения при объявлении переменных с помощью инициализатора. Он указывается после имени переменной в виде знака равно и задаваемого значения.

```
double x = 1.0, y = 0.0, z = 0.0;

int i1 = 256, i2 = 128;
```

Вообще говоря, в инициализаторе можно использовать и *выражения*, вычисляя значения переменных через уже инициализированные

```
double x = 1.0, y = 2.0, z = x + y;
```

Второй способ состоит в том, чтобы **присвоить** значение переменной во время работы программы. Так можно не только задать значение переменной, но и изменить его. Конструкция присваивания аналогична инициализатору, используется знак равенства, но формально они отличаются:

```
double x = 1.0, y = 2.0, z, r;

/* ... */

r = 5.0;
z = x + y;
```

Третий способ — ввод данных с устройств ввода, например с помощью `scanf`.

Следует отметить, что инициализировать и присваивать значения переменных можно не обязательно данными строго объявленного типа: целое число автоматически и однозначно приводится к числу с плавающей точкой, поэтому

```
double x = 1;
```

суть корректная конструкция.

При работе с переменными следует руководствоваться следующими соглашениями:

- Имена переменных должны говорить сами за себя, то есть соответствовать смыслу хранимых данных. Использовать однобуквенные переменные за редким исключением не рекомендуется (например, хранить значение координаты x в переменной `x` — хорошее решение, это говорящее имя, хотя и однобуквеннное, также для счетчиков принято использовать переменную i, для количеств и размеров может быть использована переменная n и т.д.).
- Для каждой переменной должно быть четко сформулировано (в комментарии или хотя бы в голове), что именно хранится в данной переменной.

- В Си принято использовать имена переменных, записанных строчными буквами. Следует придерживаться этого правила если нет веских причин сделать обратное.

Заметим, что неиспользуемая переменная не является смысловой ошибкой сама по себе, но компиляторы могут быть настроены на предупреждение об их наличии в программе: вероятнее всего, такую переменную следует убрать как излишнюю, но есть и вариант, что программист объявил ее с какой-то целью и "забыл" реализовать соответствующий код (например, коэффициент в формуле). Поэтому следует отслеживать такие предупреждения и устранять проблему соответствующим образом.

Форматированный ввод и вывод данных

Ввод и вывод осуществляется с помощью функций `printf` и `scanf`. Это функции *форматированного ввода и вывода*, так как они осуществляют вывод и вывод данных в определенном формате.

Следует отметить, что почти все функции в Си имеют фиксированное количество аргументов, каждый из которых должен быть определенного типа. Для `printf` и `scanf` это не так: только первый аргумент фиксирован, он обязателен и должен быть строкой, это т.н. **строка формата**. Остальные аргументы опциональны, их количество не детерминировано, они могут иметь любой тип. В контексте данной функции эти аргументы называются *полями*. То, как именно будут прочитаны и выведены поля, определяется содержимым строки формата. Для каждого поля в строке следует отнести комбинацию символа `%` и указания на то какого типа данных очередное поле и в каком формате будет выведено указанное поле.

В качестве полей функции `printf` могут выступать любые выражения (в т.ч. константы и переменные), именно значения выражений будут выведены на экран. Для вывода данных типа `int` поле `%d`, типа `unsigned int` — `u` для вывода в десятичной системе, `o`, `x` и `X` — для вывода в восьмеричной, шестнадцатеричной и шестнадцатеричной заглавными буквами соответственно.

Для вывода вещественных чисел можно использовать `%lf` — вывод в простом формате, `%lE` или `%le` — для вывода в экспоненциальной форме (выведется заглавная или строчная E соответственно) или `%lG` или `%lg` для выбора более оптимального варианта между `%lf` и `lE`/`le`.

Для вывода знака % следует указать `%%`.

Кроме задания собственно выводимого типа данных, после `%` можно регулировать формат выводимых данных.

Пример.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int m;
6      double b, c;
7
8      m = 7;
9
10     printf("m␣=␣%d\n",m);
11     printf("m␣=␣%3d\n",m);
12     printf("m␣=␣%03d\n",m);
13     printf("m␣=␣%+03d\n",m);
14
15     b = 1.23456789;
16     c = 1.23456789E5;
17
```

```
18      printf("b␣=␣%3.4lf,␣b␣=␣%3.4le,␣b␣=␣%3.3lE,␣b␣=␣%3.5lG␣c␣=␣%3.5lG␣\n", b, b, b, b, c);
19
20      return 0;
21  }
```

Выведет

```
m = 7
m =   7
m = 007
m = +07
b = 1.2346, b = 1.2346e+00, b = 1.235E+00, b = 1.2346 c = 1.2346E+05
```

В **строке 11** указано вывести не мене трех цифр, недостающие дополняются пробелами, в **строке 12** — нулями, а в **строке 13** дополнительно выводится знак + даже для положительных чисел. В **строке 18** для вещественных чисел комбинация из двух чисел через точку имеет следующий смысл: первое число по-прежнему число выводимых цифр, дополняемых пробелами или нулями, второе — число цифр, выводимых после точки (для `lf` и `le`/`lE`) или общее число значащих цифр (для `lg`/`lG`).

Форматы ввода для `scanf` аналогичны, хотя `X`, `lE`, `lG` не используются, а `lf`, `le`, `lg` имеют одинаковый смысл. Однако, у функции есть существенное различие при передаче полей, куда будет осуществляться запись введенных данных. Во-первых, это должны быть именно переменные — то есть ячейки памяти, куда можно записать значение. Во-вторых, в функцию нужно передавать не сами переменные, а адреса ячеек памяти, соответствующих переменным. Дело в том, что в Си параметры передаются **по значению**, то есть значения переменных или иных данных, указанных в качестве аргумента функции, копируется из области памяти, где хранятся данные вызывающей функции, в другую область памяти, где хранятся данные вызываемой функции. Для того, чтобы получить из переменной ее адрес, необходимо предварить символом `&`: адрес все равно будет передан по значению (как копия), но `scanf` сможет записать данные в ячейку памяти, по указанному адресу.

Следует обратить внимание, что вообще говоря стандарт Си оставляет на совести программиста проверку корректности указания длины и спецификатора, равно как и количества полей. Только современный компилятор, настроенный на вывод соответствующих предупреждений, сообщит что что-то сделано не так. Вообще говоря, даже в этом случае он обязан создать работающую программу. Однако, в случае `printf` результат вывода может оказаться неожиданным:

```
#include <stdio.h>

int main()
{
    int i = 1, j = -3;
    double a = 3.7, b = 10;

    printf ("%lf\n", i);
    printf ("%d␣%d␣%d\n", a);

    return 0;

}
```

может вывести что-то похожее на

```
0.000000
```

```
4195827 -338725056 2147483639
```

В первом случае так распозналась целочисленная переменная `i` как типа `double`, во втором — переменная `a` типа `double` "распалась" на две целочисленные (обычно `double` занимает вдвое больше байт, чем `int`), а в качестве третьего значения вообще вывелся т.н. "мусор", то есть какие-то значение, случайно оказавшиеся в памяти в соответствующем поиску поля функцией `printf` месте, при том, что само поле отсутствует.

Вообще говоря, результат работы такой программы **непредсказуем**.

Для `scanf` ситуация еще хуже: если `printf` просто выведет "странные" значения, то эта функция может *перезаписать* данные, которые не должны быть перезаписаны. Это могут быть как другие переменные, так и часть кода программы, что приведет к неверным результатам или краху ее работы в странный момент.

```
#include <stdio.h>

int main()
{
    int i = 1, j = -3;

    scanf ("%lf", &i);

    printf ("%d␣%d\n", i, j);

    return 0;
}
```

при попытке ввода 1.5 выведет что-то подобное (приводится в качестве примера, результат работы данной программы также непредсказуем)

```
1.5
0 1073217536
```

Здесь не только переменная `i` не ввелась корректно, еще и нарушилось значение переменной `j`.

Совсем неприятно, если забыть символ `&`: тогда значение переменной будет воспринято как адрес и в эту ячейку памяти будут записаны данные. Где будет эта ячейка памяти — непредсказуемо, это может быть одна из переменных программы, а может быть часть ее кода. В небезопасных операционных системах так может быть нарушена и целостность операционной системы. В самом неблагоприятном случае может оказаться, что введенные данные приведут к появлению в части кода программы верной, но нежелательной машинной инструкции: например, команды уничтожения всех данных на компьютере. Вероятность того, что это произойдет случайно ничтожно мала, но такая ошибка может быть использована со злым умыслом, для порчи или кражи данных.

Аналогичные неприятности могут возникнуть, если в функциях форматированного ввода и вывода указать больше полей в строке формата, чем передать значений или переменных для вывода и ввода: откуда и куда будут выводиться недостающие данные непредсказуемо. "Недостаток" указанных в формате полей к большим неприятностям не приведет, но свидетельствует о ошибке программирования: зачем указаны неиспользуемые поля в аргументах — они лишние или про них забыли?

При работе с функциями форматированного ввода и вывода следует соблюдать особую осторожность. *Важно получать и исправлять все предупреждения компилятора, касающиеся строки формата. Это крайне важно для* `printf` *и критично для* `scanf`.

Ввод и вывод строк является более сложной задачей, выпадающий за рамки данной части.

Операции: арифметика и присваивание

Арифметические операции в Си — `+` (сложение), `-` (вычитание), `*` (умножение), `/` (неполное частное или отношение), `%` (получение остатка от деления). Операции могут быть применены как к данным (константам) и переменным, так и к результатам других операций, что позволяет записывать составные арифметические выражения, причем Си поддерживает общепринятый порядок (приоритет) выполнения операций — умножение и деление (в том числе получение остатка) выполняется раньше сложения и вычитания. Регулировать приоритет можно с помощью круглых скобок. Например

```
r1 = (x1 + x2) * x3;
r2 = x1 + x2 * x3;
r3 = x1 + (x2 * x3);

t1 = 1 / 3 + 5;
t2 = 1.0 / 3 + 5;
```

Выражения, присваиваемые в `r2` и `r3` одинаковы, в `r1` — отличается. В `t1` запишется 5, в `t2` — приближенное представление $5\frac{1}{3}$.

Данные, к которым применяются операции, называются **операндами**.

Операция во многих смыслах похожа на вызов функции — входные данные (операнды) обрабатываются с помощью некоторой подпрограммы, в результате чего получаются выходные данные (результат операции). В Си в отличие от функций, у которых тип возвращаемого значения фиксирован, тип возвращаемого значения операции зависит от типа операндов. Для арифметических операций он будет соответствовать "наибольшему" из типов операндов (например, при сложении двух однотипных операндов тип результата будет соответствовать типу операнда, при сложении короткого и длинного целого результат будет иметь тип длинного целого, а при сложении `int` и `double` результат будет типа `double`). Именно по этому принципу получается, что деление двух целых чисел будет целым. Также при умножении двух чисел типа `int` может возникнуть переполнение даже если результат может быть записан в `long int`.

Вычисление выражения можно представлять себе как последовательный процесс замены операций с операндами на результат применения операций к операндам, начиная с самой **приоритетной** операции.

Следует также отметить, что *присваивание также является операцией*. Ее приоритет ниже почти всех других операций, поэтому она выполняется последней. Отличие присваивания от арифметических операций в том, что она не только вычисляет некоторое значение, но еще и *изменяет значение в ячейке памяти*, то есть состояние системы и программы. Такое явление называется **побочным эффектом операции**. Пример.

```
1 int n = 5, m = 2;
2 double d = 1.5, r;
3 /* ... */
4
5 r = m + ((n + 2) * 3) / d + 2.5;
```

В **строке 5** вычисление осуществляется, например, следующим образом:

```
1 r = m + ((n + 2) * 3) / d + 2.5;
2 r = m + (7 * 3) / d + 2.5;
3 r = m + 21 / d + 2.5;
4 r = m + 19.5 + 2.5;
5 r = 21.5 + 2.5;
6 r = 24.0;
```

```
7 /* здесь следует применение побочных эффектов */
```

Вначале вычисления идут согласно скобкам, в **строке 4** при отсутствие скобок вычисления идут слева направо, последним шагом происходит изменение значения переменной `r`. Присваиваемое выражение имеет тип `double`. Следует учесть, что все результаты промежуточных вычислений хранятся в некоторых ячейках памяти, их можно рассматривать как *неявные переменные*.

На самом деле присваивание не только изменяет значение переменной, но возвращает присвоенное значение. Причем если при отсутствии скобок арифметические операции выполняются слева направо, то операции присваивания — справа налево, поэтому такое присваивание корректно:

```
a = b = c = 1;
```

До сих пор рассматривались операции, имеющие *два операнда*. Такие операции называются **бинарными**. Существуют и операции, имеющие *один операнд* — **унарные**. Например, существуют унарный минус `-`, результат применения которого к числу (данным), записанному справа от него, есть изменение знака числа, унарный плюс `+`, не меняющий число, но часто удобный для симметризации записи.

Унарной операцией является и взятие адреса переменной — `&`, это операция с единственным операндом, переменной, адрес которой необходимо получить. Унарной операцией является и **приведение типа**, рассмотренное на примере получения вещественного отношения целых чисел: `(double) n1 / n2` — указание типа данных в скобках есть приведение типа.

Помимо традиционных арифметических операций в Си введены две особые операции **инкремента** `++` и **декремента** `--` — увеличения или уменьшения значения целочисленной переменной на единицу. Они могут применяться только к переменным и созданы для того, чтобы менять значения переменой *быстро* — как в плане скорости написания программы, так и в плане ее работы. Это унарные операции, но они присутствуют в двух вариантах — когда операнд записывается справа от операции (**префиксный инкремент** и **префиксный декремент**) и когда операнд записывается слева от знака операции (**постфиксный инкремент** и **постфиксный декремент**).

В плане *побочных эффектов* они выполняют одну и ту же работу — изменяют значение переменной на 1. Поэтому можно записать

```
int i = 1;

++i; /* теперь i = 2 */
i++; /* теперь i = 3 */

--i; /* теперь i = 2 */
i--; /* теперь i = 1 */
```

Различие между ними в том, какое значение *возвращает* данная операция: постфиксные возвращают старое значение переменной, префиксные — измененное. Поэтому

```
int i = 1, j;

j = ++i; /* теперь i = 2, j = 2 */
j = i++; /* теперь i = 3, j = 2 */
```

При этом префиксный инкремент и декремент — более быстрые в плане выполнения, они не требуют хранения предыдущего значения переменной, поэтому **всегда следует отдавать предпочтение префиксному варианту инкремента и декремента, если нет причины**

использовать постфиксный, то есть если предыдущее значение переменной не надо вернуть в выражение.

Для удобства программирования в Си введены операции **арифметического присваивания** — `+=`, `-=`, `*=`, `/=`, `%=`. Рассмотрим их на примере `+=`, остальные ведут себя аналогично. Левым операндом такой операции должна быть переменная, правым некоторые данные. Операция увеличивает значение переменной на значение правого операнда, то есть `a += b` то же самое, что `a = a + b`, но выполняется быстрее, так как происходит непосредственно в ячейке памяти `a`, без хранения промежуточного результата в неявной переменной и переноса его в нужную ячейку. Например,

```
int n = 1, m = 2;

n += 2;       /* Теперь n=3 */
m += n + 3;   /* Теперь m=8 */
```

Последнее действие выполняется именно таким образом, так как операции арифметического присваивания низкоприоритетны, то есть сначала будет применено сложение.

Следует отдавать предпочтение операциям арифметического присваивания, а не конструкциям вида `a = a + /*...*/`.

Операции арифметического присваивания возвращают присвоенное значение и выполняются справа налево, поэтому такая запись возможна.

```
int n = 2, m = 3;

n += m += 3;     /* сначала m = 6, затем n = 8 */
```

К сожалению, операции возвращают не переменную, поэтому запись вида

```
int n = 2, m = 3;

/* ошибка */ (n += m) += 3;     /* хотим сначала n = 5, затем n = 8, m неизменно */
```

является ошибочной в стандартном Си.

Неопределенное и непредсказуемое поведение

Синтаксис операций в Си представляется очень гибким и свободным, это позволяет писать громоздкие вычислительные конструкции. Однако такие конструкции не всегда работают корректно. С одной стороны это сделано для того, чтобы программисты не злоупотребляли этой свободой в ущерб читаемость кода, с другой потому, что Си — еще и язык, направленный на создание быстро работающих программ: некоторые вещи, касающиеся порядка и способа вычисления значения выражения не стандартизированы, создатели компиляторов имеют возможность самостоятельно выбрать более оптимальный путь.

Рассмотрим простой пример.

```
int i = 5, j;
j = ++i + ++i;
```

Чему равно `j`? Это зависит от того, как считать. Первый вариант, применяется левый инкремент, `i` становится равным 6, это значение возвращается, применяется правый инкремент, `i` становится 7, это значение возвращается `6+7` есть 13. Однако, реально скорее всего будет применен другой способ. Применяется один из инкрементов, это префиксный инкремент, это быстрейшая операция, она просто изменяет значение переменной `i`, т.е. в ячейке оказывается

6. Применяется второй инкремент, тоже меняет значение `i` — оно стало 7. Теперь складывается `i + i` — ответ 14.

Реально результат зависит от компилятора и его настроек.

Более того, вообще говоря Си не регламентирует порядок вычисления операндов:

```
int i = 5, j;
j = i++ + i;
```

Может быть сначала будет взято значение i (5, правый операнд), затем изменено значение `i` и взято предыдущее значение (5, левый операнд) — ответ 10? А может быть сначала будет взято значение `i` (5, левый операнд), а затем изменено и взято новое (6, правый операнд) — ответ 11?

Такое явление называется **неопределенным поведением**, под этим термином понимается то, что поведение программы не определено стандартом, результат определяется компилятором. Рассмотренное ранее *непредсказуемое* поведение связанное, например, с использованием неинициализированных переменных, не определяется даже компилятором и зависит от состояния компьютера в момент запуска программы.

Чтобы избежать неопределенного поведения необходимо следовать таким правилам.

- Если эксперимент показал, что исследуемый код работает именно так, это еще не значит, что он будет работать именно так всегда: на другой платформе, компиляторе, при других настройках он может дать другой результат. Только строгое следование требованиям стандарта, описанию возможностей языка, математическому описанию его синтаксиса и т.п. гарантирует то, что программа даст предсказуемый результат.
- Необходимо подключать соответствующие предупреждения компилятора и исправлять их.
- Следует исключить более чем однократное изменения значения одной и той же переменной в одном выражении (с помощью инкремента, арифметического присваивания, присваивания и т.п.).
- Следует исключить использование значения переменой в том же выражении, где ее значение изменяется. Это касается как операндов операций, так и аргументов функции, как одной и той же операции/функции, так и разных операций и функций, входящих в одно выражение. Единственный случай, когда это допускается — использование значения изменяемой переменной в правой части операции присваивания: в этом случае стандарт гарантирует, что изменение значения переменной произойдет после всех остальных вычислений.

Размер данных и переменных

В стандарте Си не декларируется **длина** для большинства основных типов данных, то есть количество байтов, отводимых для хранение переменной каждого типа. Гарантируется лишь, что тип `char` является строго однобайтовым. (Правда для `char` не гарантируется, является ли он знаковым или нет, для конкретизации нужно использовать `signed char` или `unsigned char`, впрочем, чаще этот тип используется не для вычислений, а для хранения кодов символов, знак которых не важен.)

Среди целочисленных типов выделяют “классический” (`int`), “короткий” (`short`), длинный (`long`), в некоторых компиляторах поддерживается и тип `long long`. В строке формата при форматированном вводе и выводе для них следует использовать формат поля `i`, `hi`, `hl`, `hL` и т.п. При этом вообще говоря часть или все эти типы могут совпадать по длине.

Базовым типом является `int`, так как его размер соответствует т.н. *машинному слову*, размеру единовременно обрабатываемых данных, размеру ячейки памяти, доступ к которой

осуществляется за один акт, значения типа `int` выравниваются по ячейкам памяти, доступ к которым осуществляется быстрее всего и т.п. Если нет явных причин использовать другие целочисленные типы, в вычислениях следует использовать `int`, хотя в целях экономии при хранении больших количеств небольших целочисленных значений использовать `short`, если нужно, наоборот, представит большие значения — `long` и т.п.

Среди типов, предназначенных для хранения вещественных чисел, различают стандартизированные типы одинарной точности (`float`, формат `f`), двойной точности (`double`, формат `lf`), и нестандартизированный формат повышенной точности (`long double`, формат `Lf`). В качестве базовых рекомендуется использовать вычисления с двойной точностью.

Математические функции

Как уже было отмечено, стандартной библиотекой языка Си предоставляется ряд функций. Для того, чтобы функции стали доступны в программе, необходимо подключить соответствующий заголовочный файл. При вызове функции указывается имя функции, затем в круглых скобках перечисляются ее **аргументы** через запятую.

Функция обычно также имеет **возвращаемое значение** — данные некоторого типа. В качестве аргументов функции могут выступать любые выражения соответствующего типа. Следует отметить, что и в качестве операнда может выступать выражение некоторого типа. Правда, если, как уже было отмечено, для одной и той же операции от типа операндов может зависеть как именно будет выполняться операция и какой будет тип результата операции, то в Си почти все функции имеют фиксированное количество и тип аргументов (исключение составляют функции неопределенного числа аргументов, такие как `printf` и `scanf`), а также тип возвращаемого значения (без исключений).

Таким образом, функции можно применять к результатам операции и наоборот, уровень вложенности выражений неограничен. Такой подход обеспечивает “естественность” выражений в Си — их аналогичность математическим выражениям, где используются числа, переменные, операции и функции.

Как и операции, функции с аргументами образуют выражения, представляют собой способ получения данных, при вычислении выражений функция с аргументами “заменяется” на результат применения некоторого алгоритма к аргументам.

Например, в файле `stdlib.h` (то есть для его использования в начале программы нужно указать

```
#include <stdlib.h>
```

определена функция `abs`, функция нахождения модуля (абсолютной величины). Функция имеет единственный аргумент типа `int` и возвращает число типа `int`.

В файле `math.h` определен ряд математических функций. *Замечание: в некоторых системах, например GNU/linux, для их использования недостаточно просто записать*

`#include <math.h>`,

может понадобиться подключить библиотеку `m` *в настройках компилятора.*

Функция	Кол-во и тип арг-в	Тип и смысл возвращаемого значения
`fabs`	1, `double`	`double`, абсолютная величина числа
`sin`	1, `double`	`double`, синус угла, заданного в радианах
`cos`	1, `double`	`double`, косинус угла, заданного в радианах
`tan`	1, `double`	`double`, тангенс угла, заданного в радианах

`exp`	1, `double`	`double`, экспонента
`log`	1, `double`	`double`, натуральный логарифм числа
`pow`	2, оба `double` (a, b)	`double`, возведение в степень a^b
`sqrt`	1, `double`	`double`, квадратный корень
`floor`	1, `double`	`double`, наибольшее целое число, не превосходящее аргумент
`round`	1, `double`	`double`, округление до ближайшего целого

Обратите внимание на разницу между функциями `abs` и `fabs`: одна работает с целыми числами, другая с вещественными. Отметим, что в Си существует **неявное** приведение типа, оно используется, если операнд операции или аргумент функции имеет тип, отличный от требуемого, но может быть приведен однозначно. Например, вызвать `fabs` для целого числа с получением вещественного результата можно без дополнительных ухищрений — например, `fabs(-3)` Си приведет -3 к -3.0, а функция вернет 3.0. Вызвать `abs` для вещественного числа в зависимости от компилятора или нельзя, или можно с неявным приведением — потерей дробной части, что может привести с неожиданным результатам: `abs(-1.5)` есть целое число 1.

Также следует обратить внимание, что такие функции как `floor` и `round` возвращают вещественное по формату, хотя и целое по содержанию число. Если нужно иметь именно целочисленное по типу данных значение, можно воспользоваться явным приведением:

```
int i;
double d;

/* ... */

i = (int) round (d);
```

Это далеко не полный перечень функций. В рамках данного (да и вообще практически любого) курса не ставится задача изучить *все функциям всех библиотек.*

Лабораторная работа №2

Знакомство с языком Си.

Задание 2.1. *Напишите следующие программы (данные вводятся с клавиатуры)*

(a) вычисление модуля (абсолютной величины) целого числа;

(b) вычисление модуля (абсолютной величины) вещественного числа;

(c) вычисление синуса и косинуса введенного угла в радианах;

(d) вычисление синуса и косинуса введенного угла в градусах.

С помощью строки формата добейтесь того, чтобы выводимый результат результат выглядел красиво и читабельно. Указание. Для получения числа *pi* можно использовать т.н. *символическую константу* M_PI, заданную в файле `math.h` как псевдоним для соответствующей явной константы.

Задание 2.2. *Напишите программу, позволяющую определить размер переменных различных типов и результатов операций с помощью операции sizeof.*

Задание 2.3. *Выполните ряд ошибочных примеров из текста.* Обратите внимание на предупреждения, выдаваемые компилятором.

(a) Неверное указание типа данных и количества полей при вводе и выводе.

(b) Неопределенное поведение.

(c) Использование неинициализированных переменных.

(d) Наличие неиспользуемых переменных.

(e) Использование библиотечной функции без подключения соответствующего файла.

Задание 2.4. *Даны следующие переменные*

```
double a = 2, b = 3, c = -1;
int i = 5, j = 2;
```

Определите тип и значение выражений.

(a) `a + b`

(b) `a + b - 1.0`

(c) `sin (a * M_PI) / j`

(d) `pow(i, j)`

(e) `i / j + c`

Прототип функции

Для описания функций в Си используются т.н. **прототипы** — особые языковые конструкции, указывающее имя функции, количество и тип аргументов, а также тип возвращаемого значения. Их назначение в языке рассмотрено ниже, здесь же следует акцентировать внимание на том, что прототип функции, снабженный информацией о том, что делает функция, какой смысл каждого из ее параметров, какой смысл возвращаемого значения, является достаточной информацией для того, чтобы данную функцию можно было использовать в программе.

Поэтому в описании (документации) к библиотекам информация о функциях предстает именно в таком виде. Например, описание функций `abs` и `fasb` может быть записано следующим образом:

Вычисление абсолютного значения целого числа.
`int abs (int n);`

Вычисление абсолютного значения числа с плавающей точкой.
`double abs (double d);`

А для двухаргументной функции возведения в степень документация может быть представлена подобным образом.

Вычисление `x` в степени `y`.
`double pow (double x, double y);`

Во всех случаях тип данных аргумента указывается перед его именем, тип возвращаемого значения — перед именем функции. Имя аргумента может быть опущено. Понимания прототипов и способе информирования о том, как работает функция — первый шаг к необходимому для программиста умению чтения документации с целью изучения работы библиотечных функций.

Основы представления числовых данных и связанные проблемы

Целые числа без знака как правило представляются в памяти в двоичной системе счисления естественным образом. Однако, тип данных вводит ограничение на максимальное количество

разрядов, а, значит, и наибольшее число, которое может быть записано в нем. Например, если тип данных `unsigned int` представляется с помощью 4 байт, т.е. 32 бит, то максимальное значение, которое может быть в нем записано $2^{32} - 1 = 4294967295$ (минимальное — 0).

Для знакового типа `int` диапазон возможных значений составляет от $-2^{31} = 2147483648$ до $2^{31} - 1 = 2147483647$ (асимметрия связана с особенностью кодирования отрицательных чисел и тем, что 0 фактически кодируется как положительное число).

Если результат вычисления превышает максимальное или ниже минимального значения происходит **переполнение**. Си никогда не проверяет переполнение: работа программы не останавливается, факт переполнения в программу не сообщается. При этом результат операции будет вообще говоря непредсказуемым: сложение двух положительных знаковых чисел может оказаться отрицательным, произведение может оказаться меньше сомножителей, а может и быть положительным и большим, но все равно неверным. Поэтому при составлении алгоритма и выборе исходных данных следует внимательно следить, чтобы они умещались в указанный тип.

Заметим, что приведенные минимальные и максимальные значения ориентировочные, реальные зависят от компилятора и компьютера, так как в разных системах используются разные длины целых типов данных. Для того, чтобы определить реальный размер типа данных в процессе исполнения программы можно использовать унарную префиксную операцию `sizeof`. Она может быть применена как к имени типа, так и к данным. Например,

```
char c;
int i;

printf("%lu␣%lu␣%lu␣%lu", sizeof c, sizeof i, sizeof double, sizeof 'A');
```

Результат операции `sizeof` имеет тип, который обозначается как `size_t`. Вообще говоря, такого типа в самом языке Си нет, он приравнен к одному из беззнаковых целочисленных типов (самому большому), поэтому вероятнее всего строка для него будет `lu` или `Lu`, что, однако, нестандартизировано. Во избежании недоразумений можно приводить результат к `int` и использовать формат `d`:

```
char c;
int i;

printf("%d␣%d␣%d␣%d", (int) sizeof c, (int) sizeof i,
                       (int) sizeof double, (int) sizeof 'A');
```

Заметим, что переполнение может возникнуть даже при использовании максимально длинного целого типа, а, например, при сложении двух чисел типа `int` результат может переполниться, даже если он умещается в `long int`, ведь результат сложения однотипных операндов соответствует типу операнда:

```
int n1 = 1000000, n2 = 1000000;
long int m1, m2;

m1 = n1 * n2            /* переполнение - результат умножения int */
m2 = (long int) n1 * n2 /* переполнения нет - результат умножения long int */
```

Тип переменной, в который присваивается результат никак не влияет на тип вычисляемого в правой части выражения.

Вещественные числа моделируются с помощью чисел с плавающей точкой. Это числа, представленные в виде

$$m \cdot 2^p$$

, где m и p — целые числа со знаком, называемые, соответственно, **мантисса** и **порядок**. Они представляются в виде целых двоичных чисел фиксированной длины.

Для простоты в десятичной системе счисления такое представление можно разобрать на следующих примерах. Пусть для мантиссы отводится 6 десятичных знаков, для порядка 2. Тогда $2/3 \approx 666667 \cdot 10^{-6}$, $\pi \approx 314159 \cdot 10^{-5}$, $1234567890 \approx 123456 \cdot 10^4$, $12/5 = 24 \cdot 10^{-1}$ точно (хотя в двоичном случае дробь будет бесконечной и представление приближенным).

Таким образом при работе с вещественными числами может возникнуть:

- **переполнение порядка** (например $1 \cdot 10^{90} \times 15 \cdot 10^{10}$ невозможно представить в рассматриваемом формате) — в этой ситуации число примет специальное значение **бесконечность** (положительная или отрицательная); такой же результат может быть получен при делении конечного конечного числа на 0.
- **потеря точности**: если произойдет переполнение мантиссы, то младшие разряды будут удалены, а порядок увеличен, т.о. потеряется точность представления числа; также это может произойти, например, при сложении чисел разных порядков: $151 \cdot 10^1 + 412 \cdot 10^{-3}$, то есть $1510 + 0.412$ окажется равным $151041 \cdot 10^{-1}$.
- **возникновение "не числа" (NAN)** — когда результат не может быть представлен в виде числа или бесконечности, например делении 0 на 0, сложении разнознакомых бесконечностей и т.п.

Заметим, что при делении целого числа на 0 работа программа останавливается с ошибкой, в то время как для чисел с плавающей точной программа будет работать в любом случае и даже выдавать в некотором смысле корректный результат. Еще одной специфичной для целых чисел проблемой является деление или умножение на -1 наименьшего целого числа, т.к. результат непредставим в данном типе в силу асимметрии, а, следовательно, непредсказуем.

Уточним также, что отрицательные целые числа как правило представляются с помощью т.н. обратного дополнительного двоичного кода. Например, чтобы закодировать число -5 в 2 байтах необходимо, во-первых, представить его в двоичном виде соответствующей длины: 0000000000000101, затем инвертировать все разряды 1111111111111010 (обратный код) и добавить 1: 1111111111111011, что и будет результатом представления. Аналогично -1 всегда представляется набором единиц, равным длине типа данных.

Сложение чисел с плавающей точкой неассоциативно. Хотя для вещественных чисел

$$(a + b) + c = a + (b + c),$$

если взять числа с плавающей точкой $a = 10^{20}$ (`a=1E20`), $b = -10^{20}$ (`b=-1E20`), $c = 1$ (`c=1`), то `a+b` есть 0, соответственно $(a + b) + c = 1$, а `b+c` есть снова `b`, то есть `-1E20`, так как на реальном компьютере мантисса скорее всего не содержит 20 десятичных знаков и прибавление $c = 10^0$ к числу 20-го порядка не повлияет него, соответственно $a + (b + c) = a + b = 0$. Этот эффект реально можно наблюдать при исполнении соответствующего кода программы. Поэтому *к числам, полученным в результате вычисления на компьютере следует относиться с осторожностью и учитывать точность вычислений.*

Лабораторная работа №3

Представление чисел в компьютере

Задание 3.1. *Предположим, что имеется некоторый целочисленный двухбайтовый тип. Каков результат следующих операций в данном типе:*

(a) $32000 \cdot 2$;

(b) $300 \cdot (-200)$;

(c) $65535 + 2$;

Рассчитать без компьютера отдельно для знакового и беззнакового типа. Учесть, что отрицательное число в беззнаковом типе превратится в некоторое положительное, слишком большое положительное в знаковом — в отрицательное. Для отрицательных чисел в знаковом типе использовать дополнительный код.

Задание 3.2. *Даны следующие числа (записаны в десятичной системе счисления)*

(a) 128

(b) 78

(c) 3245

Найти сумму их десятичных и шестнадцатеричных цифр. Обратить внимание, что сумма цифр представления любой системы счисления может быть записана в любой системе счисления: записать сумму десятичных цифр в шестнадцатеричной системе и наоборот.

Задание 3.3. *Рассмотрим гипотетический двухбайтовый формат чисел с плавающей точкой. Пусть 6 бит отведено под порядок, 10 — под мантиссу (целые числа со знаком в дополнительном коде). Записать, как будут кодироваться следующие числа (каждый из 16 бит).*

(a) 1;

(b) 32;

(c) -32;

(d) 100;

(e) 0.2;

(f) π.

Задание 3.4. *Определить результат вычисления в типе данных из предыдущей задачи*

(a) 1000+1;

(b) -12345+5.

Задание 3.5. *Придумать и отобразить в Си программе примеры переполнения (знакового и беззнакового) для целых чисел, потери точности и переполнения порядка, получения бесконечности и не числа для вещественных чисел с реальными типами данных языка Си. Написать соответствующие программы.*

§1.3. Создание и оформление программ

При создании программ следует руководствоваться рядом принципов, рекомендаций и соглашений по написанию *хорошего* кода. Под этим понимается не только эффективно и правильно работающий код: в первую очередь исходный код программы должен быть хорошо читаемым и воспринимаемым человеком, что в свою очередь позволяет минимизировать количество ошибок в нем.

Условно процесс создания программ состоит в следующем.

1. Прежде чем написать программу, следует четко определить, что конкретно она делает, какие данные она получает на вход, что выдает на выходе. Если программист не может что-либо сформулировать на естественном языке (например, на своем родном или на

рабочем языке фирмы), то программист не сможет это реализовать на языке программирования.

При этом рекомендуется использовать подход **черного ящика**, под которым понимается такое устройство, для которого *известно, что именно оно делает*, т.е. преобразует входные данные в выходные по известному, описанному правилу, но при этом для которого *не важно, как именно оно это делает*, то есть сам процесс преобразования данных не рассматривается.

Действительно, для *пользователя* программа представляет собой черный ящик — программа выполняет определенные, известные, документированные действия, пользователю известно как добиться выполнения этих действий, но не важно, как именно программа достигает результата.

В то же время задачей *программиста* является создание "внутренности" черного ящика, т.е. реализация выполнения требуемых действий. По факту, однако, программист также пользуется черными ящиками других программистов — в качестве таковых, например, выступают функции: благодаря прототипу и описанию известно, что они делают и как их использовать, но для их использования не важно как они это делают.

Таким образом, словесная формулировка задачи должна представлять собой описания внешней стороны черного ящика — выполняемых действий, входных и выходных данных.

Всякую программу следует начинать с комментария (как в смысле начала процесса создания программы, так и в смысле расположения комментария в начале программы), содержащего соответствующее описание, а также авторства кода.

2. Следующий шаг — определить как именно программа преобразует входные данные в выходные математическим образом, т.е. подготовить расчетные формулы и модели.
3. Далее следует определять компьютерный формат представления входных и выходных данных (количество и тип соответствующих переменных), а также алгоритмической (пошаговой) реализации математического преобразования, выявление промежуточных данных (количество и тип вспомогательных переменных).
4. Дальнейшая задача — разделение программы на части (участки, функции, модули). Об этом разделении речь пойдет в следующих главах. На данном этапе обучения при написании кода следует как минимум отделить пустыми строками интерфейсную часть (отвечающую за взаимодействие программы с пользователем) и рабочую часть (то есть собственно реализуемый алгоритм).
5. Далее следует собственно написание кода программы.

Всякая программа должна быть качественно оформлена. Программа пишется не только для компьютера, но и для человека, который будет ее читать — проверять, сопровождать, модифицировать. Даже если этим человеком будет лишь автор программы, через некоторое время есть высокий шанс забыть тонкости и детали создания кода и столкнуться с необходимостью восприятия своей программы как чужой. Поэтому программа должна быть написана хорошо читаемой.

Это также очень важно и для процесса отладки, поиска ошибок. Код программы следует читать и редактировать: как и в случае с литературным произведением, первичный код не более, чем черновик. При прочтении могут быть как выявлены незаметные на первый взгляд ошибки, так и улучшена производительность и читаемость кода.

- Программа должна быть снабжена заголовочным комментарием, поясняющим кто является автором программы и что эта программа делает, а также пояснением ключевых

нетривиальных мест кода. Комментарий должен быть предложением, а не обрывком фразы. Злоупотреблять комментарием, однако, не следует: например, нежелательно пояснять действия инструкций языка, т.к. предполагается, что читать знает язык Си.

- В коде программы следует грамотно переносить строки (одна инструкция — одна строка, или делать больше разрывов строк для наглядности). Строка должна целиком помещаться на экран (обычно это не более 60–70 символов, даже если современные мониторы позволяют отобразить более длинные строки, превышать это значение не следует, так как возможности восприятия человеком все равно ограничены).
- Следует отделять части кода, отвечающие за различные этапы работы программы, пустыми строками, а также следить за ровными столбцами везде, где это возможно. В частности, вложенные в фигурные скобки блоки кода (например, тело функции `main`) следует делать с отступом в 4 пробела (это т.н. "**правило четырех пробелов**"), во многих случаях столбцы (отступы) следует соблюдать при разрывах длинных строк кода, при кодировании однотипных выражений разной длины (выравнивание, например, по знаку операции присваивания, комментариям и т.п. — "**правило столбцов**"). Сравните два варианта оформления идентичной программы:

```
int k; /* Что делает k. */
unsigned long number; /* Опишите, что делает number. */
int l; /* Что делает l. */
k = 5; /* Здесь идет комментарий. */
numset(str); /* Здесь второй комментарий. */
l = k; /* А здесь третий. */
```

читается хуже, чем такой:

```
int k;                  /* Что делает k.                */
unsigned long number;   /* Опишите, что делает number.  */
int l;                  /* Что делает l.                */

k = 5;                  /* Здесь идет комментарий.      */
numset(str);            /* Здесь второй комментарий.    */
l = k;                  /* А здесь третий.              */
```

Во втором случае объявления переменных отделены от выполняемых действий пустой строкой ("**правило абзацев**"), а комментарии выровнены по вертикали.

- Все имена, вводимые программистом (переменных, программы) должны быть говорящими. Более того, следует руководствоваться **принципом наименьшей неожиданности**: если программа названа "синус", она не должна считать квадратный корень. То же самое для переменных: переменная `x` не должна хранить координату y, не следует называть переменную `hour`, если она будут использоваться для хранения номера года и т.д.

  От однобуквенных имен переменных следует отказываться за исключением случаев, когда одна буква действительно говорит сама за себя (например, если переменная `x` используется для хранения координаты x) или общепринятых сокращений и обозначений (`i` — счетчик, `n` — количество и т.п.) Следует полностью исключить использование хаотических имен переменных типа `aks2f`.

  При использовании имен переменных следует учитывать некоторые соглашения по типам данных.

  – переменные, чьи имена начинаются с `i`, `j`, `k`, `l`, `n`, `m` — целые;

– переменные, чьи имена начинаются с `a`, `b`, `c`, `d`, `e`, `f`, `x`, `y`, `p`, `q`, `r` — вещественные;
– переменные, чьи имена начинаются с `c` — символьные;
– переменные, чьи имена начинаются с `s` — строковые.
– переменные, чьи имена начинаются с `b` — логические.

Как видно, правила не жесткие (важнее использовать говорящие имена). Также следует учесть, что переменная `i` используется как счетчик, `s` может использоваться для целочисленного размера и т.п., но будет очень странно, если в программе встретится целочисленный `x` или вещественный `k`.

В Си для записи имен переменных принято использовать строчные буквы. В исключительных случаях возможно использование верхнего регистра, но это должно быть четко обосновано.

- Программа должна сообщать пользователю всю необходимую информацию, а именно: что требуется ввести, что именно выведено, если произошла ошибка — в чем она состоит как пользователь может ее исправить и т.п..
- Следует экономить машинные ресурсы: время и память. Если проблема экономии памяти на первых этапах обучения вряд ли актуальна (например, отказ от объявления переменной в пользу более сложного выражения, как уже было сказано, память не экономит), то задача экономии времени может встать довольно быстро: необходимо избегать повторяющихся вычислений, проверок условий и т.п.

§1.4. Линейные программы

Следующая лабораторная работа посвящена **линейным программам**, т.е. программам, не содержащим условий и циклов, последовательность исполнения инструкций которой не зависит от входных данных. Во всех задачах следует исключить использование условного оператора.

Лабораторная работа №4

Линейные программы

Задание 4.1. *Дано трехзначное целое число. В нем*

(a) зачеркнули первую слева цифру и приписали ее справа;

(b) зачеркнули первую цифру справа и приписали ее слева.

Вывести полученное число. Для получения цифр числа использовать вычисление остатка, а не логарифмирование и потенцирование.

Задание 4.2. *С начала суток прошло n секунд (n — целое). Найти количество минут и секунд, прошедших с начала последнего часа. Вывести в формате мм:сс.* (Например, $n = 11107$ — 3 часа, 5 минут, 7 секунд; вывод: 05:07).

Задание 4.3. *Даны координаты трех вершин треугольника:* $(x1, y1)$, $(x2, y2)$, $(x3, y3)$. *Найти его*

(a) периметр

(b) площадь.

Задание 4.4. *Даны вещественные положительные числа a, b, c. На прямоугольнике размера* $a \times b$ *размещено максимально возможное количество квадратов со стороной c*

(без наложений). Найти

(a) количество квадратов, размещенных на прямоугольнике;

(b) площадь незанятой части прямоугольника;

(c) долю занятой части (процент заполненности квадратами) прямоугольника.

Глава 2. Основы структурного программирования

§2.1. Условия и ветвления

Условные оператор и операции сравнения

Проверка условий является важным элементом программирования. Уже отмечалось, что функция представляет собой совокупность операторов, разделенных точкой с запятой. **Процесс исполнения функции есть процесс последовательного исполнения этих операторов.** Во многих случаев в зависимости о входных данных (точнее часто и в зависимости от результатов их предварительной обработки) нужно исполнить разные операторы или группы операторов.

Эту задачу решает условный оператор. У него две формы: полная

```
if ( /* выражение */ )
    /* оператор_1 */;
else
    /* оператор_2 */;
```

и неполная.

```
if ( /* выражение */ ) /* оператор */;
```

Суть оператора в следующем: выражение, записанное в круглых скобках, рассматривается как условие.

Если оно истинно, то полная форма оператора выполняет `оператор_1`, `оператор_2` игнорируется, если ложно — игнорируется `оператор_1`, выполняется `оператор_2`. Далее управление передается следующему за условным оператору.

Краткая форма выполняет единственный оператор-аргумент тогда и только тогда, когда условие истинно, по окончании работы в любом случае передает управление следующему за условным оператору.

Разумеется, оператор-аргумент может быть составным: *в Си там где можно оказать один оператор, можно указать и блок операторов, заключенный в операторные (фигурные) скобки.*

Условные выражения можно формировать, например, с помощью **операций сравнения чисел**: строго меньше (<), строго больше (>), меньше либо равно (<=), больше либо равно (>=), не равно (!=), равно (==). Если первые 4 операции достаточно естественны, а для понимания пятой лишь следует иметь в виду, что ! в Си есть символ отрицания, то на равенство нужно обратить особое внимание: *не следует путать равенство и присваивание.*

```
1  if (a < b) c = a + b;
2
3  if (x == y)
4      printf("Числа␣равны\n");
5  else
6      printf("Числа␣не␣равны\n");
7
8  if (n >= m)
9  {
10     x = pow (n, m);
11     y = pow (m, n);
```

```
12 }
13 else
14 {
15     x = pow (m, n);
16     y = pow (n, m);
17 }
```

Обратите внимание на следование *правилу четырех пробелов* для операторов, "вложенных" в `if`, а также на наличие точки с запятой в строке 4 и отсутствию ее в строке 12.

При использовании условного оператора следует учитывать ряд обстоятельств.

- Условные операторы могут быть вложенными. Обычно это работает нормально, но в некоторых случаях может вызывать неоднозначное прочтение: операторы следует не только корректно форматировать, но и указывать фигурные скобки, даже если оператор один. Компилятор может предупреждать о неоднозначности прочтения.

```
/* В этом примере else относится ко второму if
 * отступ вводит в заблуждение человека,
 * но игнорируется компилятором
 */
if (cond1)
    if (cond2)
        statement_2;
else
    statement_3;

/* Это тот же самый код,
 * только по другому корректно() форматирован
 */
if (cond1)
    if (cond2)
        statement_2;
    else
        statement_3;

/* А здесь else корректно отнесен
 * к первому оператору if
 */
if (cond1)
{
    if (cond2)
        statement_2;
}
else
    statement_3;
```

- Часто вложенные условные операторы можно оформлять в виде цепочки

`if...else if...else`.

```
/* Сложный для восприятия вариант */
if (a == b)
```

```
    printf ("a␣=␣b\n");
else
    if (a == c)
        printf ("a␣=␣c\n");
    else
        if (a == d)
            printf ("a␣=␣d\n");
        else
            printf ("Совпадений␣не␣найдено\n");

/* Тот же самый код, но форматированный по другому */
if (a == b)
    printf ("a␣=␣b\n");
else if (a == c)
    printf ("a␣=␣c\n");
else if (a == d)
    printf ("a␣=␣d\n");
else
    printf ("Совпадений␣не␣найдено\n");
```

- Внутри операторного блока — после открывающейся фигурной скобки — можно объявлять переменные. Только эти переменные будут видны лишь внутри данного операторного блока, после использовать их нельзя.
- Важная особенность Си: **в качестве условия может выступать выражение любого типа**. При этом истиной считается любое ненулевое значение, ложью только ноль. Поэтому **нельзя путать** присваивание и равенство. Запись типа `if (n = 2)` не только изменит значение переменной `n`, но и создаст тождественно верное условие — ведь присваивание возвращает присвоенное значение, а 2 есть истина. Современные компиляторы могут выдавать предупреждение о том, что присваивание используется как сравнение.

```
#include <stdio.h>

int main()
{
    int n;

    printf ("Введите␣целое␣число\n");
    scanf ("%d", &n);

    if (n = 2)                          /* тождественно истинное условие        */
        printf ("Число␣равно␣2\n");     /* для всех n будет выведена эта строка */
    else
        printf ("Число␣не␣равно␣2\n")   /* недостижимый код                     */

    return 0;
}
```

- В Си нет отдельного логического типа. Все операции сравнения возвращают значение типа `int`, либо 0 (ложь), либо 1 (истина). Тип `int` выбран потому, что он наиболее оптимален в смысле скорости доступа и выравнивания по блокам памяти.

- Последнее обстоятельство делает бессмысленным сравнение на не совпадение с нулем. "`if`(a != 0)"то же самое, что `if (a)`: **условный оператор как раз и проверяет, не является ли выражение нулем**.

  Первая запись создаст дополнительное действие — вычислить значение выражения `a != 0` (получится 0 или 1), а затем проверить, равно ли 0 полученное выражение, вместо того, чтобы просто проверить, равно ли само `a` нулю.

  Конечно, во многих случаях запись сравнения с нулем выглядит более понятной и наглядной, например, когда она соответствует сути алгоритма или формулы, то есть если надо действительно проверить на (не)равенство нулю некоторой математической величины. В таких случаях указание `!= 0` оправдано.

  Однако в других моментах, например, когда проверяется, что переменная `a` задана (имеет нетривиальное значение, определено, существует, верно) или когда проверяется результат, возвращаемый *логической функцией*, то есть функцией, которая сама по себе проверяет некоторое условие, то запись, подобная `if(cond)` ("если верно условие"), `if (is_ok(expr))` ("если выражение проходит проверку"), `if (p)` ("если `p` задано") и т.п. являются более соответствующие сути вещей.

- Сравнивать вещественные и целые числа между собой можно (хотя для вещественных чисел результат может оказаться некорректным из-за ошибок округления), а вот **результат сравнения знакового и беззнакового целого неопределен.** Это связано с тем, что компилятор не знает, приводить ли беззнаковое число к знаковому или наоборот. Действительно, если сравнивать -1 (знаковый) и 5 (беззнаковый) — кто из них больше? Как действовать — привести -1 к беззнаковому и получить самое большое число данного типа (например, в однобайтовом типе -1 знаковый есть 11111111, при прочтении как беззнакового получится 255, что больше 5) или 5 привести к знаковому (оно останется равным 5, что больше -1)?

  В данном случае корректный результат дал второй способ, но если сравнивать -1 и слишком больше, чтобы быть приведенным к знаковому, беззнаковое число, то правильным может оказаться первый способ или оба способа могут дать неверный результат. Компилятор не может предугадать, какой способ сработает для заранее неизвестных ему данных, но может предупредить о неопределенности сравнения знакового и беззнакового целого.

Операции сравнения и логические операции

Как уже было сказано, в Си имеется 6 операций сравнения. Это бинарные операции, в качестве операндов могут выступать значения числовых типов, возвращаемое значение — всегда типа `int` и всегда либо 0 (ложь) либо 1 (истина).

`<`	строго меньше
`<=`	меньше либо равно
`>`	строго больше
`>=`	больше либо равно
`==`	равно
`!=`	не равно

В качестве операндов, разумеется, могут выступать не только собственно числа (константы) и переменные, но и выражения.

Часто приходится проверять не одно условие, а одновременное выполнение нескольких условий или выполнение одного из условий и т.п. Для этого используются **логические операции**,

которые на вход берут операнды любых типов, в частности `int`, как результат операций сравнения, на выходе возвращают значение типа `int`, либо 0 (ложь), либо 1 (истина).

Операции следующие: логическое И (`&&`, истинно только когда оба операнда истинны), логическое ИЛИ (`||`, истинно когда хотя бы один из операндов — истина), логическое НЕ (`!`, отрицание, меняет истину на ложь и наоборот).

Не следует путать логические операции и побитовые (не рассматриваемые в пособии): использование `&` вместо `&&` и `|` вместо `||` часто некорректно, хотя во многих случаях визуально может работать правильно.

Значение логических операций в зависимости от значения операндов можно представить в виде *таблиц истинности*.

Операция `!` (унарная):

операнд	значение
0 (ложь)	1 (истина)
не 0 (истина)	0 (ложь)

Операция `&&` (бинарная):

левый операнд	правый операнд	значение
0 (ложь)	0 (ложь)	0 (ложь)
0 (ложь)	не 0 (истина)	0 (ложь)
не 0 (истина)	0 (ложь)	0 (ложь)
не 0 (истина)	не 0 (истина)	1 (истина)

Операция `||` (бинарная):

левый операнд	правый операнд	значение
0 (ложь)	0 (ложь)	0 (ложь)
0 (ложь)	не 0 (истина)	1 (истина)
не 0 (истина)	0 (ложь)	1 (истина)
не 0 (истина)	не 0 (истина)	1 (истина)

Следующий код проверки совпадения значения трех переменных корректен.

```
if (a == b && b == c)
    printf ("Числа␣равны\n");
else
    printf ("Числа␣не␣равны\n");
```

Этот код **не корректен**.

```
if (a == b == c)  /* Ошибка !!!! */
    printf ("Числа␣равны\n");
else
    printf ("Числа␣не␣равны\n");
```

Это верно записанная программа на Си, но она не проверяет равенство трех чисел: вместо этого проверяется равенство результата сравнения `a` и `b` — 0 или 1 — и `c`, то есть истиной будет комбинация $a = 2$, $b = 2$, $c = 1$ или $a = 2$, $b = 3$, $c = 0$.

Логические операции имеют более низкий приоритет, чем операции сравнения, поэтому дополнительные скобки в строке `if (a == b && b == c)` не требуются. В то же время `&&` имеет

более высокий приоритет, чем `||`, хотя в любом случае при комбинировании нескольких различных бинарных логических операций в одно условие рекомендуется использовать скобки во избежании ошибок.

Благодаря операции отрицания `if (a == 0)` можно записывать как `if (!a)`.

Тернарное условие

Часто проверить условие можно и без использования условного оператора. Для этого в Си есть тернарная **условная операция** `?:`. У нее три операнда: условие и два выражения, первое есть значение операции при истинности условия, второе — при ложности. Например, найти наибольшее из двух чисел можно таким образом:

```
if (a > b)
    max = a;
else
    max = b;
```

но более коротким будет код

```
max = a > b ? a : b;
```

а пример из предыдущего параграфа можно сократить до

```
    printf (a == b && b == c ? "Числа␣равны\n" : "Числа␣не␣равны\n");
```

Условная операция сокращает код и рекомендуется к использованию.

Оператор выбора

Хотя использование цепочки `if...else if...else` удобно, часто удобнее воспользоваться **оператором выбора**.

```
if (i == 1)
{
     /* случай i == 1 */
}
else if (i == 2)
{
     /* случай i == 2 */
}
else if (i == 3)
{
     /* случай i == 3 */
}
else if (i == 4)
{
    /* случай i == 4 */
}
else
{
    /* остальные случаи */
}
```

записывается как

```
switch ( /* выражение */ )
{
   /* объявления */

   case /* константа 1 */ :

      /* операторы, выполняемые если выражение == константа 1 */
      break;

   case /* константа 2 */ :

      /* операторы, выполняемые если выражение == константа 2 */
      break;

   /* ... */

   default :

      /* операторы, выполняемые если выражение не равно ни одной константе */
}
```

использование `break` чаще всего необходимо, так как иначе выполнение программы не прервется только потому, что встретился следующий помеченный оператор и будет исполняться до самого конца оператора-аргумента `switch` (или до первого `break` в другом `case`). Это используется, например, в случаях, когда двум выбираемым значениям соответствует одно действие (случай ИЛИ для множественных `if`):

```
if (i == 1)
{
     /* случай i == 1 */
}
else if (i == 2 || i == 3)
{
     /* случай i == 2 или 3 */
}
else if (i == 4 || i == 5 || i == 6)
{
     /* случай i == 4 или 5 или 6 */
}
else
{
    /* остальные случаи */
}
```

превращается в

```
switch (i)
{
    case 1:
            /* случай i == 1 */
            break;
    case 2:
    case 3:
```

```
            /* случай i == 2 или 3 */
            break;
    case 4:
    case 5:
    case 6:
            /* случай i == 4 или 5 или 6 */
            break;
    default:
            /* остальные случаи */
}
```

Оператор `switch` может генерировать более быстрый код, так как вместо проверки условий сразу передает управления нужному блоку, однако имеет свои ограничения: выражение должно быть целочисленным, а после `case` должна следовать константа (явные целочисленные данные).

Проверка корректности ввода

Использование `scanf` связано с определенным риском: пользователь может ввести данные, которые не могут быть преобразованы в соответствующий формат (ввести буквы вместо числа и т.п.). В связи с этим важно проверять, все ли числа были прочитаны корректно. *Функция* `scanf` *возвращает число фактически корректно прочитанных полей.* В случае неверного ввода можно, например, завершить программу с ошибкой. Например, для ввода двух чисел

```
1  /* Нахождение максимума двух чисел
2   * (C) Авторы пособия, 2020
3   * Входные данные: два целых числа
4   * Выходные данные: наибольшее из двух чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     int n1, n2, max;
12
13     printf ("Введите␣два␣целых␣числа:␣");
14     if (scanf ("%d%d", &n1, &n2) != 2)
15     {
16         printf ("Неверный␣ввод,␣следует␣ввести␣два␣целых␣числа\n");
17         return 1;
18     }
19
20     max = n1 > n2 ? n1 : n2;
21
22     printf ("Наибольшее␣из␣введенных␣чисел␣%d\n", max);
23
24     return 0;
25 }
```

В строке 17 оператор `return` не только возвращает значение, но и завершает работу функции `main`, а вместе с ней и всей программы, с кодом возврата 1, что сообщает операционной системе о том, что программа завершилась с ошибкой.

Упрощение логических выражений

Исходя из определения логических операций легко проверить прямым перебором всех возможных значений операндов (т.е. составив соответствующую таблицу истинности) следующие логические законы. Здесь логические операции приводятся в нотации Си, но под знаком равенства понимается математическое равенство значений выражения при всех возможных значениях операндов, в случае Си — соответствующего целого числа, 0 (ложь) или 1 (истина); A, B и C — некоторые выражения, интерпретируемые как логические.

- **законы коммутативности:**
  - (1) $A \texttt{||} B = B \texttt{||} A$;
  - (2) $B \texttt{\&\&} A = A \texttt{\&\&} B$;
- **законы ассоциативности:**
  - (3) $A \texttt{||} (B \texttt{||} C) = (A \texttt{||} B) \texttt{||} C$;
  - (4) $A \texttt{\&\&} (B \texttt{\&\&} C) = (A \texttt{\&\&} B) \texttt{\&\&} C$;
- **законы дистрибутивности:**
  - (5) $A \texttt{\&\&} (B \texttt{||} C) = (A \texttt{\&\&} B) \texttt{||} (A \texttt{\&\&} C)$;
  - (6) $A \texttt{||} (B \texttt{\&\&} C) = (A \texttt{||} B) \texttt{\&\&} (A \texttt{||} C)$;
- **законы де Моргана:**
  - (7) $\texttt{!}(A \texttt{||} B) = \texttt{!}\, A \texttt{\&\&} \texttt{!}\, B$;
  - (8) $\texttt{!}(A \texttt{\&\&} B) = \texttt{!}\, A \texttt{||} \texttt{!}\, B$;
- **закон двойного отрицания:**
  - (9) $\texttt{!}(\texttt{!}\, A) = A$;
- **законы поглощения:**
  - (10) $A \texttt{||} (A \texttt{\&\&} B) = A$;
  - (11) $A \texttt{\&\&} (A \texttt{||} B) = A$;
- **закон исключенного третьего:**
  - (12) $A \texttt{||} (\texttt{!}\, A) = 1$;
- **закон противоречия:**
  - (13) $A \texttt{\&\&} (\texttt{!}\, A) = 0$;
- **законы логических констант:**
  - (14) $1 \texttt{||} A = 1$;
  - (15) $0 \texttt{||} A = A$;
  - (16) $1 \texttt{\&\&} A = A$;
  - (17) $0 \texttt{\&\&} A = 0$;
- **законы логических операций:**
  - (18) $A \texttt{\&\&} A = A$;
  - (19) $A \texttt{||} A = A$;

Использование данных законов во многих случаях позволяет упростить логическое выражение, используемое в программе, или сделать его более эффективным с точки зрения количества проводимых компьютером вычислений.

Следует иметь в виду, что благодаря законам (14) и (17) в случае если левый операнд операции логического ИЛИ и И определен в не 0 и 0 соответственно, вычисление правого операнда не требуется, поэтому соответствующее выражение Си не вычисляется.

Лабораторная работа №5

Логическое выражения

Задание 5.1. *Даны следующие переменные*

```
double a = 2, b = 3, c = -1;
int i = 5, j = 2;
```

Определите тип и значение следующих выражений. Указание: отрицание имеет более высокий приоритет, чем арифметические операции и операции сравнения.

(a) `a + b < c`

(b) `a == i - 1 && j`

(c) `!(c == -1)`

(d) `!j`

(e) `a && b || c`

(f) `a == 1 || b == 3 && c == 5`

(g) `a && !c == 0`

Задание 5.2. *Используя логические законы, упростите выражения.* Указание. Отрицание строго неравенства есть обратно нестрогое неравенство и наоборот (по крайней мере это верно для целых чисел).

(a) `!(m < n && !(n > l))`

(b) `!(m == n || !(n > l) && n != 0)`

Особенности сравнения чисел с плавающей точкой

При вычислениях с плавающей точкой неизбежны **ошибки округления**, то есть потеря точности. Это делает проверку на равенство и неравенство вещественных чисел не вполне корректной операцией: математически равные числа могут отличаться, например, на последнюю цифру мантиссы, что сделает их неравными с точки зрения компьютера, или, наоборот, математически различные числа могут оказаться равны в силу ошибок округления.

Для разрешения данной проблемы можно использовать сравнение с некоторой точностью ε: считать два числа a и b равными, если $|a - b| < \varepsilon$, однако числа a и b в разных случаях могут быть разных порядков и использовать один и тот же ε для всех случаев некорректно. Например, при сравнении чисел первого десятка отличие в 0.1 может быть существенным, а для чисел $\sim 10^{100}$ разница в 10^{90} может оказаться на грани точности.

Поэтому, вместо упомянутой выше **абсолютной погрешности** лучше использовать **относительную погрешность**:

$$\left|\frac{a-b}{a}\right| < \varepsilon$$

или даже

$$\left|\frac{a-b}{\max(a,b)}\right| < \varepsilon$$

если порядок этих чисел тоже может отличаться существенно. Это не идеальное решение, но идеального решения просто не существует из-за того, что потерь точности в большинстве случаев не избежать. Данный подход решает проблему неассоциативности сложения в приведенном на стр. 2.4-е примере: если брать сравнение, например, с относительной погрешностью 10^{-10}, то результаты при обоих вариантах расстановки скобок равны.

Кроме того, при работе с типами с плавающей точкой в Си следует учитывать возможность, что результат операции окажется бесконечностью (INF) или не-числом (NaN).

Арифметические операции и операции сравнения чисел с плавающей точкой с положительной или отрицательной бесконечностью дают естественный результат: например, сумма и произведение бесконечности и конечного числа есть бесконечность, сумма разнознаковых бесконечностей — NAN, положительная бесконечность больше любого конечного числа и отрицательной бесконечности и т.д. В свою очередь арифметические операции с не-числом всегда дают NAN, сравнение с NAN всегда дают ложь, в том числе NAN не равно самому себе, только NAN != NAN есть истина.

Поэтому, для вещественных чисел `!(a<b)` не то же самое, что `a>=b`. Кроме того, код

```
if (a < b)
    printf ("a␣меньше␣b\n");
else if (a > b)
    printf ("a␣больше␣b\n");
else if (a == b)
    printf ("a␣равно␣b\n");
```

не перебирает все возможные ситуации, нужен еще случай

```
if (a < b)
    printf ("a␣меньше␣b\n");
else if (a > b)
    printf ("a␣больше␣b\n");
else if (a == b)
    printf ("a␣равно␣b\n");
else
    printf ("a␣или␣b␣-␣не␣число\n);
```

Бывает полезно сгенерировать не число или бесконечность. В новых стандартах Си существуют соответствующие константы (`NAN` и `INF`), в общем случае не-число получается как `0.0/0.0`, бесконечность — как `1.0/0.0`, однако проверка на то, что значение является не-числом осуществляется не с помощью `a==NAN` (это всегда ложь), а с помощью `a!=a` (это истинно если `a` — не-число).

Лабораторная работа №6

Ветвления и условия

Задание 6.1. *Дан номер года (положительное целое число). Определить количество дней в этом году.* Обычный год содержит 365 дней, високосный — 366 дней. Високосным считается год, делящийся на 4, за исключением тех годов, которые делятся на 100 и не делятся на 400 (например, годы 300, 1300 и 1900 не являются високосными, а 1200 и 2000 — являются).

Задание 6.2. *Известны год, номер месяца и день рождения каждого из двух человек. Определить, кто из них старше.*

Задание 6.3. *Вводится число от 1 до 99 — сумма в рублях. Вывести эту сумму строкой, с указанием количества рублей и правильным согласованием падежей.* (Например, 20 — "двадцать рублей", 32 — "тридцать два рубля", 41 — "сорок один рубль".) Указание. Использовать `switch`.

Задание 6.4. *Вводится число от 1 до 99 — возраст в годах. Вывести возраст строкой, с указанием количества лет, правильным согласованием падежей и слова год.* (Например, 20 — "двадцать лет", 32 — "тридцать два года", 41 — "сорок один год".) Указание. Использовать `switch`.

Задание 6.5. *Даны длины сторон двух треугольников. Определить подобны ли эти треугольники.* Указание: стороны могут вводиться в произвольном порядке, предполагается сортировка трех чисел с помощью `if`.

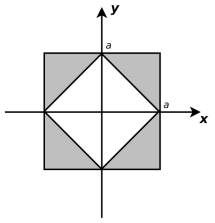
Задание 6.6. *Даны длины трех отрезков a, b, c. Определить вид треугольника, образован-*

ного отрезками такой длины: тупоугольный, прямоугольный, остроугольный, вырожденный в отрезок, несуществующий.

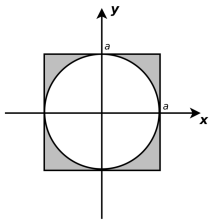
Задание 6.7. *Даны четыре вещественных числа — координаты противоположных углов прямоугольника* $(x1, y1)$, $(x2, y2)$, *стороны которого параллельны осям координат, и координаты пробной точки* (x, y). *Определить, находится ли точка внутри прямоугольника.* Учесть, что углы могут вводиться в любом порядке, неизвестно какой из них выше, какой левее.

Задание 6.8. *Даны области, определенные параметром a и точка с координатами* (x, y). *Определить, находится ли точка внутри затемненной обрасти.*

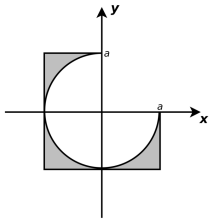
(a)



(b)



(c)



Задание 6.9. *Написать программу анализа квадратного уравнения* $ax^2 + bx + c = 0$. *Рассмотреть все возможные варианты: два вещественных корня, один кратный корень, два комплексных корня, линейное уравнение с одним корнем* ($a = 0$, $b \neq 0$), *уравнение, не имеющее корней* ($a = 0$, $b = 0$, $c \neq 0$), *любое вещественное число является корнем* ($a = b = c = 0$). *Вывести полученный случай и корни (если есть).* Указание. Исключить повторения вычислений (дискриминанта, корня из него и т.п.) и проверок одного и того же условия.

Замечание. В задачах 5–8 рассмотреть вариант, учитывающий, что ответ на поставленный вопрос может быть дан лишь с некоторой точностью. Использовать вводимую пользователем относительную погрешность.

§2.2. Циклы

В Си существует три оператора цикла — с предусловием, с постусловием и особый оператор `for`, представляющий с собой цикл с постусловием, хотя традиционно использующийся в качестве цикла со счетчиком.

Цикл с предусловием записывается как

```
while ( /* выражение */ ) /* оператор */;
```

с постусловием

```
do { /* список операторов */ } while ( /* выражение */ );
```

Разумеется, цикл с предусловием также может применяться к составному оператору

```
while ( /* выражение */ ) { /* список операторов */ }
```

А вот у цикла с постусловием фигурные скобки обязательны.

В качестве условия может выступать выражение любого типа данных. Тело цикла исполняется до тех пор, пока условие истинно (то есть значение выражения — не ноль).

Цикл с предусловием сначала проверяет условие, затем, если оно истинно, исполняет оператор. Цикл с постусловием сначала исполняет оператор, затем проверяет условие. Таким образом, цикл с постусловием приведет к выполнению тела цикла хотя бы один раз.

Оператор (в т.ч. составной оператор, группа операторов), выполняемый циклически, называется **телом цикла**.

Третий оператор цикла аналогичен по названию и своему предназначению циклам со счетчиком других языков программирования, однако имеет более широкий синтаксис:

```
for ( /* Инициализатор */ ; /* Условие */ ; /* Итератор */ ) /* Оператор */;
```

по сути является циклом с предусловием и с точностью до некоторых тонкостей эквивалентен следующей конструкции:

```
/* Инициализатор */ ;
while ( /* Условие */ )
{
    {
        /* Оператор */;
    }
    /* Итератор */;
}
```

В качестве инициализатора, условия и итератора могут выступать любые выражения. Цикл сначала вычисляет инициализатор, затем выполняет цикл с предусловием, вычисляя итератор после каждого прохода.

В качестве инициализатора, условия и итератора могут выступать любые выражения (но не операторы!), однако по понятным причинам инициализатор и итератор должен иметь побочные эффектов. Любое из трех выражений может быть оставлено пустым, тогда итератор и инициализатор просто не вычисляются, а пустое условие считается всегда истинным.

Обычное использование цикла `for` — со счетчиком:

```
for (i = 0; i < 10; ++i) printf ("%d␣%d\n", i, i*i);
```

В Си для целочисленных счетчиков типично использование нестрого неравенства. Разумеется, вместо `++i` можно указать, например, `i--` или `i+=2`. Разумеется, счетчик может быть и вещественным, а может изменяться не только в итераторе, но и в теле цикла.

При работе с циклами следует учесть ряд обстоятельств.

- С помощью оператора цикла можно создать программу, работа которой никогда не завершится. Например, указать 1 в качестве условия или составить условие таким образом, что сколько бы раз тело цикла не исполнялось, оно не станет ложным: **в теле цикла (или в хотя бы итераторе в случае `for`) следует обеспечить изменение условия**. Простейшие случаи бесконечного цикла:

  ```
  while (1);
  ```

```
do {} while (1);

for (;;);
```

- Цикл с постусловием всегда выполняет тело цикла хотя бы один раз. В то же время цикл с предусловием может быть не выполнен ни разу. В программировании чаще встречаются ситуации, когда при некоторых данных не нужно выполнение тела цикла, чем гарантированное требование по крайней однократного выполнения тела. Поэтому рекомендуется использовать цикл с предусловием, за исключением случаев, когда преимущество использование цикла с постусловием четко обосновано.
- С условиями в циклах (как и в условных операторах) связан риск перепутать равенство и присваивание. Следующий цикл бесконечен.

```
while (i = 1) do_something();
```

- Переменная, определенная в теле цикла, не видна в условии цикла, следующий код также создает бесконечный цикл.

```
int i = 1;
while (i)
{
    int i = 0;
}
```

Особенно велик риск создать такой бесконечный цикл при использовании цикла с постусловием:

```
int i = 1;
do
{
    int i = 0;
}
while (i);
```

Здесь, хотя переменная `i` в условии стоит *после* объявления и изменения значения одноименной переменной в теле цикла, она находится *вне* тела цикла (за пределами операторных скобок), поэтому является переменной `i`, объявленной до тела цикла, а не внутри тела цикла.

Вообще говоря лучше избегать использования одноименных переменных на разных уровнях вложенности. Оператор

```
int i = 1;
do
{
    i = 0;
}
while (i);
```

корректен (в том смысле, что не создает бесконечный цикл: тело цикла исполнится один раз, корректность данного примера в смысле решаемой задачи оставлена за скобки, подобные короткие примеры удобны для *локализации ошибок* и демонстрации типичных ошибок).

- Операторы цикла и условные операторы могут быть вложенными друг в друга.

  Структурное программирование основано на том, что любой алгоритм можно закодировать с помощью трех **структур управления** — последовательности, ветвления и цикла. Последовательность в Си определяется последовательностью записи операторов (условно через точку с запятой).

  Одна из основных идей структурного программирования в том, что структуры управления регулирует порядок выполнения операторов или операторных блоков, то есть составных операторов Си. Программа состоит из вложенных блоков кода, выполняющих некоторое обособленное действие, выбор, порядок исполнения и повторения исполнения этих блоков определяется условиями.

- При оформлении операторов цикла также следует использовать правило четырех пробелов — делать соответствующий отступ для тела цикла, чтобы визуализировать уровни вложенности и сделать легко видимым какой оператор тела к какому циклу или условию относится.

- Оператор `continue` приводит к непосредственному переходу к следующей итерации тела цикла, при этом для цикла `for` происходит вычисление итератора, оператор `break` прерывает исполнение цикла (самого внутреннего) и передает управление следующему за данным оператором цикла оператору. Использование данных операторов нарушает принципы структурного программирования и не рекомендуется за исключением случаев, когда это оправдано производительностью или читабельностью кода. В учебных задачах их лучше избегать.

Лабораторная работа №7
Циклы

Задание 7.1. *Дано целое положительное число. Найти*

(a) сумму его цифр

(b) наибольшую из его цифр

(c) наименьшую из его цифр

Замечание. Использовать цикл `while` для последовательного получения и отсечения цифр, возведение в степень исключить. Решить в двух вариантах: для цифр только десятичного представления, для цифр представления в заданной системе счисления. В последнем случае учесть, что ввод числа и его p-ичных цифр (суммы цифр) может и должен осуществляться в десятичной системе счисления.

Задание 7.2. *Дано целое положительное число. Визуализировать проверку его делимости на 3 с помощью признака делимости: циклическое сложение всех цифр числа с получением нового числа, к которому применяем тот же признак, до тех пор пока не получится однозначное число, для которого делимость проверяется условием совпадения с делящейся на 3 цифрой. На каждом шаге программа должна отображать складываемые цифры числа и их сумму, на последнем шаге вывести ответ.* Цель — не проверить делимость на 3 (для этого нужно использовать операцию вычисления остатка), а именно визуализировать признак.

Задание 7.3. *С клавиатуры вводится последовательность строго положительных вещественных чисел. Заранее количество вводимых чисел неизвестно, признак конца ввода — 0 (в последовательность не входит). Найти*

(a) наибольшее из этих чисел;

(b) наименьшее из этих чисел;

(c) наибольшее (наименьшее) из четных чисел списка;

(d) наибольшее (наименьшее) среди чисел списка, кратных заданному числу (вводится предварительно);

(e) наиболее близкое значение число списка (в смысле модуля разности значений) к заданному числу (вводится предварительно);

(f) сумму этих чисел;

(g) среднее арифметическое этих чисел;

(h) среднее геометрическое этих чисел;

(i) наибольшее и второе по величине число;

(j) наименьшее и второе снизу число;

(k) длину самого длинного участка возрастания чисел;

(l) длину самого длинного участка убывания чисел;

(m) длину самого длинного участка монотонного следования чисел, с указанием того, был это участок возрастания или убывания.

Более одного вводимого числа списка одновременно в памяти не хранить. Там где это возможно по условию, решить в двух вариантах: когда ответом является значение (число), и когда ответом является номер введенного числа.

Задание 7.4. *Составить программу для возведения заданного натурального числа в третью степень используя следующую закономерность*

$$\begin{aligned} 0^3 &= 0 \\ 1^3 &= 1 \\ 2^3 &= 3+5 \\ 3^3 &= 7+9+11 \\ 4^3 &= 13+15+17+19 \\ 5^3 &= 21+23+25+27+29 \end{aligned}$$

Задание 7.5. *Напечатать таблицу значений функции на отрезке $[a,b]$ с шагом h.* Функцию выбрать самостоятельно. Использовать цикл `for`. Таблицу напечатать "красиво", с заголовком, выравниванием по столбцам и аккуратным форматированием чисел.

Задание 7.6. *Напечатать таблицу значений выражения $(x^2-p^2)/(x-p)-p$ для некоторого p и для x из отрезка $[a,b]$ с шагом h. Обратить внимание, что математически данное выражение равно x. Рассмотреть случаи, когда x и p отличаются на большое количество порядков (5, 10, 15, 20 и т.п.). Всегда ли в результате вычисления получается x? Как объяснить именно такие результаты?*

Глава 3. Процедурно-модульное программирование

§3.1. Функции

Синтаксис написания функции

Код функции в Си состоит из **заголовка** и **тела функции**. Заголовок представляет собой следующую конструкцию:

<тип возвр. значения> <имя функции> (<тип пар-ра 1> <имя пар-ра 1>, <тип пар-ра 2> <имя пар-ра 2>, ...) >

Параметров может не быть, тогда скобки должны быть пустыми. Возвращаемого значения может не быть, тогда в качестве его типа нужно указать ключевое слово `void`. Такие функции называются void-функции. По формату заголовок функции соответствуют ее прототипу, но после него не ставится точка с запятой.

После заголовка в фигурных скобках следует тело функции. Оно представляет собой составной оператор, по сути блок операторов, список операторов, заключенных в фигурные скобки. В начале блока можно объявить **локальные** переменные, то есть переменные, которые будут доступны внутри функции и только внутри ее.

Возврат значения осуществляется с помощью оператора `return`. Оператор не только возвращает значение, но и прерывает работу функции. В void-функции можно использовать `return` без параметров, чтобы прервать ее работу.

Примеры.

1. Функция, возвращающая квадрат целого числа.

```
int sqr (int x)
{
    return x*x;
}
```

   Замечание. Следуя стилю стандартной библиотеки, функцию, возвращающую квадрат числа типа `double` следовало бы назвать `fsqr`.

2. Функция, вычисляющая целую степень числа (наивным, неэффективным образом).

```
int intpow (double x, unsigned n)
{
    double r = x;     /* локальная переменная для подсчета результата  */

    if (n == 0)       /* возведение в нулевую степень                  */
        return 1;     /* сразу возвращаем 1 и завершаем работу функции */

    for (; n>1; --n)  /* необходимо проделать n-1 умножение             */
        r *= x;

    return r;
}
```

   В Си все параметры в функции передаются по значению, то есть копируются, поэтому изменение формального параметра `n` внутри функции никак не скажется на значении фактического параметра в том месте, где функция была вызвана.

3. Функция, проверяющая, верно ли, что указанное целое число является точным квадратом.

```
int is_square (unsigned n)
{
    unsigned i = 0;
    while (i * i < n) i++; /* пока потенциальный корень не превысит n */

    /* здесь два варианта, либо i^2=n, те.. n --- точный квадрат
     * либо i^2>n, тогда n не является точным квадратом
     */

    return i * i == n;
}
```

Это логическая функция, она возвращает логическое значение (0 или 1). Соответственно, проверка с ее помощью будет выглядеть как `if (is_square(k))` иди `if (!is_square(k))` — пример ситуации, когда использование `!=0` или `==0` не только создаст дополнительные ненужные действия, но и ухудшит воспринимаемость программы, выведет на первый план детали реализации (представление логических данных в Си), а не суть.

Обратите внимание, что в последней строке было бы избыточно писать

```
    if (i * i == n)
        return 1;
    else
        return 0;
```

или

```
    return i * i == n ? 1 : 0;
```

так как фактически требуется именно значение указанного в функции выражения (0 или 1), а указанные конструкции не вычисляют новой информации.

Методика написания функции

Развитие концепций и языков программирования преследует ряд целей, направленных на облегчение труда программистов, повышение его эффективности, в т.ч. сокращения времени работы над задачей и количества ошибок в программах, а также облегчение сопровождение программ.

Основные задачи, стоящие перед развитием средствами разработки программ следующие:

- повторное использование кода, как написанного для других задач, так и в рамках одной задачи, в том числе исключение повторяющихся участков кода (в повторяющихся участках кода может возникнуть повторяющаяся ошибка, ее технически трудно исправлять, так как исправлять надо будет во всех местах, аналогичная проблема возникает при необходимости модифицировать или улучшить данный код);
- разбиение (декомпозиция) сложной задачи на более простые подзадачи, которые легче охватить “одним взглядом” и решить;
- повышение уровня абстракции (абстрагирование от деталей реализации ранее решенных задач при их использовании; возможность за малое количество шагов использовать готовое решение относительно сложной подзадачи; отдаление от низкоуровневых, аппаратных, инструментов и приближение используемых абстракций к понятиям и инструментам решаемых задач);

- разделение труда программистов.

В случае с функциями повторное использование кода достигается за счет того, что вместо повторного написания кода или его копирования вызывается функция; сложная программа разбивается на функции, решающие более простые задачи, также в них легче избегать находить ошибки (легче правильно написать, осознать, протестировать, отладить и проверить короткий код каждой функции отдельно, чем всей неразделенной на части программы целиком); повышение уровня абстракции достигается естественным образом (для работы с функцией достаточно знать, как она вызывается и что делает, но не важно как именно она это делает — всякая функция представляет собой отдельный черный ящик); облегчается сопровождение программ (модификация деталей реализации — внутренности черного ящика —- при сохранении внешности черного ящика не должна сказываться на работоспособности остальных функций); разделение труда программистов возможно путем распределения задачи написания отдельных функций программы между членами коллектива программистов.

При написании функций следует учитывать упомянутые цели существования аппарата функций, поэтому для того, чтобы качественно написать функцию, необходимо следовать ряду рекомендаций и соглашений.

- Каждая функция должна решать какую-то свою задачу. Если это расчетная функция, то она должна производить вычисление, если интерфейсная — отвечать за ввод и вывод и т.п. Смысл разделения программы на функции не просто в том, чтобы разделить большую задачу на малые подзадачи, но и чтобы эти подзадачи были достаточно независимы: например, в этом случае модификация пользовательского интерфейса становится возможной без изменения рабочих функций.

  В частности строгое разделение интерфейсной и рабочей части является важной методикой программирования.

- Прежде чем написать функцию, следует четко сформулировать, что именно она делает, какие у нее входные и выходные данные (внешняя сторона черного ящика). Название функции и параметров должно говорить само за себя (“принцип наименьшей неожиданности”).

  Всякая функция должна снабжаться комментарием, поясняющим, что она делает, каков смысл каждого из ее параметров, что означает возвращаемое значение. Это фактически документация к использованию данной функции.

  В идеале функция должна быть универсальной и библиотечного качества (если, конечно, это возможно, т.е. она не чисто вспомогательная), то есть быть приспособленной для обработки любых данных в любых условиях, в том числе чтобы могла быть использована и в другой программе, под это и следует затачивать ее прототип и описание.

- Следует отказаться от использования глобальных переменных.

  Не следует путать локальные переменные и формальные параметры функции: у них принципиально различная задача — первые нужны для хранения данных внутри функции (промежуточных вычислений и т.п.), вторые — для обмена данными между функциями (входных и выходных данных алгоритма).

В соответствии с последними двумя правилами хорошим прототипом для функции вычисления синуса будет

```
double sin (double x);
```

а плохими —

```
void sin ();
```

и

```
double sin ();
```

или

```
void sin (double x);
```

так как в них входные и/или выходные данные могут поступать в глобальных переменных (тогда функцию невозможно использовать в математических выражениях) или вводиться/выводиться пользователем (тогда функцию вообще нельзя использовать нигде кроме узкой задачи вычисления синуса введенного с клавиатуры числа и/или вывода этого значения на экран).

Лабораторная работа №8

Функции.

В данной лабораторной главная задача — написание функции, то есть рабочей части программы. Однако также требуется написание и интерфейсной части, которая позволит проиллюстрировать и протестировать работу функции.

Задание 8.1. *Написать функцию, находящую цифровой корень DR натурального числа n в системе счисления p.*

$$DR_p(n) = \begin{cases} n, & n < p, \\ DR_p(S_p(n)), & n \geqslant p, \end{cases}$$

где $S_p(n)$ — сумма p-ичных цифр числа n. Рекурсию не использовать.

Задание 8.2. *Напишите функцию, которая определяет является ли первый принимаемый параметр некоторой натуральной степенью второго параметра (для (n, m) указать верно ли, что есть некоторый $k \in \mathbb{N}$, такой что $n = m^k$, числа n и m целые.)*

Задание 8.3. *Написать функцию*

(a) которая принимает натуральное число n и выводит на экран квадрат из звездочек со стороной n;

(b) с тремя целыми параметрами m, n, d, изображающую в текстовом окне терминала при помощи символов * рамку размером m на n толщиной d;

Задание 8.4. *Написать функцию, которая определяет количество разрядов введенного целого числа.*

Задание 8.5. *Написать функцию приближенного вычисления квадратного корня из числа a с помощью метода итераций*

$$x_0 = a,$$
$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right).$$

с некоторой относительной погрешностью ε (итерации прекратить когда

$$|(x_n - x_{n-1})/x_n| < \varepsilon$$

). Учесть, что аргументом функции может оказаться отрицательное число (результат — не-число), положительная бесконечность (результат — она же). Замечание. Цикл `do..while` позволит написать более короткий код (без повторений).

Задание 8.6. *Написать функцию, вычисляющую приближенное значение следующих функций, используя указанный степенной ряд.* Бесконечную сумму заменить конечной, остановившись на первом слагаемом, которое окажется меньше (по модулю, если необходимо) величины ε. Прямое вычисление степени и факториала исключить, заменив нахождение значения каждого последующего слагаемого через предыдущее домножением на соответствующий коэффициент. Параметрами функции должны быть аргумент x и число ε.

(a) $\exp x = \sum\limits_{n=0}^{\infty} \dfrac{x^n}{n!}$;

(b) $\sin x = \sum\limits_{n=0}^{\infty} \dfrac{(-1)^n x^{2n+1}}{(2n+1)!}$;

(c) $\cos x = \sum\limits_{n=0}^{\infty} \dfrac{(-1)^n x^{2n}}{(2n)!}$;

(d) $\ln(1+x) = \sum\limits_{n=1}^{\infty} \dfrac{(-1)^{n-1} x^n}{n}$;

(e) $\operatorname{sh} x = \sum\limits_{n=0}^{\infty} \dfrac{x^{2n+1}}{(2n+1)!}$;

(f) $\operatorname{ch} x = \sum\limits_{n=0}^{\infty} \dfrac{x^{2n}}{(2n)!}$;

Параметр-указатель как выходной параметр

Синтаксически у функции может быть только одно возвращаемое значение. Однако иногда возникают ситуации, когда выходные данные алгоритма предполагают возврат более, чем одного значения. Это возможно при использовании **указателей**: если в функцию передать адрес переменной, то, используя этот адрес, можно изменить значение данной переменной: хотя сам адрес будет скопирован, это будет все тот же номер ячейки памяти, куда можно записать данные.

Указатели объявляются с помощью **символа указателя** `*` перед именем переменной или параметра. Например, `double *x` — указатель на переменную типа `double`, `int *n` — указатель на переменную типа `int`. Тип указателя важен, чтобы Си знал, значение какого типа содержится в указываемой ячейке памяти.

Получение указателя из переменной осуществляется с помощью операции **взятия адреса** `&`. Получение данных по указателю — с помощью унарной префиксной операции **разыменования** `*`. Эти данные можно как прочитать, так и изменить, то есть в примерах выше с `*x` и `*n` можно обращаться как с обычной переменной.

Замечание. Строго говоря Си различает выражения R-value, значения которых можно только прочитать, выражения L-value, значения которых можно прочитать и записать, и void-выражения, у которых нет значения. Void-выражения являются результатом вызова void-функции. Такие выражения могут выступать в качестве оператора-выражения или использоваться в качестве инициализатора и итератора цикла `for`, но не могут служить операндами операций и аргументами функций. Выражения L-value — выражения, ассоциированные с ячейкой памяти. К таким относятся, в частности, собственно переменные и результат разыменования указа-

теля. Только выражения L-value могут являться левым операндом операции присваивания, в т.ч. арифметического присваивания. Только выражения L-value могут выступать в качестве операнда операций инкремента и декремента. Только выражения L-value могут выступать в качестве операнда операции взятия адреса, т.к. только у ячейки памяти есть адрес. Все остальные выражения являются выражениями R-value. Терминология R (right, правое) и Left (left, левое) value (значение) связано именно с тем, что только L-value может выступать в качестве левого операнда операций присваивания, а R-value может выступать только в качестве правого операнда. В случае ошибки использования R-value там, где должно быть L-value компилятор выдаст соответствующую ошибку.

Следующая функция возвращает и остаток и неполное частное двух целых чисел. При этом, функция возвращает 0, если деление произведено успешно и -1 в случае ошибки.

Это один из вариантов проинформировать о проблеме, возникшей внутри функции: возвращаемое значение есть не более чем сигнал о успехе или ошибке в ее работе, в то время как сами выходные данные передаются через параметр-указатель.

```
/* Вычисление остатка и неполного частного
 * входные параметры:  n   - делимое, m - делитель
 * выходные параметры: div - неполное частное, mod - остаток
 * возвращает 0 в случае успеха, -1 в случае ошибки деление( на 0)
 */
int divmod (int n, int m, int *div, int *mod) /* указатели - выходные параметры */
{
    if (!m) return -1;

    *div = n / m; /* присваивание значения разыменованным указателям */
    *mod = n % m; /* как в обычные переменные                        */
    return 0;
}
```

Использовать данную функцию можно, например, так

```
int i, j, d, m;
/* ... */
if (divmod(i, j, &d, &m))  /* взяты адреса переменных для записи результата */
    printf("Ошибка, деление на ноль\n");
else
    printf("%d / %d = %d, %d %% %d = %d\n", i, j, d, i, j, m);
```

Функцию можно усовершенствовать: в примере выше если один из указателей `NULL`, выполнение функции приведет к краху программы. Это можно избежать вернув -1 при получении нулевого указателя. Однако лучше предоставить пользователю функции возможность корректно отказаться от ненужного при конкретном вызове выходного параметра — передать `NULL`.

```
/* Вычисление остатка и неполного частного
 * входные параметры:  n   - делимое, m - делитель
 *                     ( могут быть NULL, если не нужны )
 * выходные параметры: div - неполное частное, mod - остаток
 * возвращает 0 в случае успеха, -1 в случае ошибки деление( на 0)
 */
int divmod (int n, int m, int *div, int *mod)
{
    if (!m) return -1;
```

```
    if (div != NULL) *div = n / m; /* проверяется на 0 именно указатель */
    if (mod != NULL) *mod = n % m; /* писать mod != NULL излишне         */
    return 0;
}
```

Идентификатор NULL определен в библиотечном файле <stddef.h>, автоматически подключаемом всеми другими файлами стандартной библиотеки. Фактически он соответствует указателю на ячейку памяти с нулевым номером, поэтому проверка "if (div)" вместо более точной "if (div != NULL)" допустима.

Лабораторная работа №9

Функции и параметры-указатели

Задание 9.1. *Написать функцию, одновременно вычисляющую сумму и число p-ичных цифр целого числа.* Так как данная функция не может завершиться с ошибкой, одно из выходных чисел можно сделать возвращаемым значением.

Задание 9.2. *Написать функцию, которая упорядочивает значение двух целочисленных переменных: параметрами являются адреса двух переменных a и b, если $a > b$, функция меняет их значения местами.* В данном случае параметры-указатели будут одновременно и входными и выходными данными.

Задание 9.3. *Написать функцию, которая сортирует значения трех вещественных переменных.*

§3.2. Перечисления

Перечисляемый тип — тип данных, множество значений которого есть конечный набор идентификаторов. Это особый тип данных, создаваемый программистом: множество допустимых значений определяется в программе, значения этого типа можно сравнивать между собой. В Си перечисляемый тип сводится к целочисленному, а идентификаторы, которые могут использоваться в качестве значения типа становятся целочисленными константами (тип int). Например,

```
enum day {SUN, MON, TUE, WED, THU, FRI, SAT};

enum day d = MON;
```

Такая конструкция, во-первых, вводит тип данных, при этом собственно именем типа является именно сочетание enum и идентификатора (в данном примере enum day, а не просто day), что требует указание именно такого сочетания при объявлении переменных данного типа.

Во-вторых, вводятся константы данного типа, по факту целочисленные, по умолчанию — все различные, нумерация начинается с нуля. Так как перечисления являются целочисленными константами их можно использовать в операторе выбора:

```
switch (d)
{
case SUN: printf ("Sunday\n");   break;
case MON: printf ("Monday\n");   break;
/* ... */
case SAT: printf ("Saturday\n"); break;
default:  printf ("Invalid␣value\n");
```

```
};
```

Указание метки `default` обычно необязательно, так как других значений (если не присваивать переменной целочисленные значения напрямую) не возникает, но желательно, а вот перебрать все варианты необходимо.

Замечание. Избежать требования указывать `enum` каждый раз при объявлении переменной можно используя конструкцию `typedef`. В примере выше это можно достичь с помощью

```
enum day {SUN, MON, TUE, WED, THU, FRI, SAT};
typedef enum day day;
```

или

```
typedef enum day {SUN, MON, TUE, WED, THU, FRI, SAT} day;
```

или даже

```
typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT} day;
```

Во всех случаях `typedef` объявляет `day` как псевдоним для создаваемого перечисления (в первых двух случаях — псевдоним для `enum day`, в последнем — псевдоним для анонимного перечисления), поэтому объявление

```
day d;
```

станет корректным.

Лабораторная работа №10

Перечисления

Задание 10.1. *Написать функцию анализа квадратного уравнения $ax^2 + bx + c = 0$. Функция должна возвращать целочисленную константу-перечисление, информирующую о полученном случае (два вещественных корня, один кратный корень, два комплексных корня, линейное уравнение с одним корнем ($a = 0$, $b \neq 0$), уравнение, не имеющее корней ($a = 0$, $b = 0$, $c \neq 0$), любое вещественное число является корнем ($a = b = c = 0$), среди коэффициентов есть бесконечность или не-число). Корни (если есть) должны возвращаться через дополнительные параметры-указатели (учесть, что может быть передано* `NULL`*, тогда корень вычислять и возвращаться не надо). В интерфейсной части программы проанализировать возвращаемое функцией перечисление с помощью* `switch`*, вывести случай и корни (если есть).*

§3.3. Модули

Разделение программы на модули в Си

На практике программы бывают достаточно большие, состоящие из значительных по объему разделов и подзадач, решающихся с помощью набора взаимосвязанных функций, многие из которых являются чисто вспомогательными. Непрактично помещать весь исходный код программы в один файл уже хотя бы потому, что данный файл трудно будет осознать, в нем будут присутствовать разнородные и слабо связанные между собой группы функций.

Кроме того, если программа создается группой разработчиков, трудно организовать одновременную их работу с единственным файлом. Также при отладке программы при малейшем изменении файла требуется его полная компиляция, что может занять значительное время.

Использование модульного программирования решает все те же задачи развития средств разработки, но уже на другом уровне:

- повторное использование кода — вместо того, чтобы копировать код функции из другой программы достаточно подключить модуль;
- разделение труда программистов — разные модули одной программы могут создаваться разными людьми;
- сокрытие деталей реализации и повышение уровня абстракции — для работы с модулем достаточно знать его интерфейсную часть, то есть какие возможности он предоставляет, но не важно как именно он это делает (модуль расширяет понятие черного ящика до группы функций и типов данных, отвечающих за некоторый круг задач или подзадачу, вспомогательные функции выпадают из программного интерфейса модуля);
- облегчение сопровождения программ —- модификация деталей реализации при сохранении интерфейса не должна сказываться на работоспособности остальных модулей.

В Си модули как элемент программы не поддерживаются, концепцию модульного программирования можно реализовать при помощи **раздельной компиляции**.

- Программа разбивается на несколько файлов исходного кода (с расширением `.c`), содержащих реализацию групп функций, решающих выбранную подзадачу — реализация модуля.
- Для того, чтобы функции, реализованные в различных `.c`-файлах могли вызывать друг друга, создаются заголовочные файлы (с расширением `.h`), содержащие прототипы функций и объявления типов данных (например, перечислений), подключаемые по мере необходимости — интерфейсная часть модуля.
- Каждый `.c`-файл компилируется отдельно в объектный модуль (с расширением `.o`), затем объектные модули **компануются** в исполняемый файл программы.
- При сборке проекта IDE компилирует только те `.c`-файлы которые были изменены сами или подключают измененные `.h`-файлы (обычно решается при помощи сравнения времени создания объектного модуля и рассматриваемых файлов исходного кода и заголовочных файлов). Это значительно сокращает время компиляции при малых изменениях кода.

Создание заголовочных файлов

При создании заголовочных файлов следует руководствоваться рядом требований и рекомендаций.

- Следует выбирать говорящее имя заголовочного файла, соответствующее тому, какую задачу решает данный модуль.
- Всякий заголовочный файл должен быть защищен от двойного подключения, в противном случае может возникнуть ошибка компиляции из-за двойного определения одних и тех же идентификаторов. Делается это с помощью комбинации, подобной

```
#ifndef FILE_H_INCLUDED
#define FILE_H_INCLUDED

/* Здесь код заголовочного файла */

#endif /* FILE_H_INCLUDED */
```

  Идентификатор `FILE_H_INCLUDED` может быть любой, но должна быть уникальной среди всех заголовочных файлов, обычно она формируется из имени этого файла.

- В заголовочном файле должны присутствовать только объявления: введение типов данных, прототипы функций, символические константы и т.п. Никаких определений, в том числе кода функций и тем более глобальных переменных там быть не должно, это вызовет ошибку двойного определения при компоновке.
- Всякое объявление в заголовочном файле следует снабжать комментарием, что именно означает или делает вводимый идентификатор (функции, типа данных и др.). Комментарий к функции следует помещать именно в заголовочный файл. Также заголовочный файл (как и файлы исходного кода) следует снабжать общим комментарием, поясняющим, какая группа задач или подзадача решается с помощью данного модуля и кто его автор. По сути комментарий в заголовочном файле — почти готовая документация к модулю.
- Совершенно необязательно, чтобы все функции из файла исходного кода, имели прототип в заголовочном файле: часть из них может быть внутренними, представлять собой детали реализации, и не быть вызываемой из вне модуля. Объявления таких функций следует предварять ключевым словом `static`, для того, чтобы исключить их использование вне модуля.

Лабораторная работа №11

Модули

В данной лабораторной главная задача — написание модуля (заголовочного файла и файла исходного кода), то есть рабочей части программы. Однако также требуется написание и интерфейсной части, которая позволит проиллюстрировать и протестировать работу модуля.

Задание 11.1. *Написать модуль анализа свойств целых неотрицательных чисел. Интерфейс модуля должен предоставлять функции*

(a) вычисления суммы p-ичных цифр числа;

(b) вычисления количества p-ичных цифр числа;

(c) вычисления цифрового корня числа в системе счисления p;

(d) определения, делится ли число на сумму своих p-ичных цифр;

(e) определения, делится ли число на каждую из своих p-ичных цифр;

(f) определения, делится ли число на каждую из своих четных p-ичных цифр;

(g) определения, делится ли число на каждую из своих нечетных p-ичных цифр;

(h) определения, делится ли число на каждую из своих нечетных p-ичных цифр;

(i) определения, делится ли число на наименьшую из своих p-ичных цифр;

(j) определения, делится ли число на наибольшую из своих p-ичных цифр;

(k) определения, является ли число палиндромом.

(l) определения, является ли число n простым (перебором делителей до $\lfloor\sqrt{n}\rfloor$).

Задание 11.2. *Написать модуль анализа квадратного уравнения $ax^2 + bx + c = 0$. В заголовочном файле модуля следует определить перечисляемый тип, информирующий о результате анализа, (два вещественных корня, один кратный корень, два комплексных корня, линейное уравнение с одним корнем ($a = 0$, $b \neq 0$), уравнение, не имеющее корней ($a = 0$, $b = 0$, $c \neq 0$), любое вещественное число является корнем ($a = b = c = 0$), среди коэффициентов есть бесконечность или не-число), и функция анализа, возвращающая соответствующую*

константу-перечисление, а также корни (если есть) через дополнительные параметры-указатели (учесть, что может быть передано NULL, тогда корень вычислять и возвращаться не надо). В интерфейсной части программы проанализировать возвращаемое функцией перечисление с помощью `switch`*, вывести случай и корни (если есть).*

§3.4. Рекурсия

Функция в Си может вызывать сама себя как напрямую (прямая рекурсия), так и опосредованно (непрямая рекурсия). Это достигается за счет того, что для каждого вызова функции создается полный экземпляр всех локальных переменных и параметров данной функции, поэтому различные вызовы одной и той же функции работают с разными значениями параметров и локальных переменных.

При написании рекурсивных функций следует учитывать ряд замечаний и рекомендаций.

- Часто код рекурсивной функции проще и понятнее, однако нередко менее эффективен. Отдавать предпочтение рекурсии следует только в том случае, когда затраты труда программиста по разработке итеративного алгоритма заведомо не окупятся эффективностью этого алгоритма. Также простая рекурсивная функция может быть использована для проверки корректности работы сложной итеративной функции путем сравнения выдаваемых результатов.
- При написании рекурсивной функции следует проследить, чтобы был реализован **выход из рекурсии**: при некоторых значениях параметров функция должна завершиться без вызова себя, причем к этому значению параметров рекурсия должна прийти обязательно.

  Данные функций хранятся в т.н. **стеке** программы, размер которого ограничен операционной системой, поэтому бесконечная рекурсия приведет к ошибке переполнения стека, а не к бесконечному циклу.

Лабораторная работа №12

Рекурсия

Задание 12.1. *Написать функцию, вычисляющую*

(a) сумму цифр числа (только рекурсивно);

(b) цифровой корень числа (только рекурсивно);

(c) наибольший общий делитель (с помощью алгоритма Евклида) — рекурсивно и итеративно;

(d) наименьшее общее кратное с использованием предыдущей функции и соотношения

$$\text{НОК}(n, m) \cdot \text{НОД}(n, m) = n \cdot m.$$

Задание 12.2. *Написать рекурсивные и итеративные функции, вычисляющую значение в точке x многочлена специального вида порядка n, определяемого рекуррентным соотношением:*

(a) многочлен Чебышёва 1-го рода

$$T_0(x) = 1, \quad T_1(x) = x,$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

(b) многочлен Чебышёва 2-го рода

$$U_0(x) = 1, \quad U_1(x) = 2x,$$

$$U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x).$$

(c) многочлен Лежандра

$$P_0(x) = 1, \quad P_1(x) = x,$$

$$P_{n+1}(x) = \frac{2n+1}{n+1}\,x\,P_n(x) - \frac{n}{n+1}\,P_{n-1}(x).$$

(d) многочлен Лаггера

$$L_0(x) = 1, \quad L_1(x) = 1 - x,$$

$$L_{n+1}(x) = \frac{1}{n+1}\left((2n+1-x)L_n(x) - nL_{n-1}(x)\right).$$

(e) многочлен Падована

$$P_1(x) = 1, \quad P_2(x) = 0, \quad P_3 = x,$$

$$P_n(x) = xP_{n-2}(x) + P_{n-3}(x).$$

(f) многочлен Фибоначчи

$$F_0(x) = 0, \quad F_1(x) = 1,$$

$$F_n(x) = xF_{n-1}(x) + F_{n-2}(x).$$

(g) многочлен Эрмита

$$H_0(x) = 1, \quad H_1(x) = x,$$

$$H_{n+1}(x) = xH_n(x) - nH_{n-1}(x).$$

Задание 12.3. *Написать рекурсивную функцию, которая печатает в обратном порядке цифры p-ичной ($p \leqslant 16$) записи целого неотрицательного числа.*

Задание 12.4. *С клавиатуры вводится последовательность ненулевых вещественных чисел. Заранее количество вводимых чисел неизвестно, признак конца ввода — 0 (в последовательность не входит). Написать рекурсивную функцию (не используя цикл), которая находит*

(a) сумму этих чисел;

(b) наибольшее из этих чисел;

(c) наименьшее из этих чисел;

Указание. Функция, предоставляемая интерфейсом модуля, должна быть без параметров и может вызывать требуемую рекурсивную функцию, доступную только в реализации модуля.