Ловягин Никита Юрьевич
Алимова Ольга Викторовна
Ловягин Юрий Никитич

Информатика

Часть 1.

Основы структурного и процедурно-модульного программирования на языке Си

курс лекций

для студентов прикладных
математических специальностей

Санкт-Петербург
2023

Оглавление

1. Общие сведения об информатике и компьютерах **6**
1.1. Данные и алгоритмы . . . 6
1.2. Алгоритмическая неразрешимость и трансвычислимость . . . 8
1.3. Числа и цифровые компьютеры . . . 10
Системы счисления . . . 10
Биты и байты . . . 11
1.4. Устройство компьютера и архитектура . . . 12
Концептуальная схема устройства компьютера . . . 12
Архитектура компьютера . . . 12
Адресация . . . 13
1.5. Функции и библиотеки . . . 16
1.6. Программные абстракции и языки программирования . . . 17
Языки программирования . . . 17
Интерфейс и уровни абстракции . . . 19
Трансляция . . . 20
Типы данных . . . 20
Сравнение и классификация языков программирования . . . 21
1.7. Разработка программного обеспечения . . . 22
Жизненный цикл программного обеспечения . . . 22
Технология программирования . . . 23
1.8. Цифровое представление данных . . . 24
Представление целых чисел . . . 24
Представление вещественных чисел . . . 25
Представление символов . . . 27
Представление строк . . . 27
Логические значения . . . 28

2. Синтаксис языка Си **30**
2.1. Общие сведения о языке Си . . . 30
2.2. Простые типы данных языка Си . . . 31
2.3. Синтаксические единицы языка Си . . . 33
2.4. Токены . . . 33
Ключевые слова . . . 34
Идентификаторы . . . 34
Константы . . . 35
Операции . . . 39
Пунктуационные знаки . . . 39
О разделении токенов . . . 41
2.5. Выражения . . . 41
2.6. Деклараторы . . . 43
Объявление переменных . . . 43
Прототипы функций . . . 46
2.7. Операторы . . . 48
Пустой оператор . . . 48
Оператор-выражение . . . 48
Оператор возврата значения функции . . . 49
Оператор безусловного перехода . . . 49
Составной оператор . . . 49
Условный оператор . . . 50
Операторы цикла . . . 51
Множественный выбор и оператор выбора . . . 52
2.8. Функции . . . 55
2.9. Файл исходного кода . . . 56
2.10. Комментарий . . . 56
2.11. Область видимости идентификаторов . . . 57
2.12. Препроцессор . . . 58

Заголовочные файлы . . . 58
Символические константы . . . 59
2.13. Перечисления . . . 60
2.14. Декларация typedef . . . 61

3. Семантические особенности Си **62**
3.1. Арифметические операции . . . 62
3.2. Операции присваивания . . . 64
Прямое присваивание . . . 64
Арифметическое присваивание . . . 64
3.3. Операции сравнения, логические операции, условная операция . . . 65
Операции сравнения . . . 65
Логические операции . . . 66
Тернарная условная операция . . . 67
3.4. Битовые операции . . . 67
3.5. Адреса и указатели . . . 69
3.6. Константные данные . . . 75
3.7. Главная функция . . . 77
3.8. Форматированный ввод и вывод . . . 77
Форматированный вывод . . . 78
Форматированный ввод . . . 80
3.9. Другие операции . . . 83
Приведение типа . . . 83
Определение размера . . . 83
Операция запятая . . . 84
3.10. Виды выражений: L-value и R-value . . . 85
3.11. Вычисление выражений . . . 86
Общая идея вычисления выражений . . . 86
Регламент вычисления выражений . . . 87
3.12. Неопределенное поведение . . . 88
3.13. Стек вызовов и сегмент стека . . . 90
3.14. Рекурсия . . . 92

4. Составные типы данных **95**
4.1. Массивы . . . 96
4.2. Связь массивов и указателей . . . 98
4.3. Многомерные массивы . . . 99
4.4. Массив как аргумент функции . . . 100
4.5. Строки символов . . . 102
4.6. Ввод и вывод строк . . . 104
4.7. Обработка массивов в стандартной библиотеке Си . . . 106
4.8. Структуры . . . 107
4.9. Функциональный указатель . . . 110

5. Ввод и вывод **112**
5.1. Стандартные потоки ввода и вывода . . . 112
5.2. Открытие и закрытие файла . . . 113
5.3. Текстовые файлы: форматированный ввод и вывод . . . 115
5.4. Стандартные потоки и использование `stderr` . . . 116
5.5. Символьный ввод и вывод . . . 117
5.6. Строковый ввод и вывод . . . 118
5.7. Двоичный доступ к файлам . . . 119
5.8. Форматированный ввод и вывод в строку и обратно . . . 120
5.9. Параметры командной строки . . . 121

6. Парадигмы и методология программирования **122**
6.1. Ошибки в программах . . . 122
6.2. Критика оператора перехода . . . 123
6.3. Парадигма структурного программирования . . . 123
6.4. Процедурное программирование . . . 124
6.5. Модульное программирование . . . 125
6.6. Правила и соглашения хорошего кода . . . 132
Общие принципы . . . 132
Правила оформления . . . 133
Вопросы эффективности и минимизации ошибок . . . 136
Принципы процедурно-модульного программирования . . . 136
6.7. Тестирование и верификация программ . . . 139
Тестирование программного обеспечения . . . 139
Верификация программного обеспечения . . . 140
Проверка условий при исполнении Си-программы . . . 141

7. Тонкости работы с указателями и управления памятью **142**
7.1. Динамическое выделение памяти . . . 142
Сегменты памяти . . . 142
Статическое и динамическое . . . 143
Функции динамического выделения и освобождения памяти . . . 143
Изменение размера выделенного блока . . . 145
Замечания . . . 146
7.2. Арифметика указателей . . . 146
7.3. Примеры реализации обработки строк . . . 149
Копирование строк . . . 149
Копирование блоков памяти . . . 150
Вычисление длины строки . . . 151
7.4. Ошибки при работе с памятью . . . 152

Приложение **155**

A. Базовые сведения о компьютере, операционной системе и файлах **155**
A.1. Компьютер — твердый и мягкий продукт . . . 155
A.2. Общие сведения о файлах . . . 155

B. Дополнительные сведения о представлении чисел **157**
B.1. Общие замечания . . . 157
B.2. Таблица операций для расширенных чисел с плавающей точкой . . . 158
B.3. Переносимость двоичного представления чисел . . . 158
Целые числа . . . 158
Вещественные числа . . . 159

C. Пример изменения стека вызова **159**

D. Дополнительные возможности языка Си **162**
D.1. Неполный тип данных . . . 162
D.2. Макросы . . . 164
D.3. Классы хранения: `extern` и `static` . . . 165
D.4. Встраиваемые функции . . . 167
D.5. Функции с переменным числом аргументов . . . 168

E. Полный список вопросов для самопроверки **169**

Глава 1. Общие сведения об информатике и компьютерах

§1.1. Данные и алгоритмы

Информация *(говоря условно, образно)* — то что представляет собой ответ на какой-то вопрос, то из чего можно извлечь знания, сведения об окружающем мире и протекающих в нем процессах, об идеях и понятиях, независящих от формы представления этих сведений.

Слово "информация" происходит от латинского слова со значением, близким к словам "разъяснение", "представление". Единого определения понятия "информация" не существует, в различных науках и отраслях знаний могут даваться разные определения. В данном курсе нет нужды в формальном определении, поэтому значение данного слова считается интуитивно ясным, исходя из данного описания.

В качестве примера информации может выступать любое повествовательное предложение естественного языка человеческого общения, таблицы сведений, код ДНК и др.

Данные — набор значений, информация, представленная в форме, приемлемой для восприятия. Например, одно число (температура, размер файла и т.д.), набор чисел (дата, координаты вектора и т.д.), таблица сведений, текст, книга и др.

Данные могут обрабатываться (восприниматься и т.д.) как человеком, так и автоматическим устройством. Форма для восприятия разными объектами может различаться: для человека это число, написанное на бумаге с помощью графических знаков, для компьютера — число, записанное в электронном носителе информации с помощью различных состояний вещества носителя, соответствующие нулям и единицам. Специализированные устройства, как и всякое орудие труда, созданы для того, чтобы с его помощью можно выполнять определенные аспекты этого процесса быстрее, эффективнее, надежнее.

Информатика *(от "информация" и "автоматика", а также "математика")* — наука об информации (в широком смысле), наука о хранении, обработке, передаче, анализе и представлении информации (в узком смысле).

Также используется формулировка "наука об автоматизации хранения, обработки, передачи, анализа и представления информации", что отождествляет информатику с англоязычным термином **computer science**, но отделяет от **data science** (наука о данных) — наука о получении знаний из данных в различных формах (статистические и др. методы).

Компьютер — устройство для хранения, обработки, передачи, анализа и представления информации.

Компьютер *(более строго)* — устройство для выполнения операций над данными в соответствии с инструкцией.

Действительно, можно обратить внимание, что все многообразие возможностей компьютера (с бытовой точки зрения) сводится к приему (вводу), преобразованию (вычислениям), передаче (между частями компьютера или различными компьютерами) и представлении (визуализации) данных.

Более формально вместо слова инструкция следует употреблять понятие алгоритма. Существуют строгие, формальные математические, определения алгоритма (Алгорифм Маркова, Машина Тьюринга, Машина Поста и др.), но на данном этапе они не рассматриваются.

Алгоритм *(интуитивно)* — конечный набор действий (инструкций), которые необходимо выполнить пошагово для достижения результата.

Инструкции алгоритма могут содержать условия и переходы на другие шаги. Из бытовых примеров алгоритма можно привести описание маршрута (как добраться транспортом из одного места в другое с пересадками, в более сложном виде — в зависимости от ситуации, например, времени ожидания по расписанию, выбрать тот или иной маршрут), рецепт приготовления блюда, порядок пользования техникой, из математических — алгоритм Евклида поиска наибольшего общего делителя, порядок решения квадратного уравнения и т. п.

Алгоритм должен быть конечной системой инструкций ("**определенность**"), однозначно (без двучтений — "**детерминированность**") определяющей пошаговый процесс ("**дискретность**"), который позволяет получить из начальных данных ("**входные данные**") требуемый результат, ("**выходные данные**") за конечное число шагов.

Определенность, детерминированность, дискретность — обязательные свойства, присущие алгоритму. Хотя "бытовой" интуитивный алгоритм, понимаемый как последовательность инструкций, в общем случае может и не обладать данными свойствами, следует ограничить под термином "алгоритм" только те наборы инструкций, которые данными свойствами обладают.

"Правильный" или "хороший" алгоритм должен быть **конечным** и **массовым** (быть способным решать весь класс подобных задач, отличающихся только входными данными, для любых входными данных завершаться

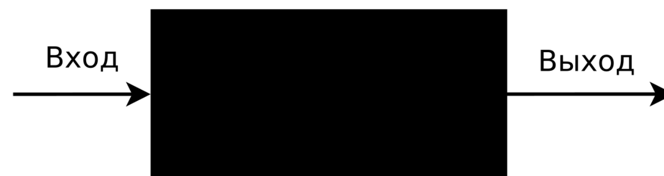


Рис. 1.1: Черный ящик.

за конечное число шагов с достижением результата или сообщать, что результат недостижим; точнее следует понимать, что отрицательный результат тоже является результатом). Неприятной является ситуация, когда для всех или каких-то данных алгоритм никогда не завершится, то есть, например, попадет в бесконечный цикл повторения одного и того же или нескольких шагов ("зациклится").

Бытовой алгоритм, например, приготовления пищи, и математический, например, алгоритм Евклида, являются "хорошими" алгоритмами, они завершаются за конечное число шагов. Легко придумать пример "зацикливающегося" алгоритма ("выполнять одно и то же действие"). Бывают ситуации, когда алгоритм неприменим к некоторым входным данным (например, с помощью алгоритма Евклида невозможно найти наибольший общий делитель ботинка и табуретки), поэтому нужно дополнять данный алгоритм так, чтобы при попытке применить его к чему-то неподходящему (в данном примере ко всему, кроме целых чисел) алгоритм сообщал об ошибке.

Ключевым моментом для любого алгоритма являются входные и выходные данные.

Черный ящик — модель устройства, имеющего вход для получения информации и выход, для получения результатов работы, для которого известно, какое именно преобразование входных данных в выходные происходит, но не как именно: внутренне устройство слишком сложное для описания или не является существенным при применения устройства для решении некоторой задачи.

То есть про черный ящик известно, *что именно* он делает, но *как* он это делает, неизвестно или неважно — не нужно, для его использования. Например, для переключения каналов на телевизоре с помощью пульта дистанционного управления не нужно знать, как это работает, достаточно знать, что при нажатии на кнопку нужного канала переключение произойдет. В качестве абстрактного примера можно представить себе аппарат, вычисляющий наибольший общий делить двух чисел. Схематически черный ящик изображен на рис. 1.1. Также можно отметить, что внутренность черного ящика — детали реализации — может быть скрыта в силу секретности (авторской, коммерческой и т.п.), но, вообще говоря, основная идея данного понятия этого не содержит.

Черный ящик — очень важный термин для понимания процесса программирования. Главная мысль применения черного ящика в том, что его можно использовать для выполнения определенной работы без вникания в то, как он эту работу будет выполнять, то есть оперировать (в мозгу) меньшим объемом данных, выполняя при этом более сложные действия, тем самым повышая эффективность труда.

Например, в случае с телевизором мы не задумываемся *как именно* телевизор переключает каналы при нажатии кнопки на пульте (читает в настройках сведения о частоте, соответствующей номеру программы, подстраивает частоту гетеродина и т.п.), вообще говоря, это избыточная для пользователя информация: хотя она, безусловно, полезна для общего и профессионального развития, прокручивать ее в голове каждый раз при переключении канала, во-первых, бесполезно, так как при нормально работающей системе это никак не поможет процессу, во-вторых, вредно, так как приведет к потере времени и сил, в-третьих, практически невозможно, так как внутри черных ящиков возникают другие черные ящики: как именно происходит чтение настроек, как именно происходит подстройка частоты, как именно пульт отправляет, а телевизор прочитывает сигнал — где сделать остановку раскрытия этих "матрешек"? Придется дойти до фундаментальных физических законов? Какой объем займет это объяснение?

В основе программирования лежит процесс составления алгоритма, который по входным данным получит выходные, представляющие требуемый результат.

Компьютерная программа — реализация (представление) алгоритма в форме, пригодной для выполнения компьютером.

Условно, алгоритм — внутренность черного ящика, это дело программиста, внешность — сторона пользователя. Задача программиста создать и реализовать свой черный ящик, однако на практике ему придется работать с другими черными ящиками, созданными другими программистами — ранее реализованными алгоритмами, уже решенными задачами, чаще всего не вникая в детали реализации.

Фактически на практике программист работает с **абстрактными** объектами — применяет в своей программе те или иные инструкции, которые решают ту или иную задачу в соответствии с описанием. Каждый "уровень

вложенности" "матрешек"-черных ящиков повышает **уровень абстракции**, тем самым позволяя за равный по объему набор инструкций — то есть примерно равный объем трудозатрат — решить более сложную задачу.

Подробнее об абстракциях и их уровнях будет изложено в параграфе 1.6.

Однако, в процессе обучения необходимо разобраться и с тем, как решаются более простые задачи, и с низкими уровнями абстракции, как в силу того, что за редким исключением невозможно сразу, без постижения азов, реализовать то, что никто никогда не делал, так и для понимания сути применяемых инструкций и повышения общего уровня мастерства программиста.

Вопросы для самопроверки.

1. *В чем различие понятий "информация" и "данные"?*
2. *Приведите примеры данных.*
3. *Что такое информатика?*
4. *Что такое компьютер?*
5. *Что такое алгоритм (интуитивно)?*
6. *Приведите примеры алгоритмов, отличных от упомянутых в тексте.*
7. *Какими свойствами должен обладать "хороший" алгоритм (интуитивно)?*
8. *Что такое входные и выходные данные алгоритма?*
9. *Приведите примеры алгоритмов, незавершающихся за конечное число шагов для некоторых, но не всех входных данных.*
10. *Что такое черный ящик?*
11. *В чем различие понятий "алгоритм" и "компьютерная программа"?*

§1.2. Алгоритмическая неразрешимость и трансвычислимость

Не всякая задача может быть решена *алгоритмическим образом.* Суть проблемы в том, что алгоритмическим решением задачи является построение алгоритма, который для любых возможных входных данных задачи получит ее решение за конечное число шагов. Однако, это не всегда возможно. Функции, для которых можно построить алгоритм, находящий их значение, называются **алгоритмически вычислимыми**. Задачи, для которых можно построить алгоритм, находящий их решение, называются **алгоритмически разрешимыми**. Соответственно, если алгоритм, который находит значение функции или решение задачи за конечное число шагов построить невозможно, такие функции и задачи называют **алгоритмически невычислимыми** и **алгоритмически неразрешимыми** соответственно. Вместо "задача" часто также используется термин "проблема". В некотором смысле всякий алгоритм p можно воспринимать как функцию входных данных d, значение которой — выходные данные $p(d)$, при условии, что алгоритм, будучи примененным к d завершается за конечное число шагов.

Классический пример алгоритмически неразрешимой задачи — проблема остановки.

Задача: требуется написать алгоритм, который сообщал бы 1, если данный алгоритм p на входных данных i останавливается, и 0, если зацикливается навечно. Пусть $\#p$ — запись алгоритма p (т.е. какое-то формальное изложение его инструкций). Можно считать, что данная запись представляет собой некоторое натуральное число: т.к. число возможных букв и цифр, используемых в записи ограничено, а сама запись конечна, то множество возможных записей алгоритма счетно, т.е. можно пронумеровать все возможные тексты.

Рассмотрим соответствующую функцию h двух аргументов:

$$h(\#p, i) = \begin{cases} 0, \text{ если } p(i) \text{ зацикливается,} \\ 1, \text{ если } p(i) \text{ завершается.} \end{cases}$$

Требуется написать алгоритм вычисления значения данной функции.

Рассмотрим некоторую алгоритмически вычислимую для любых значений аргументов функцию $f(x, y)$ и алгоритм $g(i)$ со следующим свойством:

$$g(i) = \begin{cases} 0, \text{ если } f(i, i) = 0, \\ \text{бесконечный цикл в противном случае;} \end{cases}$$

такой алгоритм существует, хотя для некоторых входных данных и не завершается за конечное число шагов: алгоритм для f существует, добавляется лишь проверка одного условия и бесконечный цикл.

Рассмотрим значения $h(\#g, \#g)$ — значение функции h на алгоритме g и данных g — зацикливается ли $g(\#g)$? Рассмотрим также значение $f(\#g, \#g)$. Они не совпадают:

- если $f(\#g, \#g) = 0$, то $g(\#g) = 0$, тогда $h(\#g, \#g) = 1$, так как $g(\#g)$ завершается;
- если $f(\#g, \#g) \neq 0$, то $g(\#g)$ зацикливается, тогда $h(\#g, \#g) = 0$,

то есть f не может быть той же функцией, что и h. Поскольку за f взята любая алгоритмически вычислимая (за конечное число шагов) функция, то никакая алгоритмически вычислимая (за конечное число шагов) функция не может являться функцией h, то есть не существует алгоритма, который за конечное число шагов вычисляет значение функции h.

Суть доказательства в том, что функция h — функция задачи, которую надо решить алгоритмически, то есть построить алгоритм вычисления h за конечное число шагов. Функция f — произвольная алгоритмически разрешимая функция, то есть функция, для которой существует соответствующий алгоритм. Алгоритм g подобран специальным образом. С его помощью показано, что значение функции h для него не совпадает с f, то есть h не совпадает ни с одной алгоритмически вычислимой функцией. Таким образом, h не является алгоритмически вычислимой.

Данное доказательство условно, базируется на том, что всякий алгоритм можно закодировать некоторым натуральным числом (т.е. можно считать, что у всех упомянутых здесь функций аргументы и значения — натуральные числа), в теории алгоритмов существует строгое доказательство. Проблема может показаться искусственной, но она четко иллюстрирует, что алгоритмически неразрешимые проблемы существуют.

Кроме того, даже если алгоритм решения задачи существует, на выполнение всех шагов может потребоваться недостижимое количество времени.

Пример: рассмотрим колоду из 36 карт и некоторый пасьянс. Вопрос: каков процент раскладов, при котором пасьянс сходится?

Предположим, что проверка сходимости выполняется за одну операцию. Даже в этом случае нам надо выполнить $36! \approx 3.72 \cdot 10^{41}$ операций, пусть современный процессор (ядро) выполняет $3 \cdot 10^9$ (3 ГГц) операций в секунду, Пусть компьютеры станут в миллиард (10^9) раз мощнее, пусть в мире будет 10 миллиардов жителей, у каждого будет 10 компьютеров по 10 ядер, т.е. на всей Земле будет 10^{12} процессоров, тогда для данного вычисления всеми компьютерами мира понадобится

$$\frac{3.72 \cdot 10^{41}}{3 \cdot 10^{9+9+12}} \approx 1.24 \cdot 10^{11} \text{ секунд} \approx 4000 \text{ лет.}$$

А если взять 54 карты, то $54! \approx 2.3 \cdot 10^{71}$, то есть примерно на 30 порядков больше возраста Вселенной.

Алгоритмически разрешимые задачи, для вычисления решения которых недостаточно времени существования Земли (или любого “разумного” для практического получения решения времени), называются **трансвычислительные**.

Правда, если ограничиться колодой в 16 карт (только “картинки”), то можно решить задачу современным компьютером в 10 процессоров 3 ГГц:

$$\frac{2.09 \cdot 10^{13}}{3 \cdot 10^{9+1}} \approx 697 \text{ секунд} \approx 10 \text{ минут :}$$

даже если на проверку одного расклада понадобятся десятки тысяч процессорных операций, то все равно время будет измеряться часами и сутками, а не годами и веками. Время перебора пропорционально количеству возможных комбинаций — $n!$. Факториал и показательная функция (экспонента) растут очень быстро: задачи, где количество комбинаций для перебора растет как экспонента или факториал от количества входных данных n, могут быть решены за разумное время при малых значениях n, но с ростом n — в том числе до значений, представляющих реальную практическую ценность — становятся трансвычислительными.

Кроме того, для записи данных может не хватать всех доступных средств хранения информации, то есть формально алгоритм может существовать, но для его выполнения может не хватать ни времени, ни емкости, поэтому нужно разрабатывать приближенные методы (в предыдущем примере нет необходимости искать точное количество сходящихся раскладов, достаточно узнать процент сходимости с точностью до нескольких цифр после точки), создавать более качественные алгоритмы, а также не забывать про аналитические решения.

Вопросы для самопроверки.

1. *Что такое алгоритмически неразрешимая задача?*
2. *В чем разница между алгоритмически неразрешимой и трансвычислительной задачей?*
3. *Какое значение для науки и практики имеет существование алгоритмически неразрешимых и трансвычислительных задач?*

§1.3. Числа и цифровые компьютеры

Ранее в русском языке вместо слова компьютер (англ. *computer*, “вычислитель”) использовался термин ЭВМ (электронно-вычислительная машина). Нельзя, однако, считать и этот термин русскоязычным, так как слова “электронный” и “машина” имеют латинские корни.

В любом случае, в дополнение к указанному выше определению, компьютер является устройством, производящим вычисления автоматически.

Цифровой компьютер *(digital computer)* — устройство для выполнения операций над данными, представленными в виде чисел.

Современные промышленные компьютеры используют исключительно целые числа, вещественные числа представляются приближенно с помощью целых, работают с конечными объемами данных и конечными инструкциями, работают в двоичной системе счисления.

Системы счисления

Утверждение 1. Для любого $p \in \mathbb{N}$, $p > 1$ всякое натуральное число $n \in \mathbb{N}$, $n \neq 0$ можно единственным образом представить в виде

$$n = d_0 \cdot p^0 + d_1 \cdot p^1 + d_2 \cdot p^2 + \cdots + d_m \cdot p^m, \text{ где } 0 \leqslant d_1, d_2, \ldots, d_m < p,\ d_m \neq 0.$$

Число 0 представляется посредством принятия $m = 0$ и $d_0 = 0$.

Число p называется основанием системы счисления, а числа d_0, d_1, ..., d_m — p-ичными цифрами числа n. Это записывается как

$$\overline{(d_m d_{m-1} \ldots d_1 d_0)}_p.$$

(При записи чисел цифрами черта и скобки не используются, скобки здесь поставлены для того, чтобы визуально отделить основание системы от индекса цифры).

Пример:

$$312_5 = 2 \cdot 5^0 + 1 \cdot 5^1 + 3 \cdot 5^2 = 2 \cdot 1 + 1 \cdot 5 + 3 \cdot 25 = 82 \equiv 82_{10}.$$

Нужно, однако, понимать, что цифры — лишь условные знаки, что десятичная система не является выделенной, просто мы привыкли к ней. Казус в том, что в этом примере основание системы записано в десятичной системе с помощью десятичных цифр. Если бы мы привыкли к пятеричной системе, то предыдущая запись выглядела бы так:

$$312 \equiv 312_{10} = 2 \cdot 10^0 + 1 \cdot 10^1 + 3 \cdot 10^2 = D2_{20}.$$

Здесь буква D используется как обозначение несуществующих цифр старше 4: A — 10 (соответствует десятичной цифре “5”), B — 11 (соответствует десятичной цифре “6”) и т.д. до цифры E — 14 (соответствует десятичной цифре “9”). При использовании десятичной системы как базовой такие обозначения используются для систем с основанием выше 10, например, в 16-ричной системе появляются цифры A, B, C, D, E и F.

Важно понимать, что цифры — графические знаки, а не числа. Этим знаком сопоставляется числовое значение. Числа существуют независимо от цифр. В качестве цифр могут использоваться любые знаки, например, можно определить пятеричную систему счисления и ее запись таким образом: символу `@` *сопоставляется числовое значение ноль, символу* `!` *— один,* `#` *— два, символу* `*` *— три, символу* `&` *— четыре. Тогда число из примера будет записано как* `#!*`*, что соответствует десятичному числу 82 (в привычной записи арабскими цифрами). Однако представляя себе пятеричную систему с пятью арабскими цифрами, мы получаем приведенную выше запись “312”.*

Причем, скорее всего привычным был бы счет в пятеричной системе (основание которой записывалось бы как 10), а из “нестандартной” десятичной (основание которой записывалось бы как 20) осуществлялся бы перевод:

$$D2_{20} = 13 \cdot 20^1 + 2 \cdot 20^0 = 13 \cdot 20 + 2 \cdot 1 = 310 + 2 = 312 \equiv 312_{10}.$$

Системнонезависимая запись числа осуществляется, например, указанием соответствующего количества палочек или точек — “единичная” система счисления:

$$(82 \text{ палочки}) = (||||||||) \cdot (||||||||||)^{(|)} + (||) \cdot (||||||||||)^{()} = (|||) \cdot (|||||)^{(||)} + (|) \cdot (|||||)^{(|)} + (||) \cdot (|||||)^{()},$$

() — отсутствие палочек, число ноль. Поэтому в замечании выше корректнее бы было сопоставлять символам именно такое значение (`@` — (), `!` — (|), и т.д.), чтобы абстрагироваться и от словесно-языкового обозначения цифр и чисел.

Утверждение 2. Для любого $p \in \mathbb{N}$, $p > 1$ всякое ненулевое целое число $n \in \mathbb{Z}$, $n \neq 0$ можно представить в виде

$$n = \pm(d_0 \cdot p^0 + d_1 \cdot p^1 + d_2 \cdot p^2 + \dots + d_m \cdot p^m),\text{ где } 0 \leqslant d_1, d_2, \dots, d_m < p,\ d_m \neq 0,$$

причем такое представление единственное. Это записывается как

$$\pm\overline{(d_m d_{m-1} \dots d_1 d_0)}_p.$$

При $n = 0$ аналогично $m = 0$, $d_0 = 0$.

Утверждение 3. Для любого $p \in \mathbb{N}$, $p > 1$ всякое вещественное число $a \in \mathbb{R}$ можно представить в виде

$$a = \pm \sum_{k=-\infty}^{m} d_k \cdot p^k,\text{ где } 0 \leqslant d_k < p,\ d_m \neq 0.$$

Это записывалось бы как

$$\pm\overline{(d_m d_{m-1} \dots d_1 d_0 . d_{-1} d_{-2} d_{-3} \dots)}_p,$$

если бы можно было записать бесконечное число цифр. Для рациональных чисел можно привлекать периоды или, в некоторых случаях, запись может оказаться конечной, т.е. найдется некоторое $i_0 < 0$, такое, что для всех $k < i_0$ $d_k = 0$, тогда эти незначащие нули опускаются.

Про единственность не утверждается, так если имеет место цифра $p-1$ в периоде начиная с j-й позиции (т.е. при всех $i < j$ имеет место $d_i = p-1$, а $d_{i+1} < p-1$), такое число равно числу, у которого 0 в периоде начиная с j-й позиции, а более старшая цифра на 1 больше (т.е. при всех $i < j$ имеет место $d_i = 0$, а d_{i+1} заменено на $d_{i+1} + 1 < p$, например, в десятичной системе $0.(9) \equiv 1.(0)$. С этой оговоркой и выделением нуля единственность имеется.

Показатели степени в p-ичном представлении числа называются номерами **разрядов** в числе. Поскольку от номера разряда — положения, позиции в записи — цифры в числе, зависит числовое значение этой цифры, такие системы счисления называются **позиционными**.

В противоположность позиционной системе записи чисел, система *римских цифр* позиционной не является, следует заметить, что проводить арифметически операции над числами, записанными римскими цифрами, гораздо сложнее (так как неосуществимы действия "в столбик").

Биты и байты

Двоичная система счисления (две цифры 0 и 1) не является экономичной, но ее просто реализовывать аппаратно — есть ток, нет тока.

Бит *(**bi**nary digi**t**, частица, порция)* — один разряд двоичной системы счисления, единица измерения количества информации (данных), физическая ячейка, способная представить такой разряд.

Байт *(byte)* — минимальная адресуемая ячейка памяти компьютера, минимальный объем информации, обрабатываемый компьютером одновременно. Исторически под байтом понималась совокупность битов, необходимая для представления одного символа. В большинстве современных компьютеров 1 байт — 8 бит, но не на всех типах компьютеров это так.

Любую информацию можно измерить количеством битов, необходимых для ее представления. Адресовать отдельный бит — слишком ресурсоемко (увеличивается размер адреса, уменьшается скорость доступа). Чаще всего информация в компьютере представляется целым количеством байтов.

Вопросы для самопроверки.

1. *Что такое цифровой компьютер?*
2. *Дайте определение цифры в позиционной системе счисления.*
3. *Как именно зависит значение цифры от позиции в записи?*
4. *Приведите пример перевода числа из восьмеричной системы счисления в троичную, как его бы записывал человек, для которого троичная система счисления — основная (число ||| записывается как 10).*
5. *Что такое бит? Что такое байт?*

§1.4. Устройство компьютера и архитектура

Концептуальная схема устройства компьютера

С концептуальной точки зрения компьютер включает в себя следующие устройства.

Устройство управления *(УУ)* — блок, выполняющий машинные команды.

Арифметико-логическое устройство *(АЛУ)* — блок, выполняющий арифметические и логические вычисления.

Центральное процессорное устройство *(ЦПУ)* — объединяет в себе АЛУ и УУ.

Оперативно-запоминающее устройство *(ОЗУ, память с произвольным доступом)* — память, в которой хранятся данные и машинный код исполняемых программ. Как правило, данная память является энергозависимой, т.е. данные в памяти стираются при отключении питания.

Постоянное запоминающее устройство *(ПЗУ, память, доступная только для чтения)* — энергонезависимое устройство хранения, в котором содержатся данные, подгружаемые в ОЗУ при включении компьютера (встроенное программное обеспечение, *firmware*, "прошивка" — не путать с жесткими дисками и т.п.). Примером такого ПО может служить BIOS или современное специфическое для производителей компьютеров встроенное ПО, объединенное под стандартом UEFI. Несмотря на название, в современных компьютерах данная память как правило перезаписываемая.

Устройство ввода-вывода — устройство, которое может обмениваться информацией с компьютером. Может подключаться внутри системного блока или как отдельное (периферийное) устройство. Бывают как устройства, способные только вводить или только выводить информацию (устройства ввода и устройства вывода соответственно), так и выполняющие обе эти функции.

Шина — устройство передачи данных между частями компьютера.

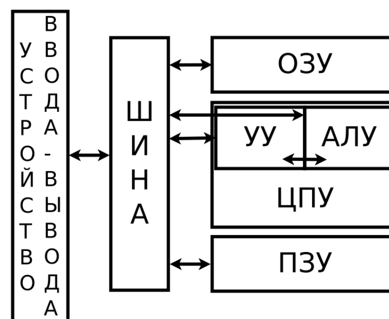


Рис. 1.2: Схема ЭВМ.

В данной классификации не только такие устройства, как мониторы, мыши, принтеры и клавиатуры, но и жесткие диски являются устройствами ввода-вывода. По сути они сами являются отдельным компьютером, со своим процессором, памятью, встроенным ПО и другими компонентами, обменивающимися информацией с другим устройством с помощью специализированных аппаратных средств — **аппаратного интерфейса** (то же самое относится к видеокартам и другим внутренним устройствам расширения возможностей компьютера, к периферийным устройствам и т.д.). Обмен данными между компьютерами по сети или кабелю также относится к вводу-выводу информации.

Архитектура компьютера

Архитектура — аппаратная и математическая организация устройства; набор правил и методов, описывающих функционал и реализацию устройства. Таким образом, может идти речь о конкретной архитектуре, включающей в себя организацию конкретного типа устройств — компьютера в целом, а также отдельных его частей (процессоров, видеокарт, накопителей данных и т.д.). В частности, **архитектура процессора** определяет набор машинных команд, поддерживаемых данным типом процессоров. Под архитектурой также может пониматься абстрактная концепция, лежащая в основе такой организации.

Компьютеры на основе разных процессоров (разных фирм, разных версий) имеют разную архитектуру (т.к. используют разный набор машинных команд). В то же время в основе современных компьютеров лежит **архитектура Фон Неймана**, которая определяется следующими принципами.

Принцип двоичного кодирования: вся информация кодируются двоичными цифрами 0 и 1.

Принцип программного управления: программа состоит из последовательности команд, выполняемых последовательно друг за другом, команда может представлять собой условный или безусловный переход.

Принцип адресности: память состоит из пронумерованных ячеек, процессору в произвольный момент доступна любая ячейка по ее номеру (адресу).

Принцип однородности памяти: команды и данные неразличимы, хранятся в одной и той же памяти, обрабатываются одинаковым образом.

Команды также кодируются в двоичной системе счисления. Это **машинные команды**, декодируемые и исполняемые устройством управления.

Принцип однородности позволяет программировать компьютеры (в отличие, например, от простейших калькуляторов) — загрузить программу с устройства ввода-вывода и исполнить ее.

На рис. 1.3 приведен пример работы программы с условными переходами.

Следует отметить, что современные компьютеры расширяют архитектуру фон Неймана, например, в них добавлены *аппаратные прерывания* — переход управления к другому адресу по сигналу от внешнего устройства, которые позволяют избежать постоянных проверок наличия ввода с устройства и экономить процессорное время (например, по нажатии клавиши клавиатуры происходит прерывание и выполняется обработчик данного нажатия, при отсутствии такого механизма пришлось бы регулярно, каждые несколько десятков миллисекунд или даже чаще проверять наличия нажатий клавиш, движений мыши и т.д.).

Термин "архитектура" может относиться не только к аппаратной организации компьютера, но и к компьютерным программам (**архитектура программ**, **архитектура операционных систем** и т.д.)

Адресация

Если бы программа кодировалась с указанием явного адреса ячейки памяти, где содержатся данные или куда передается управление (т.е. по какому адресу процессору следует прочитывать исполняемый код), то от компьютера требовалось бы всегда подгружать программу в один и тот же диапазон памяти, что крайне непродуктивно, например, сложно реализовать многозадачную среду, когда в каждый момент работает несколько программ.

На самом деле в коде программы указывается некоторый относительный адрес: относительно начала программы, относительно начала блока данных и т.д. Более того, программа работает с виртуальным адресным пространством.

Адресное пространство — набор адресов памяти, доступных программе в данный момент.

Виртуальное адресное пространство — набор адресов (номеров) ячеек, связываемых с физическими адресами ОЗУ операционной системой.

Процесс — набор исполняющихся в данный момент команд, имеющих доступ к отдельному (виртуальному) адресному пространству.

При запуске программы создается процесс. В многозадачной системе может исполняться несколько процессов одновременно (даже если есть всего один ЦПУ — иллюзия параллельной работы нескольких программ создается поочередным исполнением каждого запущенного процесса в течении небольшого отрезка времени), но они не видят данные друг друга (виртуальное адресное пространство параллельных процессов проиллюстрировано на рис. 1.4). Это ситуация решает как проблему адресации (одинаковые адреса, прописанные в машинном коде программы разных процессов будут ссылаться на разные физические ячейки памяти), так и безопасности — невозможно случайно изменить данные другого процесса, невозможно (без соответствующего разрешения операционной системы) прочитать конфиденциальные данные (например, пароли) у другого процесса.

Вопросы для самопроверки.

1. *Постройте и поясните схему ЭВМ.*
2. *Приведите примеры компьютерных комплектующих и периферийных устройств (принтер, жесткий диск, карта памяти, монитор, мышь, центральный процессор, оперативная память). Поясните, к каким узлам схемы ЭВМ они относятся. Замечание. Материнская плата содержит в себе различные элементы, включая шину и ПЗУ.*
3. *Каковы положения архитектуры фон Неймана?*
4. *Каким образом кодируются команды и данные в архитектуре фон Неймана?*
5. *Каковы свойства памяти в архитектуре фон Неймана?*
6. *Что такое архитектура компьютера?*

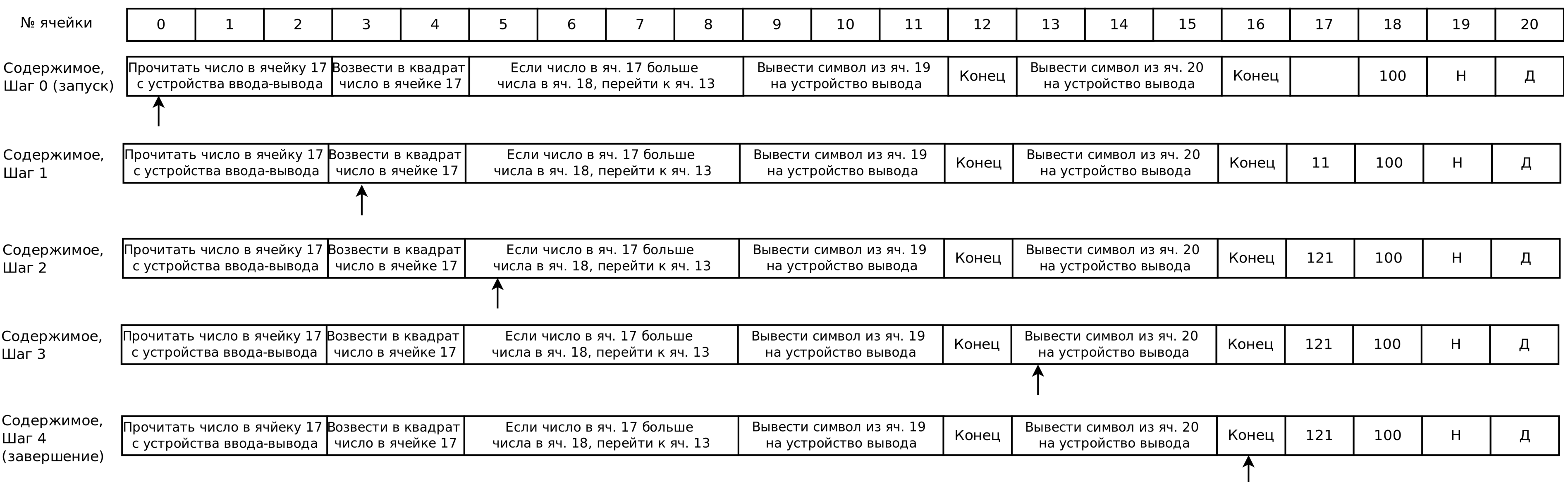


Рис. 1.3: Пример изменения состояния памяти с загруженной программой. Приведены понятные человеку описания команд и данных. В реальных компьютерах команды, их размер в памяти и представление данных может отличаться. Стрелкой отмечена ячейка памяти, где находится команда, которая будет выполнена на данном шаге работы программы. Различные команды и данные в реальных компьютерах могут занимать различное количество ячеек (байтов). Перед запуском программы значение в ячейке 17 не задано, в реальных компьютерах там может оказаться непредсказуемое значение. На нулевом шаге с устройства ввода поступило число 11. На предпоследнем шаге программа вывела на устройство вывода символ “Д”.

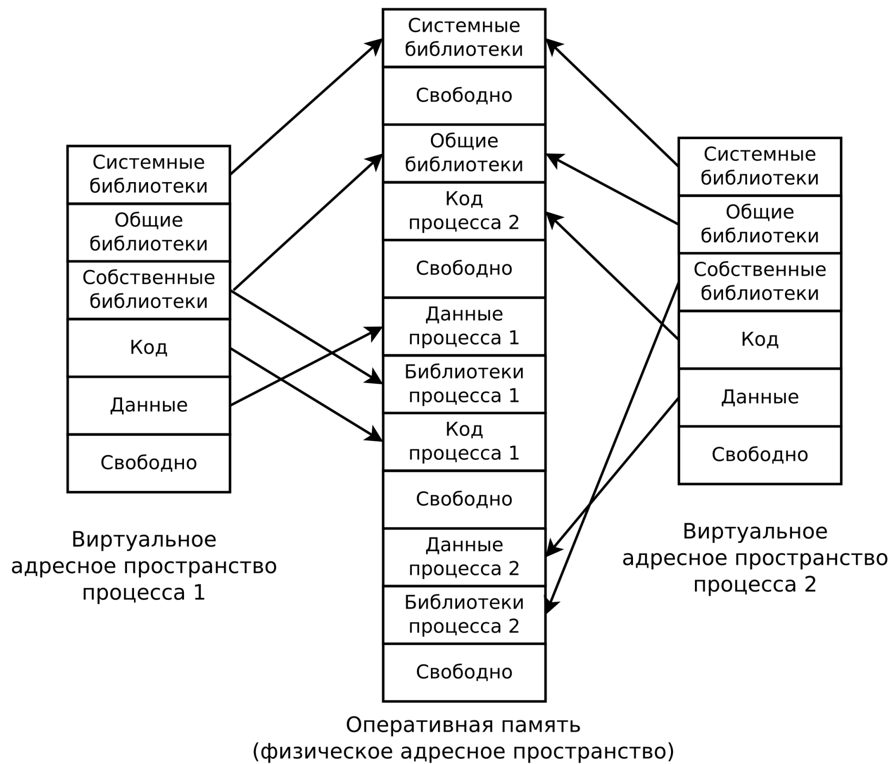


Рис. 1.4: Виртуальное адресное пространство и физическая память. Стрелками проиллюстрировано сопоставление физических адресов виртуальным, осуществляемое операционной системой. Все номера ячеек, используемые процессами, являются номерами ячеек виртуального адресного пространства, доступа к физическим адресам и виртуальным адресам другого процесса нет. При затребовании процессом дополнительной памяти или при запуске нового процесса будет выделен блок свободной памяти.

7. *Опишите, что именно делает программа, изображенная на рис. 1.3: какое именно преобразование входных данных в выходные производит алгоритм.*
8. *Что такое процесс?*
9. *Что такое виртуальное адресное пространство процесса?*
10. *Зачем нужно виртуальное адресное пространство процесса?*

§1.5. Функции и библиотеки

Создавать программы, которые работают непосредственно с аппаратурой компьютера и содержат полный код каждого, в том числе однотипного, действия невыгодно и небезопасно. В целях повторного использования ранее написанного кода используются **функции**.

Функция *(процедура, подпрограмма, метод)* — набор программных инструкций, выделенный в самостоятельную единицу.

Указанные понятия не совсем тождественны, однако в данном курсе будем использовать термин "функция" как наиболее универсальный и близкий к языку программирования Си.

Идейно всякая функция представляет собой реализацию некоторого алгоритма, преобразующего входные данные в выходные определенным образом. В таком контексте эти данные называются **параметрами** функции. У функции могут быть как входные, так и выходные параметры. Применительно главным образом к входным параметрам применяется также термин **аргумент** функции: по аналогии с аргументами математических функций, для обозначения тех данных, к которым была применена функция.

Всякая функция представляет собой черный ящик: для ее использования достаточно знать что она делает (внешняя сторона ящика), при этом как именно она это делает (детали реализации) неважны.

При программировании акт написания вызова функции является немногим более сложной (трудоемкой) задачей, чем акт написания машинной команды, сама же функция может содержать в себе большое количество команд и вызовы других функций, то есть использование функций позволяет при незначительном увеличении трудозатрат существенно (степенным образом) увеличить производительность труда программистов. Функция — один из *способов повторного использования кода*, возможность решать новые задачи, опираясь на уже решенные.

Вызов функций на машинном уровне реализован следующим образом (рис. 1.5):

- в нужное место памяти записываются значения входных параметров, точка возврата — место кода, откуда следует продолжить выполнение после завершения функции (т.е. сохраняется адрес машинной команды, следующей за командой вызова функции), а также резервируется место для сохранения выходных параметров;
- начинается исполнение кода функции, в частности вычисляются выходные параметры;
- по команде завершения функции, осуществляется переход в сохраненную точку возврата;
- прочитываются выходные параметры функции.

Под "записыванием" и "прочитыванием" входных и выходных параметров следует понимать копирование данных из области памяти, содержащей данные вызывающей функции, в область памяти, содержащей данные вызываемой функции, и обратно.

Функции могут содержаться в исполняемых файлах программы, как составная их часть, а также в отдельных файлах, представляющих собой "коллекцию" функций — **библиотеках**.

Библиотека — совокупность функций, которые могут вызываться из прикладных программ; обычно представляет собой отдельный файл, содержащий код данных функций и информацию о способах доступа к ним (с расширением so (shared object) в Linux, dll (dynamic link library) в Windows); бывают системные, прикладные и т.п.

Примеры возможных вызовов показаны на рис 1.6.

Заметим, что с определенной точки зрения операционная система компьютера представляет собой в частности набор библиотек функций, с помощью которых программы могут взаимодействовать с аппаратной частью компьютера. Совокупность аппаратной и программной среды, в которой возможно исполнение компьютерных программ называется **компьютерной платформой**. Выделяют, соответственно, аппаратную платформу (архитектуру компьютера) и программную платформу (операционную систему или другую среду исполнения). Разнообразие компьютерных платформ ставит перед программистами задачу совместимости программ с различными платформами (кроссплатформенности, мультиплатформенности).

Вопросы для самопроверки.

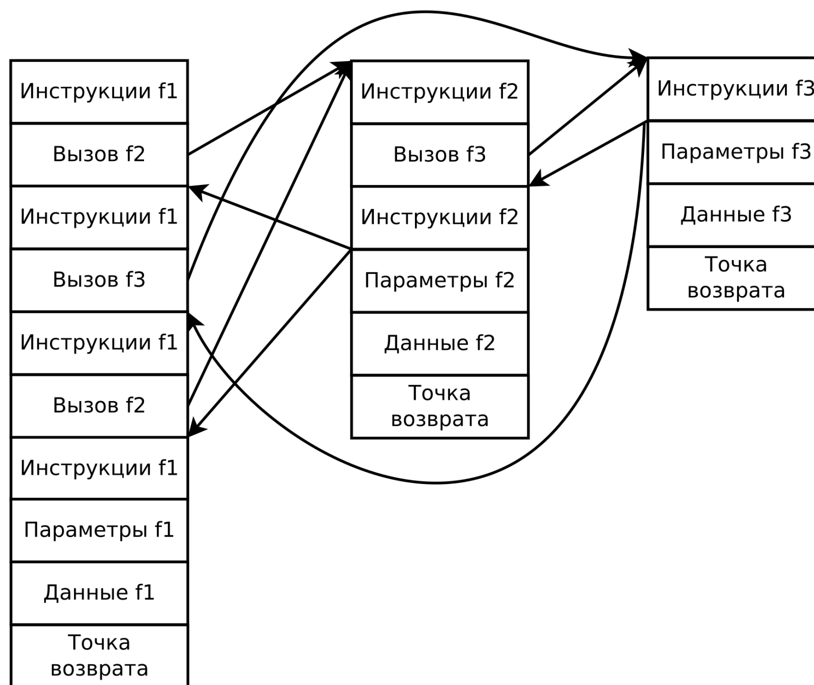


Рис. 1.5: Иллюстрация вызова функции. Условно изображены инструкции и данные функции, а также моменты вызова функций. Стрелками показаны передача управления при вызове и возврате. Возврат неоднозначен — для того, чтобы знать куда передать управление, хранится точка возврата: после выполнения инструкций функции производится переход на инструкцию, содержащуюся в ячейке с сохраненным номером. Столбцы сделаны для наглядности, реально память линейна.

1. *Что такое функция?*
2. *Как осуществляется вызов функции?*
3. *Какую задачу решает аппарат функций?*
4. *Что такое библиотеки функций?*
5. *Кто является поставщиком библиотек функций?*

§1.6. Программные абстракции и языки программирования

Языки программирования

Программирование непосредственно **в машинных кодах** возможно, но требует, во-первых, написание трудных для восприятия двоичных кодов машинных команд, во-вторых, ручное вычисление адресов данных и точек перехода в коде. Существует **мнемоническое кодирование** — наглядное, буквенное представление команд. Однако программирование в мнемокодах не снимает проблемы необходимости вычисления адресов (и даже усугубляет ее, так как заранее предсказать размер той или иной команды в байтах становится сложнее). **Язык ассемблера** решает эту проблему, добавляя к мнемокодам автоматически вычисляемые адреса по меткам.

Для автоматизации определения адреса данных или кода в памяти в языке ассемблера вводится понятие **метки** — поименованной точки (отдельного байта) в участке кода или данных программы. Таким образом, вместо обращения по конкретному адресу — например, для перехода в коде, для вызова функции, для чтения и изменения данных — становится возможным указывать имя метки.

Однако все три подхода имеют еще ряд недостатков: необходимо самостоятельно следить за способом кодирования данных (например, ассемблер не предупредит при попытке сложить вещественные числа как целые: ввиду разного способа представления их в памяти для этого используются разные машинные команды), крайне затруднена логическая организация (структуризация) кода, получаемый код годится только для выбранной архитектуры и операционной системы, то есть не является кроссплатформенным. Кроме того, использование чисто машинных команд требует написание огромного количества строк кода даже для решения простейших задач, хотя технически вызов функций из библиотек в машинном коде возможен. Частично задачи структурирования и вызова функций решается макроассемблером, но проблема ручного контроля за кодированием данных и платформозависимость остаются.

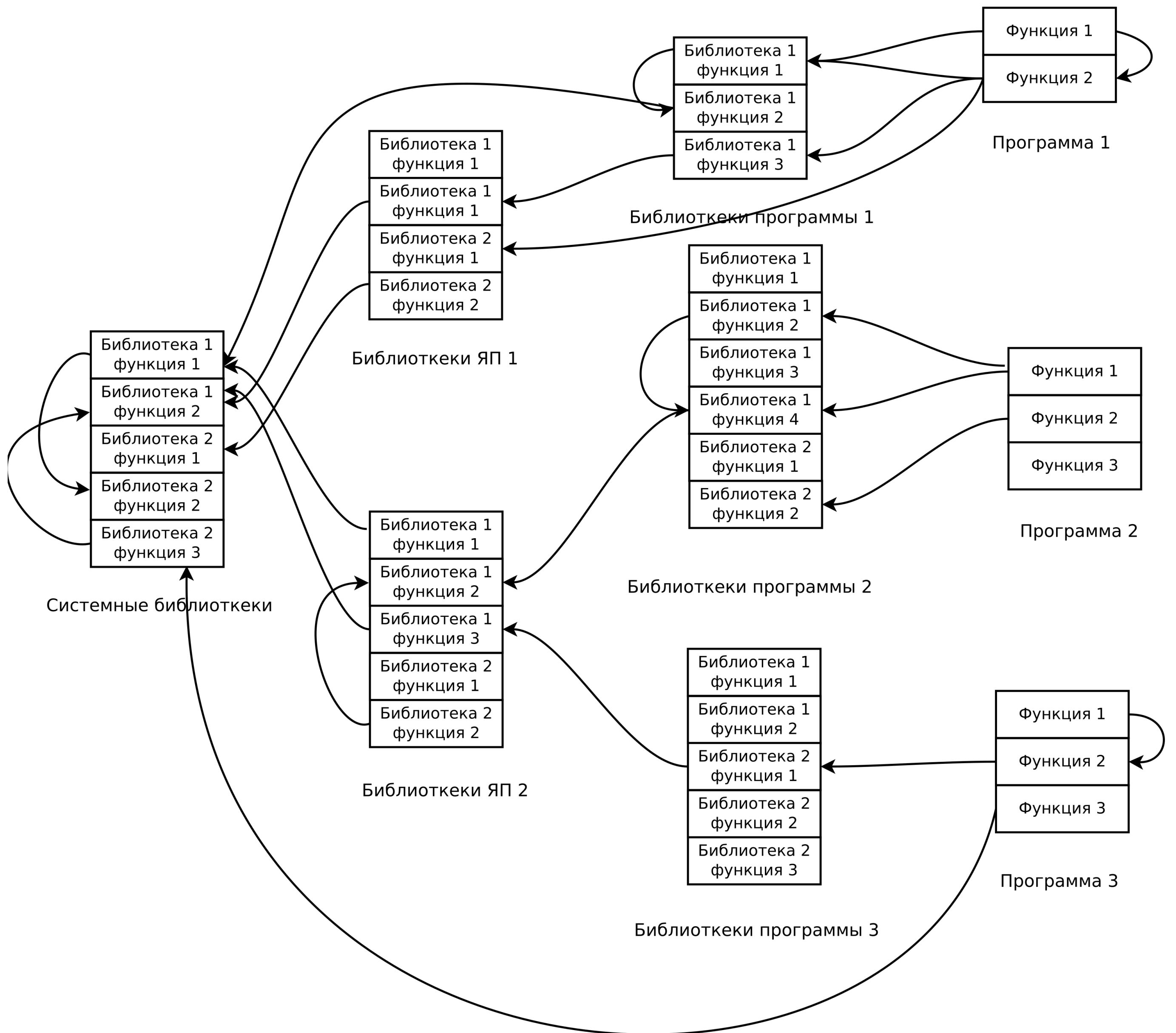


Рис. 1.6: Иллюстрация работы программ с использованием библиотек. Примеры возможных вызовов показаны стрелками.

Указанные проблемы стимулировали развитие различных **языков программирования** — формальных искусственных языков для записи компьютерных программ.

Говоря о языках вообще, следует отметить, что разделяют **естественные языки**, которые возникли в процессе эволюции как средство общения живых существ, и **искусственные языки**, созданные человеком целенаправленно, например — эсперанто, эльфийские языки и языки программирования. Из искусственных выделяют также формальные языки, чьи правила образования слов строго детерминированы. Языки программирования могут считаться формальными языками, хотя они имеют свойство эволюционировать — отходить от стандарта, использовать диалекты компилятора и т.п.

Следует также отметить, что всякий язык характеризуется **алфавитом** — набором используемых символов, **синтаксисом** — правилами образования слов и **семантикой** — смыслом этих слов. Совокупность правил образования слов также называется **грамматикой** (данная терминология упрощенная, но отражает суть). Также используется термин **прагматика** — зависимость смысла от контекста (в естественных языках такая ситуация встречается часто, например, многозначные слова и т.д., но и в искусственных языках возможны контекстно-зависимые значения слов).

Интерфейс и уровни абстракции

Абстракция — представляет собой набор возможностей (функций, команд и других), реализованных аппаратно или посредством других функциональных возможностей, принадлежащих абстракциям более низкого **уровня**.

Абстракции условно группируют по уровню (или в слои) абстракции по "степени удаленности" от аппаратной реализации — от машинных команд и двоичного кодирования данных.

Использование функций и библиотек позволяет повышать **уровень абстракции**.

Во-первых, абстракция является способом сокрытия деталей реализации определенного набора функциональных возможностей, позволяющих программисту не держать в голове лишнюю информацию, переход на следующий уровень абстракции незначительно (обычно, не более, чем линейно) увеличивает усилия программиста по вызову одной функции, но повышает сложность решаемых программой задач при том же числе строк кода.

Следует отметить, что даже программирование посредством машинных инструкций представляет собой работу с черным ящиком: их аппаратная реализация не является существенной информацией при их использовании.

Во-вторых, абстракция является средством повышения переносимости программ: изменение абстракции более низкого уровня не должно требовать изменения в программах, использующих более высокий уровень. Например, программа, написанная на языке программирования без использования системных функций, а только с использованием функций данного языка, может быть перенесена с одной операционной системы на другую, т.е. является **кроссплатформенной**.

В-третьих, абстракцию можно считать инструментом безопасности: разрешение прикладным программам работать с аппаратным уровнем напрямую фактически открывает доступ ко всем данным всех пользователей для всех пользователей, ядро ОС может ограничить такой доступ.

Как на уровне аппаратных средств (аппаратной архитектуры), так и на программном уровне необходимо взаимодействие компонентов — например, двух устройств, двух программ или их частей, программы и человека. Такое взаимодействие осуществляется с помощью **интерфейса**

Интерфейс — совокупность средств, с помощью которых компоненты компьютерной системы могут взаимодействовать и обмениваться информацией. Компоненты и средства могут быть как программными, так и аппаратными, соответственно выделяют **аппаратный** (например, интерфейс USB) и **программный** интерфейс, а также **пользовательский** интерфейс, обеспечивающий взаимодействие пользователя и компьютера.

Всякая библиотека предоставляет некий программный интерфейс для взаимодействия с ней другой программы (или программиста при написании программы с использованием этой библиотеки) — набор доступных для вызова функций и других программных абстракций.

В этом смысле программный и пользовательский интерфейсы представляют собой программные абстракции.

Язык программирования высокого уровня — язык программирования, позволяющий использовать конструкции высокого уровня абстракции.

Данные конструкции ближе к объектам, с которыми оперируют люди при решении поставленных задач, чем машинные команды. Всякий язык высокого уровня основан на некоторой идее — "философии" написания программы на нем, облегчающей процесс программирования, позволяющей структурировать программу, а также содержит в себе стандартную (встроенную в язык) библиотеку функций, предоставляющую готовые решения (реализации алгоритмов) типовых задач.

Язык ассемблера считается языком программирования **низкого уровня абстракции** (низкоуровневым).
Суммируя вышеизложенное, условно можно выделить следующие уровни абстракции:

- аппаратный уровень (физическая реализация);
- уровень машинных команд (низший уровень программной реализации);
- уровень встроенного ПО (осуществляется прямой доступ к устройствам ЭВМ);
- ядро операционной системы (драйвера устройств, управление устройствами и программами, унификация интерфейсов однотипных разнородных устройств, обеспечение безопасности);
- библиотеки (операционной системы, языков программирования высокого уровня сторонних производителей);
- языки программирования высокого уровня (синтаксис);
- системные и прикладные программы.

Абстракции, реализованные с помощью программных средств, называются **программными абстракциями**.

Трансляция

Программирование на любом языке, включая мнемокод и ассемблер, требует перевода текста программы — **исходного кода** — в машинный язык. Данный процесс называется **трансляцией**, выполняется с помощью особой программы, называемой транслятор. Трансляция бывает по крайней мере двух видов: **интерпретация** (перевод текста программы в машинный код с немедленным исполнением, выполняется с помощью специальной программы — интерпретатора) и **компиляция** (перевод текста программы в **исполняемый код**, соответствующая программа называется компилятор).

Интерпретируемая программа работает медленнее, чем компилированная, кроме того, для работы программы на интерпретируемом языке от пользователя требуется устанавливать интерпретатор на свой компьютер.

Компилированная программа работает быстрее, но не все языки поддерживают компиляцию по ряду причин. Например, сценарии оболочки (Bash) и языки Python, PHP — интерпретируемые. Одна из вещей, неподдающихся компиляции — обращение к переменной по имени, которое хранится в другой переменной (для компилятора переменные существуют только в исходном коде, а после компиляции переменная будет заменена адресом ячейки памяти — к какому адресу обращаться, если имя пришло из входных данных и заранее неизвестно?), исполнение кода языка, записанного в переменной и др.

Компиляция переводит один файл исходного кода программы в один **объектный модуль** — представление программы, близкое к машинному коду. В дальнейшем требуется **компоновка** (связывание) — объединение нескольких объектных модулей (если их несколько) и связывание с библиотеками, в результате получается **исполняемый модуль**, который может требовать для исполнения наличие библиотек, но не наличие компилятора.

Часто процессы компиляции и компоновки объединены в одну программу, в простейшем случае наличие объектного модуля незаметно. Вообще говоря, он создается всегда, ведь без компоновки, во-первых, невозможно определить адреса ячеек памяти, находящиеся впереди компилируемого в данный момент кода (например, если в программе функция вызывает функцию, код которой в исходном коде находится ниже по тексту: неизвестно сколько байт займет еще не компилированный код) — поэтому всякая компиляция из исходного кода в исполняемый проводится в два этапа. Во-вторых, компилятор не может выставить адреса ячеек памяти, приходящих из других модулей и библиотек, на месте таких адресов в объектном модуле оставляются пронумерованные пропуски, а компоновщик заполняет их нужными значениями.

Типы данных

В соответствии с архитектурой фон Неймана данные и команды в современных компьютерах кодируются в двоичной системе счисления.

Тип данных — абстракция, характеризующая множество допустимых значений данных и их представление (кодирование) в памяти, а также набор операций, которые можно совершать с этими значениями.

(Слово “множество” здесь употребляется в строгом математическом смысле. Множество допустимых значений данных может быть конечным или счетно-перечислимым, хотя на практике оно будет ограничено размерами доступной памяти или адресного пространства.)

Тип данных в языке программирования задает, во-первых, какой именно размер в памяти (число байтов) занимает значение данного типа — это число также называется **длиной**, во-вторых, определяет как именно

могут и не могут обрабатываться данные этого типа (например, целые и вещественные числа складываются по-разному).

Типизированные языки программирования — языки, поддерживающие типы данных. Различают языки со **статической** и **динамической** типизацией: в первых типы данных определяются на стадии написания кода программы (компиляции), во вторых — во время выполнения программы. Типизация также может быть **строгой** (также называемой сильной) и **нестрогой** (также называемой слабой). В первом случае гарантируется отсутствие ошибок на стадии исполнения программы ввиду строгих проверок соответствия типа данных в процессе компиляции, то есть **типобезопасность**, во втором есть возможность обойти подобные проверки или они не всегда производятся.

Типы данных разделяют на **простые** (или **примитивные**, **примитивы**) — базовые, встроенные в язык, не строящиеся через другие типы, и **составные** (или **структурные**) — типы, строящиеся через другие типы данных, часто определяемые пользователем.

К составным типам данных относят, например, **массивы** — комбинации значений одного или различных типов, называемых **элементами массива**, доступ к которым осуществляется по **индексу**. Индекс при этом может является либо целым числом (**индексные массивы**), либо данными другого типа, например строкой (**ассоциативные массивы**). Заметим, что только индексные массивы однотипных элементов с плотным (без пропусков) следованием целочисленных индексов являются достаточно легко реализуемыми программно (абстракциями низкого уровня), ассоциативные массивы требуют реализации сравнительно сложных алгоритмов. При этом практически во всех случаях значение индекса может быть как указано в коде программы, так и представлять собой значение, хранимое в памяти (быть данными программы, храниться в переменной и т.п.).

Другой пример составного типа данных — **структура**, представляющая собой комбинацию элементов различных типов данных, называемых **полями**, доступных по **имени**. Обычно имя поля указывается в коде программы в явном виде и не может являться данными программы (что отличает структуру от ассоциативного массива разнотипных элементов).

В языках программирования высокого уровня все данные относятся к одному из типов данных. Также используется понятие **переменной** — поименованной ячейки памяти (фактически метки), характеризующейся определенным типом данных, а значит и длиной (размером ячейки). Адрес переменной (ячейки памяти) вычисляется компилятором автоматически.

При этом само значение номера ячейки памяти — **адреса** — представляет собой особый тип данных. Его также можно сохранить в переменной. Переменная, содержащая адрес другой переменной (ячейки памяти) называется **указателем**. С помощью указателя можно получить доступ к данным в той ячейке памяти, на которую он указывает.

Массивам, структурам и указателям посвящены соответствующие параграфы данного пособия.

Сравнение и классификация языков программирования

Сравнение низкоуровневых и высокоуровневых языков можно представить в виде таблицы:

Возможность	Машинный код	Мнемокод	Ассемблер	ЯВУ
человекочитаемый код	нет	да	да	да
необходимость трансляции	нет	один проход	да	да
автоматическое вычисление адресов	нет	нет	да	да
платформозависимость	да	да	да	необязательно
использование функций и библиотек	возможно	возможно	возможно	да
собственная библиотека языка	нет	нет	нет	да
контроль типов данных	нет	нет	нет	да
логическая организация программы	нет	нет	возможно	да

Считается, что программа, написанная на языке программирования высокого уровня, будет работать медленнее, чем программа, написанная на ассемблере или в кодах. Отчасти такое мнение справедливо, но следует учесть, что современные трансляторы поддерживают **оптимизацию** кода и учет ряда особенностей современных процессоров, что позволяет создавать более быстрый код, чем написанный на ассемблере напрямую. Вероятно, идеальный код на ассемблере будет работать быстрее, чем идеальный код того же самого кода алгоритма, написанный на языке программирования высокого уровня, но, во-первых, создание первого требует многократно бо́льших трудозатрат и несопоставимо большей квалификации программиста (труднодостижимой в современных условиях), а во-вторых, в виду регулярного появления новых процессоров и архитектур такой код будет устаревать гораздо быстрее, чем код, написанный на языке высокого уровня. В то же время различные языки программирования высокого уровня обладают разной эффективностью, и одна и та же программа, напи-

санная на разных языках, будет работать с разной скоростью. Интерпретируемые языки в этом смысле обычно медленнее компилируемых еще и потому, что требуется время на трансляцию кода в процессе исполнения.

К настоящему времени создано несколько сотен или даже тысяч языков программирования высокого уровня. Часть из них не получили практического применения, некоторые были популярны, но к настоящему времени практически не используются (мертвы), некоторые продолжают использоваться и сейчас. "Наилучшего" (абсолютно универсального) языка программирования не существует, для различных задач могут подходить разные языки, как разные инструменты решения этих задач. Языки программирования классифицируют по ряду признаков:

- по уровню — низкого (ассемблер) и высокого;
- по назначению — общего (C, C++, JAVA) и специального (PHP для web-разработки, TeX для форматирования текстов);
- по используемой парадигме (совокупности идей, принципов и понятий, определяющих организацию вычислений и структуру программы) — процедурно-модульный (C, Pascal), объектно-ориентированный (C++, JAVA), функциональный (РЕФАЛ, lisp) и др.;
- по использованию — промышленный (C, C++, PHP), учебный (КуМир, отчасти BASIC, PASCAL), эзотерический (для исследования возможностей программирования, например язык FALSE);
- по целевой аудитории — для программистов (C, JAVA), для пользователей (VB Script в MS Office), для инженеров (Fortran),
- и по другим признакам: со строгой и нестрогой типизацией, по уровню безопасности получаемого кода и т.д.

Важной характеристикой языков программирования является **полнота по Тьюрингу** — свойством языка, означающим, что на данном языке возможна программная реализация решения любой алгоритмически разрешимой задачи (вычисление алгоритмически вычислимой функции). Все указанные выше языки полны по Тьюрингу, включая язык TeX, который, хотя и относится к языкам разметки документов, в этом смысле может считаться полноценным языком программирования, в отличие, например, от другого языка разметки документов — HTML, который полным по Тьюрингу не является.

Вопросы для самопроверки.

1. *Что такое тип данных?*
2. *Зачем нужны типы данных?*
3. *Что определяет тип данных?*
4. *Что такое интерфейс?*
5. *Что такое абстракция, уровень абстракции?*
6. *Какие уровни абстракции можно выделить?*

§1.7. Разработка программного обеспечения

В принципе для создания программы на языке высокого уровня достаточно текстового редактора и компилятора (с компоновщиком, отдельно о нем в большинстве случаев говорить не будем, подразумевая этот процесс неотъемлемой частью получения исполняемого кода).

Однако на практике часто применяется еще и **отладчик** — интерпретатор особого толка (часто работающий с исполняемым кодом программы), позволяющий исполнять программные инструкции пошагово, отслеживая изменения данных программы в процессе ее работы.

Кроме того, используется **интегрированные окружения (среды) разработки**, объединяющие в себе редактор и интерфейс для вызова компилятора и отладчика, запуска программы, визуализирующий процесс отладки.

Жизненный цикл программного обеспечения

Традиционно выделяют следующие этапы разработки программы на языке высокого уровня, называемые также **жизненным циклом программного обеспечения**.

1. Постановка задачи (формулирование на естественном языке или подготовка технического задания).
2. Моделирование (решение задачи на математическом уровне, разработка вычислимой модели).
3. Алгоритмизация (выбор алгоритма решения задачи).

4. Проектирование (разработка структуры программы, разделение на части, функции, модули и т.п.).
5. Кодирование (написание исходного кода программы).
6. Тестирование и отладка (поиск и устранение ошибок).
7. Анализ результатов (перевод их на естественный язык и т.п.).
8. Документирование (подготовка сопроводительной информации для пользователей).
9. Публикация (передача заказчику и т.п.)
10. Сопровождение (консультации пользователей, модификация программы, исправление ошибок и др.).
11. Конец жизненного цикла (прекращение поддержки, вывод из эксплуатации).

Замечание. Несмотря на то, что документирование в этом списке находится в конце разработки, во многих случаях разумно начинать с документирования (на стадии постановки задачи или проектирования), уточнять документацию в процессе кодирования, и лишь оформлять ее по завершении разработки. Хорошей практикой является написание программы в соответствии с требованиями к функционалу и интерфейсу, а не наоборот, поэтому значительная часть документации в каком-то смысле первична.

Технология программирования

Технология программирования — набор методов и средств для создания программного обеспечения.

Технология — термин относящийся к созданию методов промышленного производства.

Научный-технический прогресс, хотя и начинается от требований практики и анализа данных опыта и наблюдений, в дальнейшем часто проходит этапы от науки (фундаментальной) к прикладной науке, далее к технологии (применение научного знания для решения практических задач) и созданию техники ("инженерии") — причем между этими этапами нередко проходят десятки и сотни лет. Так, математические основы программирования были заложены в 19 веке, алгоритмов — в середине 20 века, цифровые технологии вошли в жизнь людей в конце 20 века.

Программирование — изначально и наука, и искусство, так как требует и развития математического инструментария, и творческого подхода при разработке. Любое создание чего-то нового есть творение. Также часто требуется создание видеографических и звуковых образов, сопровождающих программу. Это, однако, в современной технологии является задачей не программиста, а соответствующих специалистов. Сейчас различные аспекты программирования охватывают весь диапазон от фундаментальной науки до техники.

Вопросы для самопроверки.

1. *С какими проблемами сталкивается программист при написании программы на машинных кодах?*
2. *Что такое язык ассемблера и какие задачи он решает?*
3. *Что такое язык программирования высокого уровня?*
4. *Что такое транслятор и какие виды трансляторов бывают?*
5. *Чем отличается компилятор от интерпретатора?*
6. *Чем компиляция отличается от компоновки?*
7. *Какие виды языков по их происхождению бывают?*
8. *Что такое синтаксис?*
9. *Что такое семантика?*
10. *Как классифицируются языки программирования?*
11. *Что такое отладчик и зачем он нужен?*
12. *Что такое интегрированная среда разработки?*
13. *Какие этапы разработки программного обеспечения выделяют?*
14. *Что такое технология программирования?*

§1.8. Цифровое представление данных

Представление целых чисел

Целые числа представляются целым количеством байтов n, обычно n — степень двойки, такой формат называется представлением **фиксированной длины**. Под *длиной*, соответственно, понимается число битов (или байтов), отводимых под представление числа.

Целые числа без знака. При использовании n байтов можно представить числа от 0 до $2^{8n} - 1$ (0..255, 0..65535, …) естественным образом.

Целые числа со знаком.

Для отрицательных чисел существует несколько вариантов представления.

Прямой код. Отводится один бит под знак (0 — плюс, 1 — минус), далее записывается абсолютная величина (модуль) числа, как обычное положительное число. Диапазон возможных значений — от $-2^{8n-1} + 1$ до $2^{8n-1} - 1$ (-127..127, -32767..32767, …), при этом возникает положительный и отрицательный ноль.

Обратный код. Для представления отрицательного чисела все разряды двоичного представления модуля числа инвертируются. Диапазон возможных значений аналогичен прямому коду, также присутствует положительный и отрицательный ноль.

Дополнительный код. Для представления отрицательного числа, все разряды двоичного представления модуля числа инвертируются, как при обратном коде, но к результату добавляется 1. Диапазон возможных значений — от -2^{8n-1} до $2^{8n-1} - 1$ (-128..127, -32768..32767, …) — ноль оказывается положительным числом. Этот способ удобен тем, что позволяет проще реализовать сложение и вычитание. Сложение (как отрицательных, так и положительных чисел) производится так, как будто это числа фиксированной длины без знака. Если для представления результата сложения длины не достаточно, старшие биты отбрасываются. Операция получения противоположного числа реализуется очевидным образом по определению. Вычитание заменяется сложением с противоположным числом. Именно поэтому в современных процессорах чаще всего используется именно дополнительных код.

Можно показать, что при таком подходе операции сложения и вычитания являются согласованными, то есть дают корректный результат при условии, что этот результат входит в диапазон допустимых значений типа данных заданной длины.

Пример: представление в однобайтовом знаковом числе ($n = 1$) числа -9:

$$-9 = -00001001 = 11110110 + 1 = 11110111.$$

Вычитание $4 - 9$:

$$00000100 + 11110111 = 11111011 = 11111010 + 1 = -00000101 = -5.$$

С операциями над целыми числами связаны три проблемы, три ситуации, приводящие к ошибкам.

1. Переполнение. Проблема существует из-за представления в виде чисел фиксированной длины. Из-за того, что отрицательные числа могут быть представлены разными способами, то, что результат переполнения будет одинаков на всех компьютерах, не гарантируется.

   Например, в однобайтовом представлении для беззнаковых чисел вычисления производится в классе вычетов по модулю 256:

   $$200 + 100 = 11001000 + 01100100 = 00101100 = 44.$$

   Для чисел со знаком в обратном дополнительном коде возможна такая ситуация:

   $$100 + 100 = 01100100 + 01100100 = 11001000 = 11000111 + 1 = -00111000 = -56.$$

   Тот же пример в прямом коде даст ответ -72.

   По умолчанию компьютером проверка на переполнение не производится в целях экономии времени, на разных компьютерах при переполнении результат может быть разным, поэтому переполнения следует избегать.

   Компьютеры устроены таким образом, что операции над целыми числами фиксированной длины производятся *аппаратно* за один шаг. Альтернативой этому подходу является *программная* реализация арифметики над числами произвольной длины (фактически над наборами цифр в некоторой системе счисления) — т.н. **длинной арифметики**, работающей медленнее аппаратной, но ограничивающей размер чисел размерами доступной памяти.

2. Деление на ноль. В целых числах невозможно представить результат такой операции, поэтому, если в случае переполнения будет выдан неверный результат, в случае деления на ноль произойдет аварийная остановка программы.

3. Допустимость числа -2^{8n-1} в n-байтовом знаковом представлении также не гарантируется, так как возможно только в дополнительном коде. Знаковое деление или умножение такого числа на -1 приводит неверному результату или к аварийной остановке. Например, для одного байта результат -128/-1 не может быть представлен (128 выходит за диапазон), поэтому тоже приводит к ошибке, которая выразится либо в некорректном результате (снова -128), либо в аварийной остановке программы в зависимости от вида компьютера, языка программирования, компилятора и возможности провести операцию с более длинным числом.

Представление вещественных чисел

Вещественные числа представляются приближенно в виде чисел **с плавающей точкой** (floating point):

$$r = m \cdot 2^p,$$

где m — мантисса, p — порядок. Например, в десятичной системе — с десятичными p, m и показателем степени — в таком формате числа записываются следующим образом: $21.57 = 2157 \cdot 10^{-2}$, $1100 = 11 \cdot 10^2$ и т.п. (В компьютере также используется запись $2.157E1$, $2157E-2$, $1.1e+3$ и т.д., заглавная или строчная буква "E" — десятичный порядок).

Мантисса и порядок — знаковые целые фиксированной длины, то есть под каждый из них отводится определенное количество битов, чаще всего записываемые в прямом, а не дополнительном коде (т.е. используется 1 бит знака и n битов модуля, позволяя представить числа в диапазоне $-2^n..2^n$ (под мантиссу обычно отводится в несколько раз больше битов, чем под порядок).

Стандарт IEEE 754, определяющий формат чисел с плавающей точкой, вводит дополнительные значения: **Бесконечность** (положительная, отрицательная — $\pm$Inf), **не число** (NaN), положительный и отрицательный ноль.

Самое маленькое по модулю ненулевое число — $\pm 1 \cdot 2^{-p_{max}}$, где p_{max} — максимальное число, которое может быть записано в отведенные под модуль порядка биты. Все ненулевые числа, меньшие его по модулю, округляются до нуля, такое округленное значение называется **машинным нулем** — нулем, получаемым компьютером, но не являющимся таковым строго математически. Отрицательный ноль возникает при округлении слишком малого по модулю отрицательного числа, то есть, когда получаемое значение строго отрицательное, но больше самого маленького строго отрицательного числа, которое может быть представлено в формате с плавающей точкой.

Бесконечность является результатом деления на ноль или при возникновении слишком большого числа (переполнении, превышении максимально возможного значения порядка). Знак бесконечности определяется естественным образом, в т.ч. при делении ненулевого числа на отрицательный ноль. Результатом арифметических действий над бесконечностью и конечным ненулевым числом является бесконечность.

При делении нуля на ноль, при сложении разнознаковых бесконечностей, делении бесконечности на бесконечность и др. действиях возникает значение, известное в математике как "неопределенность", которая в IEEE 754 обозначается как *не число*.

Арифметические действия с бесконечностью дают бесконечность или не число (например, при делении двух бесконечностей, сложении разнознаковых бесконечностей), арифметические действия с не числом всегда возвращают не число. Сравнение бесконечности и чисел проводится естественным образом, но не число не равно ни какому-либо числу, ни бесконечности, ни даже самому себе.

Если целочисленное деление на ноль приведет к аварийной остановке программы, то никакая операция с вещественными числами не приводит к ошибке, результат есть всегда. Это несомненный плюс, так как во многих случаях программы не должны останавливаться никогда (например, если они управляют реакторами, самолетами и т.п.), однако важно правильно учитывать возможность появления такого результата.

Подробные таблицы арифметических операций и операций сравнения для расширенных чисел с плавающей точкой приведены в приложении B.2.

Замечание. Иногда вместо NaN используется IND (недетерминированный).

Примеры.

- Предположим, что используется десятичное представление чисел, 2 разряда отведено под порядок и 3 под мантиссу (не считая знака). Тогда произведение чисел $143 \cdot 10^{80}$ и $-12 \cdot 10^{40}$ будет отрицательной бесконечностью, так как порядок 10^{120} непредставим. Сумма $456 \cdot 10^{-2} + 742 \cdot 10^{-2} = 1198 \cdot 10^{-2}$, что не может быть представлено из-за ограничения мантиссы, поэтому округлится до $120 \cdot 10^{-1}$. Сумма $125 \cdot 10^{-2} + 121 \cdot 10^{-4} = 1.25 + 0.0121 = 1.26121$, что округлится до $1.26 = 126 \cdot 10^{-2}$, то есть младшие разряды числа меньшего порядка будут утеряны.

- Пусть
$$a = 10^{20}$$
$$b = -10^{20}$$
$$c = 1$$
Тогда
$$(a+b)+c = (10^{20} - 10^{20}) + 1 = 1,$$
но
$$a + (b+c) = 10^{20} + (-10^{20} + 1) = 10^{20} + (-10^{20}) = 10^{20} - 10^{20} = 0,$$
то есть $(a+b)+c \neq a+(b+c)$.

  В последнем варианте при вычислении $-10^{20} + 1$ в случае, когда разрядность мантиссы меньше необходимых для хранения результата в точном виде 21 десятичной цифры, что это для современных компьютеров практически всегда так, результат будет округлен до -10^{20}. То есть данный пример, в отличие от предыдущего, не привязан к системе счисления представления и может наблюдаться на реальных компьютерах. Поскольку какова бы ни была разрядность мантиссы (число представимых p-ичных цифр), порядок практически всегда позволяет хранить значение, большее этого числа, поэтому подобрать достаточно большой порядок для наблюдения данного феномена в условиях реальной вычислительной системы, работающей с числами с плавающей точкой, можно практически всегда.

- Рассмотрим число 1/5, представимое конечной десятичной дробью 0.2. В двоичной системе эта дробь является периодической $0.01100110011\cdots = 0.(0110)$. Предположим, что в мантиссе 8 двоичных разрядов. Тогда это число будет представлено как
$$11001100_2 \cdot 2^{-10} = (2^7 + 2^6 + 2^3 + 2^2) \cdot 2^{-10} = 204/1024 = 0.19921875,$$
если 10 разрядов
$$1100110011_2 \cdot 2^{-12} = (2^9 + 2^8 + 2^5 + 2^4 + 2^1 + 2^0) \cdot 2^{-12} = 819/4096 = 0.199951171875.$$

  При передаче из переменной с меньшей точностью в большую, значение в первой переменной будет дополнено нулями и превращено в 1100110000, что не равно значению, получаемому сразу при вычислении с большей точностью.

  При циклическом прибавлении таких чисел абсолютная ошибка будет накапливаться:

$$11001100 \cdot 2^{-10} = 0.19921875 \text{ (погрешность } 0.00078125,\ 0.39\%)$$
$$11001100 \cdot 2^{-10} + 11001100 \cdot 2^{-10} = 11001100 \cdot 2^{-9} = 0.3984375 \text{ (погрешность } 0.0015625,\ 0.39\%)$$

$$11001100 \cdot 2^{-9} + 11001100 \cdot 2^{-10} = 10011001 \cdot 2^{-8} = 0.59765625 \text{ (погрешность } 0.00234375,\ 0.39\%)$$
$$10011001 \cdot 2^{-8} + 11001100 \cdot 2^{-10} = 11001100 \cdot 2^{-8} = 0.796975 \text{ (погрешность } 0.003025,\ 0.38\%)$$
$$11001100 \cdot 2^{-8} + 11001100 \cdot 2^{-10} = 11111111 \cdot 2^{-8} = 255/256 = 0.99609375 \text{ (погрешность } 0.00390625,\ 0.39\%)$$

  Если исходное число сразу умножить на 5 (101), то также получится
$$1111111100 \cdot 2^{-10} = 11111111 \cdot 2^{-8} = 0.99609375 \neq 1.$$
  Относительная погрешность в данном примере не растет, хотя бывают ситуации, когда и это не так.

  В реальном компьютере результат может быть не таким, за счет округления и автоматических поправок есть шанс, что ошибка будет нивелирована, и на простых примерах такая погрешность не всегда видна, но риск всегда существует. В подобных ситуациях циклическое прибавление 0.2 к 0 пока результат меньше 1 может остановиться на 7 шаге, приближенно равным 1.2, а не на 6, приближенно равным 1.0, поэтому в таких случаях следует использовать целочисленный счетчик i и вычислять $0.2 \cdot i$.

Таким образом видно, что *нельзя слепо верить числу, полученному компьютером, надо учитывать погрешность.* При работе с вещественными числами необходимо учитывать следующие ограничения.

1. Переполнение порядка. Приводит к появлению бесконечности, даже если математически результат является конечным числом.

2. Возможность возникновения в результате NaN с последующим продолжением выполнения программы и сохранением данного "ответа".
3. Ошибки округления и потеря точности. Мантисса переполниться не может, в этом случае просто увеличивается порядок, но происходит потеря точности — младшая цифра "исчезает". В частности, *сложение чисел с плавающей точкой не является ассоциативным!* Даже рациональные числа за редким исключением нельзя представить точно в формате чисел с плавающей точкой. Число m/n, где m и n целые, взаимно простые, $n > 0$, может быть представлено конечной p-ичной дробью, если все простые множители знаменателя являются делителями основания системы, т.е. если $n = p_1 \cdot p_2 \cdot \dots \cdot p_k$, где p_i — простые числа (такое разложение единственное), то p_i должно быть делителем основанием системы p, иначе дробь будет бесконечной периодической.
4. Зависимость результата не только от способа представления данных в компьютере (например, если не используется стандарт IEEE 754), но и от настроек работы процессора и компилятора, например, выбором стратегии округления и оптимизации кода: арифметические выражения могут преобразовываться к математически равнозначному для вещественных чисел, но не являющемся таковым для чисел с плавающей точкой.

Как следствие, операции сравнения вещественных чисел в общем случае некорректны из-за ошибок округления и представления чисел. Вместо них, можно использовать сравнение с некоторой точностью, например считать, что $a = b$ если $|a-b| < \varepsilon$, где ε мало. Однако например $\varepsilon = 0.0001$ можно считать малым по сравнению с числами ~ 10–1000, но если a и b сами малы $\sim 10^{-20}$, то такое ε в качестве погрешности использоваться не может. Наоборот, если числа слишком большие, $\sim 10^{50}$, то малой может считаться погрешность в 10^{40}, а 10^{10} сведется к машинному нулю. Как вариант, в общем случае можно использовать относительные погрешности и другие методы. Например, относительная погрешность может вычисляться таким образом $|(a-b)/(max(a,b)| < \varepsilon$. Заметим, что, универсальных рецептов не существует и нужно учитывать особенность конкретной задачи (погрешность относительно исходных данных, относительно характерных величин, приемлемого для задачи уровня и т.д.).

Математическая теория (аксиоматика) для данной алгебры, вообще говоря, не разработана.

Кроме того, двоичное представление чисел с плавающей точкой зависит от платформы и системы: переносимости данных на двоичном уровне (при сохранении в файл и т.д.) в общем случае нет. Дополнительные сведения о представлении и переносимости чисел представлены в приложении B.1 и B.3, с которыми рекомендуется ознакомиться после изучения типов данных языка Си и работы с двоичными файлами соответственно.

Представление символов

Изначально один символ представлялся одним байтом. Так можно представить всего 256 различных символов. Этого достаточно, чтобы закодировать управляющие символы, буквы латинского алфавита, буквы одного из дополнительных языков, знаки препинания и т.п., но недостаточно чтобы закодировать все буквы всех языков. Последняя проблема решается выбором кодировки символов.

Кодировка символов *(codepage)* — таблица соответствия номера (кода) и символа алфавита (буквы, цифры, знака и др.).

В памяти компьютера нет символов, есть только коды. Символы появляются тогда, когда их нужно отобразить, т.е. сопоставить коду графический знак. Выбранная кодировка символов отвечает за то, какой символ алфавита будет отображен, а **шрифт** — то, как этот символ будет выглядеть.

Первая половина кодировки (коды 0–127) постоянная, содержит управляющие символы, буквы латиницы, знаки препинания и цифры. Вторая половина — переменная, содержит символы псевдографики, буквы других языков. Кодировки русского языка: cp866 (кодировка DOS, в Windows используется в консольных приложениях), cp1251 (кодировка Windows в графических приложениях), koi8-r (кодировка Linux) и др.

Существуют кодировки, где одному символу может сопоставляться несколько байтов: Unicode (UTF-16 — фиксировано 2 байта, UTF-8 от 1 байта для первой половины таблицы и последовательности из 2 и более байтов, начинающихся с символов с кодом от 128 до 255) — достаточно для представления практически всех символов практически любых известных сейчас алфавитов, включая иероглифы.

Представление строк

Существуют различные способы представления строк в компьютере. Два простейших способа из них — Си-строки и паскалевские строки.

Первый способ состоит в том, что строке отводится необходимое количество байтов (если длина строки известна заранее) или с запасом (если в момент написания программы длина строки неизвестна, например, строка будет введена с клавиатуры), плюс 1 символ, концом строки считается управляющий символ с кодом 0. Недостаток: нельзя создать строку, содержащую символ с кодом 0. Однако, данный недостаток за редким исключением несущественен, так как символ с кодом ноль иначе как для обозначения конца строки не используется.

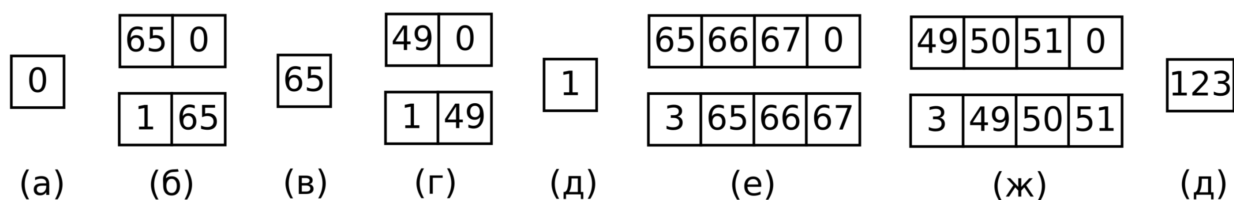


Рис. 1.7: Примеры представление строк и символов, клетки символизируют байты, числа в них — десятичный код символа; (а) — пустая строка (Си и паскаль), (б) — строка с символом “A” (Си вверху, паскаль внизу), (в) — символ “A”, (г) — строка с числом 1 (Си вверху, паскаль внизу), (д) — число 1 (1 байт), (е) — строка “ABC” (Си вверху, паскаль внизу), (ж) — строка “123” (Си вверху, паскаль внизу), (з) — число 123 (1 байт).

Второй способ: под каждую строку отводится фиксированное количество байтов (256), в нулевом байте хранится фактическая длина строки (число от нуля до 255 — 1 байт). Недостаток: длина строки ограничена.

Общий недостаток этих методов — необходимость отведения места под “лишние” символы остатка строкового буфера в случае, когда длина строки на момент написания программы неизвестна.

Примеры: пустая строка (в Си — просто символ с кодом 0, в паскале — аналогично), строка из одного символа (в Си — код символа и символ с кодом ноль, в паскале — символ с кодом 1 и код требуемого символа), примеры также приведены на рис. 1.7. Следует обратить внимание на разницу в представлениях между символом и строкой, содержащей этот символ, числа и строкового представления числа.

Логические значения

Логические данные — данные, которые могут принимать ровно два значения, истина и ложь.

Например, результат операции сравнения есть либо истина, либо ложь. Над самими логическими данными также возможны особые алгебраические операции — логические операции. Наиболее существенные для программирования — отрицание (¬), конъюнкция (&, “логическое и”), дизъюнкция (∨, “логическое или”), эквивалентность (≡, “равенство”, “равнозначность”), неравнозначность (⩔, “разделительное или”).

Это операции на двухэлементном множества $\{И, Л\}$, подобно арифметическим операциям над числами (на множестве натуральных чисел, например), преобразуют элементы данного множества.

Таблицы операций выглядит следующим образом:

	И	*Л*
¬	*Л*	*И*

Отрицание одного значения истинно тогда и только тогда, когда значение ложно.

&	*И*	*Л*
И	*И*	*Л*
Л	*Л*	*Л*

Конъюнкция двух значений истинна тогда и только тогда, когда оба значения — истина.

∨	*И*	*Л*
И	*И*	*И*
Л	*И*	*Л*

Дизъюнкция двух значений ложна тогда и только тогда, когда оба значения ложны.

≡	*И*	*Л*
И	*И*	*Л*
Л	*Л*	*И*

Эквивалентность двух значений истинна тогда и только тогда, когда оба значения равны.

⩔	*И*	*Л*
И	*Л*	*И*
Л	*И*	*Л*

Разделительное или двух значений истинно тогда и только тогда, когда оба значения различны.

Для операций существует приоритет: ¬, &, ∨ и ⩔, ≡. Таким образом $A \vee B \,\&\, C$ то же самое, что $A \vee (B \,\&\, C)$, а не $(A \vee B) \,\&\, C$.

Чаще для логических операций и значений логических выражений составляют т.н. **таблицы истинности** — в первых столбцах таблицы перебираются все возможные комбинации значений переменных, в последнем — значение выражения. Дело в том, что любое логическое выражение, в данном случае рассматриваемое как логическая функция n переменных, имеет конечное число — 2^n — возможных комбинаций значений аргументов.

Пример: таблица истинности для дизъюнкции и конъюнкции:

A	B	$A \vee B$	$A \& B$
И	*И*	*И*	*И*

- **законы коммутативности:**
  (1) $A \vee B = B \vee A$;
  (2) $B \,\&\, A = A \,\&\, B$;
  (3) $B \equiv A = A \equiv B$;
- **законы ассоциативности:**
  (4) $A \vee (B \vee C) = (A \vee B) \vee C$;
  (5) $A \,\&\, (B \,\&\, C) = (A \,\&\, B) \,\&\, C$;
- **законы дистрибутивности:**
  (6) $A \,\&\, (B \vee C) = (A \,\&\, B) \vee (A \,\&\, C)$;
  (7) $A \vee (B \,\&\, C) = (A \vee B) \,\&\, (A \vee C)$;
- **законы де Моргана:**
  (8) $\neg(A \vee B) = \neg A \,\&\, \neg B$;
  (9) $\neg(A \,\&\, B) = \neg A \vee \neg B$;
- **закон двойного отрицания:**
  (10) $\neg\neg A = A$;
- **законы поглощения:**
  (11) $A \vee (A \,\&\, B) = A$;
  (12) $A \,\&\, (A \vee B) = A$;
- **закон исключенного третьего:**
  (13) $A \vee \neg A = И$;
- **закон противоречия:**
  (14) $A \,\&\, \neg A = Л$;
- **законы логических констант:**
  (15) $A \vee И = И$;
  (16) $A \vee Л = A$;
  (17) $A \,\&\, И = A$;
  (18) $A \,\&\, Л = Л$;
- **законы логических связок:**
  (19) $A \,\&\, A = A$;
  (20) $A \vee A = A$;
  (21) $A \equiv A = И$;
  (22) $A \veebar A = Л$;

Таблица 1.2: Законы логических операций.

Другой пример — таблица истинности для выражения $\neg A \equiv B \& C$, вычисленная по действиям:

A	B	C	$\neg A$	$B \& C$	$\neg A \equiv B \,\&\, C$
И	И	И	Л	И	Л
И	И	Л	Л	Л	И
И	Л	И	Л	Л	И
И	Л	Л	Л	Л	И
Л	И	И	И	И	И
Л	И	Л	И	Л	Л
Л	Л	И	И	Л	Л
Л	Л	Л	И	Л	Л

Для данных операций верны следующие законы, приведенные в таблице 1.2.

Данные законы можно доказать путем составления таблиц истинности для левой и правой части — значения соответствующих логических функций совпадают.

Для понимания способа представления логических данных в компьютере в первом приближении можно считать, что ложь кодируется нулем, истина единицей, однако в разных языках программирования и платформах используются различные способы. Для такого представления достаточно одного бита, но в большинстве случаев бит — неадресуемая ячейка памяти. Поэтому используется как минимум один байт, реально чаще всего больше из соображений скорости доступа к определенным участкам памяти (словам).

В данном параграфе намеренно использовались общематематические, а не специфические для языков программирования, обозначения для логических операций и констант, чтобы подчеркнуть первичность, универсальность и независимость от программирования данных математических законов, однако следует учесть, что и в математике такая система обозначений не единственная.

Вопросы для самопроверки.

1. *Как представляются целые числа в компьютере?*

2. *Как моделируются вещественные числа в компьютере?*
3. *Что такое числа с плавающей точкой?*
4. *Какие ошибки могут возникнуть при работе с целыми числами?*
5. *Какие ошибки могут возникнуть при работе с числами с плавающей точкой?*
6. *Что такое код символа?*
7. *Что такое таблица символов (кодовая страница)? Что от нее зависит?*
8. *Что такое шрифт?*
9. *Чем отличаются Си-строки от паскалевских строк? Какие преимущества и недостатки у этих способов представления строк?*
10. *Поясните различие между строкой, содержащий символ "цифра 0", символом "цифра 0" и числом 0.*
11. *Перебирая всевозможные комбинации значений логических переменных, докажите приведенные логические законы.*

Глава 2. Синтаксис языка Си

§2.1. Общие сведения о языке Си

Язык программирования **Си** (**C**) — компилируемый язык программирования высокого уровня общего назначения. Разработан в 1980-х годах для нужд ОС Unix, в настоящее время является кросс-платформенным, то есть исходный код программы, написанный с использованием только стандартизированных средств языка (без использования системно-зависимых библиотек, компиляторо-зависимых расширений и т.п.), может быть скомпилирован в исполняемый код на любой платформе, для которой существует реализация языка Си.

Иногда Си называют языком низкого или "среднего" уровня, поскольку он близок к машинным, ассемблерным и системным конструкциям, но, следуя определению, Си является языком программирования высокого уровня. Однако Си действительно приближенный к машинным и системным инструкциям язык, что позволяет использовать его для системного программирования, большинство современных операционных систем написаны на Си и их системные функции представлены на Си. Это же делает язык "быстрым", т.е. программа, написанная на Си, будет работать быстрее, чем аналогичная, написанная с использованием большинства других языков, что делает данный язык привлекательным для проведения больших объемов вычислений, расчетов и численного моделирования.

Язык Си разработан в 80-х годах 20 века и имеет устоявшееся ядро, за время его существования создано большое количество библиотек, что позволяет реализовывать на нем в том числе и прикладные программы, программы с графическим интерфейсом для различных графических окружений и систем, серверá, языки программирования и др. виды приложений. Язык де-факто стал эталонным, так как лег в основу многих других языков программирования, под него разработаны многие стандарты форматирования кода, средства проверки, отладки, документирования программ и т.д.

Другие преимущества и недостатки языка Си вообще и его выбора для данного курса можно свести к следующему.

1. Обучение языку Си полезно как минимум по трем причинам.

   Во-первых, благодаря своему условно низкому уровню он позволяет вручную проработать, увидеть и проконтролировать сложность высокоуровневых конструкций. Можно условно ориентироваться на следующее обстоятельство: *то, что сложно реализовать в Си (в смысле количества кода и сложности применяемых языковых конструкций) будет выполняться долго и наоборот, то, что пишется легко и коротко, выполняется быстро*, это дает интуитивное понимание расхода ресурсов создаваемой программы. В этом, однако, есть и недостаток: создание современных высокоуровневых прикладных программ в Си может быть более трудоемкой задачей, чем на некоторых современных языках общего и прикладного назначения.

   Во-вторых, Си лег в основу многих новых и активно развивающихся и использующихся языков программирования, в т.ч. C++, C#, JAVA, PHP — зная Си изучить их будет гораздо проще.

   В-третьих, Си "живой" промышленный язык, он реально используется в программировании.

2. На самом деле не так важно, какой язык программирования изучать, главное изучить **парадигмы** (значение этого термина объясняется в параграфе 6), приемы и алгоритмы. Алгоритмы можно вообще записывать с помощью т.н. **псевдокода** — фактически на структурированном естественном языке. Квалифицированный программист должен уметь реализовывать алгоритм, написанный на псевдокоде, на любом

известном ему языке программирования, а изучение новых языков не должно представлять сложности: сложнее всего изучить первый язык, позже становится понятно, что в языках, основанных на одинаковых парадигмах, используются фактически одни и те же понятия, пусть даже они и выглядят по-разному, с некоторыми отклонениями в деталях.

В данном курсе псевдокод не используется, алгоритмы записываются на языке Си.

3. Основная сложность (в некотором смысле недостаток) языка Си в том, что в самом языке отсутствуют средства автоматической проверки на многие ошибки выполнения программы. Си — небезопасный язык, согласно известному выражению, “*Си позволяет выстрелить себе в ногу*”, в нем нет предохранителя, “защиты от дурака”. Конкретика этих слов будет понятна лишь по мере изучения языка (примерами могут послужить отсутствие проверки на выход индекса элемента массива за диапазон, обращение к памяти, переполнение и т.д.). Важно, что Си требует повышенной внимательности. Это обстоятельство является и положительным моментом при обучении основам программирования, так как программист должен быть внимательным, в том числе внимательным к деталям и мелочам. Плюсом данной стороны языка для промышленного программирования является тот факт, что лишние проверки замедляют работу программы, а предназначение Си — быть “быстрым”, то есть экономить ресурсы. Также имеется концептуальная причина, по которой в язык Си (как и в ОС Unix) не была включена “защита от дурака” — запрещая программисту (пользователю) делать “глупые” вещи (то есть ограничивая свободу) можно заодно запретить и делать “умные” (полезные), что не есть хорошо.

Си основан на парадигмах структурного и методологии процедурно-модульного программирования, описание которых приводится в главе 6. Си — живой, развивающийся язык. В разные годы утверждались разные стандарты языка Си, обозначаемые именно по годам: C89, C90, C99, C11, C14, C17, C20 и др.

Наиболее базовый язык утвержден институтом ANSI (American National Standards Institute) — ANSI C, он же C89 и именно он поддерживается большинством компиляторов. Стандарт C90 и дальнейшие принят международной организацией по стандартам ISO (ISO C). В настоящее время практически все компиляторы поддерживают стандарт C99 и более новые.

Разные компиляторы могут вносить полную или неполную поддержку стандартов, а также собственные возможности и библиотеки. Использование расширений ANSI C бывает полезно и удобно, но приводит к написанию непортируемого кода, поэтому должно быть четко обосновано.

Этот курс ориентирован именно на ANSI C с отсылками на наиболее существенные расширения, вносимые стандартом C99. Рекомендуется подключение строгого следования одному из этих стандартов в настройках компилятора.

Вопросы для самопроверки.

1. *Каково место языка Си в общей классификации языков программирования?*
2. *Каковы преимущества и недостатки языка Си?*

§2.2. Простые типы данных языка Си

Язык Си — **типизированный**, то есть, в отличие от машинных кодов, где все данные представлены равномерным образом и при каждом действии с ними необходимо следить за тем, что именно закодировано в ячейке по указанному адресу и сколько именно байтов занимают находящиеся там данные, в Си можно указать, что данные имеют определенный тип. После этого Си во многих случаях автоматически выбирает правильный машинный код для обработки данных данного типа, проверяет, корректно ли применяемое действие к данным указанного типа или нет, предупреждая программиста об ошибке или потенциальной ошибке. Си является языком со статической типизацией. Однако Си относится к языкам с *нестрогой типизацией*, т.к. защиту от некорректного применения действия можно обойти, а в некоторых случаях она не предусмотрена вовсе.

В Си простыми типами данных являются типы для представления целых чисел и чисел с плавающей точкой, остальные (строки, массивы, структуры и др.) типы строятся через них.

Имена типов данных в Си могут состоять из одного или нескольких слов, один и тот же тип может иметь как полное многословное имя, так и допускать сокращения до одного-двух слов, т.е. иметь синонимичные названия.

Конкретный размер значения типа в байтах часто не регламентирован стандартном и определяется реализацией языка.

1. “Символьный” тип `char` — целочисленный тип размером строго в один байт (длина строго 1 байт, гарантировано стандартом независимо от используемых кодировок символов), может быть как знаковым, так и беззнаковым (в зависимости от компилятора) и на восьмибитных компьютерах позволяет хранить число от -127 до 127 (включая -128 или -0) или от 0 до 255 соответственно, фактически предназначен для хранения кода символа.

Отдельного типа для хранения символов не существует, в памяти коды и символы неразличимы, различие возникает при их использовании — например, программист может выбрать, что следует выводить: шрифтовое отображение символа или его код.

2. Целочисленные типы.

   (a) Однобайтовый целочисленный
   со знаком: `signed char`;
   без знака: `unsigned char`;

   всегда строго один байт, на самом деле `char` это один из этих двух типов в зависимости от реализации.

   (b) Короткий целочисленный
   со знаком: `signed short int`, `signed short`, `short int`, `short`;
   без знака: `unsigned short int`, `unsigned short`;

   должен быть не короче, чем 2 байта.

   (c) Основной целочисленный
   со знаком: `signed int`, `signed`; `int`;
   без знака: `unsigned int`, `unsigned`;

   должен быть не короче, чем 2 байта, т.е. может как совпадать с `short`, так и быть больше него (4 байта).

   (d) Длинный целочисленный
   со знаком: `signed long int`, `signed long`, `long int`, `long`;
   без знака: `unsigned long int`, `unsigned long`;

   должен быть не короче, чем 4 байта.

   (e) Начиная со стандартна C99 добавляется двойной длинный целочисленный тип со знаком: `signed long long int`, `signed long long`, `long long`; без знака: `unsigned long long int`, `unsigned long long`.
   должен быть не короче, чем 8 байт.

Логика сокращений такова: по умолчанию тип знаковый (кроме `char`), т.е. если не указано ни `signed`, ни `unsigned`, то подразумевается `signed`; основной целочисленный тип — `int`, а `short` и `long` — его модификаторы, по умолчанию относятся к целочисленным типам, поэтому после них `int` можно опускать; после `signed` и `unsigned` также можно опустить слово `int`.

Как правило, в реализациях тип `short` короче (то есть имеет меньшую длину), чем `long`, в то время как `int` может совпадать с одним из них или отличаться от обоих (например, длины `short`, `int`, `long` и `long long` могут быть 16, 32, 32 и 64 или 16, 32, 64 и 64 бита соответственно).

3. Числа с плавающей точкой.

   (a) Одинарной точности `float`;
   отводится 1 бит на знак, 8 битов на порядок, 23 бита на мантиссу — всего 32 бита.

   Мантисса хранит 24 двоичных цифры — старшая цифра не хранится, т.к. подразумевается 1 (принимает значения от 2^{23} до $2^{24}-1$). Двоичный порядок принимает значения от -126 до 127, при этом нулевой порядок дает числа в промежутке $[1,2)$ (при изменении мантиссы от 2^{23} до $2^{24}-1$ включительно). Часть значений порядка зарезервирована для представления бесконечности и не-числа, а также хранения малых чисел: двоичный порядок тоже -126, но без подразумевающейся старшей единицы, мантисса принимает значения от 1 до $2^{23}-1$.

   В десятичном представлении обеспечивает 6-9 значащих цифр и десятичный порядок примерно от -38 до +38 типично, а также $\sim -45..-39$ (для малых чисел). Наименьшее по модулю ненулевое число — $\approx 1.4 \cdot 10^{-45}$, наибольшее по модулю конечное — $\approx 3.4 \cdot 10^{38}$.

   (b) Двойной точности `double` (базовый тип с плавающей точкой);
   отводится 1 бит на знак, 11 битов на порядок (десятичный порядок примерно от -308 до 308, малые числа до порядка -324), 52 бита на мантиссу — 53 двоичных цифры (около 16 десятичных цифр) — всего 64 бита. Наименьшее по модулю ненулевое число — $\approx 4.94 \cdot 10^{-324}$, наибольшее по модулю конечное — $\approx 1.8 \cdot 10^{308}$.

   (c) Повышенной точности `long double`;
   точно не регламентирован, может совпадать с `double` или быть длиннее его.

Отдельных логических и строковых типов в стандарте С99 нет. Строки представляют собой массивы символов (блоки переменных типа `char`, расположенные в памяти непрерывно). В качестве логического может выступать практически любой тип. Значения любого типа интерпретируются как логические следующим образом: **всякое ненулевое значение трактуется как истина, нулевое значение — как ложь.** На практике для хранения логических значений обычно используется тип `int` (а не `char`, выгодный с точки зрения экономии памяти, но более медленный, так как `int` выравнивается по быстродоступным ячейкам оперативной памяти и размеру шины). Кроме того, все стандартные операции и функции в Си, возвращающие логические значения, возвращают именно данные типа `int`.

Стандарт С99 вводит логические тип данных `_Bool` и псевдоним для него — `bool`, а также логические значения `true` и `false`, которые являются псевдонимами для целочисленных 1 и 0 соответственно. При этом сохраняется указанное в предыдущем параграфе правило трактование значений всех типов как логических.

Вопросы для самопроверки.

1. *Какие базовые типы данных есть в Си?*
2. *Каковы свойства типа `char`?*
3. *Как представляются логические значения в Си?*

§2.3. Синтаксические единицы языка Си

Для задания любого языка прежде всего следует ввести алфавит. Алфавит языка Си состоит из следующих символов:

- буквы (только латинского алфавита, a-z и A-Z), символ подчеркивания `_`;
- цифры (арабские) 0-9;
- служебные символы `( ) [ ] { } ; ' " # \ ! & * + - / % < = > | ^ ~ , . ? : ;`
- пробельные символы (пробел, табуляция, перенос строки).

Других символов в тексте программы присутствовать не может (это не относится к строковым и символьным значениям, которые являются данными программы, а не частью программного кода). В частности, это означает, что стандарт не допускает использование русских букв в именах переменных и функций.

Синтаксические единицы — элементы, из которых строится программа на Си — следующие.

- **символ** (символ алфавита языка, текст программы состоит из символов);
- **токен** ("слово", простейшая последовательность символов, имеющая некоторый смысл);
- **выражение** (способ получения данных);
- **декларатор** (способ задания новых идентификаторов);
- **оператор** (минимальная исполняемая единица кода);
- **функция** (фактически обособленный алгоритм, в том смысле, в котором указано в предыдущей главе);
- **файл исходного кода** (в данной главе будет использоваться термин "программа", хотя точнее будет сказать "единица компиляции").

Здесь единицы перечислены от самого низкого уровня организации до самого высокого, исключение декларатор — он находится на одном уровне с оператором или функцией. В следующих параграфах определяются данные синтаксические конструкции языка Си.

Важность понимания синтаксиса языка в том, что при неверном задании языковой конструкции компилятор сообщит синтаксическую ошибку: в этой ситуации требуется проверить корректность кода программы на соответствие синтаксическим нормам.

Вопросы для самопроверки.

1. *Из чего состоит алфавит языка Си?*
2. *Перечислите элементы синтаксиса языка Си.*
3. *Зачем нужно понимание норм синтаксиса языка программирования?*

§2.4. Токены

Токен — минимальная синтаксическая единица языка, имеющая самостоятельный смысл (в отличие от символа).

По сути токен — уточненный эквивалент бытового понятия “слово” естественного языка. В формальном языке под **словом** понимается любая последовательность букв, т.е. любая последовательность букв является корректным словом (в т.ч. и вся программа является одним словом), но не все слова имеют смысл. Минимальные слова, имеющие смысл — токены. Компилятор разбивает код программы (слово-программу) на токены по определенным правилам.

Во многих — но не во всех — естественных языках слова разделяются пробелами. В Си границы токенов могут определяться разделительными символами (пробелами, но не только), а также естественным образом, если очередной символ однозначно не может быть продолжением текущего токена.

В качестве токенов в Си могут выступать следующие объекты:

- **ключевые** (зарезервированные) **слова** — имена операторов, типов данных, модификаторы и др.;
- **идентификаторы** — имена, задаваемые для идентификации определяемых в программе элементов (переменных, функций, типов данных и др.);
- **константы** — явно вводимые в код программы значения (данные: строки, символы, числа и др.);
- **операции** — знаки операций (сложения, умножения, сравнения, логических и др.);
- **пунктуационные знаки** — разделители элементов (запятая, точка с запятой и др).

Этим группам токенов посвящены следующие пункты.

Ключевые слова

Ключевые слова — слова, встроенные в язык программирования.

По синтаксису аналогичны *идентификаторам*, но их семантика встроена в язык, а не определяется программистом. В стандартном Си это следующие слова:

```
auto    break   case     char   const    continue default  do
double  else    enum     extern float    for      goto     if
int     long    register return short    signed   sizeof   static
struct  switch  typedef  union  unsigned void     volatile while
```

Ключевыми словами являются названия типов, имена операторов и другие управляющие элементы, их смысл встроен в язык и не предназначен для переопределения. Смысл конкретных слов будет изучаться последовательно, некоторые оставлены за рамками курса — на данный момент нет необходимости знать их все, важно лишь огласить факт их существования в виду того, что использование ключевых слов в качестве имен переменных, функций и других идентификаторов не допускается. Компиляторы могут добавлять другие ключевые слова.

Идентификаторы

Идентификаторы — слова, смысл которых определяется в программе, это могут быть имена переменных, функций и других объектов.

Смысл идентификаторов в язык не встроен, его определяет программист (в программе можно использовать идентификаторы, которые кажутся встроенными, например, функции стандартной библиотеки, но формально синтаксически это не так, просто к программе подключается необходимое определение).

В качестве идентификатора может выступать любая последовательность букв (включая знак подчеркивания) и цифр, причем цифра не может быть первым символом. В этом смысле ключевое слово по синтаксису является частным случаем идентификатора.

Любой символ, отличный от цифры и буквы, будет означать, что имя идентификатора закончилось и далее следует другой токен (пример случая, когда токены разделяются не с помощью пробелов).

Синтаксические определения чаще всего записывают в **форме Бэкуса — Наура** (сокращенно **БНФ**, Бэкуса — Наура форма). Это математически строгие определения. Их следует уметь читать для понимания синтаксиса языков программирования, приводимого в документации или стандарте, и не только. Существуют различные вариации записи данной формы, обычно понятные интуитивно, в этом пособии используем одну из них. В ней определение идентификатора выглядит так:

⟨*Буква*⟩ ::=
`a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z`
`|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|_`

⟨*Цифра*⟩ ::= `0|1|2|3|4|5|6|7|8|9`

⟨*Идентификатор*⟩ ::= ⟨*Буква*⟩|⟨*Идентификатор*⟩⟨*Буква*⟩|⟨*Идентификатор*⟩⟨*Цифра*⟩

В таком определении в треугольных скобках указывается вводимое в БНФ понятие, знак '::=' читается как "равно по определению", вертикальная черта — символ перебора возможных вариантов, соответствует союзу ИЛИ. Такое определение означает, что *все последовательности символов, удовлетворяющие хотя бы одному из перечисленных вариантов, являются экземплярами определяемого понятия, в то же время другие последовательности символов таковыми не являются.* Таким образом, в определении буквы и цифры сказано, что буквой или цифрой является каждый из перечисленных символов, но никакой другой символ алфавита языка буквой (цифрой) не является.

Определение идентификатора рекурсивное, то есть он определяется через себя: отдельная буква является идентификатором, идентификатор за которым записана буква или цифра, также является (другим) идентификатором, а все остальные последовательности символов, например, отдельная цифра, последовательность цифр, последовательность букв и цифр, начинающаяся с цифры, идентификатором не являются.

По БНФ определению всегда можно проверить, является ли данное слово указанным понятием или нет. Например, `x1` является идентификатором (`x1` — `x` к которому приписано `1`, `1` — цифра; `x` — буква, т.е. `x` является идентификатором; следовательно `x1` — является идентификатором). В то же время `x*` идентификатором не является (`x*` — `x` к которому приписано `*`; `*` не является ни буквой, ни цифрой), так же как и `1x1` (`1x1` — `1x` к которому приписано `1`, `1` — цифра; `1x` — `1` к которому приписано `x`, `x` — буква; `1` не является буквой, не является идентификатором; следовательно `1x1` не является идентификатором).

БНФ также позволяет перечислить все возможные слова, попадающие под данное определение, перебором. Конечно, таких слов может быть счетное множество (счетно-перечислимое), но перечислить можно конечное число слов, длина которых не превышает любое наперед заданное число — то есть даже в этом случае потенциально можно перечислить все слова.

Замечание. Стандарт и компиляторы ограничивают максимальную длину идентификатора, не рекомендуется использование идентификаторов длиннее 31 символа.

Следует обратить внимание, что язык Си — регистрозависимый. Таким образом, `a` и `A` — разные буквы, `number`, `Number` и `numbeR` — разные идентификаторы, а `Char` не является ключевым словом.

В Си могут быть определены идентификаторы переменных, функций, параметров функций и пользовательских (задаваемых программистом) типов данных.

Константы

Константы *(явные константы, используется также термин "**явные значения**")* — данные, вводимые непосредственно в исходный код программы.

Используется также понятие данных, **жестко включенных в код программы**.

Следует отличать константы от переменных, значения которых не подлежат изменению. Фактически значения констант вычисляются на стадии компиляции и включаются в исполняемый код программы. Тип константы определяется по ее значению. В Си имеются следующие константы.

1. **Символьные константы** (**символьные литералы**). Записываются в одинарных кавычках (апострофах), например `'A'`, `'0'`, `','`, `'Ъ'`, `'"'` — может быть указан почти любой символ, но строго один (в частности, при использовании многобайтовой кодировки, например, UTF-8, кириллические буквы не являются одним символом). Представляет собой значение типа `char`. Кроме явно печатаемых символов можно задать ряд служебных символов с помощью escape-последовательностей (управляющих последовательностей):

`\a`	Звонок (устаревший, при выводе должен издаваться сигнал PC-динамика).
`\b`	Удаление предыдущего символа.
`\f`	Перевод страницы (на старых принтерах).
`\n`	Новая строка (прокрутка).
`\r`	Возврат каретки (переход к началу строки, только Windows-системы).
`\t`	Горизонтальная табуляция.
`\v`	Вертикальная табуляция (практически не используется).
`\'`	Одиночная кавычка.

`\"` Двойная кавычка.

`\\` Обратная косая черта (необходимо, т.к. одиночная косая черта создает ESC-последовательность).

`\?` Литерал вопросительного знака (для совместимости с т.н. "триграфами" — комбинации вопросительного знака и последовательности букв для отображения ряда спецсимволов, используемыми на компьютерах с ограниченными клавиатурами, не имеющими таких символов; сейчас практически не используются и отключены в компиляторах).

`\ooo` Указание кода в восьмеричной системе счисления (число от 0 до 377, например `'\0'` — нулевой символ, `'\60'` — цифра 0 в стандартной кодировке и т.д.).

`\xhh` Указание кода в шестнадцатеричной системе счисления (число от 0 до FF, например `'\x0'` — нулевой символ, `'\x30'` — цифра 0 в стандартной кодировке и т.д.).

Таким образом, с помощью последних двух последовательсностей, можно задать любой символ, при этом в памяти будет представлен его код, однобайтовое число.

2. **Целочисленные константы** задаются с помощью цифр, префиксов и суффиксов, числа можно записывать в десятичной, восьмеричной и шестнадцатеричной системе счисления.

   С помощью констант кодируются только положительные числа, отрицательные числа задаются с помощью операции "унарный минус", о которой речь пойдет в параграфе 3.1.

   - Запись в десятичной системе: любая последовательность цифр, начинающаяся не с нуля, воспринимается как число в десятичной системе счисления. После последовательности может быть указан суффикс, отвечающий за тип данных, которым определяется тип данных. По умолчанию кодируется как `int`. Если указан суффикс `u` или `U`, то число становится беззнакового типа (явное задание беззнаковости числа бывает необходимо, например, для того, чтобы исключить сравнение знакового и беззнакового значений или переполнение знакового типа числом, которое может быть помещено в беззнаковый той же длины). Если указан суффикс `l` или `L`, то число записывается в типе `long` (необходимо указывать, например, чтобы исключить переполнение) Суффиксы могут комбинироваться и следовать в любом порядке.

     Например: `1`, `21`, `12L`, `36u`, `2000Ul`.
   - Запись в восьмеричной системе: любая последовательность восьмеричных цифр, предваренных нулем кодирует целочисленную константу в восьмеричной системе счисления. Также может добавляться аналогичный суффикс. Например `010` (число 8), `016l` (число 14).
   - Запись в шестнадцатеричной системе: любая последовательность шестнадцатеричных цифр, предваренных комбинацией `0x` или `0X`, воспринимается как целое число, записанное в шестнадцатеричной системе счисления. Шестнадцатеричный цифры могут кодироваться как заглавными, так и строчными буквами `A` – `F`. Число также может завершаться суффиксом. Например, `0x10` (число 16), `0XA` (число 10), `0Xal` (число 10 типа `long`).

Можно обратить внимание, что формально в Си константа `0` является на самом деле восьмеричным нулем, так же как и символ с кодом 0 — `'\0'` кодируется восьмеричным, а не десятичным кодом, но это не играет никакой роли. А вот наивная запись с использованием незначащих (в обычных математических выражениях) нулей вида `100+010+001` в результате даст 109 вместо ожидаемого на первый взгляд 111, т.к. во втором слагаемом начальный ноль превращает десятичное число 10 в восьмеричное.

Формальное определение целочисленной константы следующее.

⟨*ненулевая цифра*⟩ ::= 1|2|3|4|5|6|7|8|9

⟨*восьмеричная цифра*⟩ ::= 0|1|2|3|4|5|6|7

⟨*десятичная цифра*⟩ ::= 0|⟨*ненулевая цифра*⟩

⟨*шестнадцатеричная цифра*⟩ ::= ⟨*десятичная цифра*⟩|A|B|C|D|E|F|a|b|c|d|e|f

⟨*десятичное число*⟩ ::= ⟨*ненулевая цифра*⟩|⟨*десятичное число*⟩⟨*десятичная цифра*⟩

⟨*восьмеричное число*⟩ ::=
    0⟨*восьмеричная цифра*⟩
    |⟨*восьмеричное число*⟩⟨*восьмеричная цифра*⟩

⟨*шестнадцатеричное число*⟩ ::=
    0x⟨*шестнадцатеричная цифра*⟩
    |0X⟨*шестнадцатеричная цифра*⟩
    |⟨*шестнадцатеричное число*⟩⟨*шестнадцатеричная цифра*⟩

⟨*суффикс беззнакового целого*⟩ ::= u|U

⟨*суффикс длинного целого*⟩ ::= l|L

⟨*суффикс целого*⟩ ::=
    ⟨*суффикс беззнакового целого*⟩|⟨*суффикс длинного целого*⟩
    |⟨*суффикс беззнакового целого*⟩⟨*суффикс длинного целого*⟩
    |⟨*суффикс длинного целого*⟩⟨*суффикс беззнакового целого*⟩

⟨*целое число*⟩ ::= ⟨*десятичное число*⟩|⟨*восьмеричное число*⟩|⟨*шестнадцатеричное число*⟩

⟨*целочисленная константа*⟩ ::= ⟨*целое число*⟩|⟨*целое число*⟩⟨*суффикс целого*⟩

Определение обеспечивает, что суффиксы могут следовать в любом регистре и в любом порядке.

Существует наглядно-схематический полуформальный способ описания синтаксиса. В частности, для целочисленной константы запись выглядит следующим образом. Десятичные числа:

1-9[0–9...][uUlL]

Восьмеричные числа:

0[0–7...][uUlL]

Шестнадцатеричные числа:

0{x|X}0–9a–fA–F[0–9a–fA–F...][uUlL]

Здесь вертикальная черта означает возможность выбора одного из вариантов, если они указаны в фигурных скобках, то необходимо обязательно выбрать один из списка, если в квадратных — то элемент необязателен, многоточие — то, что значение может повторяться сколько угодно раз, дефис определяет диапазон (естественным образом, по алфавиту или порядку цифр). Неточность данного определения в данном случае кроется в указании суффикса — суффиксы регистронезависимы, можно указывать один или оба в любом порядке, но в схеме это не отражено.

Примеры ошибочных целочисленных констант: `018` (нет восьмеричной цифры 8), `1A23` (нет десятичной цифры A), `0x1FG1` и т.д. Кроме того, ошибочным будет число, которое слишком большое, чтобы уместиться в требуемый тип (вызовет переполнение константы).

3. **Константы с плавающей точкой**. Задаются с помощью цифр, экспоненциальной части, десятичной точки и суффиксов, в десятичной системе счисления.

[0-9...][.[0-9...]][e|E[+|-]0-9...][f|F|l|L]

Т.е. число с плавающей точкой состоит из четырех частей: целой части (целое число до точки), дробной части (целое число после точки) — вместе образующих десятичную дробь, десятичного порядка (буквы e или E и целого числа, возможно со знаком) и суффикса.

Число до точки может отсутствовать, тогда целая часть считается равной нулю. Число после точки тоже может отсутствовать, тогда дробная часть считается равной нулю. Порядок может отсутствовать, тогда он считается равным 0.

Число может завершаться суффиксом `f` или `F`, который будет означать, что число имеет тип `float`, или `l` или `L`, соответствующий типу `long double`, если суффикс отсутствует, то значение имеет тип `double`.

Однако если отсутствует и десятичная точка, и экспоненциальная часть и суффикс `f` или `F`, то число окажется целочисленной константой (типа `int` или `long` при отсутствия или наличии суффикса `l` или `L` соответственно).

Строгое БНФ определение константы с плавающей точкой следующее.

⟨*цифра*⟩ ::= `0`|`1`|`2`|`3`|`4`|`5`|`6`|`7`|`8`|`9`

⟨*последовательность цифр*⟩ ::= ⟨*цифра*⟩ | ⟨*последовательность цифр*⟩⟨*цифра*⟩

⟨*дробное число*⟩ ::=
⟨*последовательность цифр*⟩`.` | `.`⟨*последовательность цифр*⟩
|⟨*последовательность цифр*⟩`.`⟨*последовательность цифр*⟩

⟨*e-символ*⟩ ::= `e` | `E`

⟨*знак*⟩ ::= `+` | `-`

⟨*экспоненциальная часть*⟩ ::=
⟨*e-символ*⟩⟨*последовательность цифр*⟩
|⟨*e-символ*⟩⟨*знак*⟩⟨*последовательность цифр*⟩

⟨*суффикс числа с плавающей точкой*⟩ ::= `f`|`F`|`l`|`L`

⟨*число с плавающей точкой*⟩ ::=
⟨*дробное число*⟩ | ⟨*дробная число*⟩⟨*экспоненциальная часть*⟩
|⟨*последовательность цифр*⟩⟨*экспоненциальная часть*⟩

⟨*константа с плавающей точкой*⟩ ::=
⟨*число с плавающей точкой*⟩
|⟨*число с плавающей точкой*⟩⟨*суффикс числа с плавающей точкой*⟩
|⟨*последовательность цифр*⟩f | ⟨*последовательность цифр*⟩F

Примеры: `1.0`, `.1`, `1.`, `1e-2`, `1.2E15`, `1f`, `.5E-17L`. Ошибочные токены: `1E`, `1E2E1`, `1.22.3`, `1E2.4`.

4. **Строковые константы** (**строковые литералы**).

   В языке Си нет специального типа для хранения строк, однако существуют строковые константы. Они представляются в памяти последовательностью байтов. Соответствующий тип данных — массив символов — будет разъяснен ниже, он обозначается как `char[`N`]`, где N — количество задействованных символов (байт). В память автоматически добавляется символ `'\0'` в конец строки, поэтому N есть длина строки плюс 1.

   Строковой константой является последовательность символов, заключенных в кавычки. В качестве символов можно использовать как явно вводимые символы, так и escape-последовательности. Примеры:

   `"123"`

   `""` (пустая строка, 1 байт)

   `"123\n"` (символ переноса строки в конце)

   `"0"` (строка из символа 0)

   `"\"Hello, World\""` (строка с кавычками внутри)

   Строковые константы могут быть разбиты: последовательные строковые константы компилятор объединяет в одну. Например, записи `"1"␣"2"` и `"12"` равносильны. Таким образом, длинные строковые константы можно разбивать на несколько строк исходного кода.

Следует четко различать следующие три значения и аналогичные им:

`0`	целое число ноль, длина соответствует длине типа `int`;
`'0'`	символ 0, один байт, хранящий целое число, его код (например, 48);
`"0"`	строка содержащая символ 0, два байта, в первом хранится код символа ноль (48), во втором — 0 (символ с кодом 0 или просто однобайтовый 0), признак конца строки;

При этом символ `'\0'` соответствует целому числу 0, хотя и отличается размером (1 байт против размера `int`).

Замечание. Формально тип явных констант отличается от общего базового названия типа данных наличием модификатора `const`, например `const int`, `const float`, `const char [5]` и т.п. Данный модификатор не меняет формата представления данных в памяти, а указывает компилятору, что эти данные не подлежат изменению. Подробно описывается в параграфе 3.6.

Операции

Операции — токен особого вида, последовательности специальных символов (`!`, `&`, `*`, `+`, `-`, `/`, `%`, `<`, `=`, `>`, `|`, `^`, `~`, `,`, `.`, `?`, `:`). Семантически операция — способ задания действия над данными. Операция, подобно функции, имеет входные данные и возвращаемое значение — выходные данные. Входные данные операции называются **операндами**. Операции в программировании идейно аналогичны математическим операциям.

Фактически операция — особый способ задания некоторой функции, у которой тоже есть входные параметры и возвращаемое значение. Однако в Си, в отличие от функций, одна и та же операция может иметь разное количество операндов, в зависимости от типа операндов может вызываться разное действие (т.е. вызываться разный программный и машинный код), возвращать значения разного типа. Кроме того, если функции в Си не встроены и предоставляются библиотекой или создаются программистом, то операции фактически встроены в язык и не могут быть добавлены программистом. В других языках программирования высокого уровня функции могут быть свободны от данных ограничений, а программисту может быть позволено вводить собственные операции, поэтому главные отличия между функцией и операцией именно синтаксические и вытекающие из них идейные особенности применения.

В Си различают **унарные** (один операнд), **бинарные** (два операнда) и **тернарные** (три операнда) операции. При этом унарные операции различаются на **префиксные** (знак операции следует до операнда) и **постфиксные** (знак операции следует после операнда). Бинарные и тернарные операции являются инфиксными (знак операции указывается между операндами).

Иногда английское название operator переводится на русский как “оператор”, но это создает конфликт терминологии с устоявшимся в русском языке использования слова “оператор” в качестве русского аналога английского термина statement. Тем не менее, не только в интернете, но и в официальной литературе можно встретить этот казус, когда два принципиально разных синтаксически и семантически объекта называются одним и тем же словом. Во избежании путаницы в данном пособии используется термин “операция”.

Кроме значения операция может иметь **побочный эффект** — изменение значений операнда, если в качестве операнда указывается переменная. Например, присваивание изменяет значение переменной, указанной в качестве левого операнда, а возвращает присвоенное значение. Список операций с их арностью, возвращаемым значением и побочным эффектом приводится в таблице 2.3 с кратким семантическим пояснением. Тип возвращаемого значения операции определяется операцией и типом операндов. Подробно операции разъясняются в главе 3. БНФ определение операции состоит из их простого перечисления.

Пунктуационные знаки

Пунктуационные знаки или **пунктуаторы** — “`{`”, “`}`”, “`[`”, “`]`,” “`(`”, “`)`”, “`,`”, “`;`”, “`*`”, “`=`”, а также последовательность из трех точек “`...`” — явно предусмотренные синтаксисом более высокоуровневых единиц (выражений, операторов, функций) последовательности символов.

Фигурные скобки служат для группировки операторов, ограничения тела функций и инициализации массивов; квадратные скобки — для объявления массива и доступа к элементам массива; круглые скобки — для определения порядка действий при вычислении выражения и объявления переменных, а также перечисления списка параметров при задании и вызове функций, при этом параметры разделяются запятой; точка с запятой является признаком конца оператора, звездочка используется при объявлении переменной-указателя, равно — для инициализации значений переменных при объявлении, точка с запятой — для объявления функций с переменным числом параметров. Каждое из этих применений будет понятно из описании соответствующего синтаксического элемента в дальнейших параграфах и главах.

При этом является ли запятая, звездочка и знак равенства операцией или пунктуатором определяется по контексту: например, при перечислении параметров функции и объявлении переменной запятая предусмотрена

Арность	Знак	Название	Пример	значение	побочный эффект
Унарные префиксиные	`++`	Префиксный инкремент	`++n`	`n`	увеличение `n` на 1
	`--`	Префиксный декремент	`--n`	`n`	уменьшение `n` на 1
	`+`	Унарный плюс	`+a`	`a`	нет
	`-`	Унарный минус	`-a`	противоположное значение, $(-a) + a \equiv 0$	
	`!`	Логическое НЕ	`!b`	отрицание `b`	
	`~`	Побитовое НЕ	`~x`	инвертирование каждого бита `x`	
	`(type)`	Приведение типа	`(double) n`	наиболее близкое значение указанного типа	
	`*`	Разыменование	`*p`	значение в ячейке по адресу `p`	
	`&`	Взятие адреса	`&a`	адрес ячейки `a`	
	`sizeof`	Получение размера	`sizeof a`	число байт ячейки `a`	
Унарные постфиксные	`++`	Постфиксный инкремент	`n++`	`n + 1`	увеличение `n` на 1
	`--`	Постфиксный декремент	`n--`	`n - 1`	уменьшение `n` на 1
Бинарные	`*`	Умножение	`a * b`	произведение `a` и `b`	нет
	`/`	Деление	`a / b`	частное `a` и `b`	
	`%`	Остаток	`n % m`	остаток от деления `a` на `b`	
	`+`	Сложение	`a + b`	сумма `a` и `b`	
	`-`	Вычитание	`a - b`	разность `a` и `b`	
	`<<`	Битовый сдвиг влево	`a << n`	сдвиг каждого бита `a` на `n` позиций влево	
	`>>`	Битовый сдвиг вправо	`a >> n`	сдвиг каждого бита `a` на `n` позиций вправо	
	`<`	Строго меньше	`a < b`	истина, если `a` строго меньше `b`, иначе — ложь	
	`<=`	Меньше либо равно	`a <= b`	истина, если `a` меньше либо равно `b`, иначе — ложь	
	`>`	Строго больше	`a > b`	истина, если `a` строго больше `b`, иначе — ложь	
	`>=`	Больше либо равно	`a >= b`	истина, если `a` больше либо равно `b`, иначе — ложь	
	`==`	Равенство	`a == b`	истина, если `a` равно `b`, иначе — ложь	
	`!=`	Неравенство	`a != b`	истина, если `a` равно `b`, иначе — ложь	
	`&`	Битовое И	`n & m`	попарная конъюнкция каждого бита `n` и `m`	
	`^`	Битовое исключающее ИЛИ	`n ^ m`	попарное разделительное или каждого бита `n` и `m`	
	`|`	Битовое ИЛИ	`n | m`	попарная дизъюнкция каждого бита `n` и `m`	
	`&&`	Логическое И	`a && b`	конъюнкция	
	`||`	Логическое ИЛИ	`a || b`	дизъюнкция	
	`=`	Прямое присваивание	`a = b`	`b`	запись значения `b` в `a`
	`+=`	Присваивание со сложением	`a += b`	`a + b`	прибавление `b` к `a` (запись значения `a + b` в `a`)
	`-=`	Присваивание с вычитанием	`a -= b`	`a - b`	вычитание `b` из `a` (запись значения `a - b` в `a`)
	`*=`	Присваивание с умножением	`a *= b`	`a * b`	умножение `a` на `b` (запись значения `a * b` в `a`)
	`/=`	Присваивание с делением	`a /= b`	`a / b`	деление `a` на `b` (запись значения `a / b` в `a`)
	`%=`	Присваивание с вычислением остатка	`n %= m`	`n % m`	запись значения `n % m` в `n`
	`<<=`	Присваивание с битовым сдвигом влево	`a <<= n`	`a << n`	запись значения `a << n` в `a`
	`>>=`	Присваивание с битовым сдвигом вправо	`a >>= m`	`a >> m`	запись значения `a >> m` в `a`
	`&=`	Присваивание с битовым И	`n &= m`	`n * m`	запись значения `n & m` в `n`
	`^=`	Присваивание с битовым исключающим ИЛИ	`n ^= m`	`n ^ m`	запись значения `n ^ m` в `n`
	`|=`	Присваивание с битовым ИЛИ	`n |= m`	`n | m`	запись значения `n | m` в `n`
	`.`	Поле структуры (по переменной)	`s.name`	значение указанного поля	нет
	`->`	Поле структуры (по указателю)	`p->name`	значение указанного поля	
	`,`	Запятая	`a, b`	`b`	
Тернарная	`?:`	Тернарное условие	`b ? x : y`	`x`, если `b` — истинно, `y` в противном случае	

Таблица 2.3: Операции языка Си: синтаксис и арность.

синтаксисом и является пунктуатором, в остальных случаях воспринимается как операция. Данная операция описывается в параграфе 3.9.

О разделении токенов

Токены могут разделяться пробелами и другими разделительными символами, например, переносом строки и табуляцией, при этом часто пробел не требуется.

Все небуквенные и нецифровые символы автоматически завершают предыдущий токен, если он идентификатор, ключевое слово или число (за исключением точки при записи числа с плавающей точкой), конец токена-операции определяется автоматически и т.д.

Завершение символьной и строковой константы символом ' и " соответственно завершает токен, любой другой символ будет начинать следующий токен за исключением ситуации, когда две строковые константы записаны подряд (т.е. разделены только пробельными символами) они будут объединены в одну строковую константу (один токен).

Небуквенный и нечисловой символ после числовой константы также будет начинать следующий токен.

При этом все следующие подряд пробельные символы — в том числе символы перевода строки и табуляции — компилятор сокращает до одного.

Присутствие неверных токенов (не попадающих не под одно синтаксическое определение токена) в программе является синтаксической ошибкой.

Вопросы для самопроверки.

1. *Что такое токен?*
2. *Как разделяются токены в языке Си?*
3. *В чем отличие понятий "токен" и "слово"?*
4. *Какие виды токенов есть в Си?*
5. *Что такое ключевые слова?*
6. *Что такое идентификатор?*
7. *Что такое форма Бэкуса-Наура и зачем она нужна?*
8. *Различает ли язык Си строчные и заглавные буквы алфавита?*
9. *Что такое константы?*
10. *Чем отличаются константы от переменных?*
11. *Какие виды констант в Си есть?*
12. *Как задаются символьные константы в Си?*
13. *Как задаются целочисленные константы в Си?*
14. *Как задаются константы-числа с плавающей точной в Си?*
15. *Почему шестнадцатеричные константы начинаются в `0x`, а не, например, просто `x` или с числа 16?*
16. *Как задаются строковые константы в Си?*
17. *В чем различие между токенами `0`, `'0'`, `"0"`, `'\0'`, , `0l`, `0.0`?*
18. *В чем различие между токенами `1`, `'1'`, `"1"`, `'\1'`, `1l`, `1.`, `.1`, `1f`, `1.0l`?*

§2.5. Выражения

Выражение — синтаксическая единица языка, определяющая способ вычисления значения.

Выражения в программировании подобны математическим выражениям, где с помощью операций и функций можно получать "новые" числа из "старых".

Фактически выражения — прямой способ преобразования данных, получение из одних данных других. Данные преобразуются с помощью функций и операций: каждая из них берет на вход некоторое количество аргументов и на выходе возвращает новое значение. При этом каждый аргумент также может являться выражением, то есть выражение — рекурсивное понятие.

Всякая константа уже является выражением, всякий идентификатор (переменная) является выражением, вызов функции (идентификатор функции с перечислением аргументов) и вызов операции (операция со своими операндами) является выражением. В качестве операндов и аргументов выступают выражения, что делает понятие выражения рекурсивным.

Для явного указания приоритета и порядка ассоциативности операций используются круглые скобки,

Приближенно выражение можно определить следующим образом:

⟨*константа*⟩ ::=
  ⟨*символьная константа*⟩
  |⟨*целочисленная константа*⟩
  |⟨*константа с плавающей точкой*⟩
  |⟨*строковая константа*⟩

⟨*бинарное выражение*⟩ ::= ⟨*выражение*⟩⟨*бинарная операция*⟩⟨*выражение*⟩

⟨*постфиксное выражение*⟩ ::= ⟨*выражение*⟩⟨*постфиксная унарная операция*⟩

⟨*префиксное выражение*⟩ ::= ⟨*префиксная унарная операция*⟩⟨*выражение*⟩

⟨*унарное выражение*⟩ ::= ⟨*постфиксное выражение*⟩ | ⟨*префиксное выражение*⟩

⟨*тернарное выражение*⟩ ::= ⟨*выражение*⟩ `?` ⟨*выражение*⟩ `:` ⟨*выражение*⟩

⟨*список аргументов*⟩ ::= | ⟨*выражение*⟩ | ⟨*список аргументов*⟩, ⟨*выражение*⟩

⟨*вызов функции*⟩ ::= ⟨*идентификатор*⟩ `(` ⟨*список аргументов*⟩ `)`

⟨*получение элемента массива*⟩ ::= ⟨*выражение*⟩ `[` ⟨*выражение*⟩ `]`

⟨*выражение*⟩ ::=
  ⟨*константа*⟩
  |⟨*идентификатор*⟩
  |⟨*вызов функции*⟩
  |⟨*унарное выражение*⟩
  |⟨*бинарное выражение*⟩
  |⟨*тернарное выражение*⟩
  |⟨*получение элемента массива*⟩
  |`(`⟨*выражение*⟩`)`

Данное определение не является точным и приводится лишь для понимания сути выражений. Точное определение можно найти в спецификации языка.

Всякое выражение представляет собой значение. Поэтому, выражение, как и любое другое значение, имеет свой тип данных. Тип выражения-константы определяется типом этой константы, выражения-вызова операции — типом возвращаемого значения операции для данного типа операндов, выражения-вызова функции — возвращаемым значением функции.

Существуют также функции без возвращаемого значения (т.н. **void-функции**), запись вызова такой функции является **void-выражением**, пустым выражением — оно не имеет значения, условно его тип данных — `void`. Такие выражения не могут быть использованы в качестве операндов операции и аргументов функции, но все-таки являются выражениями (это лишь один из примеров неточности приведенного определения).

Подробно о типах выражений рассказывается в параграфе 3.10. В таблице 2.6 приведены примеры выражений с указанием значения и типа данных. Семантика операций описывается в главе 3.

Заметим, что хотя в таблице 2.3 приведены примеры операций с указанием в качестве операнда переменной, синтаксис позволяет указывать в качестве операнда любое выражение. Однако при отсутствии скобок это может создать неоднозначность, например, если расставлены скобки, то определение операнда операции не составляет труда. Сравните

```
int a = (2 + 3) * 4; /* значение 20 */
```

и

```
int a = 2 + (3 * 4); /* значение 14 */
```

Однако при отсутствии скобок определение выражения-операнда требует дополнительных действий.

```
int a = 2 + 3 * 4; /* правый операнд + есть 3 * 4 или 3? */
```

Если скобок нет, то операнд определяется соответствии с **приоритетом**: чем его числовое значение меньше, тем "теснее" операция связывает выражение, тем раньше она вычисляется. Поскольку умножение имеет более высокий приоритет, чем сложение, то пример выше соответствует расстановке скобок вокруг первого, т.е. значение — 14.

При равном приоритете операции вычисляются в соответствии с **порядком ассоциативности** — слева направо или справа налево. В таблице 2.5 приведен перечень операций Си с указанием приоритета и порядка.

Порядок и приоритет операций определяют, как именно выражение разбивается на подвыражения, какие именно токены группируются в операнд — как именно расставляются скобки. В таблице отражено, что помимо операций в классическом смысле этого слова, понятие порядка применимо к вызову функции и получению элемента по индексу, которые имеют более высокий приоритет чем, например, разыменование, поэтому `*p[3]` равносильно `*(p[3])` и отлично от `(*p)[3]` (см. параграф 4.2). Кроме того, приоритет применяется в объявлениях: в частности, приоритет символа указателя при объявлении функций, переменных и т.п. ниже, чем у списка аргументов и размера массива.

Вопросы для самопроверки.

1. *Что такое операция?*
2. *Какие виды операций с точки зрения синтаксиса есть в Си?*
3. *Что такое выражение?*
4. *Всегда ли выражение представляет собой значение?*
5. *Как определяется тип данных выражения?*
6. *Что такое void-выражение?*
7. *Что такое приоритет и порядок операций?*
8. *Приведите примеры операций с разным приоритетом, с разным порядком ассоциативности.*

§2.6. Деклараторы

Всякий идентификатор (который не является ключевым словом) должен быть **объявлен** внутри программы.

В Си могут объявляться следующие виды идентификаторов: идентификатор переменной, идентификатор функции, идентификатор типа данных. Идентификаторы типов данных позволяют задавать пользовательские типы данных или являются псевдонимами других типов данных, их декларация описывается в соответствующих параграфах. В данном параграфе описывается декларация переменных и функций.

С точки зрения программиста переменные — это данные, с которыми работает программа, функции — участки кода программы, которые можно исполнить. Однако согласно архитектуре фон Неймана данные и команды неразличимы в памяти, поэтому с точки зрения организации в обоих случаях идентификатору соответствует некоторая ячейка памяти, в которой хранятся или данные, или код функции.

Различают **объявление** (декларацию) и **определение** идентификаторов. Объявление — сообщение компилятору о существовании идентификатора с определенными синтаксическими свойствами, определение — сопоставление идентификатору объекта, т.е. собственно резервирование ячейки памяти для переменной или сопоставление функции ее кода. При этом, в Си всякое определение является и объявлением тоже, обратное, разумеется, неверно.

Поскольку стандарт Си ограничивает возможность использования определений и объявлений различных идентификаторов в различных синтаксических элементах, нет смысла вводить обобщенную единицу "декларатор". В данный момент имеет значение следующие единицы:

- **определение переменных**;
- **объявление функции**;
- **определение типа данных**;
- **определение функции** или собственно **функция**.

Объявление переменных без определения возможно, но в данном курсе не рассматривается, описывается в приложении D.3. Функция рассматривается как отдельная синтаксическая единица в параграфе 2.8.

Объявление переменных

Переменная — поименованная область (ячейка, группа байтов заданной длины, следующих в памяти подряд) памяти; ячейка памяти, доступ к содержимому которой (чтение и запись) осуществляется по ее имени.

Приоритет	Знак	Описание	Порядок
1	`++`	Постфиксный инкремент	Слева направо
	`--`	Постфиксный декремент	
	`()`	Вызов функции, объявление аргументов функции	
	`[]`	Элемент по индексу, объявление массива	
	`.`	Поле структуры по переменной	
	`->`	Поле структуры по указателю	
2	`++`	Префиксный инкремент	Справа налево
	`--`	Префиксный декремент	
	`+`	Унарный плюс	
	`-`	Унарный минус	
	`!`	Логическое НЕ	
	`~`	Побитовое НЕ	
	`(type)`	Приведение типа	
	`*`	Разыменование, символ указателя	
	`&`	Взятие адреса	
	`sizeof`	Получение размера	
3	`*`	Умножение	Слева направо
	`/`	Деление	
	`%`	Остаток	
4	`+`	Сложение	
	`-`	Вычитание	
5	`<<`	Битовый сдвиг влево	
	`>>`	Битовый сдвиг вправо	
6	`<`	Строго меньше	
	`<=`	Меньше либо равно	
	`>`	Строго больше	
	`>=`	Больше либо равно	
7	`==`	Равенство	
	`!=`	Неравенство	
8	`&`	Битовое И	
9	`^`	Битовое исключающее ИЛИ	
10	`|`	Битовое ИЛИ	
11	`&&`	Логическое И	
12	`||`	Логическое ИЛИ	
13	`?:`	Тернарное условие	Справа налево
	`=`	Прямое присваивание	
	`+=`	Присваивание со сложением	
	`-=`	Присваивание с вычитанием	
	`*=`	Присваивание с умножением	
	`/=`	Присваивание с делением	
	`%=`	Присваивание с вычислением остатка	
	`<<=`	Присваивание с битовым сдвигом влево	
	`>>=`	Присваивание с битовым сдвигом вправо	
	`&=`	Присваивание с битовым И	
	`^=`	Присваивание с битовым исключающим ИЛИ	
	`|=`	Присваивание с битовым ИЛИ	
14	`,`	Запятая	Слева направо

Таблица 2.5: Порядок и приоритет операций.

Выражение	Пояснение	Тип данных	Значение
`2.5`	константа	`double`	`2.5`
`n`	переменная (пусть имеет тип `int`, значение 1)	`int`	`1`
`n+2.5`	бинарная операция `+` с операндами `n` и `2.5`	`double`	`3.5`
`(n + 2.5)`	выражение в скобках	`double`	`3.5`
`3`	константа	`int`	`3`
`-3`	унарная префиксная операция `-` с операндом `3`	`int`	`-3`
`-3 * (n + 2.5)`	бинарная операция `*` с операндами `-3` и `(n + 2.5)`	`double`	`-10.5`
`a`	переменная (пусть имеет тип `double`, значение 0.25)	`double`	`0.25`
`-3 * (n + 2.5) * a`	бинарная операция `*` с операндами `-3 * (n + 2.5)` и `a`	`double`	`-2.625`
`pow(-3 * (n + 2.5) * a, a)`	вызов функции с аргументами `-3 * (n + 2.5) * a` и `a`	`double`	`0.90005` (примерно)
`pow(-3 * (n + 2.5) * a, a) <= n`	бинарная операция `<=` с операндами `pow(-3 * (n + 2.5) * a, a)` и `n`	`int`	`1` (истина)

Таблица 2.6: Примеры выражений.

При создании исполняемого кода, компилятор автоматически резервирует память под переменную и вычисляет адрес (в адресном пространстве), по которому будут храниться данные.

В Си всякая переменная характеризуется именем (идентификатором) и типом данных. Тип данных определяет какой размер (длину) имеет ячейка памяти, выделяемая под переменную, а также то, как именно будет обрабатываться хранимое в ней значение при вычислении значений выражений.

Приближенно синтаксис объявление переменной выглядит следующим образом:

⟨тип⟩ ⟨идентификатор⟩;

Здесь тип — одно из известных Си имен типов данных, идентификатор — новый (ранее не задействованный) идентификатор, который станет именем определяемой переменной. Объявление должно завершаться символом “точка с запятой”.

Примеры:

```
int n;
long m;
double r;
```

Си позволяет объявить несколько переменных одного типа за раз, перечислив их через запятую, например:

```
int n, m;
```

Также можно сразу присвоить значение переменной — **инициализировать** ее при объявлении. Инициализация переменных является обязательной в Си, так как исходное значение переменной является непредсказуемым, автоматическая инициализация нулем или каким-либо иным значением не производится. Например,

```
int n = 1, m = 5 * n, k;
```

Разумеется, инициализировать значение можно и после объявления (например, присваиванием или прочтением с клавиатуры). Инициализатор переменной является особой синтаксической единицей, отличной от присваивания. В первом приближении в качестве него может выступать любое выражение. Таким образом, более точное определение объявления переменных выглядит так:

⟨*инициализатор*⟩ ::= =⟨*выражение*⟩

⟨*объявление переменной*⟩ ::= ⟨*идентификатор*⟩ | ⟨*идентификатор*⟩⟨*инициализатор*⟩

⟨*список объявлений переменных*⟩ ::= ⟨*объявление переменной*⟩
    |⟨*объявление переменных*⟩,⟨*объявление переменной*⟩

⟨*объявление переменных*⟩ ::= ⟨*тип данных*⟩ ⟨*список объявлений переменных*⟩;

Следовало бы также дать точное определение типа данных — сделать это можно путем прямого перечисления встроенных в Си типов данных с оговоркой, что идентификатор, объявленный в качестве идентификатора типа данных, также является типом данных.

Данное определение неполно, так как существует еще объявление переменных, являющихся указателями и массивами и др., описанное в соответствующих параграфах. Инициализатор переменной является особой синтаксической единицей, отличной от присваивания, в частности это различие будет заметно для указателей и массивов.

Замечание. Объявлять переменные можно внутри тела функций и вне функций. В первом случае объявляются *локальные* для данной функции переменные, то есть доступные только внутри нее, во втором — *глобальные*, которые можно использовать в нескольких функций. По ряду причин использование глобальных переменных не рекомендуется, при отсутствии веских причин следует использовать локальные переменные. Стандарт Си требует, чтобы объявления локальных переменных предшествовали коду, то есть в начале функции должен следовать блок объявлений переменных, а затем — операторы, составляющие код функции. Некоторые компиляторы свободны от данного ограничения и допускают чередование операторов и объявлений переменных, но написанная таким образом программа может быть интерпретирована как ошибочная другими компиляторами. В параграфах 2.7, 2.8, 2.9 даны необходимые строгие синтаксические определения и пояснения.

Прототипы функций

В Си есть возможность как создавать свои функции, так и использовать функции стандартной библиотеки языка, а также других библиотек.

Всякая функция в Си имеет **прототип** — объявление функции, без указания ее тела (кода). Прототип содержит в себе идентификатор (имя функции), тип возвращаемого значения функции, а также количество аргументов (арность), их тип и порядок следования.

Например,

```
int abs(int n);
double sqrt(double x);
double pow(double x, double y);
```

Здесь, `abs` — функция вычисления модуля (абсолютной величины) целого числа, принимает один аргумент типа `int` и возвращает значение типа `int`;

`sqrt` — функция вычисления квадратного корня вещественного числа, принимает один аргумент типа `double` и возвращает значение типа `double`;

`pow` — функция возведения в степень, принимает 2 аргумента типа `double` и возвращает значение типа `double`.

Прототипы могут даже указываться без имен аргументов:

```
int abs(int);
double sqrt(double);
double pow(double, double);
```

но указание имен удобно при документировании функции. Так, в последней функции можно четко указать, что параметр `x` — основание, а `y` — показатель степени, оперируя словами, а не номерами аргументы. Все три функции "настоящие", т.е. присутствуют в стандартной библиотеке.

Для функций принята следующая терминология: имена, используемые в прототипе функции называются параметрами или **формальными параметрами**, данные, к которым применяется функция — **аргументами** или **фактическими параметрами**.

В общем случае прототип функции выглядит так:

$$\langle\text{тип}\rangle\ \langle\text{идентификатор}\rangle\ \texttt{(}[\langle\text{тип}_1\ \rangle\ [\langle\text{идентификатор}_1\ \rangle]\ldots]\texttt{);}$$

т.е. функция может иметь 0 и более аргументов, они перечисляются через запятую, каждый из них может иметь свой тип. Задание аргументов похоже на определение переменных: тип и имя (которое при работе с прототипом, однако, может опускаться). Стандарт устанавливает предел в 127 аргументов, на практике не рекомендуется создавать функции, у которых больше 4-5 аргументов. Возвращаемое значение у функции одно.

За редким исключением количество аргументов функции фиксировано, однако возможно задание функции, у которой количество аргументов не определено (любое).

Формально, прототип функции задается следующим образом:

⟨*тип возврата*⟩ ::= `void` | ⟨*тип данных*⟩

⟨*параметр*⟩ ::= ⟨*тип данных*⟩ | ⟨*тип данных*⟩⟨*идентификатор*⟩

⟨*список параметров*⟩ ::= ⟨*параметр*⟩ | ⟨*список параметров*⟩`,`⟨*параметр*⟩

⟨*параметры функции*⟩ ::= | `void` | ⟨*список параметров*⟩ | ⟨*список параметров*⟩`, ...`

⟨*заголовок функции*⟩ ::= ⟨*тип возврата*⟩⟨*идентификатор*⟩ (⟨*параметры функции*⟩)

⟨*прототип функции*⟩ ::= ⟨*заголовок функции*⟩;

Заметим, что параметр может указываться с именем (идентификатором) и без. Прототип функции завершается точкой с запятой. Функция может не иметь возвращаемого значения (*void*-функция). Параметры в списке разделяются запятой, объявить несколько параметров одного типа без повторения имени типа данных (подобно объявлению переменных) невозможно. Список параметров может быть пустым, однако в этом случае считается, что функция имеет произвольное число аргументов, поэтому в качестве параметров функции без параметров следует использовать `void`. Список параметров может завершаться троеточием, тогда функция может иметь любое количество параметров после обязательных. Например,

```
int foo();
int bar(void);
int foo(double x, int n, ...);
```

первая функция может принимать любое (в том числе нулевое) количество аргументов любых типов, вторая должна вызываться без аргументов, третья — должна иметь два обязательных аргумента типа `double` и `int` после которых может следовать любое (в том числе нулевое) количество аргументов любых типов.

Второй способ задания функций без аргументов используется редко, отдается предпочтение первому варианту (указание "лишних" аргументов будет проигнорировано компилятором, но вряд ли создаст серьезную ошибку). Функции с произвольным количеством аргументов (как функция `foo` или как и функция `bar`) *не являются типобезопасными*, не обеспечивают строгой типизации, что несет в себе потенциальный риск труднообнаружимых ошибок. Создание таких функций описывается в приложении D.5, но за редким исключением не рекомендуется. Однако в стандартной библиотеке языка Си такие функции используются, поэтому синтаксис их прототипа приводится здесь.

Сейчас важно понять, что прототип — информация, достаточная для вызова функции в двух смыслах:

- компилятор знает, какое именно количество входных параметров и какого типа нужно передать в функцию, а также значение какого типа нужно взять на выходе функции: *при вызове функции все значения параметров и возвращаемое значение копируются* (компоновщик по идентификатору функции определит, где на самом деле находится код функции, т.е. по какому адресу нужно передать управление);
- программист знает аргументы какого типа нужно указывать в программе при вызове функции и какой тип имеет возвращаемое значение, стандартная документация к функции содержит ее прототип, описание выполняемого действия и описание смысла аргументов.

Таким образом, прототип функции вместе с документацией является черным ящиком: известны входные данные, известны выходные данные, известно что именно делает функция, но не сообщается как именно она это делает. Более того, документация ко всякой библиотеке приводит прототипы функций. Описание всех функций стандартной, как и любой другой, библиотеки лежит за рамками данного пособия, а чтение документации является важным навыком. Например, документация к функциям может выглядеть следующим образом:

```
/* abs: вычисление модуля
 * n - число для вычисления модуля
 * возвращаемое значение - модуль числа n
 */
int abs(int n);

/* pow: возведение в степень
 * x - основание степени
 * y - показатель степени
 * возвращаемое значение - x в степени y
 */
double pow(double x, double y);
```

В Си имена функций должны быть уникальны: один и тот же идентификатор не может соответствовать разным функциям; если идентификатор уже объявлен как имя функции с определенным набором аргументов, то другой функции с тем же именем, но, например, другим количеством параметров, объявлено быть не может.

Как и всякий идентификатор, имя функции может содержать символ подчеркивания _, так как он считается буквой. В том числе, имя функции может начинаться с этого символа, однако на практике такие функции используются для внутренних нужд стандартной библиотеки языка Си: внутри программы не рекомендуется объявлять функции, имена которых начинаются с символа подчеркивания.

Вопросы для самопроверки.

1. *Что задает объявление переменой?*
2. *Что такое инициализатор переменных?*
3. *Что такое прототип функции, каков его синтаксис?*
4. *Как компилятор использует прототип функции?*
5. *Как программист использует прототипы чужих и библиотечных функции?*
6. *Что такое void-функция?*
7. *Подумайте, каков может быть смысл создания void-функций?*

§2.7. Операторы

Оператор — наименьшая исполняемая инструкция языка, команда.

Всякая функция в Си состоит из операторов, процесс исполнения функции заключается в последовательном исполнении операторов друг за другом. Си является языком со свободным переносом строк — не различает переносы строк и пробелы, поэтому операторы могут быть записаны в одну строку или разбиты на несколько строк. Условно можно считать, что операторы в Си разделяются точкой с запятой (;), однако точнее будет сказать, что пунктуатор "точка с запятой" является составной частью синтаксиса многих операторов, но не всех.

БНФ определение оператора состоит в прямом перечислении всех операторов:

⟨*оператор*⟩ ::=
⟨*пустой оператор*⟩
|⟨*оператор-выражение*⟩
|⟨*оператор return*⟩
|⟨*помеченный оператор*⟩
|⟨*оператор перехода*⟩
|⟨*составной оператор*⟩
|⟨*условный оператор*⟩
|⟨*оператор цикла с предусловием*⟩
|⟨*оператор цикла с постусловием*⟩
|⟨*оператор for*⟩
|⟨*оператор break*⟩
|⟨*оператор continue*⟩
|⟨*оператор выбора*⟩

Разберем синтаксис операторов языка Си последовательно. Отметим, что операторы могут иметь параметры, в качестве которых могут выступать выражения, идентификаторы и другие операторы.

Пустой оператор

Пустой оператор представляет собой одиночную точку с запятой.

⟨*пустой оператор*⟩ ::= ;

Пустой оператор — оператор, которому не соответствует никакой исполняемой инструкции, он существует только синтаксически и поэтому используется редко. Можно сказать, что одиночная точка с запятой ("лишняя точка с запятой") создает пустой оператор.

Оператор-выражение

Одиночное выражение в языке Си может быть использовано в качестве оператора. Это особенность Си-подобных языков, во многих других языках выражение не является оператором. Оператор-выражение — выражение, после которого стоит точка с запятой:

⟨*оператор-выражение*⟩ ::= ⟨*выражение*⟩;

При этом, не всякое выражение имеет смысл использовать как оператор: если выражение не меняет ни значение никаких переменных или данных в памяти, ни передает никакие данные на внешние устройства — то есть не имеет **побочных эффектов**, эффектов, отличных от вычисления значения этого выражения — то вычисление значения этого выражения является бесполезным, хотя и синтаксически корректным, действием. Использование в качестве оператора выражения с наличием побочных эффектов является осмысленным действием. Заметим, что *присваивание* является бинарной операцией, поэтому оператор вида

```
a = n + 3;
```

имеет побочный эффект (изменяет значение переменной `a` на значение выражения `n+3`), корректен и синтаксически, и семантически. Семантически корректными будут операторы-выражения вызова функций ввода значений переменной с клавиатуры и вывода данных на экран.

Оператор возврата значения функции

Оператор, возвращающий значение из функции — `return` имеет следующий синтаксис:
⟨*оператор return*⟩ ::= `return ;` | `return` ⟨*выражение*⟩;
т.е. после слова `return` может быть записано любое выражение. Значение данного выражения становится тем самым значением, которое возвращает функция. Данное выражение является обязательным для не-*void*-функций, но не требуется для *void*-функций, то есть функций не возвращающих значение.

Оператор не только устанавливает значение, возвращаемое функцией, но и завершает работу функции — даже если далее в коде следуют какие-то операторы, они не будут исполнены.

Оператор безусловного перехода

Оператор безусловного перехода передает управление указанному оператору, то есть следующим оператором будет исполнен не следующий по списку, а другой выбранный оператор, который может находиться в любом месте функции — как до, так и после оператора перехода по списку. *По ряду причин использование оператора перехода является объектом критики и данный оператор не рекомендуется использовать без веских причин. Подробнее об этом написано в параграфе 6.2.* Оператор, на который производится переход, задается *меткой* — идентификатором особого рода. Данный идентификатор не требует отдельного объявления. Синтаксически оператор, которому присвоена метка, является **помеченным оператором**. Помеченный оператор — один из операторов языка Си.

⟨*Метка*⟩ ::= ⟨*Идентификатор*⟩

⟨*Оператор перехода*⟩ ::= `goto` ⟨*Метка*⟩;

⟨*Помеченный оператор*⟩ ::= ⟨*Метка*⟩ `:` ⟨*Оператор*⟩

Параметром помеченного оператора выступают идентификатор и любой оператор языка Си, метка отделяется двоеточием. При исполнении помеченного оператора в порядке очереди исполняется тот самый оператор, который был помечен. Идентификатор метки доступен для использования в операторе перехода и только в нем. Имя метки может совпадать с именем переменных и функций, что не приведет к конфликту, так как идентификатор метки не может быть использован как часть выражения. Оператор перехода передает управление оператору, помеченному соответствующей меткой, то есть следующим оператором будет исполнен помеченный оператор, в какой части функции бы он не находился, перед или после оператора перехода.

Следует обратить внимание, что оператор с меткой тоже является оператором и может быть использован везде, где по синтаксису предполагается оператор — в том числе по этой причине у оператора может быть две и более метки.

Составной оператор

Операторы комбинируются в **операторные блоки**. Блоки — списки операторов (условно — разделенных символом точка с запятой '`;`'). Блоки ограничиваются фигурными скобками '`{ ...}`' — **операторными скобками**. Такой блок операторов в Си сам является оператором, известным как **составной оператор**. В начале составного оператора может следовать список объявлений переменных (и типов данных, синтаксис которых пока не рассматривался).

⟨*список операторов*⟩ ::= | ⟨*список операторов*⟩⟨*оператор*⟩

⟨*список объявлений*⟩ ::= | ⟨*список объявлений*⟩⟨*объявление переменных*⟩ | ⟨*список объявлений*⟩⟨*объявление типа данных*⟩

⟨*блок операторов*⟩ ::= `{` ⟨*список объявлений*⟩⟨*список операторов*⟩ `}`

Объявление переменных (и типов данных) содержит в себе завершающую точку с запятой, поэтому условно объявления в списке объявлений также разделяются точной с запятой. Переменные можно объявлять в начале любого блока операторов и только в начале, хотя некоторые компиляторы свободны от этого ограничения. Список объявлений и список операторов может быть пустым.

Составной оператор, как частный случай оператора, может быть использован в качестве оператора-аргумента другого оператора, например он может быть помечен.

Идентификаторы, объявленные внутри составного оператора, доступны к использованию только внутри данного составного оператора.

Условный оператор

Условный оператор — оператор, определяющий выполнение блока кода только при выполнении некоторого условия. В Си имеет следующий синтаксис:
⟨*Неполный условный оператор*⟩ ::= `if ( `⟨*выражение*⟩` ) `⟨*оператор*⟩

⟨*Полный условный оператор*⟩ ::= `if ( `⟨*выражение*⟩` ) `⟨*оператор_1*⟩` else `⟨*оператор_2*⟩

⟨*Условный оператор*⟩ ::= ⟨*Полный условный оператор*⟩ | ⟨*Неполный условный оператор*⟩
Выражение, указываемое в скобках, интерпретируется как условие. При этом условие Си понимает следующим образом: *нулевое значение (любого типа) считается ложью, ненулевое значение (любого типа) считается истиной.* Неполный условный оператор выполняет оператор-аргумент тогда и только тогда, когда условие истинно, полный условный оператор выполняет *оператор_1* тогда и только тогда, когда условие истинно и *оператор_2* тогда и только тогда, когда условие ложно.

Синтаксис языка не требует указания точки с запятой перед `else`, но она чаще всего необходима, только относится не к оператору `if`, а к операторам, завершаемым точкой с запятой: оператору-выражению, `return` и др. Если же используется составной оператор, то после него точка с запятой не требуется, но она будет завершать последний оператор перед закрывающейся фигурной скобкой.

Следует обратить внимание на следующую тонкость языка Си в контексте записи условного оператора: переносы и форматирования не влияют на синтаксис операторов.

```
if (cond1)
    if (cond2)
        statement_2;
else
       statement_3;
```

здесь `else` относится ко второму (вложенному) `if`, эта запись эквивалентна

```
if (cond1)
{
    if (cond2)
        statement_2;
    else
         statement_3;
}
```

и должна быть отформатировано как

```
if (cond1)
    if (cond2)
        statement_2;
    else
        statement_3;
```

чтобы отнести `else` к первому (внешнему) `if` следует обязательно указывать фигурные скобки:

```
if (cond1)
{
    if (cond2)
         statement_2;
}
else
    statement_3;
```

Операторы цикла

Оператор цикла — оператор, задающий многократное повторение выполнения другого оператора, называемого **телом цикла**. Каждый акт исполнения тела цикла называется **итерацией** или **проходом**.

В Си существует три оператора цикла — **цикл с предусловием**, **цикл с постусловием** и оператор `for`.

⟨*оператор цикла с предусловием*⟩ ::= `while ( `⟨*выражение*⟩` ) `⟨*оператор*⟩

⟨*оператор цикла с постусловием*⟩ ::= `do` ⟨*составной оператор*⟩ `while ( `⟨*выражение*⟩` );`

Тело цикла исполняется до тех пор, пока условие истинно, то есть значение выражения-аргумента не является нулем.

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.

То, что в качестве аргумента оператор цикла с постусловием может выступать лишь составной оператор означает, что для конструкции `do...while` обязательны операторные скобки.

С помощью оператора цикла можно создать программу, работа которой никогда не завершится, то есть представляет собой **бесконечный цикл** — цикл, условие выхода из которого никогда не достигается. Простейший бесконечный цикл следующий

```
while (1);

do {} while (1);
```

При такой ситуации

```
while (0) /*код*/;

do {/*код*/} while (0);
```

в первом случае код не будет выполнен ни разу, во втором — один раз. Существует рекомендация отказываться от цикла с постусловием за редким исключением, когда исполнение кода хотя бы один раз действительно гарантированно необходимо.

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

⟨*Инициализатор*⟩ ::= | ⟨*выражение*⟩

⟨*Условие*⟩ ::= | ⟨*выражение*⟩

⟨*Итератор*⟩ ::= | ⟨*выражение*⟩

⟨*Оператор for*⟩ ::= for (⟨*Инициализатор*⟩ ; ⟨*Условие*⟩ ; ⟨*Итератор*⟩) ⟨*Оператор*⟩

Данный цикл по сути является циклом с предусловием и за одним исключением аналогичен следующей конструкции:

⟨*Инициализатор*⟩;
while (⟨*Условие*⟩)
{
  ⟨*Оператор*⟩;
  ⟨*Итератор*⟩;
}

Таким образом, выражение-инициализатор исполняется перед началом цикла (и должно иметь побочные эффекты). Тело цикла исполняется до тех пор, пока выражение-условие истинно (ненулевое). Условие проверяется до начала исполнения тела цикла. По окончании исполнения тела цикла исполняется итератор (также должен иметь побочный эффект). Все три элемента могут отсутствовать. Отсутствующие инициализатор и итератор опускаются, отсутствующее условие всегда истинно:

```
for (;;);
```

создает бесконечный цикл (последняя точка с запятой — пустой оператор, который становится телом цикла).

Обычное использование оператора `for` — создание цикла со счетчиком (пример выводит числа от 0 до 10 и их квадраты, итератор `++i` увеличивает значение `i` на 1):

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

В Си для целочисленных счетчиков типично использование нестрого неравенства.

Отметим, что поскольку составной оператор является частным случаем оператора, то де-факто в теле цикла (как и в условном операторе), могут присутствовать не один, а несколько операторов — заключенных в фигурные скобки. А за счет того, что условный оператор и операторы цикла сами являются операторами, то условия и циклы могут быть вложенными друг в друга. При оформлении программы рекомендуется каждый следующий уровень вложенности обозначать бóльшим отступом, добавляя 4 пробела (“правило четырех пробелов”).

Начиная со стандарта C99 в качестве инициализатора цикла `for` может также выступать декларация переменных (одного типа):
⟨*Инициализатор*⟩ ::= | ⟨*выражение*⟩ | ⟨*объявление переменных*⟩
например,

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

Порядок исполнения циклов может регулироваться операторами прерывания:

- Оператор `continue` (не имеет аргументов, синтаксис — `continue ;`) приводит к непосредственному переходу к следующей итерации тела цикла, при этом для цикла `for` происходит вычисление итератора.
- Оператор `break` (не имеет аргументов, синтаксис — `break ;`) прерывает исполнение цикла (самого внутреннего) и передает управление следующему за данным оператором цикла оператору.

Эти операторы тоже являются в некотором смысле операторами перехода и их использование не рекомендуется по крайней мере до тех пор, пока отказ от их использования не ведет к ухудшению производительности и качеству кода (на практике это случается чаще, чем в случае с оператором перехода по метке).

На порядок исполнения, определенных условным оператором и операторами цикла, влияет безусловный переход (категорически не рекомендованный к использованию в структурном программировании).

Кроме того, следует иметь в виду, что оператор `return` прерывает не только цикл, но и всю функцию.

Создание циклов, условие которых всегда истинно, может быть результатом ошибки: неверного написания условия или неверного написания (или вообще опущения) изменения переменных внутри цикла, приводящее к тому, что цикл никогда не завершится, т.е. программа “зациклится”, ее придется прерывать принудительно средствами операционной системы. Однако, часто при использовании оператора `break` условие цикла намеренно делают всегда истинным (`1`): выход из цикла произойдет при достижении условия, проверяемого условным оператором внутри цикла.

Множественный выбор и оператор выбора

Когда необходимо выбрать одно из нескольких взаимоисключающих условий, можно использовать вложенные условные операторы. Например, если следует выполнить разные блоки кода для разных значений переменной:

```
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 */
```

```
        }
```

однако нагляднее будет использовать конструкцию, известную как `if...else if... else`:

```
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 */
}
```

это тот же самый код, отличается только его форматирование. Для вложенных `if` существует рекомендация ограничиваться не более 7 уровнями вложения, лучше даже не более 3–4, для `if...else if...else` такого ограничения нет: код в любом случае останется хорошо воспринимаемым.

Есть еще один способ оформить множественный выбор: это **оператор выбора** `switch`. Формально его синтаксис такой:

⟨*оператор выбора*⟩ ::= switch (⟨*выражение*⟩) ⟨*оператор*⟩

что, однако, не дает подсказки по его использованию. На самом деле оператор производит неявный переход по особой метке, действительной в рамках блока оператора-аргумента, в зависимости от значения выражения, а метка может быть одной из следующих

⟨*метка выбора*⟩ ::= case ⟨*Целочисленная константа*⟩ | default

При этом, синтаксически метка выбора является меткой. Оператор, которому предшествует метка выбора и пунктуатор “`:`” является помеченным оператором в соответствии с синтаксисом последнего. Однако использование метки выбора для получения помеченного оператора допустимо только в теле оператора выбора.

Оператор `switch` работает следующим образом: если значение выражения-аргумента совпадает с одной из целочисленных констант метки выбора составного оператора-аргумента (т.е. с константой следующей после одного из `case` внутри его), то производится переход на эту метку, если не совпадает ни с одной — то на метку `default` при ее наличии или к следующему за `switch` оператору при ее отсутствии.

Выражение должно быть целочисленным, а метки — константами, так как этот оператор не заменяется на цепочку условий при компиляции, а генерируется более эффективный код: осуществляющий именно передача управления по адресу, вычисляемому в зависимости от значения выражения.

В составном операторе-аргументе может присутствовать оператор `break`: он завершает исполнение операторного блока и передает управление следующему за `switch` оператору.

Типичное использование оператора `switch` следующее:

```
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`).

С помощью оператора выбора предыдущий пример множественного выбора можно записать так:

```
switch (i)
{
    case 1:
        /* случай i == 1 */
    break;
    case 2:
        /* случай i == 2 */
        break;
    case 3:
        /* случай i == 3 */
        break;
    case 4:
        /* случай i == 4 */
        break;
    default:
    /* остальные случаи */
}
```

Польза от возможности опустить оператор `break` возникает, например, когда для разных значений нужно исполнить один оператор, хотя, конечно, на практике пропуск `break` — частая ошибка новичков.

```
switch (c)
{
    case 'a' :
    case 'b' :
    case 'c' :
    case 'd' :
    case 'e' :
    case 'f' :  i++;
    case 'g' :
    case 'h' :  i--;
}
```

соответствует

```
if (c == 'a' || c == 'b' || c == 'd' ||
    c == 'e' || c == 'f'
   ) i++
else if (c == 'g' || c == 'h') i--;
```

По той же причине метку `default` бывает полезно указывать в середине оператора, хотя в остальных случаях нагляднее, когда она в конце.

Вопросы для самопроверки.

1. *Что такое оператор?*
2. *Что такое оператор-выражение, какие его особенности?*
3. *Что такое оператор return?*
4. *Что такое пустой оператор?*
5. *Что такое блок операторов*

6. *Что представляет собой список объявлений переменных в операторном блоке? Где он может располагаться?*
7. *Каков синтаксис условного оператора?*
8. *Что может выступать в качестве условия в Си?*
9. *Как Си трактует значение выражения-условия?*
10. *Как работает условный оператор?*
11. *Какие есть операторы цикла в Си? Каков синтаксис каждого из них?*
12. *Как работают операторы цикла в Си?*
13. *Что такое тело цикла?*
14. *Что будет, если в цикле `for` оставить пустым инициализатор? Условие? Итератор?*
15. *Как прервать исполнение текущей итерации цикла?*
16. *Как прервать исполнение цикла?*
17. *Почему использование операторов `continue` и `break` не рекомендуется?*
18. *Что такое бесконечный цикл?*
19. *Чем чреват бесконечный цикл?*

§2.8. Функции

Ранее был определен синтаксис заголовка функции в Си, содержащий тип возвращаемого значения (или `void`), имя функции (идентификатор) и список параметров (типы и идентификаторы), перечисляемые через запятую. Будучи завершенным точкой с запятой, идентификатор функции превращается в объявление функции (прототип), а будучи завершенный составным оператором — **телом функции** — в определение функции, сопоставление этой функции коду.
⟨*тип возврата*⟩ ::= `void` | ⟨*тип данных*⟩

⟨*параметр*⟩ ::= ⟨*тип данных*⟩⟨*идентификатор*⟩ | ⟨*тип данных*⟩⟨*идентификатор*⟩

⟨*список параметров*⟩ ::= ⟨*параметр*⟩ | ⟨*список параметров*⟩`,`⟨*параметр*⟩

⟨*параметры функции*⟩ ::= | `void` | ⟨*список параметров*⟩ | ⟨*список параметров*⟩`, ...`

⟨*заголовок функции*⟩ ::= ⟨*тип возврата*⟩⟨*идентификатор*⟩ (⟨*параметры функции*⟩)
⟨*функция*⟩ ::= ⟨*заголовок функции*⟩ ⟨*составной оператор*⟩

При выполнении функции операторы блока операторов тела функции (составного оператора) выполняются последовательно друг за другом, как шаги алгоритма. Оператор `return` в функции может присутствовать сколько угодно раз, но его выполнение приводит к прекращению исполнения функции и возврата из нее. Если после оператора `return` присутствуют какие-либо операторы, они не будут выполнены — это т.н. **недостижимый код**. Сам по себе недостижимый код не является ошибкой, но его наличие в программе может свидетельствовать о присутствии других, семантических, ошибок, т.е. неправильной работы программы. Современные компиляторы способны выявлять недостижимый код и предупреждать о нем.

Следует обратить внимание, что имена параметров, указываемые в определении функции называются **формальными параметрами** — это фактически некоторые переменные, которые можно использовать внутри функции и которые имеют значение, доступное внутри функции. Выражения, передаваемые в качестве параметров функции при ее вызове — аргументы — называются также **фактическими параметрами**, то есть собственно передаваемыми значениями в конкретном случае вызова функции.

Под каждую функцию отводится собственная область памяти, где хранятся значения ее локальных переменных и параметров. Все значения параметров передаются как копия данных вызывающей функции. Изменения значений параметров внутри функции никак не скажется на данных вызвавшей функции.

Заметим, что в данном случае формальные параметры, задаваемые в заголовке функции, являются идентификаторами и определение функции определяет данные идентификаторы — как ячейки памяти, содержащие данные соответствующего типа. При вызове функции значения фактических параметров копируются в формальные, таким образом происходит инициализация значений формальных параметров. Внутри функции эти

значения могут прочитываться и изменяться (не изменяя значения фактических параметров). В этом смысле формальные параметры похожи на переменные, но синтаксически их объявление и инициализация отличается.

Вопросы для самопроверки.

1. *Что такое недостижимый код?*
2. *Приведите примеры недостижимого кода. Придумайте ситуацию, когда код является недостижим, отличную от кода, находящегося за* `return`.
3. *Как синтаксически записывается функция?*
4. *Чем отличаются заголовок и прототип функции? Почему это отличие существенно?*
5. *Является ли прототип функции объявлением? Определением?*
6. *Является ли заголовок функции с ее телом объявлением? Определением?*
7. *Является ли объявление переменных в деклараторе, определением? Объявлением?*
8. *Что делать, чтобы функция могла изменить данные вызывающей функции?*
9. *Как написать функцию, имеющую более одного выходного параметра?*
10. *Что такое неявное декларирование функции?*

§2.9. Файл исходного кода

Теперь можно ввести приближенное синтаксическое определение программы (файла исходного кода, единицы трансляции) в языке Си.

⟨*элемент программы*⟩ ::=
⟨*объявление переменных*⟩
⟨*объявление типа данных*⟩
|⟨*прототип функции*⟩
|⟨*функция*⟩

⟨*единица трансляции*⟩ ::= ⟨*элемент программы*⟩ | ⟨*единица трансляции*⟩⟨*элемент программы*⟩

По определению можно заметить, что объявления переменных могут находиться вне функции, такие переменные называются **глобальными** и не рекомендуются к использованию. Объявления, сделанные внутри составного оператора, в частности в начале тела функции, являются **локальными**. Функции могут быть объявлены только глобально (хотя некоторые компиляторы свободны от этого ограничения и позволяют объявлять вложенные функции).

Вопросы для самопроверки.

1. *Что такое глобальные переменные?*
2. *Что такое объявление и определение идентификаторов?*

§2.10. Комментарий

Комментарии — исключаемые из кода программы участки файла.

Комментарии являются важнейшим элементом программы, они предназначены для записи пояснений к коду. Другое использование на практике — временное исключение строк кода при отладке программы.

В Си комментарии записываются с помощью комбинации `/* ... */`: все, что находится между этими токенами игнорируется компилятором. Комментарий может быть многострочным. При этом, в многострочных текстовых комментариях (пояснения) рекомендуется начинать каждую строку с символа “`*`” — для того, чтобы отличать его от временно исключенного кода (особенно это полезно, если текстовый комментарий написан латинскими буквами, как и код). Например,

```
/*  Многострочный
 *  текстовый
 *  комментарий
 */

/*
    int i = 5;
    k = i + 1;
```

```
*/
```

С комментарием связно одно важное правило написания хорошего программного кода: начинать программу следует с комментариев. Для всякой программы, модуля, функции, блока должно быть четко сформулировано, что этот элемент делает. Если программист не может сформулировать что-то на естественном языке, то он не может это запрограммировать. Комментарий должен быть предложением, а не обрывком фразы.

Вопросы для самопроверки.

1. *Что такое комментарий?*
2. *Как записывается комментарий?*
3. *Как используется комментарий?*

§2.11. Область видимости идентификаторов

Всякий идентификатор имеет **область видимости** — участок кода программы, в котором данный идентификатор определен и может быть использован. Область видимости идентификатора определяется следующим образом:

- Область видимости идентификатора, объявленного внутри (в начале) блока операторов (составного оператора) ограничивается данным блоком операторов, то есть такой идентификатор виден от момента объявления до конца блока. В том числе переменная или пользовательский тип данных, объявленные внутри некоторой функции, видны только внутри этой функции.
- Идентификатор, объявленный вне функции (включая само объявление функции, которое в стандартном Си может осуществляться только вне функции) виден во всем файле исходного кода от момента объявления до конца файла.
- Идентификаторы-метки видны во всем теле функции, могут совпадать с другими идентификаторами без перекрытия.
- Параметры функции видны только в той функции, параметрами которой они являются.
- Из области видимости идентификатора исключается подмножество кода, являющееся областью видимости другого одноименного идентификатора, т.е. идентификатора, объявление которого **перекрывает** имеющийся идентификатор.

Чтобы понять суть последней фразы следует отметить, что объявлять одинаковые идентификаторы в одной области видимости не допускается, но перекрывать объявление идентификатора в блоке следующего уровня вложенности можно, но до конца этого блока будет виден (доступен) только один, самый внутренний идентификатор. Перекрытие глобальных переменных также возможно, перекрытие параметров не допускается.

Пример определения области видимости идентификаторов показан на рис. 2.1. Также следует помнить, что переменная, определенная в теле цикла, не видна в условии цикла, следующий код также создает бесконечный цикл.

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

здесь внутри тела цикла виден внутренний идентификатор `i`, а в условии виден только внешний, который в теле не изменяется. Если в случае с циклом с предусловием такая ситуация в коде обычно интуитивно понятна, то аналогичный цикл с постусловием осознать сложнее:

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

цикл также бесконечен, так как в условии используется внешняя переменная `i`, несмотря на то, что ее использование стоит после объявления внутренней переменной `i`. Это еще один из поводов отказаться от использования циклов с постусловием.

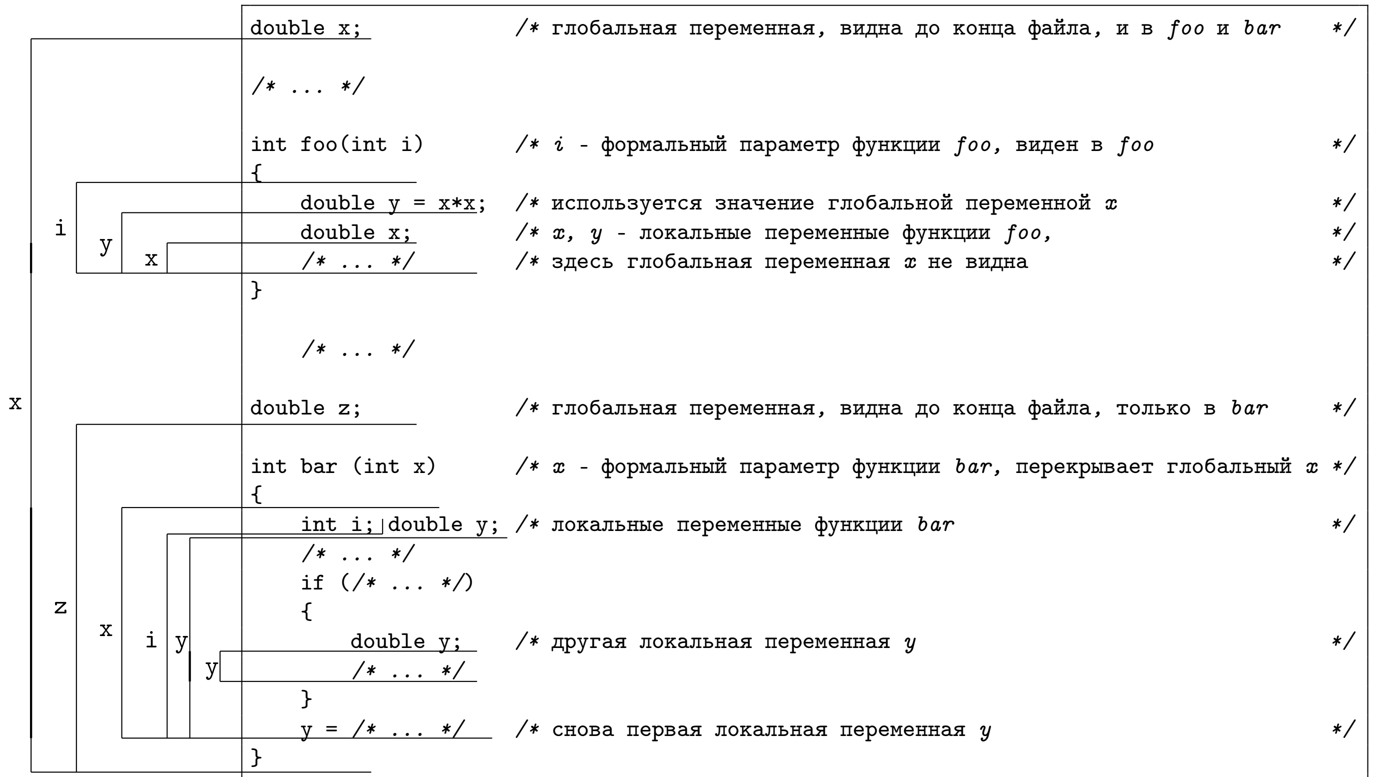


Рис. 2.1: Примеры области видимости идентификаторов переменных (показаны линиями). Жирными линиями отмечены области, перекрываемые более внутренним идентификатором. Области видимости идентификаторов функций не показаны (от момента определения, являющимся в данном случае объявлением, до конца файла).

Ключевой аспект области видимости идентификаторов в том, что одноименные идентификаторы, находящиеся в разных областях видимости, по сути являются разными идентификаторами. Они могут иметь разный смысл (разный тип, быть в одном случае идентификатором переменной, в другом — идентификатором типа данных и т.п.) При этом одноименные переменные разных областей видимости хранятся в разных ячейках памяти. Таким образом, появляется возможность независимого написания отдельных блоков кода и отдельных функций с использованием удобных для данного блока или функции переменных, без риска, что они будут использованы или изменены в другом месте кода.

Вопросы для самопроверки.

1. *Что такое область видимости идентификатора?*
2. *Как определяется область видимости идентификатора?*
3. *Что такое перекрытие идентификатора?*
4. *Как связаны одноименные идентификаторы в разных областях видимости?*

§2.12. Препроцессор

Заголовочные файлы

Глядя на определение программы (единицы трансляции), можно заметить, что функции и их прототипы могут чередоваться между собой. Более того, функция может не иметь прототипа и быть объявленной определением. *Следует отметить, что в функции корректно вызывать только те функции, которые объявлены в программе ранее*, так как область видимости идентификатора функции начинается с момента объявления.

Если функция не объявлена, но делается попытка ее вызова, Си считает, что функция объявляется неявно и все ее параметры и возвращаемое значение имеют тип `int`, что чаще всего далеко от истины, поэтому *вызов необъявленной функции является ошибкой программирования.*

В большинстве простейших случаев, конечно, можно указывать тела всех функций до того, как они будут использованы. Однако, что делать, если функция библиотечная? В этом случае в программе необходимо и

достаточно задать ее прототип. Ясно, что копировать вручную прототип каждой функции каждой библиотеки как минимум неудобно, поэтому каждая библиотека предоставляет набор **заголовочных файлов**, обычно с расширением `.h` (header). С точки зрения синтаксиса это тоже файлы исходного кода программы, как правило содержащих прототипы функций (и другие объявления — типов данных и т.п.): для обращения к объявленным там идентификаторам достаточно просто добавить код заголовочного файла в код программы.

Это можно сделать автоматически с помощью **препроцессора** — этапа обработки кода программы до собственно компиляции. Обычно препроцессор вызывается компилятором автоматически или является его составной частью. Препроцессор обрабатывает особые инструкции — **директивы**.

Директивы — синтаксические единицы препроцессора, всегда записываемые в отдельной строке и начинающиеся с символа `#`. Одной из важнейших директив является директива `#include`. Ее формат следующий:

⟨*файл*⟩ ::= `"`⟨*имя файла*⟩`"`|`<`⟨*имя файла*⟩`>`

⟨*директива include*⟩ ::= `#include` ⟨*файл*⟩

Директивы записываются в отдельной строке без точки с запятой в конце. Имя файла может быть указано в кавычках, тогда предполагается, что это пользовательский файл, и компилятор ищет в том же каталоге, где находится код программы. Если имя файла указано в треугольных скобках, то это предполагается библиотечный файл и он ищется в стандартных каталогах системы, где находятся заголовочные файлы библиотек, точнее — в каталогах поиска, подключенных в настройках компилятора.

Пример:

```
#include <stdio.h>
```

Встретив директиву `#include` препроцессор найдет указанных файл (или сообщит о фатальной ошибке, если такого файла нет) и вставит все его содержимое вместо директивы как единое целое.

Символические константы

Второй важной директивой является директива `#define`, которая позволяет в частности задать псевдоним (идентификатор) для явной константы.

Использование сложных явных констант непосредственно в программе не поощряется: например, если она используется несколько раз, ее сложнее скорректировать, константа может быть длинной и т.д. Поэтому рекомендуется отдавать предпочтения символическим константам. Константы с нетривиальным значением (отличным от 0, 1 и т.п.) называют *магическими числами* или значениями. Для магических чисел рекомендуется вводить символические константы с говорящими именами. Для тривиальных констант использование символических констант, наоборот, не рекомендуется.

Формат задания константы:

⟨*директива define*⟩ ::= `#define` ⟨*идентификатор*⟩ ⟨*код*⟩

На самом деле в качестве кода может быть подставлена любая последовательность символов, встретив идентификатор препроцессор заменит его на указанный код.

Например, в некоторых компиляторах, в файле `math.h` определена константа `M_PI`:

```
#define M_PI 3.14159265358979323846
```

и другие математические константы, хотя они и не являются частью стандарта Си.

Символические константы принято записывать заглавными буквами, в противоположность идентификаторам переменных и функций, которые записываются строчными буквами, но это не требование языка, а соглашение.

Символические константы не являются идентификаторами языка Си — их объявления де-факто глобальны, их нельзя "перекрыть" объявлениями локальных переменных, так как они будут заменены на значение в любом месте, в т.ч. и в объявлении или в присваивании, что приведет к синтаксической ошибке. Поэтому важно использовать такие константы с осторожностью и выделять их регистром, чтобы не создать путаницы с переменными.

В приложении D.2 описаны другие возможности применения данной директивы.

Вопросы для самопроверки.

1. *Зачем нужны заголовочные файлы библиотек?*
2. *Что представляют собой заголовочные файлы?*
3. *Что такое препроцессор и его директивы?*
4. *Что делает директива `#include` и какой у нее синтаксис?*

5. *Что делает директива* `#define` *и какой у нее синтаксис?*
6. *Что такое символическая константа?*
7. *Какими буквами принято записывать идентификаторы переменных? Функций? Символических констант?*

§2.13. Перечисления

Перечисляемый тип — тип данных, множество значений которого есть конечный набор идентификаторов.

Действительно, перечисление — особый тип данных, но не встроенный в язык, а создаваемый программистом. Такой тип данных требует объявления — присвоение имени типа и указания набора идентификаторов, которые может принимать значения данного типа.

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;
```

Синтаксически объявление перечисляемого типа является объявлением нового идентификатора, и может встречаться в начале блока операторов или в программе вне тел функций.

Такое объявление, во-первых, декларирует идентификатор типа 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, например

```
enum day d = MON;
```

Во-вторых, декларируются константы вводимого типа, по факту целочисленные, но если значения не указаны явно, они выбираются компилятором автоматически и обычно нумеруются с нуля, возрастая на 1. Однако, их можно задать и явно следующим образом:

```
enum day {SUN = 0, MON, TUE, WED, THU, FRI, SAT = 1};
```

однако это может привести к дублирующимся значениям (которые не запрещены). В данном случае и MON и SAT будут равны 1.

В третьих, в таком деклараторе можно также сразу объявить переменные:

```
enum day {SUN, MON, TUE, WED, THU, FRI, SAT} d1, d2;
```

На самом деле можно опустить объявление идентификатора типа, тем самым декларируя **безымянное** (называемое также **анонимным**) перечисление. В этом случае объявление переменных возможно только сразу в декларации перечисления, но не в дальнейшем коде, в котором, однако, будут доступны объявленные константы:

```
enum {SUN, MON, TUE, WED, THU, FRI, SAT} d1, d2;
```

Также можно ограничиться лишь объявлением констант:

```
enum {SUN = 1, MON = 2, TUE = 3, WED = 4, THU = 5, FRI = 6, SAT = 7};
```

Отличие констант-перечислений от символических констант (`define`) в том, что перечисления определяются компилятором, а не препроцессором, подчиняются области видимости (которой посвящен следующий параграф) для переменных, то есть видны только в том блоке, где объявлены, а не до конца Си-файла, а также могут быть только целого типа. Однако *их тоже принято указывать заглавными буквами.*

Формально синтаксис объявления перечисления следующий:

⟨*enum-префикс*⟩ ::= `enum` | `enum` ⟨*идентификатор*⟩

⟨*enum-константа*⟩ ::= ⟨*идентификатор*⟩ | ⟨*идентификатор*⟩ `=` ⟨*выражение*⟩

⟨*список enum-констант*⟩ ::= ⟨*enum-константа*⟩ | ⟨*список enum-констант*⟩`,`⟨*enum-константа*⟩

⟨*список enum-переменных*⟩ ::= ⟨*идентификатор*⟩ | ⟨*список enum-переменных*⟩`,`⟨*идентификатор*⟩

⟨*объявление enum-переменных*⟩ ::= | ⟨*список enum-переменных*⟩

⟨*enum-объявление*⟩ ::= ⟨*enum-префикс*⟩ `{` ⟨*список enum-констант*⟩ `}` ⟨*объявление enum-переменных*⟩ `;`

Вопросы для самопроверки.

1. *Что такое перечисляемый тип?*
2. *Что декларирует* `enum`*?*
3. *Что такое анонимные перечисления и как они могут использоваться?*
4. *Чем отличаются константы-перечисления от символических констант?*

§2.14. Декларация typedef

С помощью ключевого слова `typedef` можно декларировать псевдонимы для типов данных, что также является объявлением идентификаторов типа данных. Общий принцип, объясняющий синтаксис использования этого ключевого слова, следующий: если предварить объявление переменной `typedef`, то это превратиться в объявление типа данных. Например, можно написать

```
typedef float flt;
```

после чего `flt` станет именем типа, равнозначным `float`.

Это далеко не бессмысленное действие: если вдруг при развитии программы придется перейти от одинарной точности к двойной, то при наличии такого `typedef` достаточно будет заменить эту строку на

```
typedef double flt;
```

вместо того, чтобы производить замену во всех функциях типов параметров, переменных, возвращаемых значений.

Также с помощью `typedef` можно "избавиться" от необходимости дублировать `enum` перед именем перечисления:

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

day d = MON;
```

обратите внимание, что декларация

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

была бы объявлением переменной-перечисления `day`, а в декларации выше `day` стал именем типа данных, причем `enum day` и `day` становятся синонимами. Можно использовать и анонимное перечисление:

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

day d = MON;
```

но тогда объявленным будет считаться исключительно `day`, `enum day` не объявлен.

Заметим, что объявления типов данных, в т.ч. `typedef`-декларации чаще всего делаются глобальными, так как типы данных часто служат для обмена данных между функциями (с использованием параметров такого типа). Для доступности библиотечных типов данных, их помещают в заголовочный файл.

Например, в стандарте C99 имеется заголовочный файл `<stdint.h>`, в котором сделаны `typedef`-декларации стандартизированных целочисленных типов (в каждом конкретном компиляторе, для каждой конкретной платформы они должны быть введены как псевдонимы соответствующих нестандартизированных целочисленных типов; они доступны только если типы данных такой длины позволяет хранить архитектура процессора). В частности, имеются типы данных фиксированной длины, соответственно 8, 16, 32 и 64 бита: знаковые (`int8_t`, `int16_t`, `int32_t` и `int64_t`) и беззнаковые (`uint8_t`, `uint16_t`, `uint32_t` и `uint64_t`) и другие. Их использование предпочтительно, когда необходимо гарантировать количество байт, отводимых под целое число.

Также, в заголовочном файле `<stddef.h>` определен тип данных `size_t` (с помощью typedef), предназначенный для хранения размеров. Он не стандартизирован, приравнивается к одному из целых беззнаковых типов (чаще всего наибольшему). Значения данного типа, в частности, возвращает операция `sizeof`. Кроме того, для данного типа гарантируется, что его значение может хранить в себе любой индекс массива. Об использовании значений данного типа речь пойдет в соответствующих главах.

Глава 3. Семантические особенности Си

Предыдущая глава была посвящена **синтаксису** Си (хотя и не содержала его полное описание — синтаксис будет дополняться по мере введения новых понятий). Данная глава в большей степени посвящена **семантике**, то есть то, каким исполняемым действиям соответствуют синтаксические конструкции языка. В дальнейшем и синтаксис и семантика будут уточняться и дополняться, также будут изучаться **общие принципы программирования** (в дополнение к уже изученным, понятиям таким как, например, общее понятие функции, переменной, ячейки памяти, адреса, уровней абстракции и т.п., а также правила “хорошего” кода — которые не связаны с конкретным языком), и также **общематематическим основам программирования** (например, представление и обработка данных, алгоритмы и их анализ).

В учебном курсе было бы сложно раздельно изложить каждую из этих в общем-то независимых частей — однако, при изучении следует обращать внимание, к какой именно из этих четырех сторон программирования относится материал и как понятия из разных сторон взаимосвязаны между собой.

§3.1. Арифметические операции

В Си существуют следующие группы арифметических операций.

1. **Унарный плюс и минус**. Обе операции префиксные, работают как с целыми типами, так с числами с плавающей точкой.

   Унарный плюс фактически не изменяет значение, возвращает операнд, с сохранением типа.

   Унарный минус возвращает противоположное число, то есть число, которое в сумме с операндом дает 0 (в том числе для беззнаковых типов — возвращает число того же типа, которое в сумме с операндом дает ноль в результате переполнения).

   Обе операции возвращают значение, тип которого совпадает с типом операнда (в частности, не переводят из беззнакового в знаковое целое, не меняют длину и т.п.).

2. **Бинарные арифметические операции**:

   `+` сложение,
   `-` вычитание,
   `*` умножение,
   `/` деление,
   `%` вычисление остатка.

   Операции возвращают результат соответствующего действия.

   При применении к целым числам и числам с плавающей точкой одного типа (например, сумма двух значений типа `int`, разность двух значений типа `float` и т.п.), тип возвращаемого значения совпадает с типом операндов (в данных примерах — `int` и `float` соответственно). Если один из операндов — целый, а другой — например, произведение `long` и `float`, разность `double` и `short`), то целый операнд приводится к типу операнда с плавающей точкой и тип результата соответствует данному типу с плавающей точкой (в данных примерах — `float` и `double` соответственно). Если оба операнда целые или оба — с плавающей точкой, но разной длины (например, сумма `short` и `int`, разность `float` и `double` и т.п.), тип возвращаемого значения совпадает с более длинным типом операндов (в данных примерах — `int` и `double` соответственно).

   Заметим, что тип результата не зависит от того, в какую переменную он записывается.

```
int i = 100000;

long j = i * 100000;

long k = i * 100000L;
```

   Если тип `int` 32-битный, а `long` — 64-битный, то при вычислении значения выражения, записываемого в `j`, возникает переполнение (например, запишется `1410065408`, так как оба операнда имеют тип `int`), а в `k` без ошибок запишется `10000000000`, так как один из операндов имеет тип `long` благодаря суффиксу.

   Также, если оба операнда целые, то значение все равно будет целым, в том числе деление будет целочисленным, то есть вычислится неполное частное:

```
double d = 1/3;
```

В `d` запишется `0`. Если нужно получить одну треть, то нужно явным образом указать тип констант:

```
double d = 1.0/3;
```

```
double d = 1/3e0;
```

```
double d = 1./3;
```

```
double d = 1/3.;
```

и др. Вариант типа

```
double d = 1f/3;
```

возможен, но даст отличный от примеров выше результат: `1f` — число типа `float` (суффикса для `double` нет) и значение отношения будет иметь меньшую точность, чем переменная `d`.

Можно также явно привести тип вручную, используя операцию приведения типа, описываемую ниже:

```
double d = (double) 1/3;
```

или даже

```
double d = 1/ (double)3;
```

Только приведение типа позволит достичь деления с плавающей точкой в случае, когда нужно получить отношение двух целочисленных переменных или выражений, а не констант:

```
int n = 1, m = 3;
double d = (double) n/m;
```

или

```
double d = n/ (double)m;
```

Вычисление остатка применимо только к целым числам.

3. **Инкремент и декремент**. Применяются к целочисленным переменным, увеличивают значение переменной на 1 (инкремент, операция `++`), или уменьшают на 1 (декремент, операция `--`): операция не только возвращают измененное значение переменной, но и изменяют содержимое самой ячейки памяти, то есть имеет побочный эффект.

   Например,

```
int i = 1;
--i;
```

В `i` окажется 0. Заметим, что вторая строка — пример осмысленного оператора-выражения: вычисленное значение этого выражения никак не используется, но из-за побочных эффектов применение такого оператора изменяет данные программы.

Инкремент и декремент бывают постфиксными и префиксными. Их побочный эффект одинаков — увеличение и уменьшение значения переменной на 1 соответственно — но возвращаемое значение отличается: постфиксный возвращает исходное значение переменной, префиксный — полученное.

```
int i = 1;
int j = ++i;
int k = i++;
```

В `j` запишется 2 (новое значение `i`), в `k` тоже запишется 2 (старое значение `i`), а в самом `i` окажется 3.

Применять к константам и выражениям инкремент и декремент нельзя.

Тип возвращаемого значения инкремента и декремента равен типу изменяемой переменной.

Вопросы для самопроверки.

1. *Какие арифметические операции есть в Си?*
2. *Чем отличается постфиксный и префиксный инкремент (декремент)?*
3. *Как определяется тип результата арифметических операций?*
4. *Какую особенность имеет операция деления в Си?*
5. *Зачем нужен унарный плюс?*
6. *Что такое побочный эффект операции?*

§3.2. Операции присваивания

Присваивание — операция, чей побочный эффект связан с записью значения в переменную.

Инициализация позволяет установить значение переменной при ее создании (объявлении), которое делается в начале операторного блока. В ходе работы программы можно изменить значение переменной с помощью операции присваивания.

Прямое присваивание

Знак операции прямого присваивания есть символ равенства `=`. Ее левый операнд должен быть переменной, правый — выражением соответствующего типа, которое и записывается в переменную.

```
int k = 2;
k = 2 * k + 1;
```

Во многих языках присваивание осуществляется с помощью оператора, в Си это именно операция, то есть она не только изменяет значение переменной, а возвращает его (новое значение). Тип возвращаемого значения всех операций присваивания равен типу изменяемой переменной. Порядок ассоциативности присваивания — справа налево, поэтому возможна запись

```
i = j = 2 * 5;
```

В этом случае сначала правое присваивание запишет в `j` значение 10 и вернет его, а возвращенное значение присвоится переменной `i`.

Для операций присваивания документировано, что сначала вычисляется правый операнд, поэтому такие выражения работают корректно.

Арифметическое присваивание

Кроме операции прямого присваивания есть операции **арифметического присваивания**. В программировании часто встречаются действия типа “увеличить значение переменной в 2 раза”, разделить на 3 и т.п. 5 операций арифметического присваивания работают следующим образом:

знак	запись	эквивалент
`+=`	`a += ` *expr*`;`	`a = a + (`*expr*`);`
`-=`	`a -= ` *expr*`;`	`a = a - (`*expr*`);`
`*=`	`a *= ` *expr*`;`	`a = a * (`*expr*`);`
`/=`	`a /= ` *expr*`;`	`a = a / (`*expr*`);`
`%=`	`a %= ` *expr*`;`	`a = a % (`*expr*`);`

где ***expr*** — некоторое выражение. Они тоже возвращают полученное значение, которое может быть использовано в выражении.

Обратите внимание на скобки в правом столбце. Запись

```
a *= x + y;
```

будет вычислена как

```
a = a * (x + y);
```

а не как

```
a = a * x + y;
```

последнюю можно было бы записать как `(a *= x) += y`, но стандарт Си такую форму не поддерживает: возвращаемое значение операций присваивания (как, кстати, и инкремента) — только значение, но не сама переменная. А запись

```
a *= x += y;
```

вполне корректна и в силу порядка ассоциативности равносильна записи

```
a *= (x += y);
```

то есть сначала вычислится правое присваивание и изменит значение `x` — результат в `a` будет тот же, что и для `a = a * (x + y);`, но появится дополнительный побочный эффект изменения значения `x`.

Использовать арифметическое присваивание полезно, так как программа записывается короче и в общем случае может работать быстрее: вычисление проводится непосредственно в ячейке памяти, а не сохраняется во временной ячейке и копируется в переменную.

Заметим, что большинство современных компиляторов могут *оптимизировать* код, то есть создавать более эффективный по времени работы и используемой памяти код, чем если бы конструкции языка программирования заменялись на машинные инструкции напрямую. В частности, компилятор может выявить лишние временные ячейки памяти и перемещения, то есть в данном примере заменить менее оптимальный исходный код на машинный код, соответствующий исходному коду с арифметическим присваиванием. Однако оптимизация не всегда гарантируется и не всегда возможна по ряду причин, кроме того ее произвести легче при более качественном коде, поэтому пренебрегать тем фактом, что разные синтаксически, но семантически эквивалентные конструкции, могут работать быстрее, не рекомендуется.

Вопросы для самопроверки.

1. *Как записывается операция присваивания в Си и что она делает?*
2. *Чем отличается присваивание от инициализатора?*
3. *Что такое операции арифметического присваивания, как они записываются, как работают, каковы их свойства (порядок, приоритет, возвращаемое значение)?*

§3.3. Операции сравнения, логические операции, условная операция

Операции сравнения

Операции сравнения на вход берут числовые данные, возвращают данные типа `int`, значение может быть либо 0 (ложь), либо 1 (истина).

`<`	строго меньше (1 если левый операнд строго меньше правого, 0 в противном случае)
`<=`	меньше либо равно (1 если левый операнд меньше либо равен правому, 0 в противном случае)
`>`	строго больше (1 если левый операнд строго больше правого, 0 в противном случае)
`>=`	больше либо равно (1 если левый операнд больше либо равен правому, 0 в противном случае)

`==` равно (1 если операнды равны, 0 если не равны)

`!=` не равно (1 если операнды не равны, 0 если равны)

Следует обратить внимание на разницу между операцией сравнения на равенство “==” и присваивание “=”. Существует риск перепутать их в условии, например, в циклах и условном операторе:

```
while (season = WINTER) do_snow();
```

если символическая константа `WINTER` не 0, то операция присваивания ее в переменную `season` вернет ненулевое значение (истину) и цикл будет бесконечен. Современные компиляторы предупреждают о потенциальной ошибке использования присваивания вместо сравнения. Чтобы подавить это предупреждение предлагается использовать двойные скобки, если действительно нужно присваивание, а не равенство.

Операции сравнения целых чисел корректны только если сравниваются два знаковых или два беззнаковых значения, но не знаковое и беззнаковое между собой. Это связано с особенностью представления данных: например, при использовании дополнительного кода, сравнение -1 (знаковый) и 5 (беззнаковый) может выполняться двояко: можно привести -1 к беззнаковому и получить самое большое число данного типа (например, в однобайтовом типе -1 знаковое представление — 11111111; при его прочтении как беззнакового получится 255, что больше 5) или 5 привести к знаковому (оно останется равным 5, что больше -1). В данном случае математически корректный результат дал второй способ, но если сравнивать -1 и слишком большое, чтобы быть приведенным к знаковому, беззнаковое число (например, 254 в однобайтовом целом), то ни один способ не даст верный результат: при приведении к знаковому $-1 < 254$ прочитается как $-1 < -2$ (ложь) к беззнаковому — как $255 < 254$ (тоже ложь). Поэтому программисту необходимо исключить такие сравнения и самостоятельно приводить к нужному (например, более длинному, если возможно) типу данных.

Логические операции

Логические операции на вход берут любые данные, возвращают данные типа `int`, значение может быть либо 0 (ложь), либо 1 (истина).

`&&` конъюнкция (1 если оба операнда не 0, 0 если хотя бы один операнд 0)

`||` дизъюнкция (1 если хотя один операнд не 0, 0 если оба операнда 0)

`!` отрицание (унарная, 1 если операнд ноль, 0 если операнд не ноль)

Важно осознать, что логические операции и операции сравнения не отдельная структурная или синтаксическая единица языка, известная как “условие”. Это именно операции, как и арифметические операции: они имеют операнды и возвращают значение, которое может быть использовано в выражении.

Поэтому не следует ожидать, что выражение

```
int a = 5;
int b = 7 > a > 2;
```

проверит нахождение `a` между 2 и 7. На самом деле сначала вычислится значение `7 > 5` (истина, 1), потом `1 > 2` (ложь, 0) — в переменную `b` запишется 0. Аналогично

```
int x = 5, y = 5;
int b = x == y == 5;
```

запишет в `b` число 0 (`1 == 5` — ложь), а не результат проверки равенства трех чисел. Подобные условия следует реализовывать с помощью логических операций

```
if (7 > a && a > 2)
{
    /* a лежит между 2 и 7 */
}
/* ... */
if (x == y && y == 5)
{
    /* и x и y равны 5 */
}
```

Тернарная условная операция

Одним из способов проверки условий в Си является тернарная условная операция. Ее синтаксис следующий

$$e_1 ? e_2 : e_3$$

Если выражение e_1 имеет ненулевое значение (истина), то возвращается значение выражения e_2, если e_1 есть ноль (ложь), то возвращает значение выражения e_3.

Выражения e_2 и e_3 должны иметь одинаковый тип.

Пример

```
int max = n > m ? n : m;
```

запишет в `max` наибольшее из чисел `n` и `m`. Эта операция — существенное удобство Си, часто позволяет сократить запись и избежать более громоздкого условного оператора.

Вопросы для самопроверки.

1. *Как представляются логические значения в Си?*
2. *Какие операции сравнения есть в Си и как они работают?*
3. *Чем отличается операция присваивания от операции равенство?*
4. *Почему нельзя сравнивать знаковые и беззнаковые целые числа?*
5. *Какие логические операции есть в Си?*
6. *Как записывается и работает тернарная условная операция?*
7. *Как корректно проверить равенство значений трех переменных?*

§3.4. Битовые операции

В отличие от обычной арифметики, применяемой к числам с учетом их кодирования, **битовые операции** применяются к отдельным битам, т.е. применяются к каждому биту данных или к паре битов (для бинарной операции) независимо, но для всех битов одновременно.

Операции битовой арифметики — операции, выполняющие логические действия над отдельными битами.

`~` побитовое отрицание (унарная): инвертирует каждый бит операнда

`&` побитовая конъюнкция (бинарная): к каждой паре бита левого и правого операнда применяется конъюнкция

`|` побитовая дизъюнкция (бинарная): к каждой паре бита левого и правого операнда применяется дизъюнкция

`^` побитовое разделительное или (бинарная): к каждой паре бита левого и правого операнда применяется разделительное или

Следующие примеры приводятся для типа `unsigned char`.

x	01110001 (113)
~x	10001110 (142)

x	01100011 (99)
y	10101110 (174)
x&y	00100010 (34)
x|y	11101111 (239)
x^y	11001101 (205)

Аналогично арифметическим операциям, тип результата битовых операций — более длинный из типов операндов.

Не следует путать битовые и логические операции, в частности использовать битовую конъюнкцию вместо логической: они могут дать разный результат.

x	01100011 (истина)
y	10001100 (истина)
x&y	00000000 (ложь)
x&&y	00000001 (истина)

Операции **битового сдвига** “сдвигают” биты внутри ячейки вправо (>>) или влево (<<) на значение, указанное во втором операнде, который должен быть целочисленным.

Биты, выпадающие за ячейку, удаляются, выходящие биты объявляются равными 0 (сдвиг не циклический).

x	00100011 (35)
x<<1	01000110 (70)
x<<2	10001100 (140)
x<<3	00011000 (24)
x<<4	00110000 (48)
x<<5	01100000 (96)
x<<6	11000000 (192)
x<<7	10000000 (128)
x<<8	00000000 (0)
x>>1	00010001 (17)
x>>2	00001000 (8)
x>>3	00000100 (4)
x>>4	00000010 (2)
x>>5	00000001 (1)
x>>6	00000000 (0)

Правым операндом битового сдвига должно быть целочисленное значение, тип результата совпадает с типом левого операнда.

Фактически сдвиг вправо — целочисленное деление на указанную степень двойки, влево — умножение, с оговоркой на возможное переполнение (в примере выше оно произошло трижды).

Также существуют соответствующие операции присваивания: `&=`, `^=`, `|=`, `<<=`, `>>=`.

Битовые операции “чужды” привычному (арифметическому) восприятию чисел, но они ближе к аппаратной организации компьютера и работают наиболее быстро.

С помощью битового И можно проверить, какое значение имеет конкретный бит числа:

```
b = a & 1 << 5;
```

`1<<5` перемещает 1 на пятый бит (сдвиг имеет более высокий приоритет), остальные биты будут 0, значение `b` будет 0 (ложь), если пятый бит переменной `a` является 0, 1 (истина) если он является 1.

```
b = a & 1 << 5;
```

Для того, чтобы установить отдельный бит в 1 можно использовать

```
a |= 1 << 5;
```

занулить —

```
a &= ~(1 << 5);
```

С помощью битовой конъюнкции можно быстро вычислять остаток от деления на степень двойки, например остаток от деления n на 8 вычисляется как `n & 7`, в общем виде остаток от деления на 2^m — с помощью выражения вида `n & ((1 << m) - 1)`.

Операции битового сдвига следует применять осторожно для знаковых чисел, т.к. компилятор для отрицательных чисел может осуществлять т.н. *знаковый сдвиг вправо*, добавляющий слева 1, а не 0. Вообще говоря, *сдвиг вправо для знакового типа неопределен.* Битовые операции зависят от аппаратного представления целых чисел: это может быть не только дополнительный код.

Вопросы для самопроверки.

1. *Что такое операции битовой арифметики?*
2. *Чем отличается битовая конъюнкция от логической?*
3. *Что такое битовый сдвиг?*
4. *Как проверить значение отдельного бита?*
5. *Как установить значение отдельного бита? Объясните как это работает на примере.*
6. *Какие операции обеспечивают присваивание совмещенное с битовыми операциями?*

§3.5. Адреса и указатели

Как уже было сказано, доступ к данным осуществляется по их адресу в адресном пространстве. Например, переменные, формальные параметры и функции соответствуют ячейкам памяти с некоторым адресом. Причем, для глобальных переменных и функций этот адрес вычисляется, например, от начала кода программы, а для локальных переменных и формальных параметров функции — относительно начала блока данных этой функции. В любом случае в исполняемый код программы вместо идентификаторов переменных, функций и формальных параметров встраивается некоторое число (адрес), поэтому переменную можно назвать адресной константой. От символических констант ее отличает то, что реальное значение адреса (номер ячейки) вычисляется компилятором, а не задается программистом.

Всякая функция принимает аргументы по их значению, то есть значения аргументов (выражений) копируются в соответствующие места блока данных функции, которые отведены под параметры. При этом вызываемая функция *не может* изменить значения блока данных вызывающей функции через изменение значений параметров: это разные ячейки памяти.

Например, наивная попытка написать функцию, которая меняет местами значения двух переменных терпит неудачу: следующая функция не имеет побочных эффектов.

```
void swap (int i, int j)
{
    int k = i;
    i = j;
    j = k;
}
```

Ее работа проиллюстрирована на рис. 3.2

Поэтому, чтобы функция могла изменять данные другой функции, необходимо передать не значение аргумента, а *адрес* той ячейки памяти, которую нужно изменить.

В Си существует особый способ доступа к памяти по адресам — **указатели**. Указатель — это особый тип данных, хранящий адрес ячейки памяти. Указателем также называется сама переменная такого типа, которая **"указывает"** на ячейку памяти.

Размер указателя (в битах) определяется архитектурой процессора — 16, 32, 64 и др. Характеристика процессора и системы, известная как *разрядность* по сути и есть размер указателя, определяющий наибольшее число ячеек адресного пространства, которое можно задать в адресе, и, соответственно, объем доступного адресного пространства. Для 32-разрядных систем это значение составляет 4 Гб, для 64-разрядных — 16 Тб.

Память однородна, поэтому, вообще говоря, все указатели могли бы иметь один и тот же тип, однако такой подход часто бывает неудобным — в этом случае указатель не содержит информации, данные какого типа хранятся в ячейке, на которую он указывает, что позволяет программисту допустить ошибку, которую не заметит компилятор: перепутать тип данных. Поэтому главным образом используются **типизированные указатели**, то есть указатели тип которых соответствует типу хранимых данных.

Объявить указатель можно, указав "символ указателя" `*` перед именем переменной. Например,

```
int *p;
```

объявляет переменную `p` как указатель на данные типа `int`. В этом случае говорят, что `p` имеет тип `int *` — "указатель на `int`" — и можно было бы записать это как

```
int* p;
```

На самом деле `*` — отдельный токен и пробел вообще не нужен:

```
int*p;
```

однако по смыслу он удобен: вопрос лишь к чему относить звездочку, к переменной или к типу?

По смыслу лучше бы к типу, но на самом деле объявление переменных в Си имеет специфику: при записи

```
int* p, q;
```

`p` будет иметь тип `int *`, а `q` — тип `int`, то есть не будет указателем! Поэтому запись

```
int *p, q;
```

в меньшей степени вводит в заблуждение, а объявление двух указателей следует делать так

```
int *p, *q;
```

Имеется также возможность объявить нетипизированный указатель, его тип `void *` (но переменную типа `void` объявлять нельзя).

Аналогично можно объявить формальный параметр и возвращаемое значение функции как типизированный или нетипизированный указатель.

Следует дополнить БНФ синтаксис объявления переменных и функций данным обстоятельством, но это остается за рамками пособия.

Для работы с указателями используется ряд операций, главные из которых

- *присваивание* (=, бинарная): значение адреса можно скопировать в указатель, при этом можно присваивать только однотипные указатели или осуществлять присваивание в нетипизированный указатель указателя любого типа и обратно (работу с нетипизированными указателями следует выполнять с особой осторожностью);
- **разыменование** (*, унарная префиксная): операндом должен быть типизированный указатель, возвращается значение, хранящееся в указываемой ячейке, соответствующего типа (на самом деле возвращается сама ячейка памяти, то есть с разыменованным указателем можно работать как с полноценной переменной — присваивать значения и т.д., разыменовывать можно только типизированный указатель);
- **взятие адреса** (&, унарная префиксная): операндом может быть любая переменная, возвращается типизированный указатель, содержащий адрес этой переменной.

Пример

```
double i = 2.0;
double *p = &i, *q; /* записали в p адрес i, q не инициализирована */
*p = pow(7, *p);    /* в i запишется 49.0 */
q = p;              /* q указывает туда же, куда и p, то есть в ячейку i */
*q -= 5;            /* в i окажется 44.0 */
```

С помощью указателя можно как прочитать, так и изменить значение в ячейке памяти. Пример поясняется на рис. 3.1.

Корректный способ написать рабочую функцию `swap` — использовать указатели для передачи адреса данных, которые нужно изменить:

```
void swap (int *i, int *j)
{
    int k = *i;
    *i = *j;
    *j = k;
}
```

такая функция вызывается как `swap(&i, &j)`, вместо `swap(i,j)`, что есть неизбежное неудобство языка Си. Работа этой функции проиллюстрирована на рис. 3.3. Синтаксис "наивной версии" данной функции позволяет вызвать ее для любых целочисленных выражений, например корректной будет запись `swap(2, x + y)`, что является абсурдом с семантической точки зрения: нельзя поменять значения числа и суммы двух переменных. Проблема именно в том, что сама функция составлена семантически неверно.

Заметим, что менять местами сами указатели также бесполезно, т.е. функция

```
void swap (int *i, int *j)
{
    int *k = i;
    i = j;
    j = k;
}
```

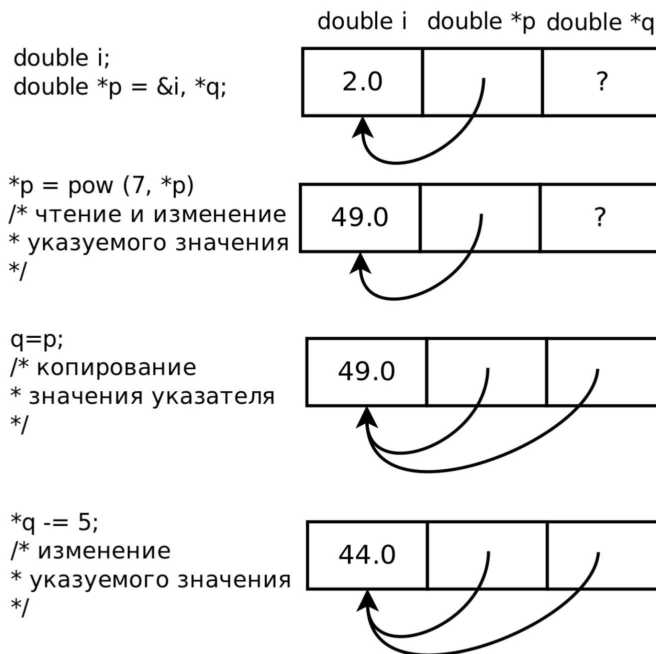


Рис. 3.1: Иллюстрация примера работы с указателем. Стрелками показаны ячейки памяти, куда указывает указатель — схематическое изображение значения указателя, представляющего собой адрес, число, точное значение которого знать невозможно и не требуется в момент написания программы, но соответствующее номеру указуемой ячейке памяти, в которую приходит стрелка. Это удобный, корректный и наглядный, способ понимать работу указателей. Следует четко отличать указатель, как отдельные данные (переменную) и данные на которые он указывает. Знак вопроса соответствует неинициализированному (непредсказуемому) значения.

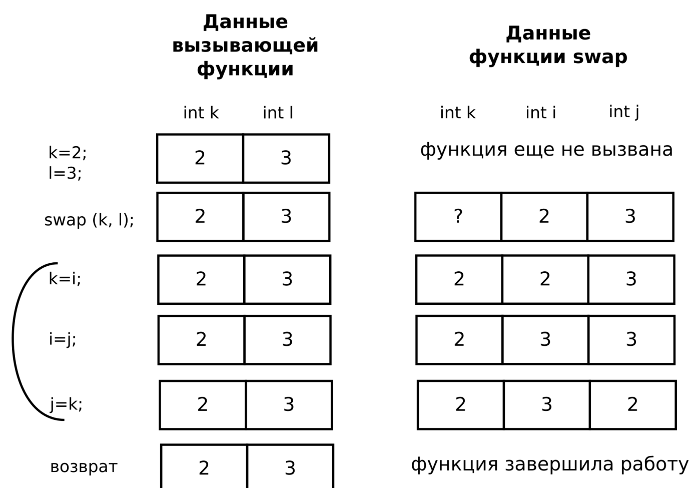


Рис. 3.2: Неверный `swap`: меняются локальные данные. Скобкой выделено исполнение кода вызванной функции.

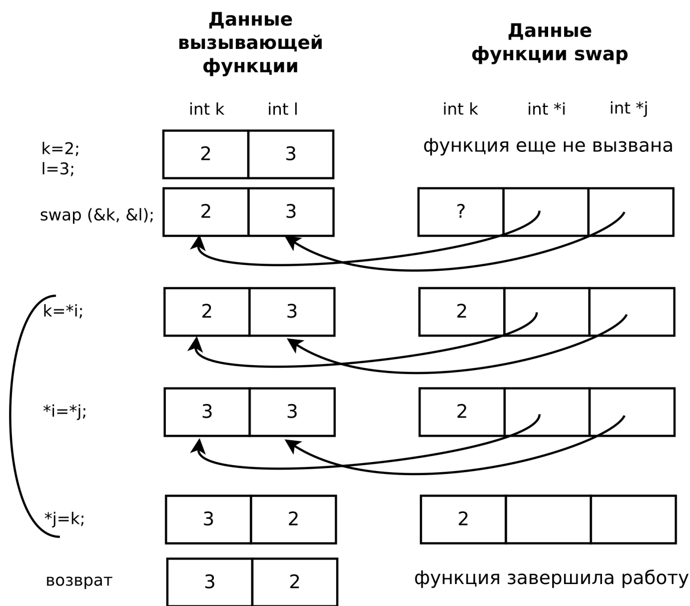


Рис. 3.3: Корректный `swap`: меняются указываемые данные. Скобкой выделено исполнение кода вызванной функции.

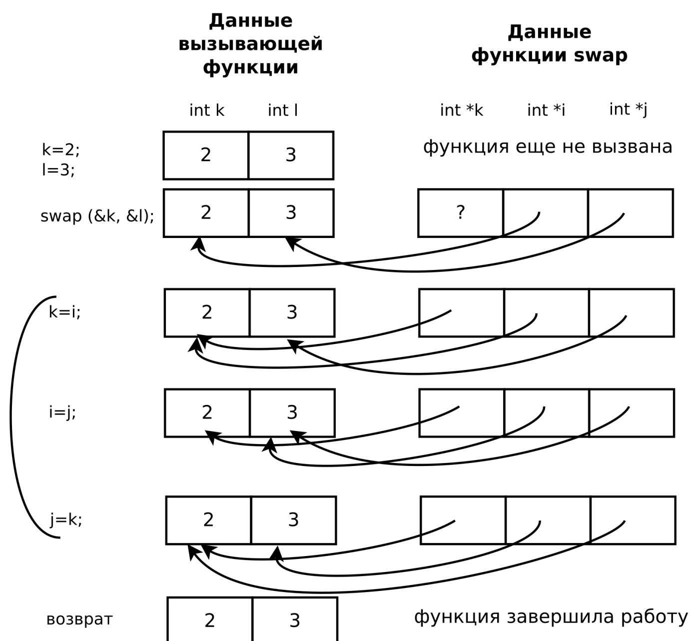


Рис. 3.4: Неверный `swap`: меняются локальные указатели. Скобкой выделено исполнение кода вызванной функции.

также не имеет побочных эффектов, меняет только внутренне данные (рис. 3.4).

С помощью параметров-указателей можно добиться того, чтобы в функции было более одного возвращаемого значения. С точки зрения архитектуры компьютера и языка Си указатель (адрес, который вычисляется операцией `&` *) является входным параметром. Однако, с алгоритмической точки зрения, ячейка, на которую он указывает может выступать как в качестве входного параметра, так и в качестве выходного (т.е. данные, содержащиеся в ней до вызова функции — входные, по завершении функции — выходные).*

Среди всех указателей выделяют нулевой, т.е. адрес со значением 0. В стандартной библиотеке (`stddef.h`) определена соответствующая символическая константа `NULL`.

Такой указатель не указывает на какую-либо определенную ячейку памяти, его нельзя разыменовывать, операция чтения-записи по данному адресу приведет к ошибке исполнения программы. В то же время такое значение переменной-указателя является четким сигналом того, что в переменной не содержится адрес какого-либо объекта, что она никуда не указывает. Это стандартное значение по умолчанию для указателя, именно им следует инициализировать указатель, если нет возможности записать в переменную адрес реальных данных.

Один из вариантов применения нулевого указателя следующий. Рассмотрим иллюстративную функцию вычисления квадратного корня из x методом Ньютона, производящую вычисления до тех пор, пока очередное приближение будет относительно отличаться от предыдущего менее чем, на ε. Пусть функция возвращает значение корня, а дополнительный выходной параметр передает число проведенных итераций:

```
/* Вычисление квадратного корня методом Ньютона
 * x - число, корень которого вычисляется
 * eps - относительная погрешность принятия последовательных приближений равными
 * *n выходной( параметр) - количество произведенных итераций
 * возвращает квадратный корень из x
 * если x < 0, x - не число или eps <= 0 возвращает не число.
 */
double sqrt (double x, double eps, int *n)
{
    double x1 = x, x2;

    if (x < 0 || x != x || eps <= 0) return 0.0 / 0.0
             /* x != x - проверка, что x является не числом,
              * 0.0 / 0.0 - получение нечисла- (NAN)
              */
    if (x == 1.0 / 0.0 || x == 0) return x;
             /* случай положительной бесконечности и
              * нуля --- результат равен x
              */



    *n = 0;
             /* Инициализация обязательна, так как иначе
              * количество итераций будет не посчитано в *n,
              * а добавлено к *n
              */

    do
    {
        x2 = x1;
        x1 = 0.5 * (x1 + x / x1)
        ++(*n);
    } while (fabs((x1 - x2) / x2) < eps);

    return x1;
}
```

Такая функция потребует от пользователя всегда заводить целочисленную переменную для хранения числа проведенных итераций, даже если эти данные не требуются. Модифицированная версия

```
/* Вычисление квадратного корня методом Ньютона
 * x - число, корень которого вычисляется
 * eps - относительная погрешность принятия последовательных приближений равными
 * *n выходной( параметр) - количество произведенных итераций игнорируется(, если NULL)
 * возвращает квадратный корень из x
```

```
 * если x < 0, x - не число или eps <= 0 возвращает не число.
 */
double sqrt (double x, double eps, int *n)
{
    double x1 = x, x2;

    if (x < 0 || x != x || eps <= 0) return 0.0 / 0.0
    if (x == 1.0 / 0.0) return x;

    if (n != NULL) *n = 0;

    do
    {
        x2 = x1;
        x1 = 0.5*(x1 + x/x1)
        if (n != NULL) ++(*n);
    } while (fabs((x1 - x2)/x2) < eps);

    return x1;
}
```

проводит вычисления с `n` только если это ненулевой указатель, поэтому если использование параметра `n` не нужно, достаточно будет вызвать

```
x = sqrt (a, 1e-3, NULL);
```

чтобы проигнорировать его. Заметим, что в отличии от числовых типов, тип указателя должен строго соответствовать типу переменной, поэтому использование

```
    int n;
    float a;
    /* ... */
    x = sqrt (a, 1e-3, &n);
```

корректно, а

```
    long n;
    double a;
    /* ... */
    x = sqrt (a, 1e-3, &n);
```

нет.

Наличие типа у указателя обусловлено не тем, что указатели имеют разный тип в смысле кодирования адреса — на самом деле в смысле способа представления данных указатели разных типов одинаковы, — а необходимостью знания типа получаемых в результате разыменования данных (а также защитой от присвоения разнотиповых указателей, для того, чтобы исключить нежелательное прочтение данных одного типа как данных другого типа).

Часто бывает полезно отказаться от типа указателя — обрабатывать указатели одинакового, независимо от того, на данные какого типа он фактически указывает. "Универсальной" формой указателя является нетипизированный указатель, объявляемый как `void *`. Такой указатель нельзя разыменовывать (более того, данных типа `void` не существует), но ему можно присвоить значение указателя любого типа. Также его можно привести к указателю любого типа.

Конечно, это может привести к грубой ошибке и непредсказуемому поведению, например,

```
    int i = 5;
    void *p;
    int *pi = &i;
    double *pd;

    p = pi;
    pd = (double *)p;

    /* Что есть *pd??? /*
```

Однако, нетипизированный указатель играет существенную роль в написании универсального (независящего от типа) кода обработки данных. Примеры использования данного указателя в стандартной библиотеке и в пользовательских функциях описаны в пособии.

Вопросы для самопроверки.

1. *Что такое указатель?*
2. *Как объявляется указатель?*
3. *Какой смысл имеет тип указателя?*
4. *Как объявляется нетипизированный указатель?*
5. *Какие операции можно выполнять с указателями?*
6. *Как получить адрес переменной (указатель на переменную)?*
7. *Как изменить значение данных по адресу (указателю)?*
8. *Что такое нулевой указатель?*
9. *Какие операции допустимы для нетипизированного указателя?*
10. *Какие ошибки можно допустить при работе с нетипизированными указателями?*
11. *Какие типичные для указателей операции недопустимы для нетипизированного указателя?*

§3.6. Константные данные

В программировании возникают ситуации, требующие ограничить возможное изменение данных. Для этого служит **квалификатор типа** `const`, используемый при объявлении переменных. Квалификаторы типа — ключевые слова, которые задают дополнительные свойства данных указанного типа, в частности `const` объявляет данные доступными только для чтения: даже если синтаксически выражение такого типа является выражением L-value, изменение его значения не допускается.

Например,

```
const int i = 1;
int j;
j = i + 1; /* корректно, прочитать значение i можно */
i = j;     /* ОШИБКА, изменять i нельзя */
i++;       /* ОШИБКА, изменять i нельзя */
```

При этом неизменяемая переменная — не то же самое, что и символическая константа: дело в том, что декларация вида

```
#define i 1
```

не создает переменную `i`, а лишь приводит к замене `i` на 1 до конца файла, константное объявление выделяет полноценную ячейку памяти и подчиняется правилам для области видимости идентификаторов.

Указатели тоже могут быть постоянными, при этом следует отличать указатель на константу и константный указатель:

```
int i = 5, j = 6;
const int *p = &i; /* указатель на константу      */
*p++;              /* ОШИБКА, изменять *p нельзя  */
p = &j;            /* корректно, изменять p можно */
```

```
int i = 5, j = 6;
int * const p = &i; /* константный указатель        */
*p++;               /* корректно, изменять *p можно */
p = &j;             /* ОШИБКА, изменять p нельзя    */
```

```
int i = 5, j = 6;
const int * const p = &i; /* константный указатель на константу */
*p++;                     /* ОШИБКА, изменять *p нельзя         */
p = &j;                   /* ОШИБКА, изменять p нельзя          */
```

Слово `const` можно указывать и после имени типа, записи слева и справа равносильны:

```
const int i = 5;
const int *p = &i;
const int * const q = p;
```

```
int const i = 5;
int const *p = &i;
int const * const q = p;
```

Однако на практике обычно используется левый вариант.

Константы требуют обязательной инициализации значения, так как его нельзя изменить присваиванием. Для указателя на константу, который сам не является неизменяемым, это, разумеется, необязательно.

```
const int i;            /* ОШИБКА, константа без значения          */
const int * const p1;   /* ОШИБКА, константа без значения          */
int * const p2;         /* ОШИБКА, константа без значения          */
const int *p3;          /* корректно, можно иницилизировать позднее  */
```

Присваивание указателя на константу указателю на неконстантные данные не является корректным, хотя и допустимым.

```
const int i = 5;
int j = 7;

const int *p;
int *q;

/* ... */

p = q;              /* корректно                    */
q = p;              /* НЕ КОРРЕКТНО, отменяет const */
q = &i;             /* НЕ КОРРЕКТНО, отменяет const */
p = &j;             /* корректно                    */
```

Отметим, что `const` следует рассматривать как "соглашение" (контракт) между программистами и компилятором:

- компилятор получает возможность применять оптимизацию с учетом того, что эти данные не будут меняться;
- программист получает уверенность, что данные, адрес которых доступен (например, в другой функции через параметр) не будут изменены.

Поэтому "превращение" константы в неконстанту — "нарушение" данного соглашения, которое следует использовать только в случае крайней необходимости. Компиляторы способны выдавать предупреждения о таких ситуациях, чтобы их подавить следует использовать явное приведение типа:

```
q = (int *) p;              /* явно отменяет const */
q = (int *) &i;             /* явно отменяет const */
```

На практике, в том числе в стандартной библиотеке, неизменяемые переменные, в том числе неизменяемые указатели, применяются редко. Однако, указатели, по которым нельзя изменять значение памяти — важный элемент программирования. Они служат для исключения ошибок, вызванных случайным изменением данных по переданному, например, в функцию указателю.

Имея данную информацию можно иметь в виду, что тип данных *строковый константы* `const char[]` совместим с типом `const char *`. Действительно, это указатель (адрес) *первого* (правильнее, конечно, нулевого) символа из ряда.

Если адрес какой-то переменной присвоен указателю на константу, то изменить значение этой переменной через данный указатель нельзя, что, однако, не исключает изменение самой переменной напрямую или через другой, неконстантный указатель:

```
int i = 5;
const int *p = &i; /* указатель на константу       */
      int *q = &i; /* обычный указатель            */
*p++;              /* ОШИБКА, изменять *p нельзя   */
*q++;              /* корректно, изменять *q можно */
i++;               /* корректно, изменять i можно  */
```

Вопросы для самопроверки.

1. *Что делает ключевое слово const?*
2. *Чем отличаются неизменяемые данные от констант?*
3. *Чем отличается указатель на константу от константного указателя?*
4. *Как объявить константный указатель на константу?*

§3.7. Главная функция

Для того, чтобы написать минимальную работающую программу, в ней нужно определить **главную функцию**. Это особая функция, имя которой есть `main`. Идентификатор `main` строго говоря не относится к ключевым словам, но функция с таким именем будет иметь особый смысл: именно ей будет передано управление при запуске программы. Простейший прототип этой функции

```
int main (void);
```

то есть функция должна возвращать `int`, а не `void`, и не иметь параметров.

Наличие целочисленного возвращаемого значения связано с тем, что всякая программа сообщает об успехе и провале своей работы операционной системе. При этом 0 — успех, а ненулевое значение — провал (идейно, чем больше число, тем выше уровень ошибки).

Таким образом, простейшая программа в Си выглядит так:

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

Все фрагменты кода из предыдущих примеров могут быть вставлены перед 3-й строкой с получением работоспособной программы, однако без вывода (да и в подавляющем большинстве случаев без ввода) данных работа программы не принесет пользы.

Вопросы для самопроверки.

1. *Что такое главная функция?*
2. *Какой прототип может быть у главное функции?*

§3.8. Форматированный ввод и вывод

Существует два способа получения входных данных в программе — жесткое кодирование (с помощью констант) и ввод с устройств ввода. Вывод выходных данных должен осуществляться на устройства вывода. Существует много различных устройств ввода и вывода и способов взаимодействия с ними, базовым является **консольный** вывод — операции ввода и вывода осуществляется с помощью текстового терминала.

В настоящее время в качестве устройства консольного ввода чаще всего выступает клавиатура, вывода — текстовое окно фиксированной длины и ширины моноширинного шрифта в графическом режиме или полный экран монитора в текстовом режиме (отметим, что текстовый режим, равно как и консольные операции, нельзя назвать “устаревшими”: они до сих пор используются во многих задачах, например, при системном и серверном администрировании, пакетной обработке файлов и т.п.).

К сожалению, стандартный ввод и вывод в стандартной библиотеке Си осуществляется с помощью “нехороших” по ряду причин функций:

- они не являются типобезопасными с точки зрения проверки типов и количества вводимых/выводимых данных, что может дать труднообнаружимые ошибки выполнения (современные компиляторы позволяют провести проверку на стадии компиляции с помощью специальных ключей);
- эти функции небезопасны с точки зрения риска перезаписать данные в памяти, что может повлечь не только непосредственное нарушение данных, взломы системы и т.д. (перезаписаны могут быть не только ячейки памяти, в которые производится попытка ввода данных, но и вообще любые данные программы);
- даже частичная проверка корректности ввода данных пользователем затруднена, полноценная во многих случаях невозможна;
- использование функций достаточно сложно.

Однако знание функций *форматированного ввода и вывода* полезно не только для языка Си, так как они являются существенным этапом развития языков программирования и находят отголоски в неожиданных местах, например, в web-программировании.

Прототипы данных функций загружаются в файле `stdio.h`, то есть в начале программы потребуется указать

```
#include <stdio.h>
```

Форматированный вывод

Вывод осуществляется с помощью функции `printf`.

`int printf(const char *format, ...);`

Параметр `format` — строка, которая будет выведена на экран, а троеточие — особый пунктуационный токен, означающий, что число параметров функции не ограничено, то есть обязательно должны присутствовать явно перечисленные параметры (в данном случае строка `format`), далее может следовать любое, в том числе нулевое, число аргументов любого типа.

Возвращаемое значение сообщает о степени успеха вывода и на первых порах может быть опущено (вызов функции — выражение, выражение имеет побочные эффект — вывод данных, и может быть использовано в качестве оператора, а его значение использовать необязательно).

Рассмотрим классический пример программы, с которой обычно начинается изучение нового языка программирования:

```
#include <stdio.h>

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

    return 0;
}
```

Обратите внимание на символ `\n`: он необходим, чтобы следующий вывод (в данном случае от других программ, оболочки и т.д.) начинался с новой строки. Заметим, что все используемые здесь языковые конструкции были определены ранее.

Можно, конечно, записать и

```
#include <stdio.h>
int main()
{
    const char *str = "Hello,␣world!\n"
    printf(str);
    return 0;
}
```

Первый параметр — выводимая строка — названа *форматом*, поскольку функция не просто выводит строку на экран напрямую, а может вывести значения выражений, передаваемые в функцию после строки, в соответствии с форматом, определяемым данной строкой.

Если в строке формата встретится символ `%`, то `printf` воспримет его как управляющую последовательность, возьмет очередное значение, в списке аргументов функции — **поле** данных — и выведет его в соответствии с указанным форматом поля.

Общий синтаксис формата поля следующий:

`%`[флаги][ширина][.точность][длина]спецификатор

указанное в квадратных скобках — все, кроме спецификатора – является необязательным.

Комбинация длины и спецификатора определяет тип данных, который ожидается в списке аргументов. Ожидаемый тип данных определяется комбинацией длины и спецификатора следующим образом:

	спецификатор					
длина	d i	u o x X	f e E g G	c	s	p
(пропущена)	int	unsigned int	float	char	const char *	void*
h	short int	unsigned short int				
l	long int	unsigned long int	double			
L			long double			

Следует обратить внимание, что форматы обрабатываются не на стадии компиляции, а на стадии выполнения, поэтому только “умный” и настроенный на соответствующие предупреждения компилятор обеспечит

проверку соответствия типа данных выводимого поля. В остальных случаях возникнет неверная работа программы, которая, впрочем, может обнаружиться далеко не сразу.

Спецификатор также определяет то, каким образом будут выведены данные:

спец.	что и как выводит	пример вывода
d или i	знаковое целое в десятичной системе	-322
u	беззнаковое целое в десятичной системе	7135
o	беззнаковое целое в восьмеричной системе	410
x	беззнаковое целое в шестнадцатеричной системе (нижний регистр букв)	2fb
X	беззнаковое целое в шестнадцатеричной системе (верхний регистр букв)	2FB
f	число с плавающей точной в десятичной системе в простом виде	392.65
e	число с плавающей точной в десятичной системе в научном виде (мантисса/порядок, строчное e)	3.9265e+2
E	число с плавающей точной в десятичной системе в научном виде (мантисса/порядок, заглавное E)	3.9265E+2
g	Использовать наиболее короткий из `%e` и `%f`	392.65
G	Использовать наиболее короткий из `%E` и `%f`	392.65
c	символ (по коду)	a
s	строка символов	Hello, world!
p	указатель (выводит адрес в каком-то формате)	b8000000
%	двойной процент выводит одинарный процент	%

Ширина может быть указана либо целым числом напрямую, либо звездочкой, тогда число возьмется из следующего аргумента, который должен иметь тип `int`. Ширина определяет ширину поля — минимальное выводимое количество символов, короткие поля дополняться пробелами или нулями, в зависимости от заданных флагов.

Точность также может быть указана либо числом напрямую, либо звездочкой, тогда число возьмется из следующего аргумента, который должен иметь тип `int`.

Для целых чисел (спецификаторы `d`, `i`, `o`, `u`, `x`, `X`) точность определяет минимальное число цифр, которое будет напечатано, если число имеет меньше цифр, оно дополняется незначащими нулями, для спецификаторов `e`, `E`, `f` и `F` — число цифр, печатаемых после десятичной точки (по умолчанию — 6), для спецификаторов `g` и `G` — максимальное число значащих цифр, которое будет напечатано, для строк (спецификатор `s`) — максимальное число символов строки, которое будет выведено (по умолчанию печатается вся строка до терминального символа `'\0'` исключая его).

Флаги могут быть следующими:

-	выравнивать по левому краю в пределах указанной ширины
+	всегда печатать знак
(пробел)	печатать пробел вместо знака плюс
#	при использовании со спецификаторами o, x и X значение предваряется 0, 0x и 0X соответственно для ненулевых чисел, при использовании с e, E, f, F, g и G принудительно печатает десятичную точку, даже если далее не следуют цифры.
0	дополняет число незначащими нулями (слева) в пределах ширины.

Пример.

```
#include <stdio.h>
#include <math.h>

int main()
{
    int m, n;
    float a, b;

    n = 15;
    m = n / 2;

    printf("m␣=␣%d\n",m);
    printf("m␣=␣%3d\n",m);
    printf("m␣=␣%03d\n",m);
    printf("m␣=␣%+03d\n",m);

    a = 82;
    b = a / 3;
    printf("b␣=␣%3.2f,␣b␣=␣%3.2e,␣b␣=␣%3.2E,␣2^b␣=␣%3.4lG\n", b, b, b, pow(2, b));

    return 0;
}
```

Выведет

```
m = 7
m =   7
m = 007
m = +07
b = 27.33, b = 2.73e+01, b = 2.73E+01, 2^b = 1.691E+08
```

Обратите внимание на ширину `l` в последнем поле: функция `pow` возвращает тип `double`.

Основной проблемой при использовании `printf` является тот факт, что данная функция “не умеет” определять тип выводимых данных (тип выражений, перечисляемых после строки формата), равно как и их количество. Это необходимо делать вручную, с помощью комбинации длины и спецификатора (причем может оказаться, что в зависимости от компилятора и используемой версии языка эти комбинации будут соответствовать разным типам данных).

Современные компиляторы позволяют подключить режим предупреждения корректности задаваемого формата. При игнорировании данных предупреждений или при отсутствии их поддержки компилятором, будет сгенерирован исполняемый файл, но при несоответствии пары длина-спецификатор или недостаче выводимых полей вывод данных будет осуществляться неверно, например, вещественное может быть интерпретировано как целое, а целое в паре с случайными данными, следующими в памяти за ним — как вещественное и т.д. Подобная неверная интерпретация, например, вещественного как целого, означает не отбрасывание дробной части, а именно двоичную интерпретацию данных одного типа, как данных другого типа — результат будет сильно неожиданным.

Поэтому с данной функцией следует соблюдать определенную осторожность.

Форматированный ввод

Ввод во многом аналогичен выводу, осуществляется с помощью функции `scanf`.

`int scanf(const char *format, ...);`

Параметр `format` — строка, в соответствии с которой будут вводиться поля. Функция прерывает работу программы ждет, пока не будут введены данные и нажата клавиша *Enter*.

Возвращаемое значение сообщает о степени успеха ввода и на первых порах может быть опущено.

Общий синтаксис формата поля следующий:

%[*][ширина][длина]спецификатор

Спец.	Описание	Ожидаемые символы

i, u	Целое	Любые число цифр, возможно предваряемые знаком (+ или -). По умолчанию — десятичные цифры (0-9), но префикс 0 предполагает восьмеричное число (0-7), а 0x — шестнадцатеричное (0-f).
d	Десятичное целое	Любые число десятичных цифр, возможно предваряемые знаком (+ или -).
o	Восьмеричное целое	Любое число восьмеричных цифр (0-7), опционально со знаком (+ или -).
x	Шестнадцатеричное целое	Любое число шестнадцатеричных цифр (0-9, a-f, A-F), опционально предваряемых 0x или 0X, и знаком (+ или -).
f, e, g	Число с плавающей точкой	Серия десятичных цифр, опционально содержащих десятичную точку, опционально предваряемых знаком и завершаемых символом e или E и десятичным целым.
c	Символ	Следующий символ.
s	Строка символов	Любое число символов — не пробелов, завершающихся первым пробелом.
p	Указатель (адрес)	Адрес в памяти (в зависимости от платформы и библиотеки).

Звездочка в формате означает, что данные прочитываются, но игнорируются (не записываются в переменную). Ширина ограничивает максимальное число символов, которое может быть прочитано. Ожидаемый тип данных определяется следующим образом:

	спецификатор				
длина	d i	u o x X	f e E g G	c s	p
(пропущена)	int*	unsigned int*	float*	char*	void*
h	short int*	unsigned short int*			
l	long int*	unsigned long int*	double*		
L			long double*		

Очевидным образом `scanf` ожидает не переменные, а именно указатели на них — иначе он не смог бы действительно ввести, а не записать значение. При этом только указатель на строку формата константный, т.е. в прототипе функции `scanf` (как и `printf`) явно указано, что строка формата не может быть изменена данной функцией.

Ввод строк с помощью `scanf` здесь не разъясняется, строкам посвящена отдельный параграф.

Пример.

```
#include <stdio.h>

int main()
{
    int i;
    double d;

    printf ("Enter␣two␣numbers:␣");
    scanf ("%d%lf", &i, &d);
    printf ("i␣=␣%d,␣d␣=␣%lg\n", i, d);

    return 0;
}
```

Примеры работы:

```
Enter two numbers: 1 2
```

```
i = 1, d = 2

Enter two numbers: 1 5e8
i = 1, d = 5e+08
```

Кроме того, как и в случае с `printf`, `scanf` не имеет возможности проверить корректность соответствия пары длина-спецификатор фактическому типу переданного выражения. Современный компилятор может лишь сгенерировать предупреждение, но исполняемый код все равно будет получен.

В отличие от `printf`, где такое несоответствие приведет лишь к к “странному” выводу, `scanf` может серьезно испортить данные и код программы. Например, если указать в строке формата ввод числа типа `double`, а передать адрес более короткой переменной типа `int`, функция перезапишет данные, следующие в памяти сразу за данной переменной. Этими данными может оказаться как другая переменная, так и участки кода, что приведет к краху программы вплоть до порчи данных системы (последнее, конечно, ничтожно маловероятно при случайной ошибке, но может быть использовано злоумышленником для взлома системы).

Аналогичный результат получится, если указывать в параметрах `scanf` не указатели, а сами переменные: функция воспримет данные как адреса и перезапишет память в непредсказуемом месте. Если число передаваемых указателей будет меньше числа вводимых полей, то `scanf` получит случайные, непредсказуемые числа в качестве указателей и также перезаписать данные в непредсказуемом месте. Ошибка с функцией `scanf` может проявиться не сразу при ее вызове во время исполнения программы, а существенно позже — при проявлении перезаписанных данных или исполнении “испорченного” кода. *Поэтому с данной функцией следует соблюдать особую осторожность и всегда отслеживать предупреждения компилятора.*

В случае ввода пользователем данных в неверном формате значения переменных могут оказаться непредсказуемыми. Поэтому следует проверять корректность обработки введенных данных: функция возвращает *количество успешно полученных полей*, то есть в этом примере должно было вернуться 2.

```
#include <stdio.h>

int main()
{
    int i;
    double d;

    printf ("Enter␣two␣numbers:␣");
    scanf ("%d␣%lf", &i, &d) == 2
        ? printf ("i␣=␣%d,␣d␣=␣%lg\n", i, d)
        : printf ("Bad␣input\n");

    return 0;
}
```

```
Enter two numbers: 1 5e8
i = 1, d = 5e+08

Enter two numbers: 1 2
i = 1, d = 2

Enter two numbers: a 3
Bad input

Enter two numbers: 1.5 3
i = 1, d = 0.5
```

Последний случай показывает, что такая проверка все-таки не панацея: первое, целое, число 1 считалось, а точка оказалась началом следующего, вещественного, числа.

Вопросы для самопроверки.

1. *Каковы недостатки функций форматированного ввода и вывода стандартной библиотеки Си?*
2. *Какая функция отвечает за форматированный вывод? Ввод?*
3. *Что такое формат и поле?*
4. *Как определяется тип данных поля?*

5. *За что в строке формата отвечают спецификатор? Длина? Флаги? Точность? Ширина?*
6. *Почему в `scanf` необходимо использовать указатели?*
7. *Что будет, если в `scanf` указать переменные, а не указатели?*
8. *Какие ошибки можно допустить при работе с `printf` и `scanf`?*

§3.9. Другие операции

Приведение типа

Как уже можно было заметить, во многих случаях Си автоматически преобразует тип данных: при присваивании, при передаче аргумента, при несовпадении типов операндов. Это т.н. **неявное приведение типа**, оно осуществляется автоматически в следующих ситуациях:

- разнотипные операнды бинарных арифметических операций, операций сравнения и операций битовой арифметики (но не сдвига) приводятся к одному, наиболее длинному из двух) типу;
- тип присваиваемого выражения приводится к типу изменяемой переменной;
- тип аргумента функции приводится к типу параметра (кроме функций с переменным числом аргументов: в частности, для `printf` и `scanf` приведение не проводится).

В целых числах приведение приводится к более длинному типу однозначно, при приведении от целого к вещественному — тоже, при этом, как уже было отмечено, при сравнении знакового и беззнакового целого направление приведения неопределено. Также неявное приведение типа не происходит при использовании функций форматированного ввода и вывода.

Иногда бывает нужно преобразовать тип явным образом, например уменьшить его (привести длинное целое к короткому, если известно, что реально значение уместится в указанный тип), превратить вещественное число в целое с потерей дробной части и т.п. Это делает операция **приведения типа**, которая является унарной префиксной операцией, символом которой есть имя типа, взятое в скобки. Возвращаемое значение — данные указанного типа. Пример

```
#include <stdio.h>

int main()
{
    printf( "integer␣=␣%i,␣symbol␣-␣%c\n", 65, (char)65 );
    return 0;
}
```

выведет

`integer = 65, symbol - A`

Также с помощью явного приведения можно сделать число с плавающей точкой из целого для деления:

```
#include <stdio.h>

int main()
{
    int i = 1, j = 2;
    double a = i / j, b = (double) i / j;
    printf( "a␣=␣%lf,␣b␣=␣%lf\n", a, b );
    return 0;
}
```

`a = 0.000000, b = 0.500000`

Определение размера

Еще одной полезной операцией, связанной с типами данных, является операция `sizeof` — операция определения размера переменной или типа данных в байтах. Возвращаемое значение имеет тип `size_t` — достаточно длинный целочисленный беззнаковый тип.

```
#include <stdio.h>

int main()
{
```

```
    int i = 1;
    float a = 1;
    double d = 1;

    double *p = &d;

    printf ("size␣of␣i␣=␣%lu\n"
            "size␣of␣a␣=␣%lu\n"
            "size␣of␣d␣=␣%lu\n"
            "size␣of␣p␣=␣%lu\n"
            "size␣of␣short␣=␣%lu\n"
            "size␣of␣long␣double␣=␣%lu\n"
            "size␣of␣void␣*␣=␣%lu\n",
            sizeof i,
            sizeof a,
            sizeof d,
            sizeof p,
            sizeof (short),
            sizeof (long double),
            sizeof (void *)
           );

    return 0;
}
```

```
size of i = 4
size of a = 4
size of d = 8
size of p = 8
size of short = 2
size of long double = 16
size of void * = 8
```

(вывод зависит от платформы, операционной системы, компилятора и его настроек). Поскольку `sizeof` является операцией, а не функцией, скобки вокруг операнда-выражения формально не требуются, но они необходимы при определении размера типов данных. Также, поскольку данная унарная операция имеет более высокий приоритет, чем бинарные, при определении размера большинства выражений их все же необходимо взять скобки для получения корректного результата.

Операция запятая

Операция запятая (`,`) — бинарная операция с низшим приоритетом, работает следующим образом:

```
expr1, expr2
```

вычисляет сначала выражение `expr1`, затем выражение `expr2`, возвращает значение выражения `expr2`. Порядок операции — слева направо, поэтому `expr1, expr2, expr3` вернет значение выражения `expr3` и т.д.

Часто данная операция используется в цикле `for`, когда нужно инициализировать или инкрементировать значения более, чем одного счетчика:

```
for (i = 5, j = 0; j < 5; ++i, --j) /* ... */
```

это будет иметь в точности тот эффект, какой бы ожидался от заведомо синтаксически ошибочной записи

```
for (i = 5; j = 0; j < 5; ++i; --j) /* ... */
```

аргументами `for` в скобках могут быть только выражения, но не операторы, а операция запятая — именно операция, поэтому ее использование для формирования выражения (в данном случае итератора и инициализатора) корректно.

Еще один пример. Пусть есть функция, которая как-то изменяет значение аргумента, но не возвращает его, а именно передает через указатель. При этом цикл нужно продолжать до тех пор, пока значение аргумента не окажется нужным. Тогда можно записать, например,

```
while (foo(&s), s != 5) /* ... */
```

и запятая позволит вызвать функцию `foo`, проигнорировать возвращенное ей значение, а в оператор `while` в качестве условия передаст именно результат вычисления `s != 5`. Без использования данной операции необходимый результат достигался бы, например, конструкцией

```
foo(&s)
while (s != 5)
{
    /* ... */
    foo(&s);
}
```

или (с не 100% тем же эффектом!)

```
s = 0;
while (s != 5)
{
    foo(&s);
    /* ... */
}
```

и другими способами (цикл `for`, `do...while` и т.п.). Одним из примеров такой функции является `scanf`.

Операция "запятая" и пунктуатор "запятая", используемом при перечислением аргументов при вызове функции, являются различными синтаксически и семантически объектами. При вызове функции запятая всегда трактуется как разделитель аргументов, для того, чтобы использовать операцию "запятая" в аргументе функции следует заключить данный аргумент в скобки.

Вопросы для самопроверки.

1. *Что такое неявное определение типа?*
2. *Когда происходит неявное приведение типа?*
3. *Как осуществляется явное определение типа?*
4. *Что делает операция sizeof?*
5. *Как определить размер (длину) типа данных? Конкретных данных?*

§3.10. Виды выражений: L-value и R-value

Выражения в языке Си относятся к трем видам.

***void*-выражение** — выражение не имеющее значение и не ассоциированное с ячейкой памяти, такое значение нельзя использовать в качестве аргумента функции или операнда операции, допускается только его использование в качестве оператора-выражения, а также инициализатора и итератора цикла `for`. Данное выражение возникает при вызове *void*-функции.

выражение L-value ("левое" значение) — выражение, соответствующее ячейке памяти, значение которой можно как прочитать, так и изменить (записать). В частности, выражением L-value являются переменные, формальные параметры функций, результат операции разыменования указателя, а также еще не рассмотренные операции получения элемента массива и поля структуры.

выражение R-value ("правое" значение) — выражение, значение которого можно лишь прочитать, но которое не связано с ячейкой памяти. Образуется как результат вызова не-*void*-функции и большинства операций. Проще определить выражение R-value как исключение — выражение, не являющееся ни L-value, ни *void*-выражением является выражением R-value. Все результаты арифметических и т.п. операций являются R-value значениями, их значение хранится во временной ячейке памяти, его изменение если и возможно теоретически, то бессмысленно.

Только L-value может быть указано в левой части операции присваивания, в правой части может присутствовать выражение R-value (чем и обусловлено их названия). Также L-value выражение требуется в качестве аргумента операций декремента, инкремента и взятия адреса (получения указателя). При этом возвращаемое значение операций присваивания, инкремента, декремента и взятия адреса — R-value. В качестве операндов остальных операций, аргументов функций а также аргументов операторов (условного, циклов, выбора) могут выступать как выражения R-value, так и выражения L-value (везде, где может использоваться выражение R-value, может использоваться и выражение L-value).

Вопросы для самопроверки.

1. *Что такое void-выражение, R-value и L-value?*
2. *Приведите примеры выражений L-value.*

§3.11. Вычисление выражений

Общая идея вычисления выражений

При вычислении выражений появляются *промежуточные значения*: выражение понятие рекурсивное, строится из других (под)выражений, каждое подвыражение также представляет собой значение некоторого типа. Для его хранения отводится специальная ячейка памяти (скорее всего временная, т.е. значение хранится в ней до тех пор, пока нужно, а затем ячейка освобождается).

Например, при вычислении

```
double x, y;
int i, b;

b = x * (3 + i) <= f(5 * y, x);
```

вычисляется четыре промежуточных значения. Можно представить их как появление неявных временных переменных:

```
double x;
int i, b;

int _t1 = 3 + i;
double _t2 = x * _t1;
double _t3 = 5 * y;
double _t4 = f(_t3, x);

int b = _t2 <= _t4;
```

Следует, однако, отметить, что в Си порядок вычисления выражений не регламентирован, если это не противоречит приоритету и порядку. Например, предыдущий пример может вычислиться и в таком порядке:

```
double x;
int i, b;

double _t1 = 5 * y;
double _t2 = f(_t1, x);
int _t3 = 3 + i;
double _t4 = x * _t1;

int b = _t4 <= _t2;
```

Наличие разных возможностей вычислить одно и то же выражение может привести к **неопределенному поведению** — ситуации, когда предсказать результат операции (поведение программы) путем анализа кода программы невозможно. На практике результат зависит от того, какой компилятор использован, с какими настройками, на какой платформе. Классический пример неопределенного поведения:

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

Чему равно `j`?

Вариант 1: сначала вычислится один инкремент, вернется значение (6), оно же будет в переменной, потом вычислится другой инкремент и вернется значение (7), потом они сложатся с получением результата 13.

Вариант 2: сначала вычислится один инкремент, в переменной окажется 6, потом вычислится другой инкремент, потом сложатся значения, находящиеся в `i`, получится 14.

При практическом исполнении данного кода вероятнее обнаружить второй вариант, так как работает быстрее и не требует дополнительной ячейки для хранения результата. Первый вариант "расшифровывается" так

```
int i = 5;
int t1 = ++i;
int t2 = ++i;
int j = t1 + t2;
```

второй — следующим образом:

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

Для того, чтобы исключить подобные ситуации, следует иметь в виду возможность произвольного порядка вычисления операндов операций и аргументов функций, а также регламент вычисления выражений. *Нельзя использовать значение переменной дважды в одном и том же выражении, если оно изменяется в этом выражении, за исключением прочтения старого значения переменной в правом операнде операций присваивания.* Также важно подключать все предупреждения компилятора, во многих случаях это позволяет выявить случаи неопределенного поведения. Следует избегать сложных выражений, с большим количеством побочных эффектов — по словам создателя языка, эта неопределенность сделана нарочно, чтобы программисты не злоупотребляли такими выражениями.

Регламент вычисления выражений

Порядок вычисления операндов любой операции, а также порядок вычисления аргументов функции при вызове, а также порядок вычисления подвыражений внутри выражения не специфицирован стандартом за исключением описанных ниже случаев — конкретный порядок зависит от компилятора, "продвинутый" компилятор может выбирать более оптимальный путь каждый раз встречая выражение.

В Си нет понятия вычисления слева-направо и справа-налево, что не следует путать с порядком ассоциативности операций: `f1() + f2() + f3()` представляется как `(f1() + f2()) + f3()` в силу того, что операция $+$ имеет порядок слева-направо, но вызов функции `f3` может произойти как первым, так и вторым, так и последним. Приоритет и порядок операций независимы от порядка вычисления выражений.

Есть два типа действий, которые могут производиться при вычислении выражения (в самом выражении и подвыражениях).

Первый — **вычисление значения**, т.е. вычисления значения, которое возвращается выражением. Это может быть как определение объекта (L-value), так и прочтение значения, присвоенного объекту (R-value).

Второй — **побочный эффект**, т.е. изменение объекта (L-value), изменение файла, вывод информации на внешнее устройство и др., в том числе вызов функции, которая делает одно из этих действий. Если побочных эффектов нет и компилятор может определить это (фактически значение выражения не используется), то выражение может не вычисляться вообще.

Основные правила вычисления выражений следующие.

1. Значения операндов операции вычисляются до значения результата операции, при вызове функции вычисление выражений-аргументов и их побочных эффектов производится до вызова функции.

2. Побочные эффекты всех операций присваивания (т.е. собственно изменение значений) выполняются после вычисления значений операндов, но до вычисления значения операции (т.е. присваивается указанное значение, а возвращается — присвоенное).

3. Вычисление значений и побочных эффектов левого аргумента логических операций `&&` и `||` производится до вычисления значений и побочного эффекта правого операнда, вычисление значения и побочные эффекты первого операнда условной операции `?:` производятся до вычисления значения и побочных эффектов второго и третьего операнда.

   При этом, если значение левого операнда логических операций дает однозначное значение всей операции (ложь для конъюнкции и истина для дизъюнкции), то правый операнд не вычисляется и его побочные эффекты не применяются. Для тернарной операции присваивания производится вычисление и применение побочных эффектов только второго или только третьего операнда, в зависимости от значения первого операнда (истина или ложь соответственно).

   Вычисление и применение побочных эффектов левого операнда операции запятая производится до вычисления и применения побочных эффектов правого операнда.

4. Вычисление значений постфиксных инкремента и декремента производится до побочного эффекта, префиксных — после (это определяет разницу их возвращаемого значения).

Возможность вычисления только одного операнда конъюнкции и дизъюнкции — принцип **"укороченной оценки"** логических выражений. В этом также проявляется разница между побитовыми и логическими операциями (например, хотя `|` и `||` часто дают одинаковый результат, первая может работать медленнее и применять побочные эффекты второго операнда там, где вторая этого не допустит). Категорически нельзя использовать

битовые "и" "или" и "не" вместо логических: хотя в некоторых случаях они дадут одинаковый результат, в других он будет различен.

Вопросы для самопроверки.

1. *Как вычисляются выражения?*
2. *Какие два процесса производятся при вычислении значения выражения?*
3. *Как регламентирован порядок при вычислении выражения?*
4. *Что такое укороченная оценка логических выражений?*
5. *Применима ли укороченная оценка к битовой арифметике?*
6. *Что такое побочный эффект?*

§3.12. Неопределенное поведение

Неопределенное поведение — ситуация, когда результат работы программы нельзя определить исходя из анализа исходного кода программы.

В общем случае результат может зависть от компилятора и его настроек, а также ряда факторов, которые принято называть случайными — состояния компьютера и операционной системы, содержимого оперативной памяти и т.п.

Иногда выделяют **непредсказуемое поведение** — ситуацию, когда результат работы программы зависит не только от генерируемого компилятором двоичного кода, но и условиями запуска программы, например, обращению к неинициализированным переменным и т.п.

Ни при каких обстоятельствах недопустимо использовать код, генерирующий неопределенное и непредсказуемое поведение, нельзя опираться на неопределенный результат даже если в каком-то конкретном случае он оказался правильным, ведь нет никакой гарантии, что он будет повторен при следующем запуске программы, при использовании другого компилятора, другой версии компилятора, других опций и настроек компилятора.

Другая ситуация — **поведение, зависимое от компилятора**, то есть ситуация, когда результат зависит от конкретной реализации, т.е. от компилятора языка и его настроек. Пример компиляторо-зависимого поведения — битовый сдвиг вправо для знаковых целых чисел может быть как знаковым, так и беззнаковым.

Полагаться на компиляторо-зависимое поведение — также как и на расширения языка, предоставляемые компилятором — можно только в том случае, когда есть гарантия, что данная программа будет всегда компилироваться с помощью данного компилятора на данной платформе, что, конечно, снижает ее переносимость.

Примеры ситуаций неопределенного поведения в Си следующие.

- Неопределенный порядок вычисления арифметических выражений. Например, при использовании

  ```
  ++i + ++i
  ```

  или иных подобных действий, при которой происходит прочтение и изменение значений одной и той же переменной (или иного объекта L-value — разыменованного указателя, элемента массива и т.д.).

  Неопределенное поведение возможно, если вычисления побочных эффектов одного выражения и другого выражения могут быть проведены в произвольном порядке, например,

  ```
  j = i++ * i;       /* что раньше - инкремент или прочтение i? */
  i = ++i + i++;     /* какой инкремент раньше?                 */
  i = i++ + 1;       /* что раньше, инкремент или присваивание? */
  f(++i, ++i);       /* какой инкремент раньше?                 */
  f(i = -1, i = -2); /* какой аргумент вычислится раньше?       */
  f(i, i++);         /* какой аргумент вычислится раньше?       */
  a[i] = i++;        /* какой операнд вычислится раньше?        */
  ```

- Использование значений неинициализированных переменных — также серьезная ошибка, дающая непредсказуемый результат. Например,

```
int i, j;
j = i + 1;  /* Значение j непредсказуео,
             * т. к. значение i непредсказуемо
             */
```

Нигде не гарантируется, что i будет равно 0 или какому-либо другому конкретному значению: на практике часто i действительно будет нулем, но на самом деле это просто случайное значение, оказавшееся в памяти по зарезервированному под переменную i адресу. При разных запусках одной и той же программы там могут оказаться разные значения.

- Смешение разнотипных указателей, например

```
int j = 1, *p = &j;
double *q = (void *) p; /* хотя данный код компилируется без предупреждений, */
double k = *q;          /* результат разыменования q непредсказуем, так как
                         * в указываемой ячейке содержится значение типа int
                         */
```

- Ошибки с типами и количеством аргументов в функциях с переменных числом аргументов, в частности `scanf` и `printf`.
- Целочисленное переполнение (результат будет зависеть от способа представления чисел).
- Выход за границы массива.
- Обращение к переменной вне зоны ее видимости, после окончания времени ее жизни, в частности возврат адреса локальной переменной из функции. Локальная переменная прекращает свое существование после выхода из блока, в котором она была определена. Обратится к ней вне него по идентификатору невозможно — переменная доступна только в ее области видимости. Однако если каким-то образом сохранить ее адрес (указатель), то обращение будет синтаксически корректным:

```
double * foo (/* ... */ )
{
     double p;
     /* ... */
     return &p;    /* недопустимо, возврат адреса локальной переменной */
}

/* ... */

double d = *foo(/* ... */); /* значение d непредсказуемо */
```

  Аналогичная ситуация возникнет при передаче указателя за границы блока в рамках одной функции:

```
double *p;
while (cond)
{
    double x;
    /* ... */
    p = &x;    /* допустимо, если *p используется в теле цикла */
}

/* ... */

double d = *p; /* значение d непредсказуемо */
```

- Отсутствие оператора `return` в функции, возвращающей значение.

  Исполнение операторов тела функции производится последовательно (с учетом переходов и структур управления) пока не будет достигнут оператор `return` или конца тела функции. Функция может иметь или не иметь возвращаемое значение (`void`-функция). Возвращаемое значение функции подставляется на место вызова функции при вычислении выражения. В *void*-функциях можно использовать оператор `return` без параметра, он прерывает работу функции, но можно его опустить — функция корректно завершится по завершении последнего оператора. В не-*void* функциях использование оператора `return` обязательно, более того, необходимо, чтобы при любых условиях (при любых значениях параметров и т.д.) он вызывался, иначе возвращаемое значение будет непредсказуемым. Например, ошибочным является код

```
float foo (float x)
{
    if ( x >= 0) return 1.0;
}
```

так как при отрицательных значениях параметра оператор `return` не исполняется, возвращаемое значение непредсказуемо. Ошибочным является и следующий код:

```
float fsignf (float x)
{
    if      (x >  0.0) return 1.0;
    else if (x == 0.0) return 0.0;
    else if (x <  0.0)
    return -1.0;
}
```

так как если параметр имеет значение `NaN`, то оператор `return` не встретится. Следует учесть эту ситуацию

```
float fsignf (float x)
{
    if      (x >  0.0) return 1.0;
    else if (x == 0.0) return 0.0;
    else if (x <  0.0) return -1.0;
    else               return 0.0 / 0.0; /* простейший способ получить NaN */
}
```

- Разыменование нулевого указателя и вообще указателя, значение которого не указывает на какой-то верный объект в памяти.

Заметим, что результатом неопределенного поведения может быть не только получение некорректных значений, но и исполнении некорректных инструкций — от аварийного завершения программы до порчи данных и повреждения оборудования.

Современные компиляторы способны выявлять ситуации неопределенного поведения в арифметических выражениях, а также использование значений неинициализированных переменных, и других ситуациях, когда синтаксически верная программа может дать непредсказуемый результат. Следует подключать предупреждения в настройках компилятора и обращать внимание на них. Ни в каких ситуациях не следует полагаться ни на неопределенное, ни на непредсказуемое поведение.

Вопросы для самопроверки.

1. *Что такое неопределенное поведение?*
2. *Приведите примеры неопределенного поведения.*
3. *Чем чревато использование значений неинциализированных переменных?*

§3.13. Стек вызовов и сегмент стека

Стек *(в общем виде)* — способ организации порядка элементов в наборе, при котором элементы удаляются в порядке, обратном порядку добавления. Точнее, стек есть хранилище данных, для которого определены операции добавления элемента и удаления элемента, причем удаляется элемент, который был добавлен последним.

Стек противопоставляется **очереди**, из которой элементы удаляются в порядке поступления. У очереди два конца, в один поступают элементы (хвост) из другого (голова) удаляются, первый пришел — первый вышел. У стека один конец (вершина), в него элементы и добавляются и убираются — первый пришел, последний вышел. Бытовой пример стека — стопка тарелок в столовой (стек с англ. переводится как “стопка”). В любой момент можно положить тарелку сверху, в любой момент можно взять тарелку сверху (необязательно, что стопка сначала заполняется, а потом опустошается до конца), но чтобы стопка действительно работала как стек, добавлять и удалять тарелки следует по одной.

Стек вызовов — стек, представляющий собой набор активных функций.

Действительно, первой вызывается функция `main`. Когда вызывается функция программы, она добавляется в стек вызовов. По мере вызовов из одной функций других количество элементов в стеке растет, при этом в вершине стека всегда находится та функция, код которой исполняется в данный момент. Когда функция завершается, она изымается из стека. Схема изменения стека вызова показана на рис. 3.5

Стек вызова — абстракция. Стек вызова отображается отладчиками программ при пошаговом исполнении, чтобы можно было увидеть посредством каких вызовов программа добралась до текущей точки кода.

В современных языках программирования, в том числе в Си, хранение параметров и данных функций организовано с помощью стека. Каждой программе отводится область памяти фиксированного размера, называемая

Действие	**структура стека**
Запуск программы (вызов `main`)	`main`
Вызов `foo`	`foo` `main`

Вызов `bar`	`bar`
	`foo`
	`main`

Вызов `fn1`	`fn1`
	`bar`
	`foo`
	`main`

Завершение `fn1`	`bar`
	`foo`
	`main`

Вызов `fn2`	`fn2`
	`bar`
	`foo`
	`main`

Завершение `fn2`	`bar`
	`foo`
	`main`

Завершение `bar`	`foo` `main`
Завершение `foo`	`main`
Запуск `fn1`	`fn1` `main`
Завершение `fn1`	`main`
Завершение `main` (программы)	стек пуст

Рис. 3.5: Изменение структуры стека вызова при вызове и завершении функций. Вершина находится вверху списка.

стек программы или **сегмент стека**. В центральном процессоре хранится указатель, изначально указывающий на последний элемент стека. Когда вызывается функция в стеке “отводится место” — а по сути перемещается указатель на нужное количество байтов — под экземпляр данных функции, то есть под все параметры функции (формальные), все локальные переменные функции, под возвращаемое значение функции и точку возврата.

Все адреса данных функции компилятор вычисляет именно относительно позиции стека. Область данных функции — это именно область памяти от указателя стека до того места, где в стеке начинаются данные предыдущей функции. По завершении функции указатель смещается на нужную позицию вверх. Иллюстрация процесса показана в приложении C.

Вопросы для самопроверки.

1. *Что такое стек как способ организации данных?*
2. *Что такое очередь как способ организации данных?*
3. *Что такое стек вызовов?*
4. *Что содержится в стеке вызовов?*
5. *Когда функция попадает в стек вызовов?*
6. *Когда функция исключается из стека вызовов?*
7. *Что такое стек программы?*
8. *Что хранится в стеке программы?*
9. *Как организовано хранение данных функций в стеке программы?*
10. *Опишите, как происходит процесс вызова функции и ее завершение в Си.*

§3.14. Рекурсия

Когда параметры и локальные переменные функции хранятся в стеке программы, ничто не мешает функции вызывать саму себя: в такой ситуации для каждого вызова функции создается новый экземпляр данных функции (локальных переменных, параметров, точки возврата) и каждый вызов функции будет работать в своей области данных, локальные переменные и формальные параметры разных вызовов одной функции будут фактически разными ячейками памяти.

Функции, которые вызывают сами себя, называются **рекурсивными**, алгоритм, предусматривающий запуск самого себя — **рекурсивным**. Алгоритмы и функции, свободными от такого явления, называют **итеративными**.

Часто код рекурсивных функций создается относительно легко, так как многие определения в математике и соответствующие вычислительные формулы являются рекурсивными. Рассмотрим рекурсивное определение факториала

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n \neq 0. \end{cases}$$

Рекурсивная функция вычисления факториала выглядит следующим образом:

```
unsigned long f1 (unsigned n)
{
    if (n == 0) return 1;
    return (unsigned long) n * f1(n - 1);
}
```

тот же результат может быть достигнут итеративным образом:

```
unsigned long f2 (unsigned n)
{
    unsigned long r = 1;
    for (; n > 1; --n) r *= n;
    return r;
}
```

На рис. 3.6 можно увидеть разницу между этими реализациями.

Следует обратить внимание на ряд моментов.

Вызов `f1(3)`

																	вз	тв	n
???	???	???	???	???	???	???	???	???	???	???	???	???	???	???	???	**Дан. f1(3)**	?	**в вызв. ф.**	**3**

Проверка условия, вызов `f1(n-1)`, то есть `f1(2)`

													вз	тв	n		вз	тв	n
???	???	???	???	???	???	???	???	???	???	???	???	**Дан. f1(2)**	?	**в f1(3)**	**1**	**Дан. f1(3)**	?	в вызв. ф.	2

Проверка условия, вызов `f1(n-1)`, то есть `f1(1)`

									вз	тв	n		вз	тв	n		вз	тв	n
???	???	???	???	???	???	???	???	**Дан. f1(1)**	?	**в f1(2)**	1	**Дан. f1(2)**	?	в f1(3)	2	**Дан. f1(3)**	?	в вызв. ф.	3

Проверка условия, вызов `f1(n-1)`, то есть `f1(0)`

					вз	тв	n		вз	тв	n		вз	тв	n		вз	тв	n
???	???	???	???	**Дан. f1(0)**	?	**в f1(1)**	0	**Дан. f1(1)**	?	в f1(2)	1	**Дан. f1(2)**	?	в f1(3)	2	**Дан. f1(3)**	?	в вызв. ф.	3

Проверка условия, возврат 1"

					вз	тв	n		вз	тв	n		вз	тв	n		вз	тв	n
???	???	???	???	**Дан. f1(0)**	**1**	в f1(1)	0	**Дан. f1(1)**	?	в f1(2)	1	**Дан. f1(2)**	?	в f1(3)	2	**Дан. f1(3)**	?	в вызв. ф.	3

Вычисление возвращаемого значения, $1 \cdot 1$

									вз	тв	n		вз	тв	n		вз	тв	n
???	???	???	???	???	???	???	???	**Дан. f1(1)**	**1**	в f1(2)	1	**Дан. f1(2)**	?	в f1(3)	2	**Дан. f1(3)**	?	в вызв. ф.	3

Вычисление возвращаемого значения, $2 \cdot 1$

													вз	тв	n		вз	тв	n
???	???	???	???	???	???	???	???	???	???	???	???	**Дан. f1(2)**	**2**	в f1(3)	2	**Дан. f1(3)**	?	в вызв. ф.	3

Вычисление возвращаемого значения, $3 \cdot 2$

																	вз	тв	n
???	???	???	???	???	???	???	???	???	???	???	???	???	???	???	???	**Дан. f1(3)**	**6**	в вызв. ф.	3

Возврат 6.

Вызов `f2(2)`

							вз	тв	r	n
???	???	???	???	???	???	**Дан. f2(3)**	?	**в вызв. ф.**	?	**3**

Инициализация `r`

							вз	тв	r	n
???	???	???	???	???	???	**Дан. f2(3)**	?	в вызв. ф.	**1**	3

Шаг цикла: увеличение `r` в `n` раз

							вз	тв	r	n
???	???	???	???	???	???	**Дан. f2(3)**	?	в вызв. ф.	**3**	3

Итератор цикла: уменьшение `n` на 1

							вз	тв	r	n
???	???	???	???	???	???	**Дан. f2(3)**	?	в вызв. ф.	3	**2**

Шаг цикла: увеличение `r` в `n` раз

							вз	тв	r	n
???	???	???	???	???	???	**Дан. f2(2)**	?	в вызв. ф.	**6**	2

Итератор цикла: уменьшение `n` на 1

							вз	тв	r	n
???	???	???	???	???	???	**Дан. f2(2)**	?	в вызв. ф.	6	**1**

вычисление возвращаемого значения, `r`

							вз	тв	r	n
???	???	???	???	???	???	**Дан. f2(2)**	**6**	в вызв. ф.	6	1

Возврат 6.

Рис. 3.6: Вычисление 3!, слева — изменение стека рекурсивной функцией, справа — итеративной. Ячейка "вз" соответствует возвращаемому значению функции, "тв" — указатель на точку возврата, знак вопроса соответствует неинициализированному значению. Жирным выделены измененные на исполненном шаге данные. При вызове функций указатель стека смещается влево, левее него — непредсказуемые данные.

- Часто код рекурсивной функции проще и понятнее, однако, нередко, рекурсивный алгоритм менее эффективен. В случае с приведенным выше примером факториала, рекурсивная функция требует выделения памяти в объеме, пропорциональном значению параметра, против фиксированного объема для итеративного варианта.

  Еще хуже ситуация с числами Фибоначчи:

$$F(n) = \begin{cases} 0, & \text{если } n = 0, \\ 1, & \text{если } n = 1, \\ F(n-1) + F(n-2), & \text{если } n > 0. \end{cases}$$

  Очевидный рекурсивный код такой функции следующий:

```
unsigned long f1 (unsigned n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return f1(n - 1) + f1(n - 2);
}
```

  Нетрудно понять, что хотя выделение памяти здесь по-прежнему пропорционально значению n (несмотря на то, что каждый вызов функции приводит к "разветвлению", обе ветви никогда не исполняются одновременно, в стеке вызовов будет только одна из них), имеет место значительный прирост времени работы программы из-за многократных вычислений одного и того же значения — рис. 3.7. Можно показать, что количество вызовов пропорционально 2^n.

  Итеративный вариант менее очевиден, но куда более эффективен и по времени, и по памяти (число операций пропорционально значению n, а объем памяти от n не зависит):

```
unsigned long f2 (unsigned n)
{
    unsigned i;
    unsigned long r1;
    unsigned long r2 = 0;
    unsigned long r  = 1;

    if (n == 0) return 0;

    for(i = 1; i < n; i++ )
    {
        r1 = r2;
        r2 = r;
        r = r1 + r2;
    }

    return r;
}
```

- Рекурсию разделяют на **линейную** и **нелинейную**. Также классифицируют **прямую** рекурсию (когда функция вызывает сама себя явно) и **косвенную** или **непрямую** (когда функция вызывается через другую функцию). Для того, чтобы организовать неявную рекурсию между функциями A и B, потребуется задание прототипа (объявления) функции, тело которой расположено в файле позже, так как иначе расположенная раньше функция не сможет ее вызвать.

- При написании рекурсивной функции следует обязательно сделать, чтобы рекурсия завершалась, то есть из рекурсии был выход. Это достигается заданием значений параметров, при которых функция должна вычисляться без вызова себя, причем к этому значению параметров рекурсия должна приводить обязательно, в том числе при нелинейной рекурсии завершаться должна каждая ветвь. Число рекурсивных вызовов, которые необходимо сделать для вычисления функции до ее завершения называется **глубиной рекурсии**.

  Если рекурсия не завершится, формально алгоритм попадет в бесконечный цикл, однако, программа не "зависнет", а некорректно завершиться из-за того, что в стеке закончится место: стек программы выделяется из оперативной памяти при запуске программы и представляет собой область фиксированного — на самом деле относительно небольшого — объема.

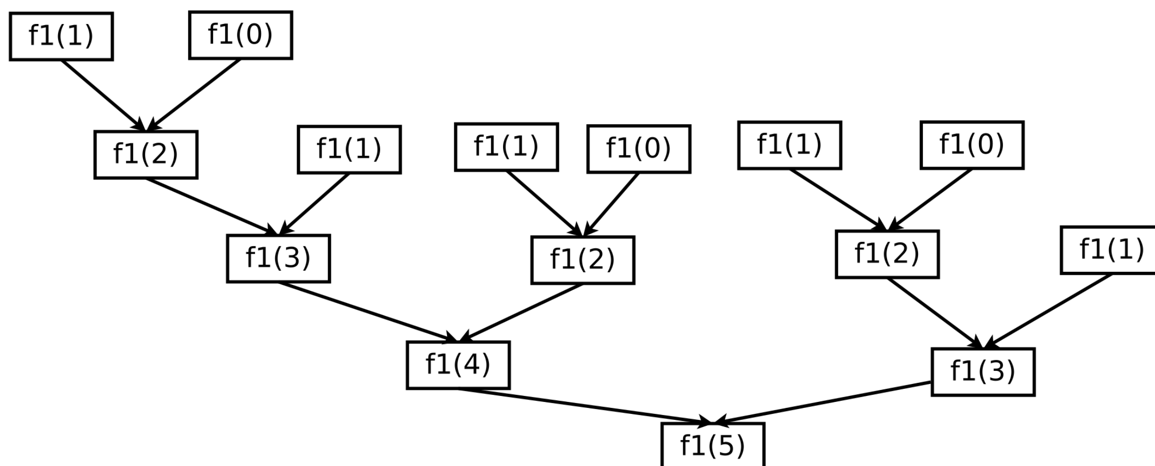


Рис. 3.7: Нелинейная рекурсия: диаграмма вызовов при вычислении числа Фибоначчи `f1(5)`. Стрелкой показаны направления вызовов, все вызовы исполняются и заводят свой экземпляр данных в стеке, среди них много повторяющихся.

Это явление называется **переполнением стека**. Обычно возникновение такой ошибки при работе программы и является следствием ошибки в коде рекурсивной функции (реже — слишком глубокой рекурсии, и гораздо реже следствием попытки создать слишком большой объем данных в одном экземпляре вызова функции).

- Общее правило, которому желательно следовать при выборе между созданием рекурсивной и итеративной функции — отдавать предпочтение рекурсии следует только в том случае, когда затраты труда программиста по разработке итеративного алгоритма заведомо не окупятся эффективностью этого алгоритма. Вообще говоря, совершенно необязательно, что рекурсивный алгоритм будет качественно менее эффективен, чем итеративный. Однако, рекурсия — тот случай, когда игнорировать расходы на вызов функции не следует.

Вопросы для самопроверки.

1. *Что в программировании понимается понимается под рекурсией?*
2. *Что такое рекурсивная функция?*
3. *Как классифицируют рекурсивные алгоритмы? Рекурсивные функции?*
4. *Каковы преимущества и недостатки рекурсивных функций?*
5. *Как организуется выход из рекурсии?*
6. *Что произойдет, если будет запущенна функция с бесконечной рекурсией?*
7. *Какая разница между бесконечным циклом и бесконечной рекурсией?*
8. *Что такое ошибка переполнения стека?*
9. *В каких ситуациях может возникнуть ошибка переполнения стека? Какая ситуация более вероятна?*
10. *В каких случаях следует реализовывать рекурсивный алгоритм?*

Глава 4. Составные типы данных

Типы данных, встроенные в язык, являются **элементарными**, то есть, не разложимые и не представимые через другие типы данных. Такие типы данных также называют **простыми**, **примитивными**, реже используется термин **скалярные**. В Си к таким типам относятся символьный, целочисленные и с плавающей точкой.

В противоположность простым типам данных существуют **составные** (так же называемые **композитными** или **структурными**) типы данных — типы, определяемые пользователем (программистом) посредством других типов. К таким типам можно отнести перечислимый тип (хотя он и сводится к элементарному типу `int`, но формально определяется пользователем), а также массивы, структуры и функциональный указатель, рассмотренные в данной главе (в частности строки — массивы символов).

Структура данных — способ организации логически (идейно) связанных данных в компьютере, при котором они могут эффективно использоваться для решения определенного круга задач.

Вообще говоря, любой способ организации данных одного или разных типов может считаться структурой данных по определению, требования логической связанности и эффективности приведены в определении лишь для указания цели использования структур данных.

Простейшим примером структурного типа данных является **массив**.

§4.1. Массивы

Массив — конечный набор значений одного типа данных, называемых **элементами** массива, расположенных в памяти непрерывно (непосредственно друг за другом).

Таким образом, массивы характеризуются, во-первых, тем, что состоят из элементов одного и того же типа, во-вторых, что эти элементы расположены в памяти по порядку. Последнее позволяет быстро определить адрес соответствующего элемента массива по его номеру, называемому **индексом**. Таким образом, массив является структурой данных, пригодной как для **последовательного** (в порядке следования элементов), так и для **произвольного доступа** к элементам, то есть для быстрого доступа к элементам в любом, продиктованным решаемой задачей, порядке, а не только в порядке следования элементов.

Замечание. В некоторых языках программирования понятие массива может быть расширено: например, может быть снято как требование на однотипность элементов массива, так и какие-либо требования к расположению элементов в памяти. Кроме того, в качестве индексов могут использоваться данные любого типа, а не только целочисленный номер элемента: такие массивы называются *ассоциативными*. Указанное выше определение относится к стандартным массивам языка Си, к классическим, “низкоуровневым” массивам. Другие виды “массивов” в данном языке могут быть созданы путем реализации соответствующих алгоритмов и использовании необходимых структур данных.

Создание массивов осуществляется следующим образом:

$$\langle\text{тип}\rangle\ \langle\text{идентификатор}\rangle\ [\langle\text{целочисленная константа}\rangle]$$

то есть при определении переменной необходимо указать в квадратных скобках число элементов массива, например,

```
int numbers[10];
```

Если значения массива не инициализированы, то они являются непредсказуемыми. Инициализировать значение массива при объявлении можно следующим образом:

$$\langle\text{тип}\rangle\ \langle\text{идентификатор}\rangle\ [\langle\text{целочисленная константа}\rangle] = \langle\text{выражение}\rangle\ , \ldots$$

Например,

```
int numbers[3] = {1, 10, 50};
```

При инициализации число элементов можно не указывать, оно будет вычислено компилятором, поэтому объявление

```
int numbers[] = {1, 10, 50};
```

равносильно предыдущему.

Однако, если число элементов указано, а инициализировано меньше, чем указано, то остальные значения автоматически инициализируются нулями:

```
int numbers[100] = {1, 10, 50};
```

В данном примере первые три элемента массива принимают значения 1, 10 и 50 соответственно, остальные 97 становятся равными нулю.

Объявление массива может следовать в любом месте объявления переменных одного типа, вперемешку с объявлениями одиночных переменных, указателей и т.п. Строго говоря, следовало бы дополнить БНФ объявления и определения переменных таким образом, чтобы учесть эту возможность.

Числом элементов массива может быть только **целочисленная константа**: данный способ объявления задает **статические массивы**, то есть массивы, чье число элементов известно на стадии компиляции и не может быть изменено в процессе работы программы. Кроме того, будучи объявленными внутри функции, такие массивы размещаются в стеке, поэтому их размер сильно ограничен — не рекомендуется создавать большие статические массивы. Создание **динамических массивов**, размещаемых в специальным образом выделяемой памяти вне стека программы, число элементов которых определяется и изменяться в процессе работы программы излагается в параграфе 7.1.

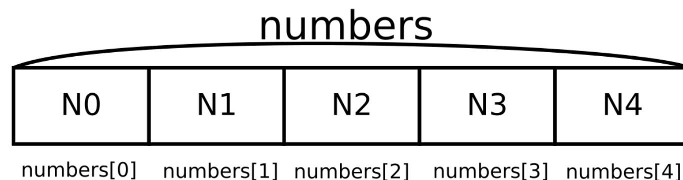


Рис. 4.1: Массив `int numbers[5]` и его элементы в памяти: элементы являются "полноценными" переменными типа `int`, операция получения элемента возвращает L-value.

Следует заметить два обстоятельства. Во-первых, стековые массивы действительно используются в программировании когда число элементов небольшое, в том числе если оно заранее неизвестно (например, для хранения координат вектора в пространстве или для хранения небольших строк). В этом случае создается массив "с запасом" — размер массива ограничивает максимальное число элементов, фактическое число элементов хранится в отдельной целочисленной переменной. Пара массив-число элементов передается в функции обработки массивов. Во-вторых, аналогичная ситуация возникает и при использовании динамических массивов — требуется хранить массив и число элементов в нем. При этом в Си *корректно написанные функции обработки массивов могут синтаксически одинаково и одинаково эффективно обрабатывать статические и динамические массивы.* Поэтому, бо́льшая часть информации данного и последующего параграфов, посвященных массивам, применима и к динамическим массивам тоже.

Доступ к элементам массива осуществляется с помощью операции взятия элемента по индексу (постфиксная операция "квадратные скобки"), например, `numbers[2]`. В качестве индекса может выступать любое целочисленное выражение (не только константа).

Результат взятия элемента массива по индексу является выражением L-value (рис 4.1).

```
numbers[2] = 5;
i = numbers[2] * numbers[1];
```

Тип данных массива обозначается как

$$\textit{type}[N]$$

где ***type*** — тип, а N — количество элементов массива, например, `int [5]`. Если число элементов неизвестно или не требует конкретизации, указываются пустые квадратные скобки, например `double []`.

Следует обратить внимание на два обстоятельства.

Во-первых, элементы массива в Си всегда нумеруются с 0, это не может быть изменено средствами языка явным образом. Такая нумерация дает наиболее быстрый доступ к элементам, поэтому подобное изменение и не требуется: при необходимости программист может добавлять (вычитать) необходимый сдвиг к индексу массива самостоятельно, отдавая себе отчет в том, что сложение (вычитание) тратит процессорное время.

Во-вторых, Си не проверяет возможность выхода индекса массива за границы диапазона, и не имеет средств подключения автоматической проверки при обращении к элементам массива. Такая проверка времяемкая операция, осуществляется программистом по необходимости в исключительных случаях (например, при проходе всех или части элементов массива с использованием цикла со счетчиком достаточно проверить только, что границы изменения счетчика не выходят за пределы допустимых значений индекса массива, а проверка при обращении к каждому элементу будет явно избыточной).

```
int numbers[3] = {1, 10, 5}, i, j;

i = numbers[2] * numbers[1];     /* Корректно */

j = numbers[3];                  /* Некорректно, чтение за пределами массива
                                  * по факту скорее всего это будет чтение значения
                                  * переменной i, так как раз по декларатору она
                                  * следует сразу за массивом, то и в памяти расположится
                                  * следом ( хотя это не гарантируется )
                                  */

i = numbers[-1];                 /* Некорректно, чтение за пределами массива,
                                  * по факту - данных другой функции или кода,
                                  * значение непредсказуемо
                                  */
```

```
numbers[-10] = 5;              /* Некорректно, перезапись данных вне массива
                                * по факту - данных другой функции или кода,
                                * последствия непредсказуемы
                                */
```

В последнем случае примера непредсказуемые последствия в лучшем случае могут заключаться в перезаписи каких-то данных и появлении неожиданных значений (необязательно числа 5, т.к. затрагиваемая ячейка содержать данные переменной другого типа или могут даже "пострадать" несколько переменных меньшей длины), в худшем — исполнении данных, которые будут интерпретированы как непредсказуемые машинные команды с вероятным крахом работы программы и менее вероятным крахом системы и потерей данных (последнее крайне маловероятно в результате "несчастного случая" — ошибки, но возможно по умыслу злоумышленника).

Таким образом, массивы в Си довольно опасны и с ними следует соблюдать осторожность.

Вопросы для самопроверки.

1. *Что такое массив?*
2. *Что такое элемент массива?*
3. *Что такое индекс элемента массива?*
4. *Как объявляется одномерный массив в Си?*
5. *Как инициализировать значение элементов массива в Си?*
6. *Как осуществляется доступ к элементам массива в Си?*
7. *Что возвращает операция `[]`?*
8. *Как нумеруются элементы массива в Си?*
9. *Как проверяются границы массива при доступе к элементам в Си?*
10. *Что может произойти при попытке доступа к элементу за границами массива в Си?*

§4.2. Связь массивов и указателей

Для того, чтобы получить доступ к данным, хранимым в ячейке памяти, необходимо и достаточно знать ее адрес и тип данных. Адрес может быть постоянным (переменная) или храниться в другой переменной (указатель). Если указатель типизированный, то он сообщает всю необходимую и достаточную информацию для доступа к данным.

Для того, чтобы добраться до элемента массива, необходимо и достаточно знать тип элементов массива и адрес первого (нулевого элемента). Всю эту информацию также сообщает типизированный указатель. Поэтому указатель может использоваться как одномерный массив.

Рассмотрим массив

```
int a[5];
```

Вычисление адреса элемента `a[2]` будет осуществляться следующим образом: к адресу начала массива (первого элемента) будет прибавлено 2, умноженное на размер элемента массива, то есть `sizeof(int)`.

Обозначим за $S(x)$ — размер данных x в байтах (результат операции `sizeof`, аналогично применимый как к значению, так и к имени типа данных), за $P(x)$ — номер ячейки памяти, в которой размещается переменная, массив (фактически его нулевой элемент), элемент массива или любое иное значение L-value. Тогда

$$P(\texttt{a[2]}) = P(\texttt{a[0]}) + 2 \cdot S(\texttt{a[0]}).$$

Нетрудно заметить, что размер (число элементов) массива `a` для вычисления адреса элемента не требуется, он нужен только при определении массива для вычисления объема выделяемой для его хранения памяти.

Таким образом, типизированному указателю можно присвоить значение массива элементов того же типа, любой указатель можно использовать как массив:

```
int a[5];
int *p = a;
```

С этого момента `p` и `a` практически равносильны:

$$P(\texttt{a[i]}) = P(\texttt{a[0]}) + \texttt{i} \cdot S(\texttt{a[0]}) = P(\texttt{p[i]}) = P(\texttt{*p}) + \texttt{i} \cdot S(\texttt{*p}) = P(\texttt{p[0]}) + \texttt{i} \cdot S(\texttt{p[0]}).$$

И можно использовать, например такой код:

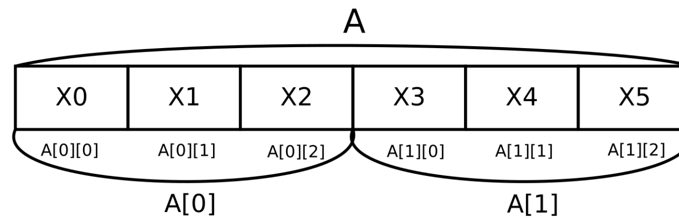


Рис. 4.2: Двумерный массив `float A[2][3]` и его элементы в памяти: элементы являются переменными типа `float`, элементы `A[0]` и `A[1]` — являются одномерными трехэлементными массивами (`тип float[3]`.

```
p[1] = 5;
a[2] = p[1];
```

(Заметим, что формулы выше являются именно математическими формулами и не должны использоваться как программный код для вычисления адресов.)

Отличие массивов и указателей в том, что у них разный размер: `sizeof(a)` вернет $5 \cdot$ `sizeof (int)`, а `sizeof(p)` вернет размер указателя, `sizeof(void*)`. Кроме того, массив, так же как и переменная — фактически "постоянный адрес", некоторая "адресная константа" — в отличие от указателя, массиву нельзя присвоить другое значение, чтобы он указывал на другие данные. (Замечание: не следует путать константы и неизменяемые данные.)

Тип данных строковой константы — массив символов `const char[]`, *но указатель* `const char *`, *является совместимым с соответствующим массивом.*

Вопросы для самопроверки.

1. *Что общего и в чем разница между указателем и одномерным массивом?*
2. *Указатель какого типа является совместимым с одномерным массивом элементов заданного типа?*
3. *Как вычисляется адрес элемента массива?*

§4.3. Многомерные массивы

Многомерные массивы являются массивами массивов. Ими можно моделировать таблицы (матрицы) или более сложные структуры данных. Соответственно, используется термины одномерные, двумерные, трехмерные и т.д. массивы.

```
float two_d[3][5];   /* двумерный массив  */
int   prop[3][4][6]; /* трехмерный массив */
```

Инициализация многомерных массивов также возможна при объявлении:

```
float a[2][3] = {{ 1.0,  2.51, 1.5 },
                 { 6e-4, 7e1,  8.9 }
                };
```

(инициализируется массив из двух элементов, каждый из которых является массивом из трех элементов).

В памяти Си представляет массив "по строкам", то есть элементы будут располагаться в порядке `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[1][0]`, `a[1][1]`, `a[1][2]`. При этом выражениям a[0] и a[1] корректны, им соответствуют данные массивного типа — трехэлементные одномерные массивы типа `float` (рис 4.2).

"Строки" массива не могут иметь разное число элементов, иначе бы речь шла уже не о многомерном массиве, а о массиве массивов разного размера (реализация такой структуры возможна и будет рассмотрена при изложении динамических массивов).

Рассмотрим n-мерный массив

$$\mathit{type}\ \ a\,[d_0]\,[d_1]\ldots[d_{n-1}]\,;$$

Вычисление адреса элемента по индексу

$$a\,[i_0]\,[i_1]\ldots[i_{n-1}]$$

(в байтах) производится следующим образом:

$$P(a\,[i_0]\,[i_1]\ldots[i_{n-1}]) = P(a\,[0]\,[0]\ldots 0) + i_0 \cdot S(a\,[0]) + i_1 \cdot S(a\,[0]\,[0]) + i_2 \cdot S(a\,[0]\,[0]\,[0]) + \cdots + i_{n-1} \cdot S(a\,[0]\ldots[0]),$$

или

$$P(a[i_0][i_1]\dots[i_{n-1}]) = P(a[0][0]\dots 0) + i_0 d_1 d_2 \cdots d_{n-1} S(type) + \cdots + i_1 d_2 d_3 \cdots d_{n-1} S(type) + \cdots + i_{n-1} S(type).$$

Для того, чтобы вычислить адрес элемента многомерного массива, Си должен знать, сколько элементов содержится в массиве, получаемом при взятии всех подмассивов, кроме одного, наибольшего. Многомерный массив является одномерным массивом массивов размерности на единицу меньшей, для вычисления адреса в этом, "внешнем", массиве не требуется знать число его элементов. Однако необходимо знать размер элемента: он равен произведению числа элементов "внутреннего" массива на размер элемента этого "внутреннего" массива (который в свою очередь может быть массивом и т.д.).

Таким образом, хотя в Си возможно объявление указателя на указатель,

```
int **a;
float ***b;
```

такие указатели не могут выступать в качестве многомерного массива:

```
float m[2][3][4];
b = m;           /* Указатели несовместимого типа */
```

`b[1]` имеет размер указателя, а не массива 2×2, укажет совершенно не туда, куда `m[1]` и т.д.

Совместимым типом указателя будет

```
float m[2][3][4];
float (*b)[3][4];
b = m;           /* Корректно */
```

Таким образом, наименование типа n-мерного массива есть

$$\langle\text{тип}\rangle\ \texttt{[]}\,\texttt{[}d_1\texttt{]}\,\texttt{[}d_2\texttt{]}\dots\texttt{[}d_{n-1}\texttt{]}$$

а указателя на него —

$$\texttt{(*}\ \langle\text{тип}\rangle\texttt{)}\ \texttt{[}d_1\texttt{]}\,\texttt{[}d_2\texttt{]}\dots\texttt{[}d_{n-1}\texttt{]}$$

Следует отметить, что скобки необходимы, так как символ указателя имеет более низкий приоритет, чем объявление массива (приоритет относится как к данным пунктуаторам так и к соответствующим операциям — разыменования и взятия элемента массива), поэтому объявления

```
float (*m1)[2];
float *m2[2];
```

неравнозначны: `m1` — указатель на двумерный массив элементов типа `float`, а `m2` — одномерный массив указателей на `float` (рис. 4.3).

Вопросы для самопроверки.

1. *Как объявляются и инициализируются многомерные массивы в Си?*
2. *Как располагаются в памяти элементы многомерных массивов в Си?*
3. *Какой тип указателя совместим с многомерным массивом в Си?*
4. *Чем отличается указатель на массив от массива указателей?*
5. *Как объявить двумерный массив указателей на `int`?*
6. *Как объявить указатель на двумерный массив указателей на `int`?*

§4.4. Массив как аргумент функции

Синтаксически массив может быть передан в качестве аргумента функции

```
void fnc_array (int a[10]);

void fnc_2d_array (int a[5][10]);
```

однако *Си никогда не передает в функцию собственно массив, передается лишь указатель на нулевой элемент*, поэтому указанные прототипы полностью равносильны следующим:

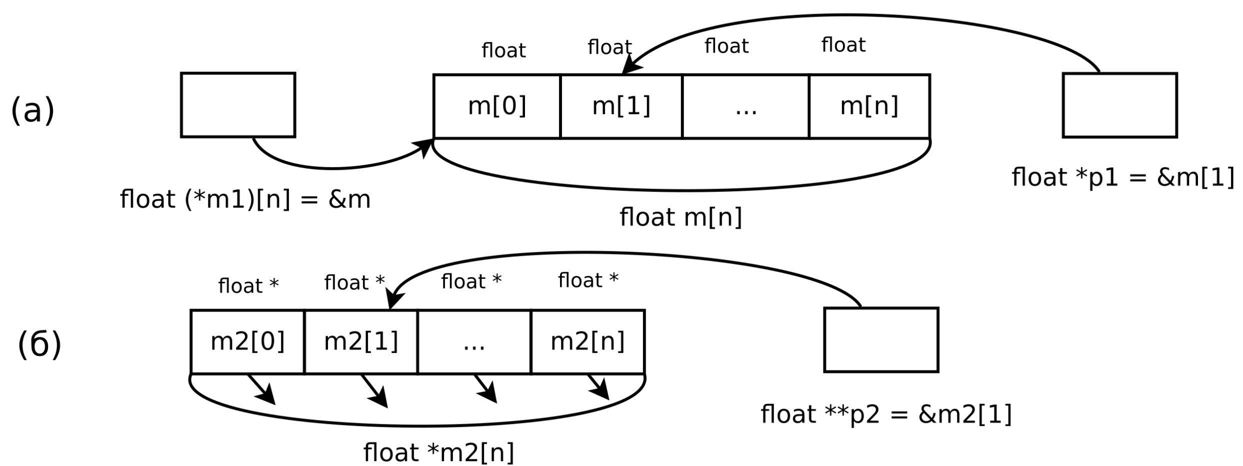


Рис. 4.3: Сравнение (а) указателя `m1` на одномерный массив `m` и (б) одномерного массива указателей `m2`. Указатели массива `m2` указывают на некоторые, возможно различные ячейки памяти (на схеме не показаны). Доступ к элементам массива `m` через `m1` может осуществляться как `(*m1)[i]`. Дополнительно показаны возможные указатели на элемент массива. `*p1` есть элемент массива `m`, `*p2` есть элемент массива `m2`, `**p2` — ячейка, куда указывает элемент массива `m2`.

```
void fnc_array (int *a);

void fnc_2d_array (int (*a)[10]);
```

Таким образом, при *передаче массива в функцию, во-первых, не передается информация о размере массива, во-вторых, элементы массива в памяти не копируются.*

Возможность указывать массивы в качестве формального параметра оставлена с целью наглядности, но указываемый размер массива ни на что не влияет и может быть даже опущен:

```
void fnc_array (int a[]);

void fnc_2d_array (int a[][10]);
```

что также будет эквивалентно передаче указателя. Можно даже указывать отрицательное значение. Однако для многомерных массивов размер внутренних массивов имеет значение: он необходим для определения размера элемента передаваемого массива в соответствии с соображениями изложенными в предыдущем параграфе.

Поскольку формальным параметром функции в любом случае будет указатель, информация о размере массива в функцию не передается, и его нельзя вычислить, например, используя `sizeof(a)/sizeof(int)`: такое действие применимо лишь в области видимости объявления массива, то есть где под массив выделена память (следует отметить, что как при обращении к элементу собственно массива, так и при обращении к элементу массива по указателю Си не проверяет допустимость значения индекса).

Если число элементов важнó, то его необходимо передать отдельным параметром. Так же делается в случае, когда при использовании статических массивов делают массив “с запасом”, а фактическое число элементов хранится отдельно.

```
/*
 * Вычисляет сумму элементов массива
 * a - массив
 * num - число элементов
 * возвращает сумму первых num элементов массива a
 */
double array_sum (double *a, size_t num)
{
    size_t i;
    double r = 0;
    for (i = 0; i < num; ++i)
        r += a[i];
    return r;
}
```

Следует обратить внимание на то, что в силу нумерации массивов с нуля при имитации цикла со счетчиком в Си-подобных языках типично используется строгое неравенство для задания верхней границы: использование `i <= num - 1` приведет к избыточному вычислению при каждом проходе цикла. Использование отрицания равенства (`!=`) возможно и равносильно в данном случае, но обычно эта операция при создании цикла со счетчиком не применяется, так как в случае замены в итераторе инкремента, например, на `i += 2` есть риск "проскочить" конечное значение с неприятными последствиями.

При передаче в функцию элементы массива не копируются, копируется только адрес массива, что является преимуществом, так как копирование большого объема данных требует времени и должно быть исключено по возможности. Однако по умолчанию при таком подходе функция может изменить содержимое массива в вызвавшей функции. Чтобы исключить такую возможность для массива, который является входным параметром, и гарантировать его сохранность, следует обязательно использовать указатель на константу:

```
/*
 * Вычисляет сумму элементов массива
 * a - массив
 * num - число элементов
 * возвращает сумму первых num элементов массива a
 */
double array_sum (const double *a, size_t num)
{
    size_t i;
    double r = 0;
    for (i = 0; i < num; ++i)
        r += a[i];
    return r;
}
```

Вопросы для самопроверки.

1. *Как передать массив в качестве аргумента функции?*
2. *Какая разница между передачей массива и указателя на нулевой элемент массива в качестве аргумента функции?*
3. *Как узнать число элементов массива-аргумента в функции?*
4. *В каких случаях при передаче массива (указателя на нулевой элемент) в функцию следует использовать `const`?*

§4.5. Строки символов

За исключением строковых констант, в Си нет особых синтаксических единиц для строк. Строковые константы имеют тип `const char[]`, строки представляются как массивы символов. Не всякий массив символов может считаться строкой, строками в Си являются нуль-терминированные строки, которые представляются в памяти как последовательности символов, завершающихся символом с кодом 0. Данный символ автоматически вставляется в конец строковых констант.

Следующие инициализации эквиваленты:

```
char s[] = "String";

char s[] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
```

Имеет право на существование и пустая строка — и она представляет собой массив из одного элемента:

```
char s[] = "";

char s[] = {'\0'};
```

При задании размера массива символов следует иметь в виду, что строка требует на один элемент массива (один байт) больше своей длины.

В стандартной библиотеке Си определены функции для обработки строк, их прототипы указаны в заголовочном файле `string.h`.

Например, функция

```
size_t strlen(const char *s);
```

вычисляет фактическую длину строки, функция

```
char * strcpy(char *dest, const char *src);
```

копирует строку src в строку dest, возвращая при этом указатель на dest, функция

```
char * strcat(char *dest, const char *src);
```

добавляет строку src к строке dest.

Например,

```
#include <stdio.h>
#include <string.h>

/* ... */

    const char *s[] = {"One", "Two", "Three"};
    const char str[] = "Number␣";

    /* ... */

    char text[20];
    /* ... */

    strcpy(text, str);
    strcat(text, s[1]);
    printf("The␣text␣is␣'%s'\n", text);
```

Тот факт, что strcpy возвращает указатель на dest, позволяет две ключевые строки из этой программы заменить одной:

```
    strcat(strcpy(text, str), s[1]);
```

Следует обратить внимание на то, что массив text должен содержать достаточное количество байтов, иначе функции strcpy и strcat могут затереть данные.

В таком контексте — когда под "будущую" строку (т.е. строку, создаваемую в процессе работы программы, чья длина на стадии написания и компиляции неизвестна) отводится массив символов "с запасом" — данный массив называется **строковым буфером**.

Переполнение строкового буфера — запись символов строки за границей массива — распространенная проблема в программировании. При копировании строки в буфер из какого-либо источника одним актом может быть перезаписан очень большой объем данных — ведь неизвестно, когда встретится символ с кодом ноль в строке src. Результатом может оказаться как крах программы, так и потеря данных, так и взлом системы хакером: переполнение буфера — один из наиболее распространенных источников уязвимостей цифровых систем.

Следует учитывать, что *Си строки небезопасны*, и при работе с ними следует проявлять особую осторожность.

Более безопасные, хотя и более медленные функции, эквивалентные предыдущим —

```
char * strncpy(char *dest, const char *src, size_t n);
char * strncat(char *dest, const char *src, size_t n);
```

ограничивают объем создаваемой строки n символами. Таким образом, более безопасная реализация была бы

```
    strncat(strncpy(text, str, 19), s[1], 19);
```

однако, данные функции не добавляют символ с кодом '\0' в конец строки, если уже скопировано n байтов, а исходная строка не закончилась, что также оставляет потенциальный источник проблемы — не-нуль-терминированную строку, при обращении к которой может произойти чтение неопределенно большого объема данных (в т.ч. потенциально конфиденциальных) — до первого встреченного в оперативной памяти символа с кодом 0 после строки.

Функция

```
int strcmp (const char *str1, const char *str2);
```

проводит посимвольное сравнение двух строк, возвращая отрицательное число, если первая строка меньше второй — то есть если первый несовпадающий символ двух строк таков, что в первой строке его код (интерпретируемый как unsigned char) меньше, — 0, если строки равны, положительное, если вторая строка больше.

Завершающий символ '\0' также участвует в сравнении, т.е. он меньше кода любого другого символа, поэтому любой префикс строки меньше самой строки.

Функция

```
char * strchr (const char *str, char c);
```

возвращает указатель на первое вхождение символа `c` в строку `str`, или `NULL`, если такого символа в строке нет.

Существуют и другие функции обработки строк: изучение всех функций стандартной библиотеки не является прямой целью данного курса, эти сведения являются справочными данными.

Вопросы для самопроверки.

1. *Как хранятся строки в Си?*
2. *Какие функции для работы со строками существуют в стандартной библиотеке языка Си?*
3. *Чем отличаются строки и массивы символов?*
4. *Что такое строковый буфер?*
5. *Что такое переполнение строкового буфера и к каким последствиям оно может привести?*
6. *Что такое нуль-терминированная строка?*
7. *К каким последствиям может привести отсутствие символа с кодом 0 в строке?*

§4.6. Ввод и вывод строк

Прототипы всех функций данного параграфа задаются в `stdio.h`.

Вывод строк может осуществляться с помощью спецификатора `%s` в `printf`. Это стандартная операция, которая не вызовет проблем, за исключением попытки передать указатель на то, что корректной нуль-терминированной строкой символов не является: в этом случае выведутся все байты до символа с кодом 0, где бы он в памяти не встретился. Если он встретится "слишком поздно", то может быть выведен огромный объем хаотических и управляющих символов, а также потенциально конфиденциальная информация.

Замечание. При выводе одиночной строки `s` может возникнуть искушение вызвать

```
printf (s);
```

вместо

```
printf ("%s", s);
```

что во многих случаях сработает, ведь `printf` выводит первый аргумент — строку — на печать. Однако эта строка обрабатывается особым образом: все сочетания символов, начинающиеся с `%`, будут восприняты как форматы полей, которых в аргументе нет, поэтому адреса выводимых данных — а как следствие и сами данные — будут непредсказуемы. Для строк, полученных от пользователя, это является источником уязвимости, в некоторых случаях "правильным" подбором строки злоумышленник может добиться вывода конфиденциальных данных.

Также вывод строк осуществляет функция

```
int puts(const char *s);
```

с очевидным смыслом. Функция

```
int putchar(char c);
```

выводит один символ. Это более низкоуровневые (и быстрые) функции, чем `printf`.

Обе функции возвращают положительное значение в случае успеха, в случае неудачи возвращается особая символическая константа `EOF`. Аббревиатура EOF расшифровывается как End-Of-File — конец файла и обычно используется для индикации достижения конца файла при его последовательном чтении. В данном случае и при многих других ошибках ввода и вывода, функции стандартной библиотеки Си возвращают данную символическую константу в качестве индикации факта возникновения ошибки.

Существует симметричная функция

```
char *gets(char *s);
```

вводящая строку `s` и возвращающая указатель на нее в случае успеха, *однако данная функция никогда не должна использоваться, так как она не проверяет размер буфера*, поэтому обязательно будет источником проблем. В отличие от функций `strcpy` и им подобных, строки-аргументы которых известны до вызова функции, поступающую от пользователя строку до вызова функции `gets` проверить невозможно, поэтому данная функция считается "запрещенной".

Также не следует использовать

```
scanf ("%s", s);
```

хотя это верная конструкция.

В последнем случае можно и нужно указывать предельную длину строки, которая будет прочитана:

```
scanf ("%10s", s);
```

При этом буфер `s` должен содержать на 1 байт больше, то есть как минимум 11. `scanf` в этом случае создаст корректную нуль-терминированную строку, даже если в нее поместится лишь часть строки, введенной пользователем.

Обратите внимание на отсутствие операции взятия адреса строки: это корректно, так как `s` уже указатель, а вызов

```
scanf ("%10s", &s);
```

может привести к ошибке, так как даже указатель на массив, как вычисленный компилятором адрес, не обязан совпадать с указателем на нулевой элемент массива, а если `s` является не массивом, а именно указателем, эти значения будут принципиально разными.

По умолчанию функция `scanf` прочитывает строку только до пробела. В современных реализациях Си существует способ задать в `scanf` символы, которые могут и которые не могут войти в строку, они указываются в квадратных скобках как перечисления или диапазоны, а также как отрицания. Так можно вводить строку с пробелами и т.п. Например,

```
scanf("%10[0-9a-zA-Z␣]", str);
```

считает элементами строки цифры, буквы и пробел, но введет не более 10 символов.

```
scanf("%10[^\n]", str);
```

будет считать строкой все, кроме символа перевода строки (знак `^` означает "кроме указанных далее символов") .

Функция

```
int getchar(void);
```

может использоваться для ввода символа (возвращает символ, приведенный к `int` или `EOF` в случае ошибки). Возвращаемое значение имеет тип `int`, а не `char` именно для того, чтобы имелась возможность вернуть 256 различных вводимых символов и отличное от них значение, соответствующее ошибке

Вопросы для самопроверки.

1. *Что делает функция `puts`?*
2. *Как выводить строки с помощью `printf`?*
3. *Что делает функция `gets`?*
4. *Почему функция `gets` считается "запрещенной"?*
5. *Как вводить строки с помощью `scanf`?*
6. *Что следует указывать в качестве строкового поля в `scanf`?*
7. *Чем лучше использование `scanf` вместо `gets`?*
8. *Можно ли использовать `%s` без ограничения числа вводимых байтов в `scanf`?*

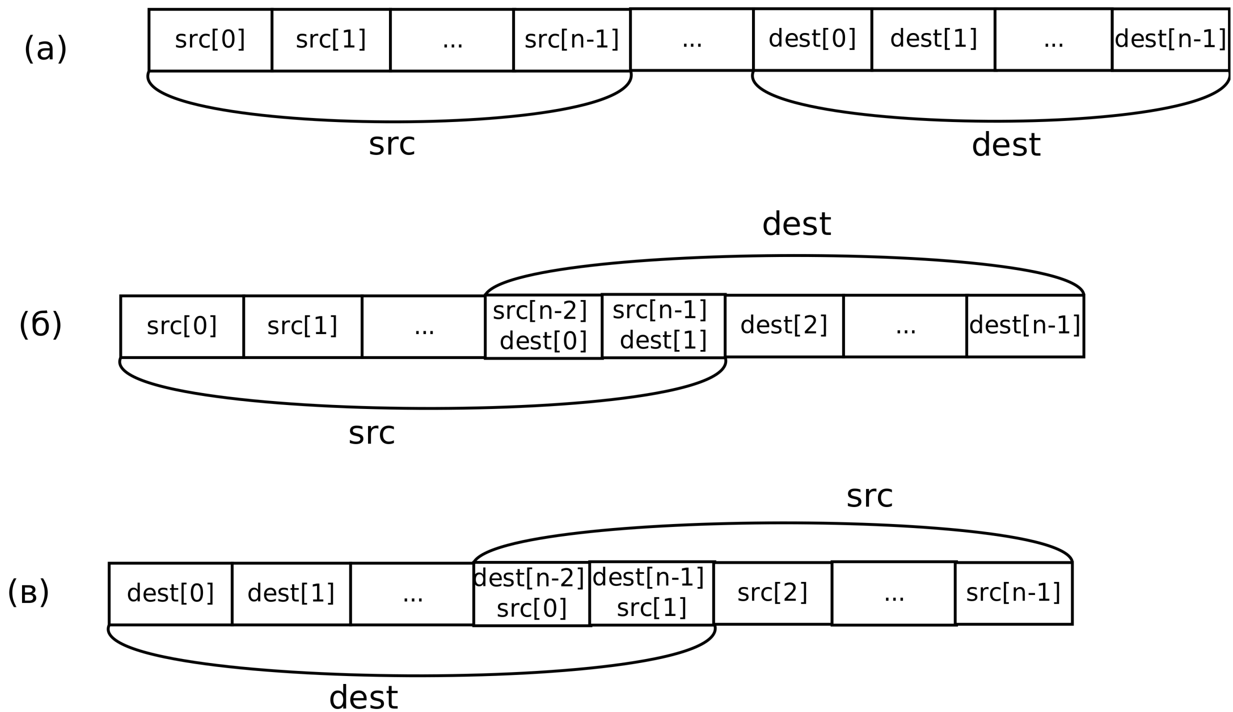


Рис. 4.4: Неперекрывающиеся блоки памяти (а), использование `memcpy` возможно, и перекрывающиеся блоки памяти (б — смещение вправо, в — смещение влево), использование `memcpy` не допускается.

§4.7. Обработка массивов в стандартной библиотеке Си

Обработка массивов — блоков данных в памяти — во многом похожа на обработку строк и осуществляется аналогичными функциями, также задаваемыми в `<string.h>` (ведь Си-строки являются частным случаем массивов):

```
/* копирует n байт из области памяти src в dest
 * области не должны пересекаться, иначе результат неопределен
 * возвращает dest
 */
void *memcpy(void *dest, const void *src, size_t n);
```

Данная функция работает с нетипизированными указателями, ей все равно какие данные копировать, важно сколько их. Например, чтобы скопировать массив `a` из `n` элементов типа `int` в аналогичный массив `b` следует вызвать

```
memcpy(b, a, n * sizeof(int));
```

Следует отметить, что компилятор не будет проверять наличие ошибок вычисления размера, в частности, если указание `sizeof(double)` вместо `sizeof(int)` приведет к копированию бо́льшего объема данных, чем требуется, потенциальной записи данных за границей буфера `b` и чтению за границей блока `a`.

Кроме того, функция не будет работать корректно, если блоки `src` и `dest` перекрываются (например, как на рис. 4.4). В частности, не допускается использование данной функции для смещения части элементов массива на заданное количество позиций вправо или влево. Однако `memcpy` — самая быстрая функция копирования данных в памяти. Функция, допускающая работу с перекрывающимися блоками — `memmove` — имеет такой же прототип, но работает медленнее, так как копирует данные через резервный блок.

Также можно отметить функции

```
/*
 * возвращает указатель на первое вхождение значения c
 * среди первых n байтов s или NULL, если не найдено
 */
void *memchr(const void *s, char c, size_t n);

/*
```

```
 * заполняет первые n байт области s значением c
 * возвращает s
 */
void *memset(void *s, char c, size_t n);
```

и другие.

Заметим, что копирование в рамках одного массива корректно, например можно использовать

```
double a[10];

/* ... */

memcpy (&a[0], &a[5], 5 * sizeof (double));
```

копирует элементы с 5 по 9 в элементы с 0 по 4 — блоки не перекрываются.

```
double a[10];

/* ... */

memmove (&a[0], &a[1], 9 * sizeof (double));
```

смещает элементы с 1 по 9 на 1 позицию влево (блоки перекрываются). Аналогичные действия можно выполнять и с помощью функций обработки строк (`strcpy` и др.) в рамках одной строки.

Вопросы для самопроверки.

1. *Что делает функция `memcpy`?*
2. *Какие ошибки могут быть допущены при работе с функцией `memcpy`?*

§4.8. Структуры

Составной тип данных “**структура**” в Си позволяет объединить в одном значении данные различных типов (в противоположность массивам). Элементы структуры называются **полями**, доступ к ним осуществляется по имени. При декларации каждой конкретной структуры (как типа данных) программист имеет возможность выбрать количество, тип и имена полей.

Структуры в частности ориентированы на моделирование объектов реального мира, например, можно создать структуру “студент”, поля которой будут хранить данные об имени (строка), группе (целое число), среднем балле (число с плавающей точкой) и т.д.

Идейно структуры позволяют объединить взаимосвязанные данные, которые обрабатываются совместно или моделируют один объект, в одну переменную. Использование структур часто позволяет избежать создания функций с большим количеством аргументов (большим считается число аргументов в 4-5 и более). Большое количество параметров может послужить сигналом, что следует объединить их в структуру: часто они кодируют какой-то один объект и одинаковые наборы параметров встречаются в нескольких функциях.

Такой подход удобен также тем, что если при модификации программы возникнет необходимость обрабатывать несколько отличный набор данных (например, передавать дополнительные данные), то достаточно будет будет внести изменение в списке полей структуры, не меняя прототип функции (конечно, этим следует пользоваться только в том случае, если добавляемые данные являются идейно связанными со структурой).

Тип данных “структура” объявляется следующим образом (аналогично перечислению):

`struct` [⟨*идентификатор*⟩] { ⟨*список полей*⟩ } [⟨*переменная*$_1$⟩, ⟨*переменная*$_2$⟩, ...];

Аналогично перечислениям, именно сочетание `struct` ⟨*идентификатор*⟩ становится именем типа данных, поэтому в дальнейшем переменные нужно будет объявлять с указанием ключевого слова `struct`. Переменные можно объявить и сразу при объявлении структуры (такой подход не используется часто, т.к. если объявляемый тип данных используется несколькими функциями, то данные переменные станут глобальными). Однако, если идентификатор структуры не указать, переменные **анонимной** структуры все равно могут быть объявлены вместе с самой структурой.

Перечисление полей осуществляется как объявление переменных (без инициализации) — после имени типа, однотипных через запятую, разнотипных через точку с запятой.

Приближенное формальное определение выглядит следующим образом:

⟨*поле*⟩ ::= ⟨*идентификатор*⟩ | `*`⟨*поле*⟩ | ⟨*поле*⟩ `[`⟨*целочисленная константа*⟩`]`

⟨*список полей*⟩ ::= ⟨*поле*⟩ | ⟨*список полей*⟩`,`⟨*поле*⟩

⟨*объявление полей*⟩ ::= ⟨*тип данных*⟩ ⟨*список полей*⟩;

⟨*список объявлений полей*⟩ ::= | ⟨*список объявлений полей*⟩⟨*объявление полей*⟩

⟨*struct-префикс*⟩ ::= `struct` | `struct` ⟨*идентификатор*⟩

⟨*список struct-переменных*⟩ ::= ⟨*идентификатор*⟩ | ⟨*список struct-переменных*⟩`,`⟨*идентификатор*⟩

⟨*объявление struct-переменных*⟩ ::= | ⟨*список struct-переменных*⟩

⟨*struct-объявление*⟩ ::= ⟨*struct-префикс*⟩ `{` ⟨*список объявлений полей*⟩ `}` ⟨*объявление struct-переменных*⟩ `;`
Список объявлений полей аналогичен списку объявлений переменных, но не допускает инициализацию. Кроме перечисленных выше вариантов — переменная, указатель, массив — поле может быть структурой и перечислением. Объявление структуры аналогично объявлению enum-перечислений — опционально объявляется идентификатор типа данных и переменные вводимого типа, допускаются анонимные структуры.

Идентификаторы полей доступны только при доступе к элементам структуры. Каждая переменная-структура в памяти представляет собой комбинацию переменных соответствующего типа — эти поля-переменные принадлежат переменной-структуре, а не типу данных "структура".

Пример:

```
struct student
{
    char name[100];
    int group, course;
    double mark;
};

struct student s1, s2;
```

Доступ к полям осуществляется с помощью операции взятия поля структуры — точка (.), ее левым операндом должна быть структура, правым — идентификатор (имя поля). Результат — значение L-value. В данном примере:

```
s1.group = 15;
r += s1.mark / n;
```

По причинам, указанным ниже, очень часто используются указатели на структуру. При этом можно осуществлять доступ к полям, предварительно разыменовывая указатель (при этом разыменование должно заключаться в круглые скобки, так как точка имеет более высокий приоритет):

```
struct student *p = &s1;
(*p).group = 15;
r += (*p).mark / n;
```

а можно воспользоваться операцией взятия поля в структуре по указателю — стрелочка `->`, что удобнее и не требует скобок:

```
struct student *p = &s1;
p->group = 15;
r += p->mark / n;
```

Структуры и массивы могут быть вложенными друг в друга произвольным образом, т.е. полем структуры может быть массив, можно создать массив структур, полем структуры может быть структура, элементом массива может быть массив и т.д. — формальных ограничений для уровня вложености нет. При создании структурных типов данных рекомендуется, чтобы для каждого типа данных было дано четкое представление (описание), что именно (какой объект) этот тип моделирует.

Можно также заметить, что формально в фигурных скобках при объявлении структуры указывается не просто список полей, а список объявлений, т.е. внутри структуры можно объявлять другие структуры и т.п. В частности, можно воспользоваться анонимной структурой подобным образом:

```
struct somestruct
{
```

```
    struct
    {
        double x, y;
    } point;
    int n;
} w;
```

У переменной `w` будет 3 поля, доступных следующим образом: `w.n`, `w.point.x`, `w.point.y`.

Следует обратить внимание, что поля есть у переменной-структуры, а не у типа данных, вызов `somestruct.n` будет ошибочным. Именно при определении структурной переменной выделяется память под все поля, каждая структурная переменная одного типа хранит полный комплект одноименных полей независимо.

Для структур определена операция прямого присваивания: если переменные имеют один структурный тип, все поля из одной переменной будут скопированы в другую.

Структура может быть параметром и возвращаемым значением функции, при этом она передается по значению, то есть копируется в памяти. *Даже если элементом структуры является массив, он будет скопирован в памяти.* Поэтому не рекомендуется передавать структуры в функции по значению (за исключением случаев, когда структура состоит из нескольких скалярных полей, но даже это нежелательно, ведь простота структуры может измениться при модификации программы). При этом входные структуры следует передавать как указатель на константу. Например:

```
void foo (const struct mystruct *in_struct, struct mystruct *out_struct);
```

задает функцию, с одним входным (in_struct) и одним выходным (out_struct) параметром-структурой.

Как и массив, структура может быть инициализирована при объявлении путем перечисления значений полей через запятую. Например,

```
struct complex
{
    double re, im;
};

struct complex i = {0, 1};
```

Это возможно и в случае, если поле является строкой или массивом:

```
struct student
{
    char name[100];
    double marks[3];
    int group, course;
};

struct student s = {"Name", {5.0, 3.5, 4.5}, 100, 1};
```

Начиная со стандарта C99 возможно указание идентификаторов конкретных полей для инициализации, например

```
struct student s = {.marks  = {5.0, 3.5, 4.5},
                    .course = 1,
                    .group  = 100,
                    .name   = "Name"
                   };
```

Вопросы для самопроверки.

1. *Что такое структура (тип данных) в Си?*
2. *Для чего используются структуры (тип данных)?*
3. *Как объявляются структуры в Си?*
4. *Что такое поля структуры в Си?*
5. *Как осуществляется доступ к полям структур в Си?*
6. *Как можно и как нужно передавать структуры в функции и почему?*
7. *Может ли существовать массив структур?*

8. *Как объявить структуру так, чтобы при объявлении переменных не нужно было указывать ключевое слово `struct`?*
9. *Перечислите операции, результатом которых являются значения L-value.*

§4.9. Функциональный указатель

Для генерации кода вызова функции компилятору необходимо иметь сведения о передаваемых и возвращаемых данных (тип, порядок и количество параметров, тип возвращаемого значения), а компоновщику — вычислить адрес ячейки кода, куда будет передано управление.

При вызове функции по ее имени следует считать, что каждому идентификатору функции, подобно идентификатору переменной, соответствует некоторая ячейка памяти, в которой содержится код функции. Фактически код функции (в соответствии с принципом однородности памяти) — тоже некоторые данные. При этом аналогично идентификатору переменной, у этих данных есть тип данных, задаваемый как прототип и определяющий способ обработки этих данных — с помощью операции вызова функции `()`. Точнее, имеет значение не весь прототип, а только тип возвращаемого значения, а также тип и порядок следования параметров, имена функции и формальных параметров роли не играют.

Соответственно, можно задать указатель, который будет указывать на данные условного *функционального типа* — **функциональный указатель**.

Общий принцип объявления указателя в Си — объявление идентификатора некоторого типа, предваренное символом указателя `*`, становится объявлением указателя на данные указанного типа. Аналогичная ситуация и с функциональным указателем: объявление функции (прототип) превращается в объявление указателя предварением имени функции символом `*`, однако в силу порядка прочтения пунктуационных знаков и операций в Си, идентификатор и символ указателя следует заключить в скобки, чтобы звездочка относилась к идентификатору, а не типу возвращаемого значения.

Пусть есть функция

```
int foo (int, double);
```

тогда указатель на функцию такого типа будет объявляться как

```
int (*pfnc) (int, double);
```

что не равносильно записи

```
int *foo2 (int, double);
```

объявляющей функцию `foo2`, которая в отличие от foo, возвращает `int *`, а не `int`.

Если же нужно объявить указатель, совместимый с прототипом `foo2`, то он будет записан как

```
int *(*pfnc2) (int, double);
```

В стандартном Си функции можно объявлять только в глобальной области видимости, функциональные указатели — как и любые другие переменные — как в глобальной, так и в локальной.

Функциональному указателю можно присваивать адреса функций совместимого типа и вызывать их через указатель:

```
int (*pfnc) (int, double);

/* ... */

int foo (int, double);
int bar (int, double);
int fnc (int, double);

/* ... */

pfnc = &foo; /* или */ pfnc = &bar; /* или */ pfnc = &fnc;

int k = (*pfnc) (1, M_PI);
```

На самом деле фактически не существует отдельного “функционального типа”, поэтому каждая функция — не более, чем адрес кода, и результатом разыменования функционального указателя будет тот же функциональный указатель, поэтому операции взятия адреса и разыменования можно опускать, что обычно и делается при написании программ на языке Си:

```
pfnc = foo; /* или */ pfnc = bar; /* или */ pfnc = fnc;

int k = pfnc (1, M_PI);
```

Функциональный указатель можно использовать в качестве параметра функции:

```
void workfnc (int (*f) (int, double))
{
    /* ... */
    i = f (k, b);
}

/* ... */

workfnc(foo);
```

Подобные объявления достаточно трудночитаемы: если одиночный параметр, являющийся функциональным указателем на функцию с простым прототипом (с малым количеством параметров, среди которых нет указателей) еще может показаться воспринимаемым, то крайне проблематично прочитать прямое объявление указателя на функцию, параметром которой является функция, параметрами которой являются несколько функций с несколькими параметрами и т.д. Поэтому в таких случаях — и вообще практически всегда при использовании функционального указателя — рекомендуется использовать `typedef`.

Здесь также действует общее правило: объявление переменной, предваренное `typedef`, превращается в объявление типа

```
typedef int (*tfnc) (int, double);

tfnc pfnc = foo;

/* или */

void workfnc (tfnc f)
{
    /* ... */
    i = f (k, b);
}
```

Пример применения функционального указателя — написание универсальной функции поэлементной обработки массива.

```
typedef int (*tf) (int *);

/* Применяет функцию f к каждому элементу n- элементного массива array */
void array_walk (int *array, size_t n, tf f)
{
    size_t k;
    for (k = 0; k < n; ++k)
    f(array[k]);
}

/* Пример функций f: взятие противоположного числа */
int inv (int *i)
{
    *i = -*i;
}

/* Пример функций f: взятие остатка от деления на 2 */
int mod2 (int *i)
{
    *i = *i % 2;
}

/* ... */
```

```
array_walk (a, n, inv);  /* заменяет все элементы массива на противоположные */
array_walk (a, n, mod2); /* заменяет четные по( значению) элементы массива на 0
                          * нечетные на 1 */
```

Вопросы для самопроверки.

1. *Что такое функциональный указатель?*
2. *Какую информацию несет тип данных "функциональный указатель"? Сам функциональный указатель?*
3. *Каков синтаксис объявления функционального указателя?*
4. *Чем отличается объявление функционального указателя от прототипа функции, возвращающей указатель?*
5. *Какие операции допустимы для функциональных указателей?*
6. *Как записать `typedef`-псевдоним функционального указателя?*
7. *Что является результатом разыменования функционального указателя?*
8. *Что является результатом взятия адреса идентификатора функции?*

Глава 5. Ввод и вывод

Для понимания материала данной главы требуется знание общих сведений о файлах, каталогах и путях. Они изложены в приложении A.2. Здесь необходимо упомянуть два ключевых момента.

- С точки зрения программирования на языке высокого уровня файл представляет собой конечную последовательность байтов, доступ к которой возможен по уникальному имени. Доступ может представлять собой чтение и запись (изменение) любого из байтов последовательности, создание пустого файла, добавление байта в конец файла.
- Побайтовый файловый ввод и вывод возможен, но работает существенно медленнее, чем ввод и вывод достаточно больших блоков данных. Функции языков программирования высокого уровня обеспечивают **буферизированный** ввод и вывод, при котором выводимые блоки сначала формируются в памяти, а затем записываются в файл и наоборот, сначала прочитываются блоки, которые затем передаются в программу. На практике это приводит к тому, что хотя произвольный доступ к любому байту файла возможен, последовательный доступ работает многократно быстрее. Основные функции работы с файлами обеспечивают именно последовательный доступ, в ассоциированном с файлом структурах ЯВУ хранится счетчик (указатель) позиции в файле, автоматически смещаемый на количество введенных-выведенных байт. Для произвольного доступа данный указатель следует смещать специальными функциями.
- Различают два *режима доступа* к файлу: **текстовый** и **двоичный**. Соответственно, при работе с файлами, предназначенными для сохранения **текста** — строк символов алфавита, знаков препинания и т.п., как правило человекочитаемых — следует использовать текстовый доступ, для файлов, предназначенных для хранения двоичных данных — представления чисел в памяти компьютера, служебных символов и т.п. — следует использовать двоичный доступ.

  При открытии файла в текстовом режиме в функциях чтения из файла происходит преобразование системозависимого кода перевода строки (`'\n'`, `'\r'`, `"\r\n"` и т.п. с Си-стандартизированный символ `'\n'` и обратно при записи. Также в текстовом режиме возможна обработка символа `'\0'` как конца строки и символа `'\x1A'` (десятичный код 26) как конца файла. В двоичном режиме никаких преобразований байтов не производится.

§5.1. Стандартные потоки ввода и вывода

Прежде чем перейти к файловым операциям в стандартной библиотеке Си, следует отметить, что на самом деле каждой программе ОС предоставляет **стандартный поток ввода**, **стандартный поток вывода** и **стандартный поток сообщений об ошибках**.

По умолчанию стандартный поток ввода — ввод с клавиатуры, стандартные потоки вывода — вывод на экран терминала. Функции `scanf`/`printf`, `getchar`/`putchar` и `gets`/`puts` на самом деле осуществляют ввод со стандартного потока ввода и вывод на стандартный поток вывода.

Программа

```
#include <stdio.h>
#include <string.h>

int main()
{
    char s[100];

    scanf ("%99[^\n]s", s);
    printf ("\"%s\"␣%u\n", s, strlen(s));

    return 0;
}
```

вводит с клавиатуры строку (до 100 символов) и выводит на экран ее в кавычках и ее длину.

Работа с потоками организована схожим с файлами образом. Более того, средствами оболочки ОС можно изменить стандартный поток как ввода, так и вывода, перенаправив их в файл. Например, пусть программа называется `strtst`, тогда команды

`./strtst <file1.txt >file2.txt`

(в ОС Linux)

`strtst.exe <file1.txt >file2.txt`

или

`strtst <file1.txt >file2.txt`

(в ОС Windows) устанавливает стандартный ввод из файла `file1.txt`, стандартный вывод — в файл `file2.txt`, вывод сообщений об ошибках сохраняется стандартным.

Тогда строка прочитается из файла `file1.txt`, а запишется в файл `file2.txt`: знак `<` перенаправляет стандартный ввод, `>` — стандартный вывод, Перенаправить можно один или оба потока сразу. О выводе в стандартный поток сообщений об ошибках и его перенаправлении речь пойдет в параграфе 5.4.

Таким образом, средствами ОС можно обеспечить ввод и вывод в файл, не прибегая к файловым операциям внутри программы, но, конечно, это подходит только если файлов не более двух и если считается допустимым требовать от пользователя использования таких конструкций. Утилиты командной строки, особенно в GNU/Linux, очень часто устроены именно таким образом: эти программы работают как фильтры, преобразуя входной поток данных в выходной. Средства интерпретатора команд оболочки предоставляет широкий набор возможностей для использования таких команд.

Следует обратить внимание, что перенаправление записи `>` в существующий файл приводит к безвозвратному уничтожению имеющихся в нем данных, поэтому пользоваться ей следует с осторожностью. Существует и другая возможность — перенаправление с помощью знака `>>`, которое приведет к дозаписи в конец существующего файла.

Вопросы для самопроверки.

1. *Что такое стандартные потоки ввода и вывода?*
2. *Откуда по умолчанию осуществляется стандартный ввод?*
3. *Куда по умолчанию осуществляется стандартный вывод?*
4. *Как перенаправить стандартный ввод из файла?*
5. *Как перенаправить стандартный вывод в файл?*
6. *С какими рисками сопряжен стандартный вывод в файл?*

§5.2. Открытие и закрытие файла

Перед началом работы с файлом его необходимо **открыть**. Открытие файла с одной стороны позволяет ассоциировать файл операционной системы с переменной (данными) в программе, с другой — сообщить ОС о том, что файл будет "занят" данной программой.

Открытие файла осуществляется с помощью функции `fopen`:

```
FILE* fopen (const char *filename, const char *mode);
```

где `filename` — имя открываемого файла (полный или относительный путь, а `mode` — режим открытия.

Прототипы всех файловых функций заданы в `<stdio.h>`

Для текстовых файлов имеются следующие режимы открытия:

режим	описание	если файл существует	если файл не существует
`r`	чтение	открывает для чтения с начала файла	ошибка (NULL)
`w`	запись	уничтожает содержимое файла	создает файл
`a`	добавление	записывает в конец	создает файл
`r+`	чтение и запись	открывает файл, чтение и (пере)запись с начала	ошибка (NULL)
`w+`	чтение и запись	уничтожает содержимое файла	создает файл
`a+`	чтение и запись	записывает в конец	создает файл

По умолчанию файл открывается в текстовом режиме, двоичные режимы получаются добавлением буквы `b` — `rb`, `wb`, `ab`, `r+b`, `w+b` и `a+b` соответственно.

Функция `fopen` возвращает указатель на данные особого — файлового — типа. Именно переменная-указатель на файловый тип является переменной, с которой ассоциируется файл. В случае неудачи (например, файл не существует или нет доступа) функция вернет `NULL`. В реальных реализациях Си `FILE` — некоторая структура, но ее поля не документированы стандартом и не важны при использовании (и не должны использоваться напрямую, так как их количество, имена и смысл — внутреннее дело стандартной библиотеки Си, зависимое от компилятора).

Заметим, что в заголовочных файлах стандартной библиотеки структура `FILE` может быть описана как неполный тип (см. приложение D.1), а память для хранения самой структуры выделяется динамически в функции `fopen` с помощью средств, описанных в 7.1, тем самым время жизни самой структуры превышает время работы функции.

По окончанию работы с файлом его необходимо закрыть. Делает это функция `fclose`:

```
int fclose(FILE *stream);
```

возвращающая 0 в случае успеха и ненулевое значение — особую константу `EOF` (определена в `stdio.h`) — в случае неудачи. Вообще-то `EOF` означает End-Of-File (конец файла) и предназначена для информирования о том, что при чтении достигнут конец файла, однако в некоторых функциях она используется с несколько другим смыслом.

Закрытие файла с одной стороны сообщает ОС, что файл освобожден программой, с другой — завершает отложенные (*буферизированные*) операции. Как уже было отмечено, запись в файл никогда не производится на диск сразу, а сначала буферизируется в памяти, пока не накопится достаточно большой блок данных. Во избежание потери этих данных закрытие файла обязательно. Также `fclose` освобождает память, выделенную для хранения структуры `FILE` в функции `fopen`. Функция `fclose` возвращает 0 в случае успеха и EOF в случае ошибки. Только после успешного выполнения `fclose` можно гарантировать, что данные действительно были записаны в файл (хотя и в этом случае есть риск потери данных уже на уровне собственной буферизации операционной системой — однако проверить это из программы невозможно даже прочитав созданный файл и его содержимое с требуемым, ведь ОС необязательно прочитает данные именно с носителя информации, а не из буфера).

Пример:

```
FILE f = fopen ("file1.txt", "w");
if (!f)
{
    /* обработка ошибки открытия файла */
}
/* работа с файлом */
if (fclose(f))
{
    /* обработка ошибки записи в файл */
}
```

Хотя при завершении работы программы Си закрывает все открытые файлы, даже если программа завершается немедленно по окончании работы с файлом, опускание вызова `fclose` — плохой стиль и источник потенциальных ошибок при последующей модификации программы.

Следует обратить внимание на то, что открытие существующего файла для записи (`w`) приводит к уничтожению содержимого файла, в общем случае эта операция необратима, поэтому использовать ее следует с большой осторожностью: можно затереть важные данные или — при наличии административного доступа — системные файлы. Однако, не следует замещать операцию записи операцией добавления (`a`) за исключением случаев, когда действительно требуется добавление в конец файла. В связи с этим нелишним будет напоминание *регулярно делать аварийные копии данных.*

Вопросы для самопроверки.

1. *Зачем нужно открытие файла?*
2. *Какой функцией осуществляется открытие файла?*
3. *Зачем нужно закрытие файла?*
4. *Какой функцией осуществляется закрытие файла?*
5. *Что такое тип данных `FILE`?*
6. *Какие есть режимы открытия файла?*
7. *Что такое константа `EOF`?*
8. *С какими рисками сопряжено открытие файла для записи?*
9. *В каких режимах открываются стандартные потоки ввода и вывода?*

§5.3. Текстовые файлы: форматированный ввод и вывод

По сути работа с текстовыми файлами осуществляется аналогично вводу со стандартного ввода и выводу на стандартный поток вывода, обычно последовательно, байт за байтом, хотя существуют функции перемещения счетчика номера позиции файла.

Для форматированного ввода и вывода используются функции

```
int fprintf (FILE *stream, const char *format, ... );
int fscanf  (FILE *stream, const char *format, ... );
```

аналогичные функциям `printf`/`scanf`: строка формата и неограниченное количество остальных аргументов имеют точно тот же смысл, только первым параметром должен быть указан открытый файл.

Аналогично, функция `fprintf` возвращает количество выведенных байт, `fscanf` — количество введенных полей или нулевое (или даже отрицательное) значение, если ничего введено не было. Кстати, при вводе с клавиатуры (использовании ее в качестве стандартного ввода) конец файла вводится с помощью комбинации `Ctrl-D` (Linux) и `Ctrl-Z` (Windows).

Если `fscanf` вернул число, меньшее чем ожидаемое количество полей, это может свидетельствовать как о нарушении формата, так и о достижении конца файла. Чтобы разделить эти два случая можно использовать функцию

```
int feof (FILE *stream);
```

возвращающую 1, если достигнут конец файла и 0 в противном случае.

Пример:

```
FILE *f;
int i;

/* ... */

while (fscanf(f, "%i", &i) == 1)
    printf("%i\n", i);

if (feof(f))
    printf("EOF reached OK\n");
else
    printf("Bad file format\n");
```

Вопросы для самопроверки.

1. *Как перенаправить форматированный ввод и вывод в файл?*
2. *Как корректно проверить достижение конца файла при форматированном вводе?*
3. *Что делает функция `feof`?*

§5.4. Стандартные потоки и использование stderr

В `stdio.h` определены константы `stdin`, `stdout` и `stderr`, которые представляют собой указатели на `FILE`, соответствующие стандартным потокам ввода, вывода и сообщений об ошибках соответственно.

Поэтому "`printf (`" эквивалентно "`fprintf (stdout, `", "`scanf (`" эквивалентно "`fscanf (stdin, `", а вместо `gets` следует использовать `fgets`, указывая `stdin` в качестве файла (хотя у этих функций и есть некоторые различия, описанные в 5.5.

Сообщения о любых ошибках, обнаруженных программой во время ее работы, следует выводить в поток `stderr`*. Также следует завершать программу с ненулевым кодом возврата. Сделать это можно с помощью* `return` *в* `main`*, из любого места программы можно завершить ее с помощью функции*

```
void exit (int status);
```

определенной в `<stdlib.h>`

Например,

```
FILE *f;
int i;

if (!(f = fopen ("file.txt", "r")))
{
    fprintf(stderr, "Could not open file\n");
    exit(2);
}

/* ... */

while (fscanf(f, "%i", &i) == 1)
    printf("%i\n", i);

if (!feof(f))
{
    fprintf(stderr, "Bad file format\n");
    exit(1)
}

printf("EOF reached OK\n");
/* ... */
exit(0);
```

Файловые и другие операции ввода-вывода — источник наибольшего количества ошибок, возникающих в процессе работы программы. Ошибки ввода-вывода могут происходить как по вине пользователя (указание неверного имени файла, запрет доступа и т.п.), так и по вине оборудования (отказ техники), а не только программиста (нарушение заявленного при открытии способа доступа к файлу и др.). Если ошибок программирования теоретически можно избежать написанием безошибочного кода, то первые два источника находятся вне контроля программиста.

При работе с файлами следует особо внимательно следить за возникающими ошибками, так как вероятность их возникновения очень высока, но при этом они не прерывают программу, и, в случае их необработки программой, приводят к молчаливому появлению неверных результатов работы.

Каждая функция стандартной библиотеки, сталкивающаяся с ошибкой, устанавливает код случившейся ошибки в глобальной целочисленной переменной `errno`, определенной в `errno.h`, однако эти коды системнозависимы. С помощью функции `perror` (`stdio.h`) можно вывести также системнозависимое сообщение о случившейся ошибке

```
void perror (const char *s);
```

где `s` — выводимая строка, за которой последует стандартное сообщение об ошибке. Это решение редко подходит при написании программ с качественным индивидуальным пользовательским интерфейсом, но может быть использовано при отладке и при написании программ со стандартным системным интерфейсом командной строки.

Поток `stderr`, разумеется, следует использоваться и для вывода иных возникших в программе ошибок.

Перенаправление потока вывода сообщений об ошибках средствами командной строки ОС также возможно, для этого следует указать

```
$ prg 2>file
```

комбинация 2> перенаправляет поток ошибок (0 — поток ввода, 1 — поток вывода).

Для того, чтобы перенаправить поток об ошибках и стандартный вывод в один и тот же файл следует указывать

```
$ prg >file 2>&1
```

`2>&1` перенаправляет второй поток в первый.

Вопросы для самопроверки.

1. *Каким файловым переменным соответствуют стандартные потоки ввода, вывода и сообщений об ошибках?*
2. *Какой тип данных у `stdin`? `stdout`? `stderr`?*
3. *Зачем нужен отдельный поток сообщений об ошибках?*
4. *Как перенаправить поток сообщений об ошибках и поток вывода в разные файлы?*
5. *Как перенаправить поток сообщений об ошибках и поток вывода в один файл?*
6. *Что делает функция `perror`?*
7. *Что такое `errno`?*
8. *Какие есть источники ошибок при файловых операциях?*

§5.5. Символьный ввод и вывод

Символьный ввод и вывод из файлов осуществляется с помощью функций

```
int fgetc (FILE *stream);
int fputc (int c, FILE *stream);
```

Следует отметить, что функция `fgetc` возвращает не `char`, а `int`. Дело в том, что кроме 256 возможных введенных символов, `fgetc` может вернуть сообщение об ошибке или достижении конца файла, которое нельзя уместить в `char`. Такой результат соответствует символической константе `EOF` (также определенной в `stdio.h`).

Функция `fputc` возвращает выведенный символ в случае успеха или `EOF` в случае провала, что аналогично объясняет использования `int`, а не `char` в качестве параметра.

Пример

```
FILE *f1, *f2;
int c;

if (!(f1 = fopen ("file_in.txt", "r")))
{
    fprintf(stderr, "Could not open input file\n");
    exit(4);
}
if (!(f2 = fopen ("file_out.txt", "w")))
{
    fprintf(stderr, "Could not open output file\n");
    fclose(f1);
    exit(3);
}



while ( ( c = fgetc(f1) ) != EOF )
    if (fputc(c, f2) == EOF)
    {
        fprintf(stderr, "Write error\n");
        fclose(f1);
        fclose(f2);
        exit(1);
    }
```

```
if (!feof(f1))
{
    fprintf(stderr, "Read␣error\n");
    fclose(f1);
    fclose(f2);
    exit(2)
}

fclose(f1);
if (fclose(f2))
{
    fprintf(stderr, "Write␣error\n");
    exit(1);
}

printf("Copy␣OK\n");
exit(0);
```

Если бы переменная `c` была объявлена как `char`, а не как `int`, то, поскольку обычно `EOF` определено как -1 (`int`), то чтение бы завершилось при получении символа с кодом 255, что соответствует -1 (в типе `unsigned char` на компьютерах с обратным дополнительным кодом). Критична именно проверка ошибки при закрытии файла, т.к. в этом случае возможна потеря буферизированных данных, в то время как в случае когда ошибка закрытия файла произошла после того, как конец файла достигнут, файл все равно был прочитан полностью.

Данные функции могут быть использованы как для доступа к текстовым, так и для доступа к двоичным файлам с очевидной разницей в поведении.

Вопросы для самопроверки.

1. *Какая функции осуществляют ввод символа (байта) из файла?*
2. *Какая функции осуществляют вывод символа (байта) файл?*
3. *Почему функция `fgetc` возвращает `int`, а `char`?*
4. *Может ли константа `EOF` иметь значение 0? 1?*
5. *Какое значение может быть у константы `EOF`?*
6. *Как корректно разделить достижение конца файла и ошибку чтения при символьном вводе?*

§5.6. Строковый ввод и вывод

Функция

```
int fputs (const char *str, FILE *stream);
```

выводят в файл строку целиком, но в отличие от `puts`, не выводит символ переноса строки. Также в файл не выводится терминирующий символ `'\0'`. В случае успеха функция возвращает 0, в случае неудачи — `EOF`.

Функция

```
char* fgets (char *str, int num, FILE *stream);
```

вводит строку из файла. Функция всегда создает нуль-терминированную строку, вводит не более чем `num-1` символов строки. Функция `fgets` ограничивает число вводимых байт, в отличие от "запрещенной" функции `gets` — размер буфера `str` должен быть не менее чем `num` байтов, в противном случае буфер может быть переполнен.

Символ перевода в строки включается в `str`, таким образом, наличие данного символа в конце введенной строки свидетельствует о том, что файловая строка введена полностью, отсутствие — о том, что исчерпан размер буфера или достигнут конец файла. Разделить последние два случая можно используя функцию `feof`.

В случае успеха функция возвращает `str`, если конец файла достигнут до начала чтения функция возвращает `NULL`, буфер `str` не меняется. Если в процессе чтения возникла ошибка, функция также возвращает `NULL` и устанавливает `errno`, но содержимое буфера `str` может быть изменено непредсказуемо.

Систематизировать это можно в следующей таблице:

ситуация	возвр. зн.	`str`	`feof`	`errno`
чтение до конца строки	`str`	содержит `'\n'`	ложь	0
нет места в буфере	`str`	не содержит `'\n'`	ложь	0
при чтении достигнут конец файла	`str`	не содержит `'\n'`	истина	0
до чтения достигнут конец файла	`NULL`	не изменяется	истина	0
ошибка чтения	`NULL`	непредсказуем	ложь	код ошибки

Конец файла может быть достигнут при чтении строки, если в последней строке файла нет перевода строки. В противном случае будет достигнут конец файла до начала чтения.

Функции строкового ввода и вывода следует использовать только с текстовыми файлами (в двоичном файле может встретиться символ `'\0'` — следует ли помещать его в строку?).

Говоря о строковом и символьном доступе к файлам, следует отметить очевидный факт, что добавление данных возможно только в конец файла, вставка и удаление из середины (в т.ч. добавление строк, добавление в строку, удаление строк, удаление части символов и т.п.) напрямую невозможны, требуется перезапись части файла. Хотя с помощью описанных в параграфе 5.7 функций перемещения счетчика позиции в файле можно заменять отдельные байты в середине файла.

Вопросы для самопроверки.

1. *Какие функции осуществляют ввод и вывод строк?*
2. *Чем отличаются функции `fgets` и `gets`?*
3. *Чем отличаются функции `fputs` и `puts`?*

§5.7. Двоичный доступ к файлам

Двоичные файлы предназначены для хранения произвольных данных, без ориентирования на переносы строк. Некоторые системы (Linux) не различают двоичный и текстовый доступ, другие, как было отмечено, проводят различия при выводе символа переноса строки. Также следует учесть, что при текстовом доступе некоторые управляющие символы некоторые ОС (в DOS/Windows это символ с десятичным кодом 26) могут воспринять как конец файла, даже если реальная длина файла больше.

Символьный (побайтовый) доступ будет работать с двоичными файлами корректно. Также для двоичного доступа имеются следующие функции ввода и вывода блоков данных

```
size_t fwrite (const void *ptr, size_t size, size_t count, FILE *stream);
size_t fread  (      void *ptr, size_t size, size_t count, FILE *stream);
```

Их параметрами являются указатель `ptr` на буфер (выводимый или вводимый массив), `size` — размер вводимого/выводимого элемента (элемента массива), `count` — число вводимых/выводимых элементов (размер массива), `stream` — используемый файл.

Функции вводят (выводят) блоки памяти (массивы) в неизменном виде. Возвращают число выведенных/введенных элементов, что позволяет проверить на наличие ошибки. Функция `fread` устанавливает `feof` при достижении конца файла.

Например,

```
size_t n, m;
file *fr, *fw;
/* ... */
int    a[n];
double b[m]
/* ... */

fread  (a, sizeof(int   ), n, fr);

fwrite (b, sizeof(double), m, fw);
```

Эти функции также осуществляют последовательный доступ к файлам.

При двоичном доступе следует учесть тот факт, что это прямой ввод и вывод данных, хранящихся в оперативной памяти. На разных платформах, разных версиях языка, даже в разных ОС на одном компьютере данные одного и того же типа могут представляться по разному, что делает двоичные файлы сильно непереносимыми. Частичное решение этой проблемы описано в приложении B.3.

Следует также обратить внимание на функции перемещения указателя на позицию в файле.

```
long int ftell (FILE *stream);
int fseek (FILE *stream, long int offset, int origin);
void rewind (FILE *stream);
```

Функция `ftell` возвращает текущую позицию в байтах от начала файла, функция `fseek` устанавливает позицию в `offset` байтов относительно origin. Для задания `origin` можно использовать символическое константы: `SEEK_SET` — начало файла, `SEEK_CUR` — текущая позиция, `SEEK_END` — конец файла. Функция возвращает 0 в случае успеха и ненулевое значение в случае ошибки. Функция `rewind` устанавливает позицию на начало файла, кроме того, она очищает флаг достижения конца файла, если он был установлен ранее.

Перемещение по файлу бывает необходимо в т.ч. при доступе к файлу для чтения-записи. Следует, однако, еще раз вспомнить, что перемещение достаточно медленная процедура, в том числе потому, что нарушает буферизацию ввода и вывода.

Вопросы для самопроверки.

1. *Какие функции осуществляют файловый ввод и вывод блоков памяти?*
2. *Какие функции читают и перемещают указатель позиции файла?*
3. *Почему использовать файл для произвольного доступа к данным не рекомендуется?*
4. *Какие сложности имеются при работе с двоичными файлами?*

§5.8. Форматированный ввод и вывод в строку и обратно

Функции `printf` и `scanf` позволяют осуществлять форматированный вывод и ввод, фактически преобразуя строки символов в данные числовых типов и обратно.

Функции `sprintf` и `sscanf` позволяют записывать результат в строковый буфер и прочитывать из строкового буфера форматированные данные:

```
int sprintf (      char *str, const char *format, ...);
int sscanf  (const char *str, const char *format, ...);
```

Первым параметром функции служит строка, с которой будет производиться операция, вторым — строка формата, далее следует перечень полей. Смысл полей и строки формата полностью аналогичен смыслу параметров функций `printf` и `scanf`, возвращаемое значение — тоже, для `sscanf` это число корректно введенных полей, для `sprintf` — число выведенных символов.

Например, с помощью функции `sscanf` можно прочитать число, представленное в строке (например, пришедшее в качестве параметра командной строки, при прочтении текстового файла или иным способом) в числовой тип данных:

```
char number[10];
int i;
/* ... */
if (sscanf(number, "%i", &i) == 1)
    printf ("Число %i", i);
else
    printf ("Ошибка формата\n");
```

Функция `sprintf` выполняет, например, обратную задачу,

```
char buffer[21];
int i;
/* ... */
sprintf(buffer, "%i", i);
```

однако с ней следует соблюдать особую осторожность: она не является безопасной, так как не проверяет количество выводимых символов. В данном примере проблема вряд ли возникнет, так как на современных платформах `int` не может быть больше 64 битов, то для его записи будет достаточно 20 символов плюс 1 символ на `'\0'`. Однако, если архитектура будущего или экзотический компилятор будет использовать 128-битные целые числа и все еще оперировать с форматом `%i` для них, то исполнение данного кода на таких системах может привести к переполнению строкового буфера со всеми вытекающими последствиями. Гораздо вероятнее неприятная ситуация возникнет при использовании формата `%s` и передаче длинной строки.

Стандарт C99 предусматривает безопасную функцию

```
int snprintf (char *str, size_t n, const char * format, ... );
```

ограничивающую размер выходного строкового буфера в `n` байт, в худшем случае последний из них будет символом с кодом 0, даже если формат еще не закончился.

Вопросы для самопроверки.

1. *С помощью каких функций можно проводить преобразование числовых данных в строковые и наоборот?*

2. *Почему функция* `sprinf` *небезопасна?*

3. *Имеет ли функция* `sprinf` *безопасный аналог?*

§5.9. Параметры командной строки

При запуске программы операционная система передает ей параметры, задаваемые пользователем или автоматически. При запуске программ с графическим интерфейсом из графической оболочки операционных систем такой подход используется, например, при передаче имени файла, при открытии файла с помощью какой-то программы.

Использование командной строки позволяет задать любое количество параметров (как правило, присутствует системно- и оболочкозаписимое ограничение на их количество и суммарную длину командной строки). Этими параметрами являются строки, разделенные пробелами.

Как правило, интегрированные окружения разработки позволяют конфигурировать параметры, с которыми будет запускаться программа при ее тестировании.

Для того, чтобы использовать данные параметры в Си-программе необходимо "превратить" их в параметры функции `main`: функция `main` должна иметь прототип

```
int main (int argc, char *argv[]);
```

или — что то же самое —

```
int main (int argc, char **argv);
```

(разумеется, вместо `argc` и `argv` можно использовать любые идентификаторы, но это не принято).

Смысл параметров следующий: `argc` — число параметров, `argv` — массив строк-параметров. При этом `argv[0]` будет собственно именем программы, той командой, которая была использована для ее запуска (так как указано пользователем или вызвавшей программой — полный или относительный путь, или просто имя программы, если поиск осуществлялся с учетом настроек системы на поиск программы по имени в определенных каталогах), `argv[1]` — первый параметр и т.д., до последнего параметра `argv[argc-1]`.

Простая программа

```
#include <stdio.h>

int main (int argc, char **argv)
{
    int i;
    for (i = 0; i < argc; i++)
    {
        printf("%i:␣\"%s\"\n", i, argv[i]);
    }
    return 0;
}
```

выводит на экран все параметры программы с их номерами. Пусть программа называется `show_args`, следующие примеры ее запуска (приведены для Linux, в Windows ситуация аналогична) иллюстрируют преобразование параметров в элементы массива:

```
$ ./show_args
0: "./show_args"

$ ./show_args 123
0: "./show_args"
1: "123"
```

```
$ ./show_args arg_1 "long␣arg␣2"
0: "./show_args"
1: "arg_1"
2: "long␣arg␣2"

$ ./show_args arg_1 "long␣arg␣2" 'another␣long␣arg'
0: "./show_args"
1: "arg_1"
2: "long␣arg␣2"
3: "another␣long␣arg"

$ /home/user/my_prg/show_args arg_1 "long␣arg␣2" 'another␣long␣arg'
0: /home/user/my_prg/show_args"
1:␣"arg_1"
2:␣"long arg 2"
3:␣"another long arg"
```

Обратите внимание, что заключение в кавычки или апострофы позволяет объединять несколько разделенных пробелами слов в один параметр, при этом сами кавычки в параметр не включаются. Параметры являются строками, объявить параметры числами автоматически невозможно, но можно использовать инструментарий, описанный в параграфе 5.8.

Вопросы для самопроверки.

1. *Что такое параметры командной строки?*
2. *Как разделяются параметры командой стоки?*
3. *Как используются параметры командной строки?*
4. *Как прочитать параметры командой строки в функции `main`?*
5. *Какие параметры может иметь функция `main`?*

Глава 6. Парадигмы и методология программирования

Данная глава в большей степени посвящена методологии программирования — принципам и правилам, позволяющим написать более качественный программный код, снизить количество ошибок в программах.

§6.1. Ошибки в программах

К сожалению, в коде программ нередко встречаются различные ошибки.

По причинам возникновения ошибки разделяют на

ошибки программирования — ошибки, возникшие из-за неправильного написания кода программы, задача программиста — свести количество таких ошибок к нулю;

ошибки пользователя — некорректные действия пользователя при работе с программой (ввод данных в некорректном формате и т.п.), задача программиста — корректно обработать такие ошибки в программе, сообщить пользователю о возникшей ошибке и способах ее устранения;

ошибки оборудования — ситуации, когда выполнение действия (обычно ввода-вывода) невозможно по причине аппаратного отказа оборудования или системными ограничениями на доступ к оборудованию, задача программиста аналогична.

По времени обнаружения ошибки разделяют на

ошибки компиляции (трансляции) — ошибки, не позволяющие компилятору (транслятору) сгенерировать машинный код из исходного, чаще всего синтаксические ошибки: исправимы легче всего;

ошибки времени исполнения программы — ошибки, возникающие в процессе исполнения программы (семантические ошибки чаще всего обнаруживаются именно на стадии исполнения): выявление таких ошибок возможно при помощи пробных (тестовых) запусков и с использованием отладчика.

Заметим, что **предупреждения компилятора** позволяют на стадии компиляции выявить ошибки, которые иначе обнаружились бы только во время исполнения программы. Предупреждения компилятора являются примером инструмента **статического анализа кода** — группы средств и методов, позволяющих находить семантические ошибки в программе без ее запуска. Разумеется, ни один компилятор и ни одно такое средство не способно найти все такие ошибки уже хотя бы в силу алгоритмической неразрешимости данной проблемы.

§6.2. Критика оператора перехода

Операторы перехода — `goto` и аналогичные ему в других языках высокого уровня — являются объектом критики:

- код с переходами трудно форматировать, сделать наглядным с помощью отступов;
- оператор перехода мешает оптимизации кода компиляторами;
- переход может привести к пропуску инициализации данных и переменных, например при переходе в середину тела цикла и т.п., с появлением непредсказуемого поведения;
- изобилие операторов перехода создает так называемый **спагетти-код** (извилистый и запутанный код — данный термин является почти официальным; также используется термин кенгуру-код из-за множество прыжков: в ассемблере оператор перехода называется jump) — такой код неудобно читать, по тексту такого кода практически невозможно понять порядок исполнения и взаимозависимость фрагментов, поиск ошибок становится крайне затруднительным.

Создатель структурного программирования Эдсгер Дейкстра сказал, что “качество программного кода обратно пропорционально количеству операторов goto в нем”. Существует шутливая переформулировка этого утверждения: “зарплата программиста обратно пропорциональна количеству операторов goto в его коде”. Из всякого правила существуют исключения, в некоторых случаях код с оператором перехода может, наоборот, оказаться более качественным, но по умолчанию от него лучше отказаться в пользу условий и циклов.

Вопросы для самопроверки.

1. *Какой критике подвергается оператор перехода?*
2. *Какова альтернатива оператору перехода?*

§6.3. Парадигма структурного программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль и методику написания компьютерных программ.

Одна из основных целей создания парадигм программирования и следования парадигме при написании программ — повышение качества кода и труда программистов, в том числе уменьшение затрат на разработку, отладку и сопровождение программ.

Каждый современный язык программирования высокого уровня реализует хотя бы одну парадигму, каждую конкретную парадигму поддерживает значительное количество языков. Изучение парадигм даже более широкий и фундаментальный процесс, чем изучение конкретных языков, так как переход с одного языка на другой в рамках одной парадигмы проще, чем переход с одной парадигмы на другую.

Приведем цитату Роберта Флойда: “если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм, то совершенствование искусства отдельного программиста требует, чтобы он расширял свой репертуар парадигм.”

Структурное программирование — парадигма программирования, в основе которой лежит принцип представления программы с помощью структур управления и блоков кода.

Блок кода — совокупность программных инструкций (операторов и деклараторов).

Блоки могут быть вложенными, могут быть пустыми. Блоки кода служат, с одной стороны, для ограничения области видимости идентификаторов, с другой — для упрощения восприятия кода, так как один блок синтаксически воспринимается как одна программная инструкция (один оператор). В языке Си блоку кода соответствует составной оператор — группа операторов, заключенная в фигурные (операторные) скобки.

Структуры управления — инструкции, регулирующие порядок исполнения кода программы.

Порядок исполнения кода программы, может регулироваться в том числе с помощью оператора перехода, однако *методология структурного программирования предполагает отказ от оператора перехода*, базируясь на следующей теореме (приводится без доказательства).

Теорема Бёма–Якопини, известная также как теорема о структурном программировании: любой исполняемый алгоритм может быть преобразован к виду, когда ход его выполнения определяется только при помощи трех структур управления: последовательной, ветвлений и циклов. (Теорема, однако, не гарантирует сохранение производительности при таком преобразовании.)

Таким образом, чистое структурное программирование основано на использовании только этих трех управляющих структур, однако такой подход не применяется, так как написание даже относительно небольших программ требует их разбиение на подпрограммы (функции). В связи с этим выделяют методологию **процедурного программирования**, в основе которой лежит разделение программы на подпрограммы, каждая из которых решает определенную задачу, но по сути такая методология включается и в структурное программирование.

Классические принципы парадигмы следующие.

1. Любая программа строится из трех базовых управляющих структур: последовательность, ветвление, цикл. Следует отказаться от использования оператора безусловного перехода. Структуры управления могут быть вложены друг в друга произвольным образом.
2. Повторяющиеся фрагменты программы следует оформлять в виде подпрограмм (функций). Также в виде подпрограмм можно оформить логически целостные фрагменты программы, даже если они не повторяются.
3. Каждую логически законченную группу инструкций следует оформить как блок. Блоки являются основой структурного программирования.
4. Все перечисленные конструкции должны иметь один вход и один выход. (Один вход в подпрограмму, один выход из подпрограммы, один вход и выход в тело цикла и т.д. — т.е. отказ от `break`, и множественных `return`, а не только `goto`, но в Си на практике часто используются множественные `return` с некоторыми ограничениями.)
5. Разработка программы производится методом “сверху вниз”, то есть сначала пишется основная программа, вместо подпрограмм указываются заглушки, потом пишется исходный код заглушек, где используются новые заглушки и т.д.

Вложенность блоков кода и структур управления в Си выражается в частности в том, что составной оператор также является оператором, и везде где аргументом оператора может выступать любой оператор, может быть использован и составной оператор (в качестве тела цикла, в условном операторе и т.п.), в то время как условные операторы и операторы цикла могут быть частью составного оператора, как один из операторов списка.

Последний принцип заслуживает пристального внимания, он позволяет четко определить круг необходимых функций. Действительно, глядя на большую задачу, можно сразу разбить ее на ряд подзадач, т.е. на группу шагов, которые нужно выполнить для ее решения. Каждая из подзадач решается тем же образом (рекурсивно!) пока не сведется к элементарным или уже решенным.

Противоположный подход потребовал бы сначала разработки мельчайших шагов до того, как станет понятно, какие из них понадобятся и понадобятся ли вообще.

Операторы `break` и `continue` тоже являются операторами перехода и нарушают структуру программы — их использование не рекомендуется по крайней мере до тех пор, пока отказ от их использования не ведет к ухудшению производительности и качеству кода (на практике проблема случается реже, чем в случае с оператором перехода по метке). Это не относится к всегда необходимому оператору `break` внутри `switch`.

Вопросы для самопроверки.

1. *Что такое парадигма программирования?*
2. *Зачем нужны парадигмы программирования?*
3. *Что такое блок кода?*
4. *Что такое структура управления?*
5. *Что такое структурное программирование?*
6. *В чем состоит теорема о структурном программировании?*

§6.4. Процедурное программирование

Процедурное программирование — методология программирования, основанная на возможности сгруппировать участки кода в подпрограммы (в терминах Си и данного курса — функций), которые можно вызвать из программы и других подпрограмм. Не следует путать процедурное и функциональное программирование, последнее не имеет никакого отношения к функциям в том смысле, в котором это слово употребляется в языке Си.

Заметим также, что не следует путать **линейные программы** — программы, алгоритмы которых не предусматривают ветвлений и циклов с **линейным программированием** — программированием задач линейной алгебры.

Процедурное программирование преследует главным образом следующие цели:

- повторное использование кода (вместо того, чтобы писать почти тождественный код повторно или копировать его, достаточно вызвать функцию);
- исключение повторяющихся участков кода (в повторяющихся участках кода может возникнуть повторяющаяся ошибка, ее технически трудно исправлять, аналогичная проблема возникает при необходимости модифицировать или улучшить данный код);
- уменьшение количества ошибок и облегчение их поиска (легче правильно написать, осознать, протестировать, отладить и проверить короткий код каждой функции отдельно, чем всей неразделенной на части программы целиком);
- разделение труда программистов (разные функции одной программы могут создаваться разными людьми);
- сокрытие деталей реализации и повышение уровня абстракции (для работы с функцией достаточно знать, как она вызывается и что делает, но не важно как именно она это делает — всякая функция есть черный ящик);
- облегчение сопровождения программ (модификация деталей реализации — внутренности черного ящика — при сохранении внешности черного ящика не должна сказываться на работоспособности остальных функций).

При написании функций следует учитывать эти цели.

Вопросы для самопроверки.

1. *Что такое процедурное программирование?*
2. *Какие цели преследует процедурное программирование?*

§6.5. Модульное программирование

Модуль — часть программы, реализующая определенную функциональность.

По сути под модулем в программировании понимают совокупность подпрограмм (функций), объявлений (констант и т.п.), типов данных (пользовательских), и других программных элементов, предоставляющих программисту некоторый *интерфейс*, посредством которого можно решать определенный круг задач. (В соответствии с ранее данным определением интерфейс — средство взаимодействия программного компонента с другим: двух модулей между собой, функции и модуля, и т.д., фактически некоторый уровень абстракции.)

Всякий модуль представляет собой черный ящик: состоит из интерфейса и **реализации**, что соответствует внешней и внутренней стороне ящика. Детали реализации в интерфейсе не видны, поэтому программисту, использующему модуль, можно в них не вникать.

Программа может состоять из нескольких модулей, что позволяет фактически оформить разделение задачи на подзадачи, протестировать отладить каждый модуль отдельно, разделить труд программистов (одному программисту достаточно знать интерфейс модуля, созданного другим программистом), облегчить модификацию программы (если изменение затрагивает только реализацию одного модуля, но не его интерфейс, не требуется затрагивать остальные модули программы), повторно использовать ранее написанный код (реализованный функционал): во многих языках программирования высокого уровня каждый модуль оформляется в виде отдельного файла, многие языки программирования предоставляют стандартные модули. Фактически, модульное программирование решает те же цели, что и процедурное, но на другом уровне.

Понятия “модуль” и “библиотека” довольно близкие, но между ними есть концептуальное и реализационное отличия.

И про программу, и про библиотеку можно сказать, что она состоит из модулей, каждый модуль является элементом программы или библиотеки. Не вполне корректно говорить, что программа состоит из библиотек, программа использует библиотеки. Вульгарно можно представлять себе, что функции объединяются в модули, а модули объединяются в библиотеки и программы. Организационно библиотека ближе к программе, чем к модулю, только в ней нет главной функции. Библиотеки не привязаны жестко к языку программирования и часто распространяются как самостоятельный программный продукт. Модули, если и распространяются отдельно, то обычно как расширение языка, поддерживающего подключение в программу модулей.

И модуль, и библиотека представляют собой коллекцию функциональности, но именно в приложении к модулю концептуально применяется понятие сокрытия деталей реализации, как о способе абстрагирования в рамках одной задачи (здесь следует вспомнить, что сокрытие деталей реализации хотя и может использоваться

как один из механизмов защиты данных от чужих глаз, но основная концепция состоит не в этом, а именно в абстрагировании).

Библиотека всегда реализуется в виде отдельного файла, модуль — необязательно. Программа, использующая библиотеку, может как включить ее в свой исполняемый код, так и использовать отдельный файл (на практике второй способ используется чаще), модуль должен быть подключен к коду программы.

Модульное программирование — концепция организации программы, при которой функциональность программы разделяется на независимые модули, каждый из которых отвечает за одну сторону функциональности.

Обычно модульное, как и процедурное, программирование не выделяют в отдельную парадигму, хотя такой термин используется. Также говорят о парадигме процедурно-модульного программирования.

Строго говоря, в языке Си нет поддержки модулей. Но поддержку модульного программирования можно организовать посредством **раздельной компиляции**.

Каждый `.c` файл представляет собой **единицу трансляции**, то есть минимальный блок исходного кода, который можно скомпилировать. Компиляция `.c` файла производится в **объектный** модуль (`.o` файл).

В объектном модуле содержится исполняемый код функций, информация об именах функций, но вызов функций, находящихся в других объектных модулях программы — как и в библиотеках — отсутствует, указывается только, что будет вызвана соответствующая функция (или использована глобальная переменная). В контексте библиотек и объектных модулей идентификаторы, ассоциированные с объектами (переменными и функциями) других модулей и библиотек называются *символами*. Заметим, что эти символы могут не совпадать с идентификаторами, использованными для тех же объектов в программе.

Различают **статическое** и **динамическое** связывание (компоновка). В первом случае связывание происходит на стадии компоновки: объединяемые модули и библиотеки формируют единый двоичный файл программы или библиотеки, вычисляются адреса всех объектов. Динамическое связывание происходит в момент запуска программы — программа и библиотеки объединяются в оперативной памяти, адреса объектов вычисляются операционной системой. Модули всегда компонуются статически. При этом возможно создание статических библиотек (расширение `.a`, `.ar`, `.lib` и др.) и динамических библиотек (`.so`/`.dll`, расширение зависит от операционной системы). Исполняемый файл программы (файл без расширения или с расширением `.exe`, расширение зависит от операционной системы) может быть скомпонован с библиотекой статически (при компоновке используется статическая библиотека, она не требуется при запуске программы) и динамически (при компоновке используется динамическая библиотека, она требуется для запуска программы). Библиотека может также использовать другие библиотеки, быть скомпонована с ней динамически (если она сама компонуется динамически) или статически (в любом случае).

Каждый файл исходного кода транслируется отдельно. Для того, чтобы функции из разных файлов могли вызывать друг друга — видели прототип функций другой единицы компиляции — каждый `.c`-файл необходимо снабжать заголовочным файлом (`.h`-файлом). Строго говоря, не требуется точного взаимно-однозначного соответствия `.c`-файлов и `.h`-файлов, они даже могут иметь разные имена, но в простейших случаях обычно делают именно так.

С точки зрения методологии модульного программирования каждому модулю соответствует заголовочный файл. Заголовочный файл содержит интерфейс модуля. В модуль также входит один или несколько файлов исходного кода. Каждый файл исходного кода (транслируется в объектный модуль отдельно) содержит детали реализации модуля.

Раздельная компиляция также обеспечивает экономию времени на сборку программы: при изменении одного файла исходного кода не требуется компилировать остальные файлы исходного кода. При изменении одного заголовочного файла достаточно компилировать только те файлы исходного кода, к которым подключается данный заголовок.

На рис. 6.1 показана диаграмма процесса раздельной компиляции и компоновки

При разделении программы на файлы следует учитывать следующие соглашения и требования.

- Формально заголовочный файл с точки зрения Си — такой же файл исходного кода, который просто вставляется препроцессором в `.c` файл, т.е. в нем может размещаться любой верный с точки зрения синтаксиса Си код. Однако в заголовочном файле не должны располагаться определения (тела функций и глобальные переменные). В некоторых случаях программа будет собираться нормально, но если заголовочный файл подключается к нескольким файлам исходного кода, то в двух и более объектных модулях окажется код одной и той же функции (дважды определенный символ), что приведет к ошибке компоновки (даже если код функций на самом деле один и тот же).

  Таким образом, в заголовочный файл включаются только объявления: прототипы функций, пользовательские типы данных, символические константы и т.п.

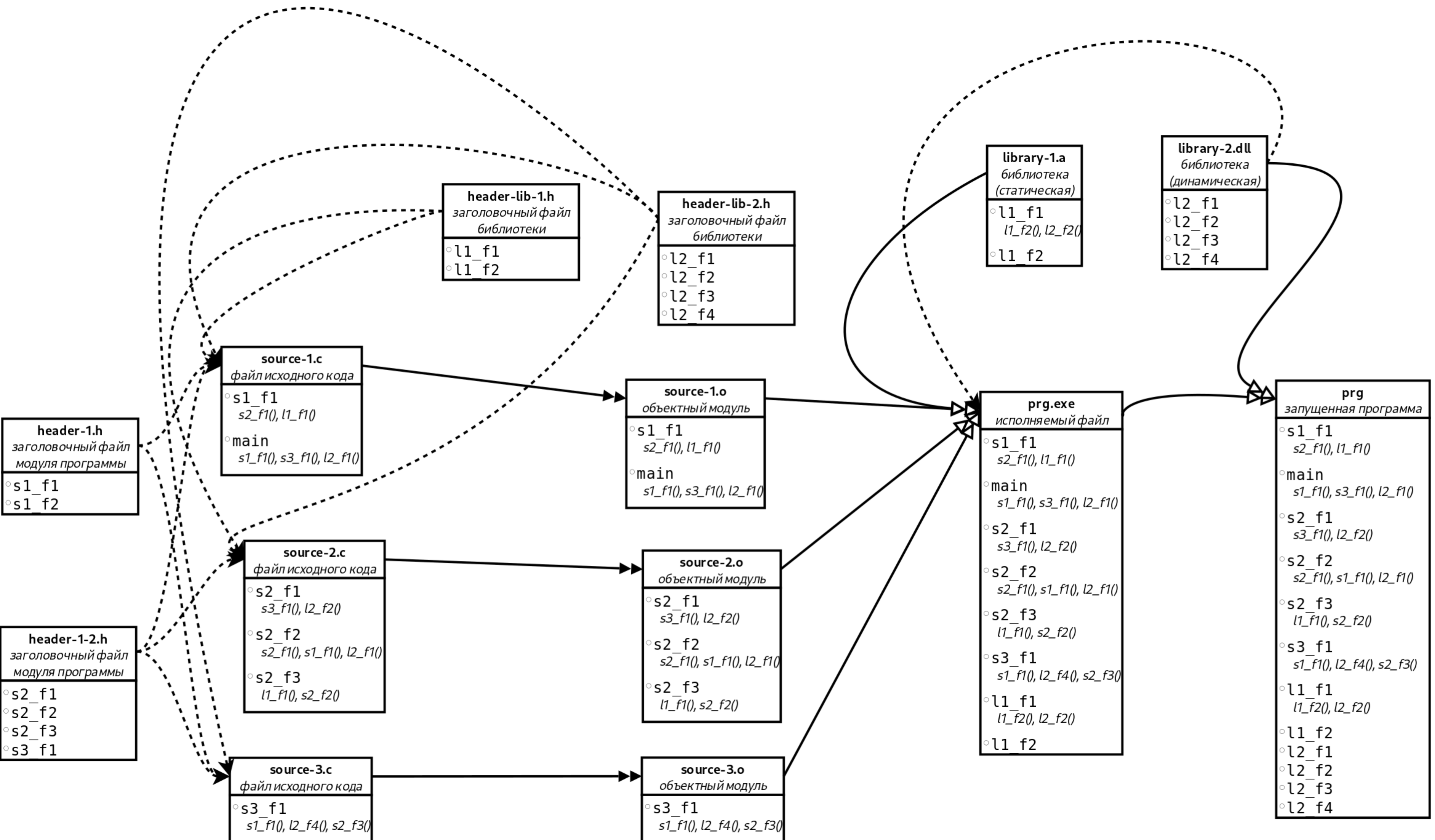


Рис. 6.1: Раздельная компиляция, двойными сплошными стрелками показаны процессы компиляции (раздельный), двойными полыми — компоновка, пунктирными — подключение заголовочных файлов (`#include` в файлах исходного кода) и библиотек (опция компоновщика при компоновке программы). В списках к `.c`-файлам, `.o`-файлам и исполняемым файлам даны функции, чьи тела там реализованы и функции, которые они вызывают, в заголовочных файлах перечислены указанные в них прототипы функций. Сплошными стрелками показаны вычисленные адреса вызовов (выставляются компоновщикам за исключением вызовов внутри единицы компиляции ранее определенных функций. Разница между заголовочными файлами модуля программы и библиотек только концептуальная, с точки зрения языка Си это одна и та же единица. При компиляции библиотек также подключаются заголовочные файлы, при компоновке — другие библиотеки.

- К ошибке компоновки — неопределенной ссылке — приведет и отсутствие тела “обещанной” функции (т.е. когда прототип в заголовочном файле есть, а кода нет).
- В одном и том же файле исходного кода один и тот же заголовочный файл может подключиться дважды (необязательно по ошибке, например, если файл `"a.c"` подключает файлы `"a.h"` и `"b.h"`, а `"b.h"` подключает `"a.h"`, то при компиляции `"a.c"` файл `a.h`’ окажется подключенным дважды, и этого невозможно избежать без изменения кода заголовочных файлов).

  Если в таком заголовочном файле содержатся не только прототипы функций (который может быть повторен в программе сколько угодно раз), а еще, например, объявляются типы данных, то это приведет к ошибке двойного определения. Поэтому каждый заголовочный файл должен быть защищен от двойного подключения. Делается это с помощью условных директив препроцессора: каждому заголовочному файлу сопоставляется уникальная символическая константа (обычно соответствующая имени файла), в начале особой директивой проверяется, определена ли эта константа. Если нет — определяется константа и следует код файла, в противном случае ничего не прочитывается. Например, в файле `"a.h"`:

```
#ifndef A_H_INCLUDED
#define A_H_INCLUDED

/* Здесь код заголовочного файла */

#endif /* A_H_INCLUDED */
```

  Парная директива `#ifndef...#endif` сообщает препроцессору подключать код, расположенный между ними, только если константа не определена. Существует также директива `#ifdef...#endif` с противоположным смыслом.

  Последний `#endif` принято снабжать комментарием (не)определенности какой директивы он соответствует, так как он может быть очень далеко от “своего” `#ifndef/ifdef`.

  Определение с помощью `#define` пустой константы (“без значения”, как во второй строка примера) — нормальное для языка Си явление.
- В директиве `#include` заголовочные файлы модулей программы указываются в кавычках, а не треугольных скобках.
- Необязательно выносить прототипы всех функций в заголовочный файл, наоборот, существование внутренних, функций модуля, вспомогательных деталей реализации, не предназначенных для выноса в интерфейс — нормальное явление.
- Следует следить, чтобы заголовочный файл содержал перенос строки в конце. Так как `#include` вставляет его как целое возможны неприятности. Например `a.h`:

```
#define A 300
```

  при подключении с `a.c`

```
#include "a.h"
int main ()
{
    return 0;
}
```

  может дать ошибку как создание константы `#define A 300int main ()` и синтаксически ошибочного кода.
- Заголовочные файлы часто подключаются не только в файлах исходного кода, но и в других заголовочных файлах (чтобы, например, там были доступны типы данных). Файлы исходного кода можно подключить в других файлах исходного кода, но не нужно, так как это нарушит раздельную компиляцию. Подключать заголовочные файлы нужно только тогда, когда их наличие действительно необходимо, опять же для сокращения времени компиляции путем сокращения количества строк кода. Порядок подключения правильно написанных заголовочных файлов значения не имеет.
- Заголовочные файлы также являются не только способом организации модульного программирования, но и способом включения в программу деклараций библиотеки. Сама библиотека подключается компоновщиком.

Пример разбиения программы на модули.

Листинг 6.1: month.h

```
 /****************************************
 * Month data type and naming           *
 * This file is a part of CALENDAR program *
 * (c) J.J. Smith, 1900                 *
 ****************************************/

#ifndef _MONTH_H
#define _MONTH_H

/* MONTH enumerator */

enum month {JAN = 0, FEB = 1, MAR = 2, APR = 3, MAY = 4, JUN = 5,
            JUL = 6, AUG = 7, SEP = 8, OCT = 9, NOV = 10, DEC = 11
           }; /* Цифровой код месяца, работа с ним быстрее */

/* Month name constants */

/* Find month name
 * @param m is a month code
 * @return corresponding string
 */

const char* mname (enum month m);
    /* Для вывода имени месяца в интерфейсе */

#endif /* _MONTH_H */
```

Листинг 6.2: month.c

```
 /****************************************
 * Month data type and naming           *
 * This file is a part of CALENDAR program *
 * (c) J.J. Smith, 1900                 *
 ****************************************/

#include "month.h"

/* Строковые константы для имен месяцев */

const char *SJAN = "January";
const char *SFEB = "February";
const char *SMAR = "March";
const char *SAPR = "April";
const char *SMAY = "May";
const char *SJUN = "June";
const char *SJUL = "July";
const char *SAUG = "August";
const char *SSEP = "September";
const char *SOCT = "October";
const char *SNOV = "November";
const char *SDEC = "December";
const char *SBAD = "";

const char* mname (enum month m)
{
    switch (m)
    {
        case JAN: return SJAN;
        case FEB: return SFEB;
        case MAR: return SMAR;
        case APR: return SAPR;
        case MAY: return SMAY;
        case JUN: return SJUN;
```

```
        case JUL: return SJUL;
        case AUG: return SAUG;
        case SEP: return SSEP;
        case OCT: return SOCT;
        case NOV: return SNOV;
        case DEC: return SDEC;
    }
    return SBAD;
}
```

Листинг 6.3: cal.h

```
 /****************************************
  * Examining a year and a month routine    *
  * This file is a part of CALENDAR program *
  * (c) J.J. Smith, 1900                    *
  ****************************************/

#ifndef _CAL_H
#define _CAL_H

#include "month.h" /* Необходимо, так требуется тип enum month */

/* Number of days in month
 * @param m is month
 * @param leap should be 1 if the year is leap and 0 otherwise
 * @return number of days in month m
 */

int mdays (enum month m, int leap);

/* Check if the year is leap
 * @param year is a year number
 * @return 1 if the year is leap and 0 otherwise
 */

int is_leap (int year);

#endif /* _CAL_H */
```

Листинг 6.4: cal.c

```
 /****************************************
  * Examining a year and a month routine    *
  * This file is a part of CALENDAR program *
  * (c) J.J. Smith, 1900                    *
  ****************************************/

#include "cal.h"

int mdays (enum month m, int leap)
{
    switch (m)
    {
        case JAN:
        case MAR:
        case MAY:
        case JUL:
        case AUG:
        case OCT:
        case DEC: return 31;
        case FEB: return leap ? 29 : 28;
        case APR:
        case JUN:
```

```
        case SEP:
        case NOV: return 30;
    }
    return 0;
}

int is_leap (int year)
{
    return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
}
```

Листинг 6.5: main.c

```
/****************************************************
 * Testing examining a year and a month routine module *
 * This file is a part of CALENDAR program             *
 * (c) J.J. Smith, 1900                                *
 ****************************************************/

#include <stdio.h>
#include "cal.h"

int main()
{
    enum month m;
    int y, i;

    printf ("Select␣month␣number␣and␣enter␣a␣year:\n"
            "%i␣%s\n%i␣%s\n%i␣%s\n%i␣%s\n%i␣%s\n%i␣%s\n"
            "%i␣%s\n%i␣%s\n%i␣%s\n%i␣%s\n%i␣%s\n%i␣%s\n",
            JAN, mname(JAN), FEB, mname(FEB),
            MAR, mname(MAR), APR, mname(APR),
            MAY, mname(MAY), JUN, mname(JUN),
            JUL, mname(JUL), AUG, mname(AUG),
            SEP, mname(SEP), OCT, mname(OCT),
            NOV, mname(NOV), DEC, mname(DEC)
           );

    if (scanf ("%i%i", &i, &y) != 2)
    {
        printf ("Bad␣input␣(should␣be␣2␣integer␣numbers\n");
        return 2;
    }

    if (i<0 || i>11)
    {
        printf ("Bad␣month\n");
        return 1;
    }

    m = i;

    printf ("Number␣of␣days:␣%i\n", mdays(m, is_leap(y)));
    return 0;
}
```

Примечание. Во многих деталях данный код можно улучшить, используя более сложные языковые конструкции, но он четко иллюстрирует создание и использование заголовочных файлов и их зависимость, а также возможность тестировать модуль.

Вопросы для самопроверки.

1. *Что такое модуль?*
2. *Каковы цели и преимущества разделения программы на модули?*

3. *Чем отличаются модули и библиотеки?*
4. *Из каких двух частей (сторон) концептуально состоит модуль?*
5. *Что такое сокрытие деталей реализации?*
6. *Что такое модульное программирование?*
7. *Как реализуется концепция модульного программирования в Си?*
8. *Что такое раздельная компиляция?*
9. *Что такое единица трансляции?*
10. *Что такое объектный модуль?*
11. *Как связаны заголовочные файлы и раздельная компиляция?*
12. *Какие преимущества дает раздельная компиляция?*
13. *Как связаны заголовочные файлы и модули?*
14. *Какую задачу решает компоновщик?*
15. *Что может и что не должно находиться в заголовочном файле?*
16. *Что следует помещать в заголовочный файл?*
17. *Какие ошибки компоновки могут возникнуть при неправильном составлении заголовочного файла и как их избежать?*
18. *Что такое защита от двойного подключения заголовочного файла и как она организуется?*
19. *Что делают директивы препроцессора `#ifdef`/`#ifndef`?*
20. *Какими правилами следует руководствоваться при разделении программы отдельные файлы исходного кода и при составлении заголовочных файлов?*

§6.6. Правила и соглашения хорошего кода

Качество программы определяется не только ее работой, но и тем, насколько читаемый (воспринимаемый) ее код. Действительно, языки программирования созданы не только для общения человека с компьютером, но и общения человека с человеком — как с другим программистом, продолжающим работу над программой, так и самим с собой: во-первых идеи и мысли, которыми программист руководствовался при написании программы, забываются со временем, во-вторых, в хорошо написанном коде легче искать ошибки и вообще легче не допускать их. Следованием правилам, соглашениям и принципам хорошего кода позволяют свести к минимуму количество ошибок в программе, облегчить процесс написания, модернизации и сопровождения программы, повысить качество программы. Приведем основные правила, касающиеся как языка Си, так и программирования в целом.

Общие принципы

- Следует тщательно выбирать названия как программы, так и ее элементов — переменных, функций и др. Это "принцип наименьшей неожиданности" — название и пользовательский интерфейс программы должны говорить сами за себя. Программа, которая называется "синус" не должна считать квадратный корень, если программа просит ввести целое число, она не должна ждать ввода числа с плавающей точкой и т.д.

  Следует использовать говорящие имена переменных. Однобуквенные переменные — не лучшее решение, за исключением таких случаев, например, как когда в переменной `x` содержится координата x, или такой классики, как когда переменная `i` используется как счетчик. Но и переусердствовать с длиной не следует, так как будет сложно вводить код. Категорически не следует использовать хаотические имена типа `xYq1x2`.

  Имена переменных разумно указывать с учетом типа. Традиционно такие переменные как `i`, `j`, `k`, `l`, `m`, `n` — целые, `x`, `y`, `z`, `p`, `q`, `r`, `a`, `b`, `c` — вещественные, `s` — строковые, `c` — символьные, `b` — логические, `p` — указатель.

  При выборе имен функций и имен параметров также следует руководствоваться данным принципом: функция `sin` должна считать синус, а не квадратный корень, а через параметр `mass` следует передавать массу, а не ускорение и т.п.

  Заметим, что в Си имена переменных и функций обычно пишутся строчными буквами, символические константы и константы-перечисления — заглавными.

- Начинать создание программы и ее элементов следует с комментариев.

  Для всякой программы, модуля, функции, блока должно быть четко сформулировано, что этот элемент делает. Для программы должно быть четко описаны входные и выходные данные и выполняемое программой действие, для функции — входные и выходные параметры, возвращаемое значение, особенности поведения — внешняя сторона "черного ящика". Комментарий должен быть написан именно до начала написания кода, это — постановка задачи, под которую следует писать код (а не, наоборот, подводить документацию под код).

  Если программист не может сформулировать что-то на естественном языке, то он не может это запрограммировать. Комментарий должен быть предложением, а не обрывком фразы.

  Под особенностью поведения функции (предупреждениями) понимаются в частности такие моменты, как корректность обработки ошибочных параметров. Например, обрабатывает ли функция указатель `NULL` (если он передан в качестве адреса выходного параметра)? Возвращает ли функция вычисления значения с некоторой точностью код ошибки при задании отрицательной или нулевой точности? Если при создании функций выбран отказ от проверок в пользу большей скорости, то следует дать предупреждение о возможном неопределенном поведении в указанной ситуации.

- Код программы следует читать и редактировать, как и в случае с сочинением, первичный код не более, чем черновик. При переработке кода следует улучшать оформление, производительность, исправлять найденные ошибки.

- Следует обязательно комментировать код, как минимум отмечать, для чего предназначена та или иная переменная и что именно делает программа, а также все ключевые участки алгоритма. Комментарий также нужно структурировать, в начале каждой строчки писать *, чтобы отличить текст от закомментированного кода.

- Следует всегда включать все предупреждения компилятора, это позволяет заметить многие смысловые ошибки программы.

Правила оформления

Конечно, всю программу можно записать в одну строчку, синтаксис это позволяет. Однако, этого делать не нужно, т.к. такой код будет совершенно невоспринимаемым. В принципе каждый программист в праве выбирать свой стиль, но есть некоторые общепринятые варианты.

- Каждый оператор, каждое объявление следует писать в отдельной строке.

- Нужно следовать **правилу четырех пробелов**: операторы вложенного блока следует писать с отступом, на четыре пробела бо́льшим, чем отступ блока более высокого уровня вложенности.

  При этом фигурные скобки блока можно писать друг под другом — это более наглядно:

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

  а можно указывать открывающуюся скобку не в новой строке, это экономит число строк:

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

- При форматировании следует использовать аккуратные столбцы везде, где можно. Код

```
int k; /* Что делает k. */
const char *str; /* Опишите, что делает str. */
int l; /* Что делает l. */
k = 5; /* Здесь идет комментарий. */
strset(str); /* Здесь второй комментарий. */
```

```
l = k; /* А здесь третий. */
```

читается хуже, чем такой:

```
int k;           /* Что делает k.            */
const char *str; /* Опишите, что делает str. */
int l;           /* Что делает l.            */

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

В последнем случае кроме прочего декларатор отделен от кода, что также повышает наглядность.

- Код следует разбивать на логические куски ("абзацы текста"), где каждый кусок выполняет одну операцию. Куски следует отделять пустыми строками — **правило абзацев**.
- Пробелы — очень важные элементы структурирования кода, как и текста естественного языка. За знаком препинания должен идти пробел, в случае с Си это относится в частности к знакам операций, которые следует отделять от операндов пробелами (исключение унарные операции `*p`, `a--` и т.п., вызовы функций `f(a1, a2)`, хотя после запятой все-таки пробел желателен, операции `.` и `->` — до и после них пробел как раз категорически не нужен) — **правило пробелов**.

  В редких, но возможных случаях, может оказаться, что отсутствие пробела создаст ошибку в программе:

```
float *p;
y=(x/*p);
f(y/* комментарий */);
```

сочетание `/*` во второй строке будет воспринято как начало комментария, а не деления на результат разыменования указателя, и программа примет вид

```
float *p;
y=(x );
```

что является синтаксически верным кодом, но не соответствующим кодируемому алгоритму.

- Более короткий блок условного оператора нужно писать вперед. Например, если это проверка ошибки. Такой код

```
if (!some_error_condition)
{
    /* Десятки строк кода */
}
else
    error(SOME_ERROR_CODE);
```

хуже, чем такой,

```
if (some_error_condition)
    error(SOME_ERROR_CODE);
else
{
    /* Десятки строк кода */
}
```

так как в первом случае есть риск, что `if` и `else` окажутся на разных экранах и их нельзя будет охватить одним взглядом, что важно, так как условие в равной степени управляет как блоков `if`, так и блоком `else`, и расположение их далеко друг от друга ухудшает наглядность и воспринимаемость кода.

В Си также часто возникает ситуация, когда при некоторых значениях параметра функция должна вернуть ошибку. Тогда код лучше писать так:

```
if (some_error_condition) return SOME_ERROR_CODE;
/* Десятки строк кода */
```

Это редкий случай, когда отход от принципа "один выход из каждого блока" (функции) оправдан, он сокращает код (не нужен `else`, скобки и отступы) и улучшает воспринимаемость кода. Это, однако, специфика Си-подобных языков, а не языков, основанных на парадигме структурного программирования вообще.

- Следует избегать конструкций, которые ничего не делают.

Неудачно	Хорошо	Пояснение
`int i = j;` `i += k;`	`int i = j + k;`	инициализируйте при объявлении

`while (i != 0) /* ... */`
`if (cond != 0) /* ... */`
`if (cond == 0) /* ... */`
	`while (i) /* ... */`
	`if (cond) /* ... */`
	`if (!cond) /* ... */`
		`!=0` ничего не делает в отношении условия, его не надо указывать никогда, последнее допускает исключения, когда нарушается восприятия смысла

`if (cond)`
`    return 1;`
`else`
`    return 0;`
 	
`return cond ? 1 : 0`
 	
`return cond ? 0 : 1`
	`return cond;`
	
	`return cond;`
	
	`return !cond;`
		не надо дважды вычислять одно и то же условие

`++i;`
`f(i);`
`--i;`
 	
`return i++;`
	`f(i + 1);`
	
	`return i + 1;`
		не модифицируйте значение, если оно используется не более одного раза

`if (i > j)`
`else if (i < j)`
`else if (i == j)`
	`if (i > j)`
	`else if (i < j)`
	`else`
		при невыполнении первых двух условий третье всегда выполнено, и его не нужно проверять; это, однако, не так для вещественных чисел, для них нужно использовать третий `else`!

- Символические константы следует использовать, когда константа встречается несколько раз, когда это сложная константа, когда есть вероятность ее изменения по мере разработки программы, когда удобно и т.п. — почти во всех случаях. Но не в простейших — не надо заводить обозначения для 0 (ложь) и 1 (истина) и т.п.

  Имена символических констант традиционно пишутся заглавными буквами, так же как и идентификаторы в перечислимом типе, которые по сути являются целочисленными символическими константами.

  В большинстве случаев не следует отдавать предпочтение `enum`-константам вместо использования целочисленных символических констант. Перечисления следует использовать тогда, как нужен именно пе-

речислимый тип, например тип данных с фиксированным количеством значений. Заметим, что 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 ("Error\n")
};
```

Указание метки `default` крайне желательно, так как в силу сводимости перечислимого типа к типу `int` возможно возникновение неперечисленного значения. Современные компиляторы способны предупреждать о пропуске значений при переборе вариантов `enum` в `switch`, что также удобно.

Вопросы эффективности и минимизации ошибок

- Если нет причины использовать постфиксный инкремент или декремент, надо отдавать предпочтения префиксному, он быстрее.
- Обязательно следует инициализировать значения переменных, чем раньше, тем лучше. Лучше всего инициализировать переменные прямо при объявлении, отказ от этого должен быть четко обоснован (например, переменная первый раз используется как выходной параметр функции, тем самым инициализируется, или если ее значение можно инициализировать только присваиванием после проведения каких-то вычислений и т.п.).
- Следует избегать использования указателей там, где это возможно, и вообще проявлять особую осторожность при их использовании.
- По возможности следует отдавать предпочтение итеративному алгоритму, а не рекурсивному. Однако последний может быть полезен в качестве быстрого чернового решения и решения-сравнения для итеративного варианта.
- Следует избавляться от повторяющихся вычислений (вычислений одного и того же значения несколько раз), запоминая вычисленные значения в переменных.

Принципы процедурно-модульного программирования

- Всякий обособленный алгоритм следует выделять в виде функции, особенно если этот участок кода используется в программе более, чем один раз.

  В подавляющем большинстве затраты времени и памяти на вызов функции несущественны, а чем короче завершенный участок кода, тем меньше в нем ошибок (считается, что число ошибок, допускаемых программистом, пропорционально числу строк кода), поэтому нужно разбивать программу на короткие независимые фрагменты.

  Рекомендуется, чтобы функция целиком влезала на экран (можно было охватить ее одним взглядом), учитывая, что в настоящее время размер экранов относительно большой, указывается и более жесткое ограничение — 10-15 строк.

  Отказ от выделения алгоритма в функцию должен быть четко обоснован.

- При создании функции нужно, во-первых, четко сформулировать, что делает функция и выбрать название, которое должно говорить само за себя. Во-вторых, при написании функции следует четко определить, какие данные функция получает на вход и какие выдает на выходе (как и всякий алгоритм — нужно задать черный ящик) и написать соответствующий прототип (заголовок) функции, четко определив какой параметр за передачу каких данных отвечает, является ли он входным и выходным, и выбрать им говорящие имена. В-третьих, функцию следует снабдить комментарием — описанием, содержащим информацию о том, что делает функция, какие параметры что означают, каково возвращаемое значение функции, какие обстоятельства влияют на поведение функции.

  Всякую функцию следует документировать, фактически комментарий и есть документация к функции. Только после написания комментария и прототипа следует приступать к написанию кода функции.

Поскольку именно заголовочный файл является интерфейсом, "лицом" модуля, содержащим прототипы функций, определение констант, типов данных и т.д., он должен быть документирован особенно качественно. Имена заголовочного файла — т.е. имена модуля — (как в общем-то и файлов исходного кода) должны говорить сами за себя. Все прототипы функций, типы данных, константы и т.д. должны быть четко прокомментированы. Правильный комментарий заголовочного файла — практически готовая документация к модулю (библиотеке). Кроме того, в начале заголовочного файла и файла исходного кода указывается заголовочный комментарий, описывающий, что это за программа и кто ее автор. Существуют специальные программные средства (Doxygen и др.), предназначенные для того, чтобы извлекать из заголовочных файлов комментарии и генерировать из них текстовую документацию.

- Всякая функция должна заниматься своим делом и только им, и только одним. Это довольно общий принцип в науке и технике, известный как "добродетель узости". (Например, дифференциация научных знаний вызвана тем, что нельзя создать теорию всего, локальная теория лучше работает в конкретной области, разделение труда повышает производительность труда человека в выбранной профессии, в современном мире уже нельзя обучить человека всем имеющимся профессиям).

  В случае с программированием как минимум следует различать **интерфейсную** часть и **реализацию** (расчетную, рабочую часть) программы — **механизм**. Функции, занимающиеся расчетами не должны заниматься вводом-выводом. "**Функция должна делать что-то одно, но делать это хорошо**". (На самом деле этот принцип разумно распространять и на целые программы.)

  Это важно, например, чтобы изменение интерфейса не требовало изменения механизма и наоборот. Можно даже сказать, что реализация (механизм) — более низкий уровень абстракции, чем интерфейс.

  Пример хорошего прототипа функции — стандартная библиотечная функция

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

  Если бы ее прототип был таким

  ```
  void sin ();
  ```

  и функция всегда считывала бы аргумент с клавиатуры, а результат выводила на экран, то такая функция была бы бесполезна: ее нельзя использовать в выражении, нельзя использовать для обработки данных, полученных в ходе расчетов, пришедших файлов и т.п.

- Рекомендуется стремиться создавать функции "библиотечного качества".

  В это понятие вкладывается универсальность, т.е. возможность применения вне программы, в которой она написана, без адаптации кода (возможность вынести в библиотеку и подключить к программе) и в разных ситуациях (при разных способах происхождения входных и использования выходных данных), четкая документированность (понятный другому программисту прототип и описание к нему), корректная обработка всех возможных значений параметров (от функции ожидается получение результата или информации об ошибке без прерывания программы и попаданий в бесконечный цикл; указание в документации при каких значениях параметров функция работать не будет допускается, если проверка корректности параметров может расцениваться как ущерб производительности), использование эффективного в смысле затрат времени и памяти алгоритма.

  Следует учесть, что часто функции программы вызывают другие функции той же программы и не могут существовать отдельно в полном смысле этого слова. Однако, это не меняет сути понятия: функция должна работать независимо от того, как реализованы вызываемые ей функции, то есть ориентироваться на них лишь как на черный ящик. Можно также говорить о библиотечном качестве группы функций, решающих определенную задачу или круг задач.

  В некоторых случаях используются и чисто внутренние, вспомогательные, функции, созданные, например, для разбиения длинного алгоритма на подзадачи и непредназначенные для внешнего использования. Однако и их следует создавать с учетом изложенных требований и рекомендаций.

- Следует избегать использования глобальных переменных. Они создают дополнительные побочные эффекты, трудноотыскиваемый механизм обмена данными между функциями, принуждают пользоваться выбранными создателем функции именами переменных, создают конфликты между функциями.

  Сравните, например,

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

```
```

и

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

Во втором случае функцию можно использовать классическим, удобным образом с получением выражений — `sqr(x)`, `sqr(y + 2)`, `sqr(2 * z + sqr(5 * a))` и т.д., в первом нужно всегда сначала присвоить что-то в переменную x, а последний пример вообще выродится в

```
x = 5 * a;
x = 2 * z + sqr();
r = sqr();
```

что представляет собой малопонятный код. Еще сложнее ситуация, когда появится 2 и более функции, и каждая будет ждать какие-то данные в глобальной переменной `x`.

“Черный ящик” функции с глобальными переменными сложнее описать и осознать.

В некоторых случаях глобальные переменные, однако, используются: например, функции стандартной библиотеки возвращают некоторое значение, сигнализирующее об ошибке, при этом устанавливают в глобальную переменной `errno` код ошибки. Это сделано для того, чтобы избежать постоянного указания дополнительного параметра для возвращения кода ошибки.

- Использовать одноименные параметры в разных функциях нормально. Использовать одноименные локальные переменные в разных функциях нормально. Они не видят друг друга и никак не конфликтуют между собой. Это удобно: при создании функции не нужно заботиться о том, какие локальные идентификаторы использованы в другой функции. Можно даже смело перекрывать глобальные переменные, если они есть, но точно в функции не нужны.

  В то же время следует избегать перекрытия имен параметров локальными переменными, перекрытия локальных переменных переменными в блоках более глубокого уровня вложенности, это источник ошибок. А вот использовать одноименные переменные в невложенных друг в друга блоках одной функции в принципе можно, они снова не видят друг друга и не перекрываются.

- Следует четко различать формальные параметры и локальные переменные функции. Они очень похожи при использовании — являются идентификатором, ассоциированным с ячейкой памяти, содержащей значение, которое можно и прочитать и изменить. Значения параметров действительно можно менять, это безопасно, так как не отразится на данных вызывающей функции (когда изменение проводится именно параметра, а не данных по адресу, на который ссылается параметр-указатель, например, изменение параметра, не являющегося указателем, или изменение собственно указателя, то есть присвоение ему адреса другой ячейки памяти), и может оказаться эффективнее, чем копировать в локальную переменную. Однако существует существенное различие в их назначении: параметры предназначены для обмена данными при вызове функции, локальные переменные — для хранения временных данных функцией.

  Каждый параметр представляет собой дополнительную степень свободы обмена данными при вызове функции. Использование параметров вместо локальных переменных — грубая ошибка проектирования программы. Следует ограничиваться минимальным необходимым количеством параметров, создание функции с 4-5 и более параметрами не рекомендуется. Локальные переменные — “внутреннее” дело функции, параметры — “внешнее”.

  Ошибкой также будет использование глобальных переменных вместо локальных — такая функция будет менять данные всей программы тогда, когда достаточно изменить локальные данные функции, а вызов из одной такой функции другой функции, оперирующей с той же глобальной переменной, нарушит работу первой функции. Глобальные переменные — дополнительный и ненаглядный механизм обмена данными между функциями.

- Если функция имеет более одного возвращаемого значения, дополнительные выходные параметры можно “организовать” с помощью указателя.

  В смысле алгоритма (черного ящика) параметры, передаваемые чисто копированием данных, являются входными. Возвращаемое значение является выходным параметром. Хотя сам указатель (как адрес)

является входным параметром функции с точки зрения Си (более того — его значение передается копированием адреса, указанного в формальном параметре), с точки зрения алгоритма, параметры, передающие указатель на данные, могут быть как входными, так и выходными. Чисто выходные параметры в этом случае никак не выделяются синтаксически, функция может просто проигнорировать значение, которое лежит по передаваемому адресу.

Однако в некоторых случаях указатели используются только для входных параметров (например, не следует копировать в функцию строки, массивы и др. большие объемы данных, достаточно передать указатель на них). В этом случае параметр следует обязательно декларировать с модификатором `const` (как указатель на константу).

- Каждую функцию следует тестировать на наличие ошибок отдельно, проверяя ее поведение и возвращаемое значение при различных комбинациях параметров.
- Большие программы нужно разбивать на несколько файлов исходного кода, на несколько модулей, в соответствии с разделением на подзадачи, на стороны функционала. Обычно это становится существенным, когда число функций в программе достаточно большое (десятки и сотни). В главный файл программы рекомендуется включать только главную функцию.

  На практике короткие программы из одного файла скорее исключение, чем правило. Профессиональные интегрированные окружения разработки ориентированы исключительно на многофайловые программы, объединяемые в "проект" или "решения" (используются и другие термины).

Вопросы для самопроверки.

1. *Какими принципами следует руководствоваться при создании функции?*
2. *Чем следует руководствоваться при написании прототипа функции и документации к нему?*
3. *Чем отличаются глобальные переменные, локальные переменные и формальные параметры? В смысле синтаксиса? В смысле семантики?*
4. *Укажите базовые правила форматирования программ.*
5. *Куда следует писать более короткий блок условного оператора и почему?*
6. *Приведите примеры избыточных и ничего не делающих конструкций в языке Си.*

§6.7. Тестирование и верификация программ

Тестирование программного обеспечения

Тестирование программного обеспечения — исследование, проводимое с целью получения информации о качестве программного продукта.

Качество программного продукта определяется следующими характеристиками:

- соответствие предъявленным требованием к дизайну и функционалу, возможность использоваться по назначению в целом;
- корректный ответ на все варианты входных данных;
- уровень производительности;
- совместимость и переносимость;
- защищенность, надежность и др.

Тестирование проводится как исследованием готового программного продукта, так и его отдельных компонентов, как с использованием знания кода программы, так и без. В связи с этим можно классифицировать тестирование по типам.

1. *По уровню тестирования.* **Модульное тестирование** — проведение тестирования каждого нетривиального компонента программы (функции, модуля), проводится "снизу вверх" (от минимальных частей программы до программы в целом). Один из наиболее важных подходов тестирования в процессе разработки. Проводится путем проверки ответа каждого компонента на возможные запросы. Набор запросов необходимо подбирать таким образом, чтобы были проверены все имеющиеся ветви кода. Для большинства современных языков программирования высокого уровня существуют инструменты для автоматизации модульного тестирования (Check, CUnit, GNU Autounit и др. для C). Следующие уровни — **интеграционное тестирование** (проверка взаимосвязи компонентов программной системы), и **системное тестирование** (тестирование программы в целом на ее соответствие требованием). Выделяют **альфа**-тестирование, проводимое разработчиками, и **бета**-тестирование, осуществляемое будущими пользователями.

2. *По доступу к коду.* Тестирование **черного ящика** (осуществляется через внешние и пользовательские интерфейсы программы) и **белого ящика** (имеется доступ к коду программы и возможность добавлять код, связанный с компонентами программы — типично для модульного тестирования). Тестирование **серого ящика** использует знания о внутренних структурах данных и алгоритмах, но тестирование осуществляется через пользовательские интерфейсы (доступность кода обеспечивает возможность сделать лучшую подборку тестов, чем при тестировании черного ящика.)
3. *Тестирование производительности.* **Нагрузочное тестирование** (сбор сведений о производительности и времени отклика на внешний запрос), **стресс-тестирование** (тестирование производительности при пиковых нагрузках). Используется также **профилирование кода** — определение времени, затрачиваемого на выполнение отдельных компонентов системы (например, для выяснения функций, на которые тратится больше всего времени). Дело в том, что если улучшить в 10 раз производительность функции, на которую тратится 1% времени, то это приведет лишь 0.1% улучшения общей производительности, а если улучшить на 10% производительность функции, на которую тратится 90% времени, то это улучшит производительность программы на 9%, поэтому профилирование позволяет направить ресурсы производителей ПО на оптимизацию наиболее существенных компонентов программной системы.
4. Также выделяют *тестирование стабильности, надежности, практичности и др.*

Верификация программного обеспечения

Тестирование программного обеспечения не может *доказать*, что программа не содержит никаких ошибок и удовлетворяет требованиям. Это может сделать **формальная верификация** — доказательство соответствия программы (алгоритма) их формальному описанию. Используется также термин **доказательное программирование**.

Одним из математических инструментов, решающих поставленную задачу, является **логика Хоара**.

Ее идея состоит в следующем. Рассмотрим тройку

$$\{P\}C\{Q\},$$

где P и Q — некоторые утверждения, P — предусловие, Q — постусловия, а C — команда (алгоритм). При этом, такая тройка, означает что если выполнено предусловие P, то после применения команды C будет выполнено постусловие Q.

Например,

```
/* {a > b} */
a -= b;
/* {a > 0} */
```

если выполнено предусловие $(a > b)$, то после замены a на $a - b$ будет выполнено постусловие, $a > 0$.

Основное утверждение гласит, что если имеет место

$$\{P\}C_1\{P'\} \text{ и}$$
$$\{P'\}C_2\{Q\},$$

то имеет место

$$\{P\}C_1C_2\{Q\},$$

то есть результат последовательного исполнении команд, при котором постусловие первой удовлетворяет предусловию второй, удовлетворяет постусловию второй команды (**правило композиции**).

Снабжение программы комментариями, указывающими пред- и постусловия участков кода, называется **корректным аннотированием программ**.

Пример.

```
#define MAX_BUFFER 1024

/* ... */

/* Две строки: */
char *s1 = /* ... */
     *s2 = /* ... */;

/* Строковый буфер для конкатенации */
char buffer [MAX_BUFFER];
```

```
/* { sizeof (buffer) = MAX_BUFFER } */

size_t s1_len = strlen (s1),
       s2_len = strlen (s2);

/* { s1_len = strlen (s1) И s2_len = strlen (s2) } */

/* Проверяем, что сумма длин строк позволяет записать
 * результат конкатенации в буфер
 */

if (s1_len + s2_len + 1 > MAX_BUFFER)
    return BUFFER_ERROR_CODE;

/* { strlen (s1) + strlen (s2) + 1 <= sizeof (buffer) } */

/* Это условие гарантирует корректность выполнения
 * следующего вызова функции
 */
strcat(buffer, s1, s2);

/* { buffer = s1 + s2 конкатенация() } */
```

Замечание. Логика Хоара не позволяет доказать, что программа завершится вообще, а не впадет в бесконечный цикл, поэтому считается частичной верификацией.

Проверка условий при исполнении Си-программы

В заголовочном файле `<assert.h>` определен макрос (“функция” препроцессора, макросы описаны в приложении D.2) `assert()`, позволяющий проводить проверку предусловий в процессе исполнения программы.

Если предусловие не выполнено, программа завершится с сообщением об ошибке, с указанием строки файла и условия, которое оказалось невыполненным. Аргументом функции является выражение, воспринимаемое как логическое.

Пример.

```
#include <assert.h>

#define N 100;

/* Тип данных vector, для хранения векторов
 * произвольной размерности в статическом
 * массиве из N >= n элементов
 */

typedef struct vector
{
    double c[N];  /* координаты  */
    size_t n;     /* размерность */
} vector;


/* Вычисляет сумму векторов v1 и v2, результат записывается в v3 */
void vector_sum (const vector* v1, const vector *v2, vector *v3)
{
    size_t i;
    assert (v1->n == v2->n); /* проверка, что складываются
                              * одноразмерные векторы
                              */

    assert (v1->n <= N);     /* проверка, что не будет выхода
                              * за границы массива
                              */
```

```
    v3->n = v1->n;
    for (i = 0; i < v1->n; ++i)
        v3->c[i] = v1->c[i] + v2->c[i];
}
```

Проверка условий требует времени и является элементом отладки, в окончательной версии программы ее разумно отключать. Для этого достаточно определить символическую константу `NDEBUG`:

```
#define NDEBUG
#include <assert.h>
```

Замечание. Ввиду отключаемости проверки `assert`, выражения-аргументы данной функции не должны иметь побочных эффектов, так как иначе они не будут применены в окончательной версии вообще (например, некорректно использовать `assert(--i)` и т.д.).

Вопросы для самопроверки.

1. *Что такое тестирование программного обеспечения?*
2. *Чем определяется качество программного продукта?*
3. *Что такое модульное тестирование?*
4. *Что такое тестирование черного, белого и серого ящика? Какие у них преимущества и недостатки?*
5. *Что такое тестирование производительности?*
6. *Что такое профилирование кода?*
7. *Что обеспечивает и что гарантирует правильно проведенное тестирование программного обеспечения?*
8. *Что такое верификация программного обеспечения?*
9. *Какова идея логики Хоара?*
10. *Что такое тройка Хоара?*
11. *Для чего и как применяется логика Хоара?*
12. *Какой аппарат языка программирования Си предназначен для проверки условий логики Хоара?*
13. *Каков синтаксис макроса `assert`?*
14. *Как отключить проверку условий `assert` во всей программе?*
15. *Какие ограничения должны быть наложены на выражения, применяемые в качестве аргумента макроса `assert`*

Глава 7. Тонкости работы с указателями и управления памятью

§7.1. Динамическое выделение памяти

Сегменты памяти

Прежде чем разобрать возможность динамического выделения памяти следует уточнить блоки (участки) памяти, доступные программе в момент работы, часто традиционно именуемые **сегментами**.

Во-первых, это два сегмента для хранения глобальных объектов программы: в одном содержится код программы (**сегмент кода**), в другом — данные, определенные как глобальные (**сегмент данных**). Данные сегменты выделяются программе при запуске (при создании процесса на стадии динамического связывания определяется их необходимый размер) и существуют до момента завершения ее исполнения. Запуск и завершение функций не приводит к изменению размера сегмента.

Во-вторых, это **сегмент стека** программы — блок, в который с помощью смещения указателя на заданную позицию отводится место под экземпляр данных каждой запускаемой функции (формальные параметры, локальные переменные, точки возврата, возвращаемое значение) и др. Обычно размер стека определяется операционной системой, он может быть фиксированным или динамически увеличиваться по мере увеличения объема хранимых в нем данных. Однако предельный размер стекового сегмента, как правило, существенно меньше объема оперативной памяти. *Все локальные массивы, объявленные внутри функций, создаются в стеке, что*

ограничивает размер данных размером остатка сегмента стека, создавая еще одно ограничение на использование таких массивов.

В-третьих, процесс может запросить выделение блока из свободной памяти. "Вся остальная память", т.е. память, не занятая данными, кодом и стеком, а также другими процессами, называется "**кучей**" (heap). Изначально такая память недоступна ни одному из процессов, чтобы процесс мог ею воспользоваться, необходимо запросить блок памяти из кучи у операционной системы. После этого данный блок резервируется за запросившим его процессом до момента освобождения.

Данный механизм и терминология были разработаны до появления в операционных системах виртуального адресного пространства процесса. Последнее, однако, не меняет сути механизма: для того, чтобы процесс мог получить блок в свободной от сегментов кода, данных и стека части собственного виртуального адресного пространства, необходимо запросить данный блок у операционной системы, которая свяжет выделенный блок с физической памятью. Если процесс обращается к части собственного виртуального пространства, не выделенной операционной системой, то это трактуется как ошибка обращения к памяти, не принадлежащему процессу, что приводит к его аварийному завершению.

Статическое и динамическое

Термин **статический** в программировании означает неизменный в процессе работы и чаще всего относится к определяемому в процессе компиляции (написания программы). Например, в Си используются только статические типы данных переменных — они фиксируются на стадии компиляции и не могут быть изменены в процессе работы. Статические массивы — массивы, размер которых не изменяется в процессе реботы программы.

Динамический, в свою очередь, относятся к определяемому и изменяемому во время исполнения программы. Например, динамические массивы — массивы, размер которых может изменяться в процессе работы программы.

Память также может выделяться статически и динамически: объем памяти под все глобальные переменные и локальные переменные данной функции в Си определяется на стадии компиляции. Это относится и к массивам. Хотя можно говорить о том, что Си динамически выделяет участок для хранения локальных данных функции в стеке, данный процесс не является динамическим выделением памяти в полном смысле этого слова: ведь сам стек выделяестся программе операционной системой статическим образом.

Массивы, выделенные в стеке, являются статическими, их размер не подлежит изменению в процессе работы. В базовом стандарте Си он также должен определяться на стадии компилияции. Хотя стандарт C99 позволят сделать это в процессе работы, например

```
int n;

scanf("%i", &n);
int a[n];
```

такой подход сопряжен с определенными рисками и по этому обычно его использование не рекомендуется. В частности, слишком большое значение `n` приведет к ошибке переполнения стека — непредсказуемому поведению и аварийному завершению работы программы: данную ошибку нельзя предотвратить и обработать в процессе работы программы. Кроме того, изменение размера такого массива в процессе работы программы невозможно.

Использование статических массивов для хранения наборов данных требует, чтобы размер массива был известен в момент написания программы, что далеко не всегда так. Обход этого ограничения путем выделения памяти в стеке с запасом и хранением фактической длины массива является фактически механизмом создания динамического массива, но такой поход, во-первых, жестко ограничивает размер данных сверху (причем ограничение программы должно быть существенно меньше размера стека), во-вторых, неоптимален в силу перерасхода памяти на хранение неиспользуемого остатка массива.

Альтернативой стековым массивам является создание массивов в памяти, выделенной из кучи.

Функции динамического выделения и освобождения памяти

Запрос на выделение памяти в Си осуществляется с помощью функции `malloc`, определенной в `<stdlib.h>`:

```
void * malloc (size_t size);
```

Единственный параметр функции — размер выделяемого блока в байтах. Функция возвращает указатель на выделенный блок или `NULL`, если выделение памяти не удалось (например, если такого объема свободной памяти нет). Гарантируется, что выделенный блок будет представлять собой непрерывный участок указанного размера.

Поскольку указатель эквивалентен массиву, а нетипизированный указатель можно присвоить типизированному, с помощью данной функции можно динамически выделять память под массивы требуемой длины. Вычисление длины блока должно быть осуществлено при вызове функции, как число элементов, умноженное на размер элемента (Си никак не может проверить корректность вычисления размера и соответствие типа, поэтому при использовании `malloc` следует соблюдать повышенную осторожность).

Далее с таким указателем, совместимым с массивом, можно работать как с "обычным" массивом, все действия будут корректны.

```
unsigned long n, i;

int *a;

scanf ("%lu", &n);

a = malloc (n * sizeof(int));

if (!a) /* проверка, выделилась ли память */
{
        /* ... обработка ошибки, завершение функции ... */
}

for (i = 0; i < n; ++i)
    scanf ("%i", &a[i]);

for (i = 0; i < n; ++i)
    printf ("%i\n", a[i]);
```

Разумеется, возможно выделение памяти не только под массив, но и под отдельную переменную. Хотя небольшую переменную проще и эффективнее создать в стеке, этот подход может использоваться в том случае, когда количество переменных неизвестно на стадии компиляции — для создания **динамических структур данных**, примеры которых описываются в следующих главах.

По завершении работы с выделенным блоком его следует освободить с помощью функции `free`:

```
void free (void *block);
```

Если это не сделать, память останется зарезервированной за процессом до момента его завершения (а в старых ОС — до перезагрузки компьютера). При потере указателя на выделенный блок (например, локальный указатель в функции, будет утерян при завершении функции, если не возвращен ей) теряется возможность как дальнейшей работы с ним, так и его освобождения. Такая ситуация называется **утечкой памяти**. Сама по себе утечка не приводит к неверной работе программы, но приводит к повышенной трате ресурса памяти (разовая утечка может показаться безвредной, но если утечка происходит в многократно вызываемой функции — например, вызываемой при нажатии на клавишу в текстовом редакторе — то программа будет занимать все больше и больше памяти во время работы, пока в конце концов не получит отказ в ее выделении от операционной системы, потому что займет всю "кучу" или превысит доступный лимит).

Функция `free` не зануляет указатель `block`, иногда это рекомендуется делать вручную, так как после освобождения `block` будет ссылаться на ячейку памяти, не принадлежащую программе, и обращение к ней может привести к ошибкам:

```
free(a);
a = NULL;
```

Разумеется, если область видимости данного указетеля заканчивается сразу или почти сразу после вызова `free`, такое действие может считаться лишним.

"Платой" за использование динамического выделения памяти является трата времени на сам акт ее выделения и освобождения. Это более медленные, по сравнению с выделением стекового блока операции, хотя время их выполнения и не зависит от количества выделяемых байт. Время зависит от количества уже выделенных блоков и характера их распределения — ведь системе необходимо найти свободный участок подходящего размера.

Функция `malloc` не гарантирует, что элементы выделяемого массива будут как-то инициализированы, по умолчанию там будут непредсказуемые данные (т.н. "мусор"). Функция

```
void * calloc (size_t num, size_t size);
```

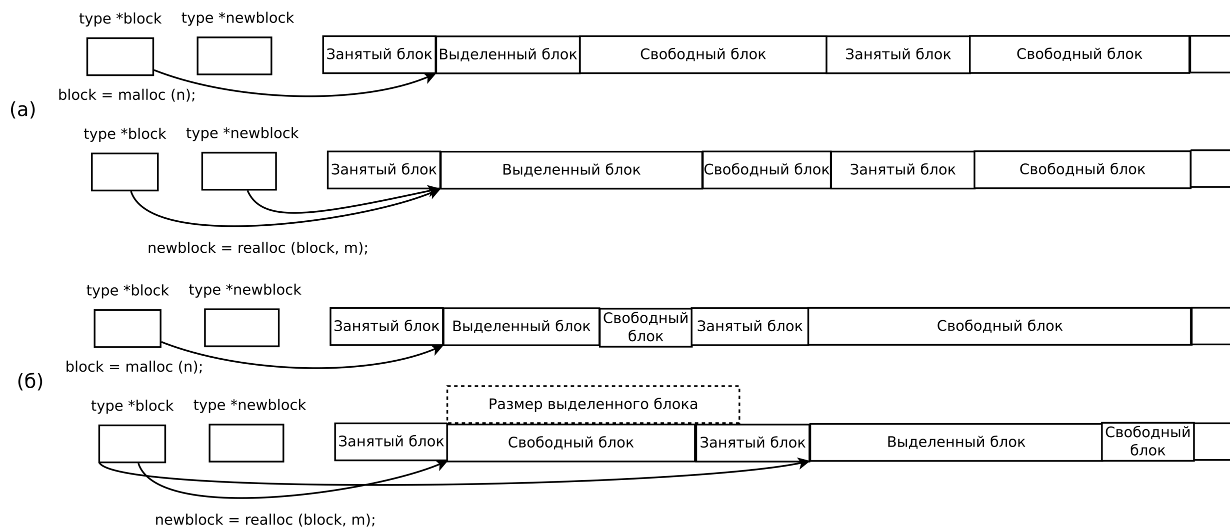


Рис. 7.1: Иллюстрация работы `realloc`: (а) — изменение размера блока без перемещения, (б) — изменение требует перемещения, после вызова функции `realloc` указатель `block` все еще указывает на старый блок, в котором находятся непредсказуемые значения, данные перемещены в новый блок.

резервирует `num` элементов размера `size`, инициализируя их нулями, однако на практике используется редко, так как на инициализацию тратится время: часто в программе следом за выделением памяти требуется инициализировать массив какими-то ненулевыми значениями.

Изменение размера выделенного блока

Размер выделенного блока можно изменить, для этого служит функция

```
void * realloc(void *block, size_t size);
```

С помощью данной функции можно как уменьшить, так и увеличить выделенный блок до размера `size`. В последнем случае может оказаться, что просто увеличить блок до нужного размера нельзя, так как участок может быть занят другими данными (рис. 7.1).

Тогда функция автоматически переместит выделенный участок в пригодное место. *Перемещение может произойти и при уменьшении размера блока*, например, если операционная система решит, что этот блок оптимальнее расположить в другом участке памяти. В случае перемещения данных время работы функции пропорционально размеру блока памяти, без перемещения — не зависит от размера блока памяти.

Функция возвращает адрес нового блока — необязательно совпадающий со старым, поэтому в общем случае использовать нужно новое значение, а использование `block` некорректно.

```
int *a, *a_new;
size_t n, n_new;

/* ... */

a = malloc(n * sizeof(int));

/* ... */

a_new = realloc(a, n_new * sizeof(int));
if (!a_new)
{
    /* обработка ошибки или работа с a как
     * корректным массивом старой"" длины
     */
}
else
    a = a_new
```

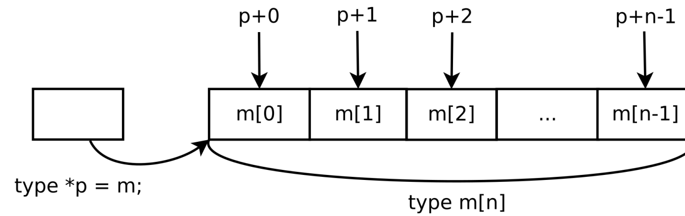


Рис. 7.2: Доступ к элементам массива с помощью арифметики указателей.

Замечания

Если аргументом `malloc` является 0, то функция возвращает `NULL`.

Если аргументом `free` является `NULL`, то никаких действий не предпринимается (поэтому нет необходимости проверять указатель на `NULL` перед вызовом данной функции). Если указатель, передаваемый в `realloc`, имеет значение `NULL`, то функция работает как `malloc`, просто выделяя память. Если передаваемый в `realloc` указатель не `NULL`, а новый размер — 0, то функция действует как `free`, то освобождает память, и возвращает `NULL`. В частности, это означает, что имеет право на существование массив из 0 элементов как указатель на `NULL` — его можно "выделить", увеличить до другого размера, массив со строго положительным числом элементов можно сократить до массива нулевой длины без дополнительных проверок.

В то же время использование в качестве аргумента функций `free` и `realloc` адреса, который не был ранее выделен из кучи (с помощью `malloc`, `realloc` или других функций) приведет к непредсказуемому результату. Это относится и к попыткам изменить размер стекового массива.

Разумеется, вызовы функций `malloc`, `realloc` и `free` не обязаны располагаться в одной функции: получающиеся локальные указатели можно передать в другие функции как возвращаемые значения или через параметр-указатель (указатель на указатель на элемент массива). Важно, чтобы в процессе работы программы каждому выделению памяти соответсвовало освобождение памяти, и осуществлялось оно как можно быстрее после того, как блок памяти становится ненужным.

Вопросы для самопроверки.

1. *Каковы недостатки статических массивов?*
2. *Каковы преимущества динамических массивов?*
3. *Каковы недостатки динамических массивов?*
4. *С помощью каких функций стандартной библиотеки выделяются блоки памяти из кучи?*
5. *Что общего между блоком памяти, выделенным с помощью `malloc`, и статическим массивом?*
6. *Каковы различия между блоком памяти, выделенным с помощью `malloc`, и статическим массивом?*
7. *Почему важно освобождение выделенного блока памяти по окончании работы с ним?*
8. *Что такое "утечка памяти"?*
9. *Как работает функция `realloc`?*

§7.2. Арифметика указателей

К типизированным указателям можно прибавлять целые числа, получая при этом новый указатель. Например, для `type *p` и `int i` указатель `p+i` указывает на ячейку с адресом

$$V(p+i) = V(p) + i \cdot S(type) = P(p[i]).$$

Используются обозначения из параграфа 4.2: $S(v)$ — размер данных в памяти, $V(p)$ — значение в ячейке памяти, для указателя это соответствующий адрес; $P(a)$ — адрес ячейки памяти. Сложение коммутативно: `p+i`≡`i+p`.

Данные действия корректны только в рамках одного массива: `p+i` есть i-й элемент массива p. Здесь следует еще раз обратить внимание на совместимость указателей и массивов, их эквивалентность при обращении к элементам (рис. 7.2).

На самом деле взятие элемента массива `a[i]` Си всегда читает как `*(a+i)`, (а в силу коммутативности это можно записать и как `i[a]`, что интерпретируется как `*(i+a)`, хотя такой подход является предельно экзотическим).

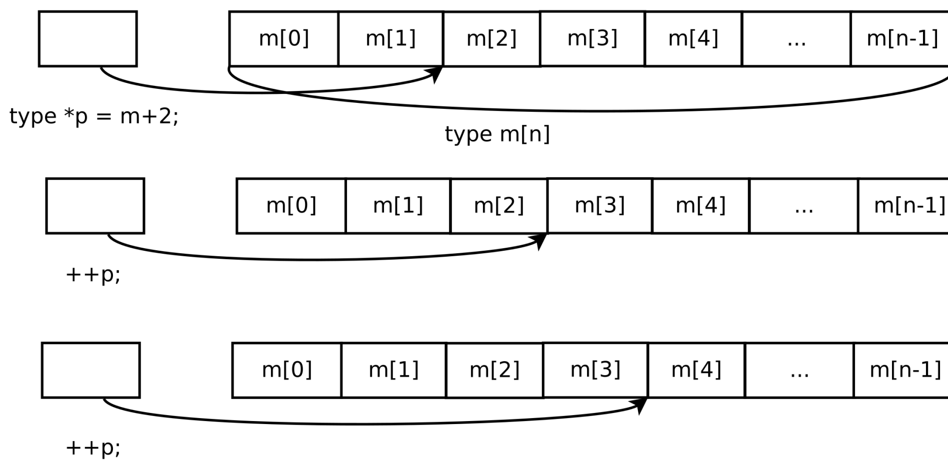


Рис. 7.3: Перемещение по массиву с помощью инкремента указателя.

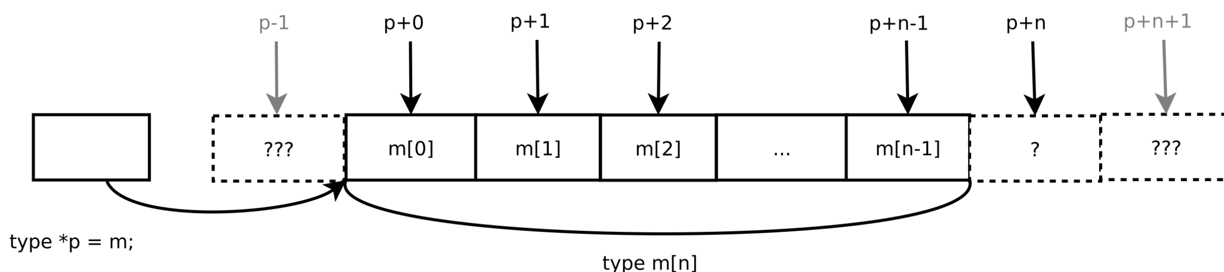


Рис. 7.4: Арифметика указателей. Значение `p+n` корректно для сравнения, хотя и должно быть разыменовано (указывает на непредсказуемые данные), значения `p+n+1`, `p+n+2` и т.д., `p-1`, `p-2` и т.д. некорректны.

Кроме того, для типизированных указателей определена операция инкремента и декремента. При этом инкремент "превращает" указатель на элемент, расположенный в памяти следом за указываемым, декремент — наоборот, то есть прибавляет или вычитает к значению указателя `sizeof(type)` байт.

Инкремент и декремент позволяет указателю "путешествовать" вправо и влево по элементу массива (рис. 7.3).

Кроме того, указатели можно сравнивать на равенство (==) и неравенство, при этом больше тот указатель, который расположен в памяти "правее".

Арифметические операции и операции сравнения корректны только для указателей, которые указывают на элементы одного и того же массива, также стандарт допускает получение указателя элемента массива, следующего за последним (но не -1-го!) — рис. 7.4.

В то же время обращение к -1 или любому другому отрицательному индексу указателя не является ошибкой априори. Например, пусть имеется массив и указатели

```
int a[10];

int *p = a, *q = p + 3;
```

В этом случае указатель `p` указывает на нулевой элемент массива `a`, q — на третий. Соответственно, `q[-1]`, как и `*(q-1)` корректно и является обращением ко второму элементу массива `a`. А вот

$$\texttt{p[-1]} \equiv \texttt{*(p-1)} = \texttt{*(q-4)},$$

недопустимо как и не гарантируется получение самих указателей `p-1`, `q-4`, `p+11`, `q+8` и т.д. Использовать указатели `p+10` и `q+7` можно, но их разыменование даст непредсказуемый результат.

Операции равенства и неравенства корректно применять к любой паре указателей, это позволяет проверить, верно ли, что два указателя указывают на одну и ту же ячейку памяти.

Арифметика указателей — более быстрый способ прохождения массива, по сравнению с использованием индексов. Сравните

```
int k;
```

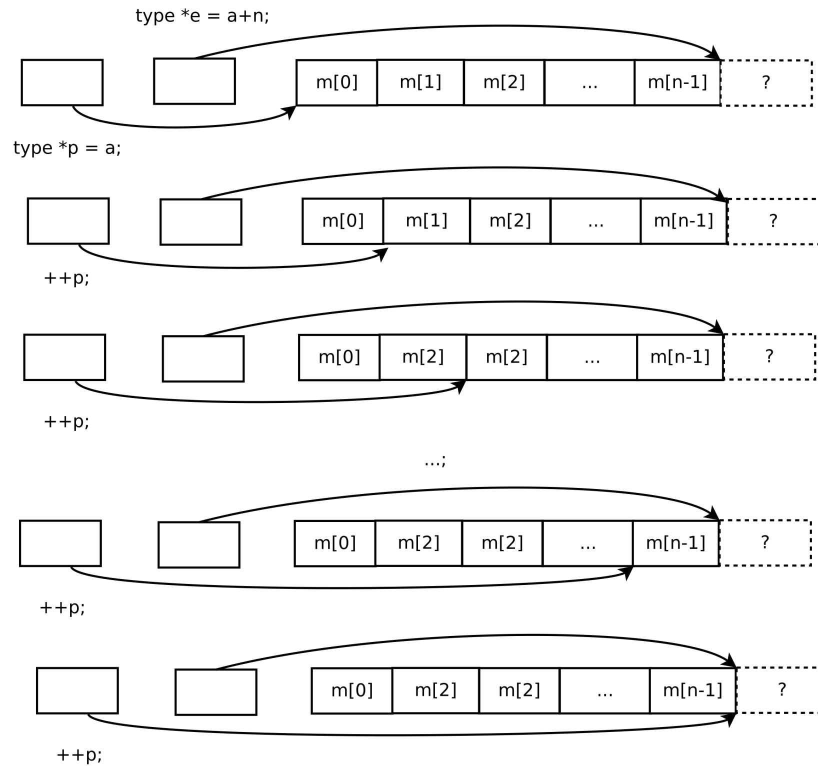


Рис. 7.5: Прохождение массива с помощью арифметики указателей. Цикл останавливается, когда достигнут элемент массива, следующий за последним.

```
for (k = 0; k < n; ++k)
    a[k] = /* ... */
```

и

```
int *p, *e = a + n;
for (*p = a; p < e; ++p)
    *p = /* ... */
```

в первом случае требуется $\Theta(n)$ инкрементов целого числа и $\Theta(n)$ сложений на вычисление адреса элемента и разыменований, во втором — $\Theta(n)$ инкрементов указателя и разыменований, а также $\Theta(1)$ вычисления адреса элемента, следующего за последним (рис. 7.5). Работа с указателями происходит быстро (арифметические действия осуществляются с той же скоростью, что и аналогичные действия над целыми числами или даже быстрее, так как процессоры затачиваются на обработку указателей).

Указатели можно вычитать с получением целого числа — количества элементов массива между ними (в частности, можно найти индекс элемента, если из указателя на него вычесть указатель на начало массива).

Арифметику можно применять и к динамическим массивам (правило, допускающее ссылаться на элемент, следующий за последним, действует и здесь):

```
unsigned n;

int *a, *i, *e;

scanf ("%u", &n);

a = malloc (n * sizeof(int))

for (i = a, e = a + n; i < e; ++i)
    scanf ("%i,␣i);
```

Для многомерных массивов ситуация аналогична (рис. 7.6).

```
int a[4][3][2];
```

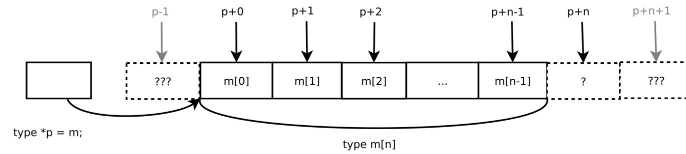


Рис. 7.6: Доступ к элементам двумерного массива с помощью арифметики указателей.

```
a[i][j][k] = /* ... */
/ * или */
*(*(*(a + i) + j) + k) = / ... */
```

В контексте арифметики указателей следует обратить особое внимание на поведение указателей на константу и константных указателей. Так, к указателям, которые сами являются константами, неприменимы операции инкремента, декремента и арифметического присваивания — их значения изменять нельзя. В то же время, эти операции применимы к (неконстантным) указателям на константу — значение указателя изменить можно, но посредством данного указателя не допускается изменение значения ячейки, на которую указатель указывает. Результатом применения арифметических операций к указателю на константу также будет указатель на константу.

В частности, если есть функция, для которой массив является входным параметром, следует использовать константный указатель для прохода по массиву:

```
void print_int_array (const int *a, size_t n)
{
    const int *p, *e;
    for (p = a, e = p + n; p < e; ++p)
        printf ("%i\t", *p);
}
```

Вопросы для самопроверки.

1. *Какие арифметические операции применимы к указателям?*
2. *Каков результат применения арифметических операций к указателям?*
3. *Что означает прибавление целого числа к указателю?*
4. *Что означает разность двух указателей?*
5. *Что показывают операции сравнения двух указателей?*
6. *Какие ограничения накладываются на применение арифметики и сравнения указателей?*
7. *Как осуществить проход массива с помощью арифметики указателей?*
8. *Почему арифметика указателей считается более быстрой, чем работа с индексами?*

§7.3. Примеры реализации обработки строк

В качестве иллюстрации работы с арифметикой указателей рассмотрим возможные реализации библиотечных функций из `string.h`. Конкретная реализация данных функций не специфицируется стандартом и может быть различна в различных компиляторах. Библиотечные реализации могут быть оптимизированы под особенности конкретных процессоров.

Копирование строк

Простейший способ — копирование с использованием индексов, пока не встретится символ с кодом 0 (он также должен быть скопирован).

```
void strcpy(char *dst, const char *src)
{
    int i = 0;
    while ( (dst[i] = src[i]) != '\0' ) ++i;
}
```

Более эффективный способ — использование арифметики указателей:

```
void strcpy(char *dst, const char *src)
{
    while ( (*dst = *src) != '\0' )
    {
        dst++;
        src++;
    }
}
```

Дополнительно можно учесть особенность постфиксного инкремента: сначала возвращать значение (указателя), потом изменять:

```
void strcpy(char *dst, const char *src)
{
    while ( (*dst++ = *src++) != '\0' ) ;
}
```

В данном случае первым скопируется нулевой символ строки, последним — символ с кодом 0. Это изменение не оптимизирует время, но сокращает код.

Также разумно убрать "ничего не делание":

```
void strcpy(char *dst, const char *src)
{
    while ((*dst++ = *src++));
}
```

Двойные скобки вокруг условия следует оставить как сигнал о том, что здесь действительно используется присваивание, а не присутствует ошибка использования присваивания вместо равенства: для того, чтобы подавить соответствующее предупреждение компилятора.

Функция `strcpy` стандартной библиотеки должна вернуть исходный указатель `dst`. Добавим соответствующее изменение:

```
char * strcpy(char *dst, const char *src)
{
    char *d = dst;

    while ((*d++ = *src++));

    return dst;
}
```

Этот вариант является компактным и изящным, но на самом деле требует один лишний инкремент двух указателей (переход на следующий после `'\0'` символ). Поэтому оптимальнее будет остановиться на варианте

```
char * strcpy(char *dst, const char *src)
{
    char *d = dst;

    while ((*d = *src)) d++, src++;

    return dst;
}
```

благодаря операции "запятая" вместо разделителя операторов, использование фигурных скобок для ограничения тела цикла не требуется.

Копирование блоков памяти

Также рассмотрим вариант реализации функции копирования массива с помощью арифметики указателей:

```
void * memcpy(void *dst, const void *src, size_t n)
{
    const char *s = src, *e = s + n;
    char *d = dst;
```

```
    while (s < e) *d++ = *s++;

    return dst;
}
```

Использование указателя `char *` вместо `void *` необходимо, так как для нетипизированных указателей невозможно разыменование, требуемое для копирования, а также стандартном не поддерживается арифметика указателей. Переменная `e` здесь используется для определения конца массива — ее значение основано на документированной возможности указывать на элемент массива, следующий за последним.

Реализация может быть улучшена копированием сначала машинными словами (данными объемом `int`), а остатка — посимвольно:

```
void * memcpy(void *dst, const void *src, size_t n)
{

        /* указатели на int для прохода словами */

    const int *is = src, *ie = is + n / sizeof (int);
    int *id = dst;

    /* указатели для прохода остатка побайтово */

    const char *s = ie, *e = s + n;
    char *d;

    while (is < ie) *id++ = *is++;

    d = (char *) id;

    while (s < e) *d++ = *s++;

    return dst;
}
```

Деление `n / sizeof(int)` целочисленно, поэтому даст наибольший размер массива элементов типа `int`, непревышающий `n` байт. Указатель `ie` будет указывать на следующую за таким массивом ячейку памяти, которая указателем `s` интерпретируется как однобайтовая. После первого цикла `id` указывает на элемент массива-назначения, следующий за последним скопированным, поэтому остаток данных следует записывать по этому адресу, что и достигается резыменованием данного адреса как однобайтового указателя `d`.

Машинное слово — объем данных, обрабатываемый процессором как единое целое (одной машинной командой, за фиксированное время). В частности, это относится к чтению и записи данных в оперативной памяти, целочисленной арифметике и др. операциям. Конкретный размер слова зависит от архитектуры процессора, как право составляет несколько байт. В частности, если размер слова и типа данных `int` составляет 4 байта, то побайтовое копирование будет работать в 4 раза медленнее, чем копирование ячейками типа `int`.

Вычисление длины строки

Строка завершается символом с кодом 0, поэтому простейший способ вычисления длины состоит в следующем:

```
size_t strlen (const char *s)
{
    size_t i = 0;

    while (s[i] != '\0') ++i;

    return i;
}
```

однако, арифметика указателей даст более быструю функцию

```
size_t strlen (const char *s)
{
```

```
    const char *c = s;
    while (*c != '\0') ++c;

    return c-s;
}
```

кроме того следует опустить “ничего не делание”

```
size_t strlen (const char *s)
{
    const char *c = s;
    while (*c) ++c;

    return c-s;
}
```

можно было бы учесть особенность постфиксного инкремента

```
size_t strlen (const char *s)
{
    const char *c = s;
    while (*(c++));

    return c-s-1;
}
```

но это потребует дополнительной операции, что неоптимально.

§7.4. Ошибки при работе с памятью

Работа с указателями является небезопасной, проводимые компилятором проверки минимальны, поэтому такие ошибки программирования обычно обнаруживаются лишь во время исполнения программы. Наиболее типичными проявлениями таких ошибок являются:

- *использование неинициализированного значения* (“мусора”) — как “обычной” переменной (в стеке и глобальной), так и в выделенной из кучи области памяти — приводит к непредсказуемым результатам вычислений или обращению к неверному адресу (если эта переменная — указатель);
- *разыменовывание нулевого указателя* обычно приводит к мгновенной аварийной остановке программы (хотя может быть мишенью атаки злоумышленника), поэтому легко обнаружимо при отладке (рекомендуется инициализировать указатели нулем и занулять указатели, которые ссылались на ячейки, которые более не могут использоваться);
- *обращение по адресу, не принадлежащему программе* — например, разыменовывание неинициализированного указателя (“мусорный” адрес), обращение к освобожденному блоку и т.п. — приводит к мгновенной аварийной остановке программы, но заранее различить по тексту кода возникнет ли именно данная ошибка или следующая в списке нельзя;
- *обращение по адресу вне пределов массива* — выход за границы массива, обращение к адресу временной переменной (например, если функция вернет указатель на переменную, которая была в стеке и удалилась по завершении функции) — одна из наиболее труднообнаружимых ошибок, так как приводит к неверным результатам вычислений, порче данных и кода, которые скажутся при исполнении других участков кода программы (возможно, даже других функций и модулей);
- *освобождение памяти, которая не была выделена* — вызов функций `free` и `realloc` для блоков, который не был выделен из кучи;
- *утечка памяти* — может привести к “разрастанию” программы в памяти, например, если многократно вызывается одна и та же функция с такой ошибкой, при достаточно долгой работе программа займет всю память и будет аварийно завершена системой.

Источниками ошибок могут служить

- использование неинициализированных переменных, массивов (в т.ч. динамических), указателей (в том числе при отсутствии проверки на корректность выделения памяти);
- неверное вычисление или учет размера блока памяти или буфера (при выделении памяти, копировании, индексации, в т.ч. из-за неправильного указания размера элемента массива);

- путаница с использованием указателей из разных блоков (например, указание в качестве конца массива фактический конец другого массива), ошибка с типом указателя, использование данных в качестве указателя и др.
- использование адреса временного элемента (адреса переменной, переставшей существовать после освобождения стековых данных по завершении функции), освобожденного блока данных (в т.ч. в результате вызова `realloc`);

Использование адреса временного элемента заслуживает особого внимания. Временными элементами являются локальные переменные и формальные параметры функций. Их адрес может быть сохранен и глобальной переменной или возвращен из функции, однако вне области видимости данной переменной обращение по данному адресу даст непредсказуемый результат, например:

```
double * foo (/* ... */)
{
    double var = /* ... */

    /* ... */

    return &var;
}
```

В данной функции присутствует ошибка, так как функция возвращает адрес временной переменной `var` и его использование вне данной функции даст некорректно. Аналогичная ситуация с массивами:

```
double * bar (/* ... */)
{
    double array [10];

    /* ... */

    return array;
}
```

Ошибкой будет и использование адреса переменной, пришедшего из другого вызова той же самой функции

```
double * foo (double *p, int n)
{
    double var = /* ... */

    if (n > 0) *p = /* ... */;

    return &var;
}

void bar (/* ... */)
{
    double *p1 = foo (NULL, 0);
    double *p2 = foo (p1, 1);

    /* ... */

}
```

так как ко второму вызову функции `foo` исходная переменная `var` перестанет существовать. (Разумеется, передача указателя на локальную переменную в качестве аргумента других функций в т.ч. вглубь по рекурсии является корректной, так как текущая функция не завершится до завершения вызванных из нее функций.). Формально недопустима и такая ситуация:

```
void foo ()
{
    double *p;

    if (/* ... */)
    {
        double var = /* ... */
```

```
        /* ... */

        p = &var;
    }

    bar(*p);

    /* ... */

}
```

Несмотря на то, что обращение производится к переменной той же функции, область видимости переменной `var` заканчивается блоком кода, в котором она определена. Компилятор, скорее всего, не будет генерировать код, смещающий указатель стека по входу в блок кода (только при вызове функции), поэтому существует высокая вероятность корректной работы данного кода, однако в общем случае это не гарантируется.

Ошибки при работе с памятью в общем случае дают непредсказуемый результат, а на практике чаще всего приводят к

- аварийному завершению программы, часто в непредсказуемом месте;
- получении неверных результатов вычислений;
- порче данных и уязвимостям (маловероятно, что это произойдет в следствие случайной ошибки, но теоретически может случиться особенно в результате умышленных действий, исполнение данных, которые окажутся верными машинными командами, в результате могут быть вызваны, например, команды уничтожения дисковых файлов или передаче секретных данных злоумышленнику).

Ошибки при работе с указателями труднообнаружимы, поэтому использование таких инструментов требует повышенной осторожности.

Существуют средства для проверки ошибок работы с памятью, в Linux это стандартная утилита valgrind, кроссплатформенная — DrMemory и др. Также некоторые компиляторы имеют опции, позволяющие внедрить соответствующие проверки непосредственно в исполняемый файл программы. Все эти методы существенно (в десятки раз) замедляют работу программы.

Рекомендуемая литература

[1] В. В. Борисенко “Основы программирования”.

[2] Б. Керниган, Д. Ритчи “Язык программирования Си”.

[3] А. Голуб “С и С++. Правила программирования”.

[4] Стандраты языка Си: C90 (ISO/IEC 9899:1990), C99 (ISO/IEC 9899:1999) и более новые.

Приложения

A. Базовые сведения о компьютере, операционной системе и файлах

§A.1. Компьютер — твердый и мягкий продукт

Компьютер представляет собой техническое устройство, для выполнения некоторых команд — программ. Таким образом, для того, чтобы получить работающий компьютер необходимо, чтобы присутствовали обе компоненты — аппаратная и программная. Их также называют англ. hardware (жесткий продукт) и software (мягкий продукт).

Аппаратная часть компьютера представляет собой совокупность различных элементов, наиболее важные из которых — *центральный процессор*, собственно выполняющий команды, *оперативная память* (где хранятся работающие в данный момент программы и данные, с которым программа работает) *устройства ввода-вывода* (позволяющие получить информацию от компьютера и передать информацию в него), *носители информации* (где данные могут сохраняться для последующего использования).

В ходе развития компьютерной техники были созданы тысячи различных моделей центральных процессоров, имеющие различный набор исполняемых команд, размер обрабатываемых данных и т.п. Говорят, что различные процессоры имеют различную *архитектуру*, часто несовместимую между собой, то есть программа, созданная для одной архитектуры, не может быть запущена на процессоре другой архитектуры.

В настоящее время некоторые из процессоров, представленных на рынке, имеют совместимую архитектуру (например, процессоры фирмы AMD поддерживают архитектуру процессоров Intel), другие — сильно отличающуюся, например процессоры архитектуры ARM несовместимы с ними.

Все программы, представленные в компьютере, можно условно разделить на 3 класса: **встроенное программное обеспечение**, **операционную систему** и **прикладные программы**. Встроенное программное обеспечение представляет собой программу, запускаемую при включении компьютера (BIOS, EFI и т.п.), основная ее задача — найти подключенные устройства и запустить операционную систему.

Задача операционной системы — унифицировать доступ к различным устройствам (например, чтобы работа с принтерами разных производителей или носителями информации различных типов была одинакова как для пользователя, так и программиста), обеспечить безопасную и удобную работу пользователей. Первая задача решается с помощью специальных программ — *драйверов* устройств, которые отвечают за зависимые от модели устройства наборы команд.

Операционные системы различны не только для различных архитектур, даже наоборот, существуют примеры реализации одной и той операционной системы для разных архитектур. В настоящее время наиболее популярные операционные системы для домашних компьютеров и мобильных устройств — Windows, GNU/Linux, MacOS, Android и др. Изучение работы с операционными системами лежит за рамками данного курса.

Следует учесть, что операционные системы предоставляют некоторый набор возможностей для программиста, без которого проблемно создать даже простейшую программу. Поэтому программы, написанные для одной операционной системы могут не работать на другой. Комбинация архитектуры, операционной системы и других аппаратных и программных факторов, влияющих на формат и работы программ, называется **платформой**. Важной задачей программиста является поддержание **кроссплатформенности** создаваемых программ, то есть возможности обеспечения работы программы на разных платформах: это делает программу доступной для более широкой пользовательской аудитории.

Программа представляет собой некоторые данные, хранимые на носителе информации. Для того, чтобы прикладная программа была исполнена ее необходимо **запустить**. Операционные системы предоставляют пользователю такую возможность. При запуске программы создается **процесс** — работающий экземпляр программы. На многих старых операционных системах одновременно могла работать только одна программа, для современных операционных систем типична ситуация, когда в ней запущены десятки процессов. Не следует путать понятия программы и процесса.

§A.2. Общие сведения о файлах

С точки зрения идеи хранения информации, данные, находящийся в памяти компьютера (оперативной памяти) уничтожаются в момент завершения программы и не могут быть доступны при ее последующем запуске. Данные, находящиеся на *долговременном носителе информации* (“долговременной памяти”: диске, карте памяти, твердотельном накопителе и др.) сохраняются при перезапуске программы и компьютера, поэтому опера-

ционные системы (и языки программирования высокого уровня) предоставляют возможность чтения и записи этих данных. На носителях информации данные хранятся в файлах.

С точки зрения организации доступа к информации **файл** — поименованная упорядоченная последовательность байтов (поименованная совокупность данных). Файл может находиться и не только на долговременном носителе информации, но и в оперативной памяти, а также храниться в архиве, передаваться по сети и т.п., поэтому говорить о файле, как о поименованной области или совокупности данных на носителе информации (диске) не совсем корректно.

Доступ к файлам осуществятся посредством функций операционной системы и поэтому в различных операционных системах может осуществляться по разному. В данный момент общими для большинства ОС персональных и микрокомпьютеров являются несколько принципов.

- Файлы группируются в **каталоги**, которые представляют собой файлы особого рода, хранящие список файлов (и каталогов), находящихся в них. Таким образом, существует самый верхний (корневой) каталог, внутри которого есть свои каталоги, образующие **дерево каталогов**. Каталоги, находящиеся внутри некоторого каталога, называются **подкаталогами**.

  В Windows и некоторых графических окружениях других ОС также используется термин “папка”, однако это понятие не тождественно понятию каталога: например, в дереве папок папка “рабочий стол” находится в вершине, а соответствующий ей каталог “спрятан” глубоко в подкаталогах; существует папка “компьютер” (на разных версиях Windows она называется по-разному — “мой компьютер”, “этот компьютер” и др.), но такого каталога нет и т.д.

- Файлы и каталоги формируют **файловую систему**. Это понятие имеет два значения: файловая система конкретного носителя информации, определяющая способ хранения файлов на данном носителе, и файловая система ОС, объединяющая файловые системы подключенных к компьютеру носителей.

  Файловые системы носителей информации различны, наиболее часто используемые из них — FAT (File Allocation Table, таблица размещения файлов; в настоящее время используется на небольших носителях информации типа карт памяти, имеет ограничение на размер файла в 4 Гб, существует улучшенная версия exFAT, смягчающая ограничения; эти ФС удобны для съемных носителей, так как не содержат информацию о правах доступа к файлам, что позволяет открывать доступ любому пользователю на любом компьютере, передавая тем самым информацию), NTFS (ФС ОС Windows NT и выше), ext и btrfs (ext2, ext3, ext4; ФС ОС Linux), HFS+ (ФС MacOS) и другие. Они не совместимы между собой, для каждой из них требуется собственный драйвер, собственная реализация файловых операций.

  Файловая система операционной системы — совокупность ФС, подключенных к компьютеру носителей информации. В разных операционных системах формируется различным образом. В Unix-подобных ОС один из носителей объявляется корневым каталогом ФС ОС, другие носители информации подключаются — **монтируются** — в пустые подкаталоги (любого уровня вложенности), тем самым превращая пустой каталог в каталог с деревом каталогов и файлами, находящимися на данном носителе с сохранением структуры вложенности.

  В ОС DOS/Windows каждому носителю сопоставляется т.н. *буква диска* (хотя карты памяти и твердотельные накопители называть дисками вообще говоря некорректно, так как в них нет ничего круглого и вращающегося) — от A до Z, диски A и B зарезервированы за дисководами, которые на современные компьютеры не устанавливаются, таким образом дерево ФС носителя информации становится деревом данного диска. В современных версиях Windows носители информации также могут быть подмонтированы в произвольный пустой подкаталог.

  Также в подкаталоги могут быть подмонтированы файловые системы, несвязанные с ФС носителей информации напрямую — сетевые, виртуальные, архивов, образов дисков и т.п.

- Уникальным именем файла (в т.ч. каталога) в ФС (ФС ОС) является полный путь к файлу. Для *nix подобных систем он начинается с символа слеша `/`, далее следует список каталогов дерева, разделенных тем же символом слеша `/`, и в конце имя файла, например

  `/home/user/dir1/dir2/file.txt`

  `/home/user/my dir/some_dir`

  В Windows путь начинается с буквы диска, затем следует комбинация `:\`, далее идет список каталогов дерева, разделенных символом обратного слеша \ и в конце имя файла, например

  `D:\dir1\dir2\file.txt`

  `C:\my dir\some_dir`

  Для обобщения полного пути в компьютерной документации используются обозначения

/path/to/file

/path/to/dir

и

D:\PATH\TO\FILE

D:\PATH\TO\DIR

соответственно и тому подобные. Windows поддерживает прямой слеш при указании пути, поэтому для обеспечения кросс-платформенности следует использовать его.

- Со всякой работающей программой ассоциирован текущий (рабочий) каталог, как правило это каталог, из которого программа запущена, хотя программа может его сменить. Поэтому доступ к файлам может осуществляться по **относительному пути**, вычисляемому относительно текущего каталога. В относительном пути можно переходить на каталог уровня ниже, просто указав его имя, или выше, указав сочетание `..` вместо имени (также символ `.` означает текущий каталог).

  Например, если текущий каталог

  /home/user/dir1/

  то относительный путь

  dir2/file.txt

  превращается в абсолютный

  /home/user/dir1/dir2/file.txt

  а

  ../../user2/file2.txt

  в

  /home/user2/file2.txt

  и т.д. (аналогично для Windows, где еще можно указывать путь относительно диска, начав с символа \, так как там отдельно существует текущий диск, а у каждого диска свой текущий каталог, смена которых происходит независимо).

- У файлов принято выделять собственно имя (название) и **расширение** (тип). Расширением считается часть имени (полного имени), следующая после последнего символа точка (`.`). Расширение соотносится с типом формата файла и программами, способными его обрабатывать. Также у файла имеются атрибуты, характеризующие время правки, создания и доступа, права доступа и файлу и др. параметры, зависящие как от ОС, так и от ФС носителя.

- Всякий файл характеризуется длиной (размером) — числом принадлежащих ему байтов. Хотя произвольный доступ к любому байту файла возможен, наиболее быстрым является последовательные чтение и запись байт за байтом, перемещая хранящийся в системе указатель (счетчик) позиции. Запись в конец файла приводит к увеличению размера файла, попытка чтения данных за последним байтом файла приводит к событию достижения конца файла, известному также аббревиатурой EOF (End Of File).

В. Дополнительные сведения о представлении чисел

§В.1. Общие замечания

Следует иметь в виду несколько важных моментов.

1. Вообще говоря, нигде, в том числе в стандарте языка Си, не гарантируется, что 1 байт есть 8 битов. Например, существуют и *семибитные* компьютеры. В стандарте Си гарантируется, что тип `char` — строго однобайтовый (т.е. 1 байт, а не 8 битов!). При этом не конкретизируется, знаковый он или нет.

2. Поскольку представление отрицательных целых чисел может осуществляться не только в дополнительном, но и в прямом коде, то при восьмибитном байте для типа `signed char` стандартом гарантируется диапазон значений от -127 до +127, для двухбайтовых целых — от -32767 до +32767 и т.д., представление в соответствующих типах чисел -128, -32768 и т.д. не гарантируется.

   На разных компьютерах переполнение может приводить к различным результатам, поэтому ориентироваться на "правильность" его значения не следует (например, нельзя рассчитывать, что -1 соответствует самому большому беззнаковому числу).

3. Порядок байтов в многобайтовых целых числах может быть различным: **от младшего к старшему** (**little-endian**, “*остроконечный*”) и **от старшего к младшему** (**big-endian**, “*тупоконечный*”). В большинстве современных компьютеров используется второй способ, но вообще говоря это не всегда так. Например, число 16416, представляющее собой комбинацию $64 \cdot 256 + 32$, в двухбайтовом целом первом случае будет записано байтами 32 и 64, во втором — 64 и 32, в четырехбайтовом — байтам 32, 64, 0 и 0, во втором — 0, 0, 64 и 32.

   В виду такого различия по разному будет вести себя операции битового сдвига.

4. Для представления чисел с плавающей точкой используются различные форматы, в том числе отличные от IEEE 745. Хотя для типов `float` и `double` гарантируется поддержка одинарной и двойной точности этого стандарта, ничего нельзя сказать про `long double`.

§B.2. Таблица операций для расширенных чисел с плавающей точкой

+	-Inf	-1	-0	0	1	Inf	NaN
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-NaN	NaN
-1	-Inf	-2	-1	-1	0	Inf	NaN
-0	-Inf	-1	-0	0	1	Inf	NaN
0	-Inf	-1	0	0	1	Inf	NaN
1	-Inf	0	1	1	2	Inf	NaN
Inf	-NaN	Inf	Inf	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

-	-Inf	-1	-0	0	1	Inf	NaN
-Inf	-NaN	-Inf	-Inf	-Inf	-Inf	-Inf	NaN
-1	Inf	0	-1	-1	-2	-Inf	NaN
-0	Inf	1	0	-0	-1	-Inf	NaN
0	Inf	1	0	0	-1	-Inf	NaN
1	Inf	2	1	1	0	-Inf	NaN
Inf	Inf	Inf	Inf	Inf	Inf	-NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

*	-Inf	-1	-0	0	1	Inf	NaN
-Inf	Inf	Inf	-NaN	-NaN	-Inf	-Inf	NaN
-1	Inf	1	0	-0	-1	-Inf	NaN
-0	-NaN	0	0	-0	-0	-NaN	NaN
0	-NaN	-0	-0	0	0	-NaN	NaN
1	-Inf	-1	-0	0	1	Inf	NaN
Inf	-Inf	-Inf	-NaN	-NaN	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

/	-Inf	-1	-0	0	1	Inf	NaN
-Inf	-NaN	Inf	Inf	-Inf	-Inf	-NaN	NaN
-1	0	1	Inf	-Inf	-1	-0	NaN
-0	0	0	-NaN	-NaN	-0	-0	NaN
0	-0	-0	-NaN	-NaN	0	0	NaN
1	-0	-1	-Inf	Inf	1	0	NaN
Inf	-NaN	-Inf	-Inf	Inf	Inf	-NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

<	-Inf	-1	-0	0	1	Inf	NaN
-Inf	*Л*	*И*	*И*	*И*	*И*	*И*	*Л*
-1	*Л*	*Л*	*И*	*И*	*И*	*И*	*Л*
-0	*Л*	*Л*	*Л*	*Л*	*И*	*И*	*Л*
0	*Л*	*Л*	*Л*	*Л*	*И*	*И*	*Л*
1	*Л*	*Л*	*Л*	*Л*	*Л*	*И*	*Л*
Inf	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*
NaN	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*

>	-Inf	-1	-0	0	1	Inf	NaN
-Inf	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*
-1	*И*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*
-0	*И*	*И*	*Л*	*Л*	*Л*	*Л*	*Л*
0	*И*	*И*	*Л*	*Л*	*Л*	*Л*	*Л*
1	*И*	*И*	*И*	*И*	*Л*	*Л*	*Л*
Inf	*И*	*И*	*И*	*И*	*И*	*Л*	*Л*
NaN	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*

⩽	-Inf	-1	-0	0	1	Inf	NaN
-Inf	*И*	*И*	*И*	*И*	*И*	*И*	*Л*
-1	*Л*	*И*	*И*	*И*	*И*	*И*	*Л*
-0	*Л*	*Л*	*И*	*И*	*И*	*И*	*Л*
0	*Л*	*Л*	*И*	*И*	*И*	*И*	*Л*
1	*Л*	*Л*	*Л*	*Л*	*И*	*И*	*Л*
Inf	*Л*	*Л*	*Л*	*Л*	*Л*	*И*	*Л*
NaN	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*

⩾	-Inf	-1	-0	0	1	Inf	NaN
-Inf	*И*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*
-1	*И*	*И*	*Л*	*Л*	*Л*	*Л*	*Л*
-0	*И*	*И*	*И*	*И*	*Л*	*Л*	*Л*
0	*И*	*И*	*И*	*И*	*Л*	*Л*	*Л*
1	*И*	*И*	*И*	*И*	*И*	*Л*	*Л*
Inf	*И*	*И*	*И*	*И*	*И*	*И*	*Л*
NaN	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*

=	-Inf	-1	-0	0	1	Inf	NaN
-Inf	*И*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*
-1	*Л*	*И*	*Л*	*Л*	*Л*	*Л*	*Л*
-0	*Л*	*Л*	*И*	*И*	*Л*	*Л*	*Л*
0	*Л*	*Л*	*И*	*И*	*Л*	*Л*	*Л*
1	*Л*	*Л*	*Л*	*Л*	*И*	*Л*	*Л*
Inf	*Л*	*Л*	*Л*	*Л*	*Л*	*И*	*Л*
NaN	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*	*Л*

≠	-Inf	-1	-0	0	1	Inf	NaN
-Inf	*Л*	*И*	*И*	*И*	*И*	*И*	*И*
-1	*И*	*Л*	*И*	*И*	*И*	*И*	*И*
-0	*И*	*И*	*Л*	*Л*	*И*	*И*	*И*
0	*И*	*И*	*Л*	*Л*	*И*	*И*	*И*
1	*И*	*И*	*И*	*И*	*Л*	*И*	*И*
Inf	*И*	*И*	*И*	*И*	*И*	*Л*	*И*
NaN	*И*	*И*	*И*	*И*	*И*	*И*	*И*

§B.3. Переносимость двоичного представления чисел

Целые числа

Переносимость двоичного представления целых чисел (например, для сохранения в файле) следует организовывать путем приведения к какому-то одному порядку байтов. Например, при сохранении на *little-endian* платформе приводить к *big-endian*-представлению, при сохранении на *big-endian* платформе сохранять без кон-

вертации, а при чтении на *little-endian* платформе приводить с *big-endian*-представления, при чтении на *big-endian* платформе прочитывать без конвертации, или наоборот.

Как правило компиляторы позволяют определить, что компиляция производится для *big-endian* платформы: в этом случае определен макрос `WORDS_BIGENDIAN`. Тогда, например, для сохранения 32-разрядных беззнаковых чисел (при наличии возможности хранения 32-разрядных чисел в данной архитектуре в заголовочном файле `<stdint.h>`, включенном в стандартную библиотеку начиная с C99, определен тип `uint32_t`) следует определить функцию конвертации следующим образом:

```
#ifdef WORDS_BIGENDIAN
uint32_t bswap32 (uint32_t n)
{
    return ((n & 0xff000000ul) >> 24) |
           ((n & 0x00ff0000ul) >> 8)  |
           ((n & 0x0000ff00ul) << 8)  |
           ((n & 0x000000fful) << 24);
}
#else
uint32_t bswap32 (uint32_t n)
{
    return n;
}
#endif
```

Каждая из 4 битовых конъюнкций выделяет свой байт представления, битовый сдвиг смещает его на нужное место, после чего применяется дизъюнкция для объединения этого в новое число. Функция обратная самой себе, то есть после такого определения ее следует применять и при чтении и при сохранении любого значения типа `uint32_t` на любой платформе.

Это, однако, не снимает проблемы вариаций представления отрицательных чисел (их можно приводить к положительным, но остается риск, что на платформе, где применяется прямой код, не будет корректно прочитано наименьшее отрицательное значение, сохраненное на платформе с прямым кодом), а также переносимость между компьютерами с байтами разной длины. Однако, именно различная "-конечность" является наиболее часто встречающейся ситуацией, которую обязательно следует учитывать.

Вещественные числа

Для сохранения вещественных чисел их можно "перевести" в целые. Во-первых, группа функций (определены в стандарте C99)

```
double      frexp  (double x,      int *exp);
float       frexpf (float  x,      int *exp);
long double frexpl (long double x, int *exp);
```

позволяют разделить число x на порядок `*exp` и мантиссу (возвращаемое значение m), причем $0.5 \leqslant m < 1$. Функции

```
double      ldexp  (double x,      int exp);
float       ldexpf (float x,       int exp);
long double ldexpl (long double x, int exp);
```

выполняют обратное действие можно — добавляют порядок `exp` к числу `x` (умножают x на 2^{exp}, например, если у $6.25 = 0.625 \cdot 2^1$ увеличить порядок на 4, получится $0.625 \cdot 2^5 = 100$). С их помощью можно восстановить число при чтении, а также свести мантиссу $0.5 \leqslant m < 1$ к целому числу — путем добавления порядка, равного числу знаков мантиссы, которое можно сохранить с учетом описанных выше способов. Однако при конвертации между типами и платформами, не поддерживающими стандарт IEEE 754 отсутствие потери точности не гарантируется.

C. Пример изменения стека вызова

Ниже приведена иллюстрация изменения стека программы при вызове, работе и завершении функций. Пример очень условный, позволяет разобрать работу стека программы не опираясь на содержательность выполняемых действий.

При добавлении элемента указатель стека (одиночная стрелка) смещается влево (или "вниз", смотря как ориентировать изображение области памяти, оба варианта используются в программировании). Адреса переменных — смещение относительно указателя стека вправо (вверх). В данном случае точка начала экземпляра

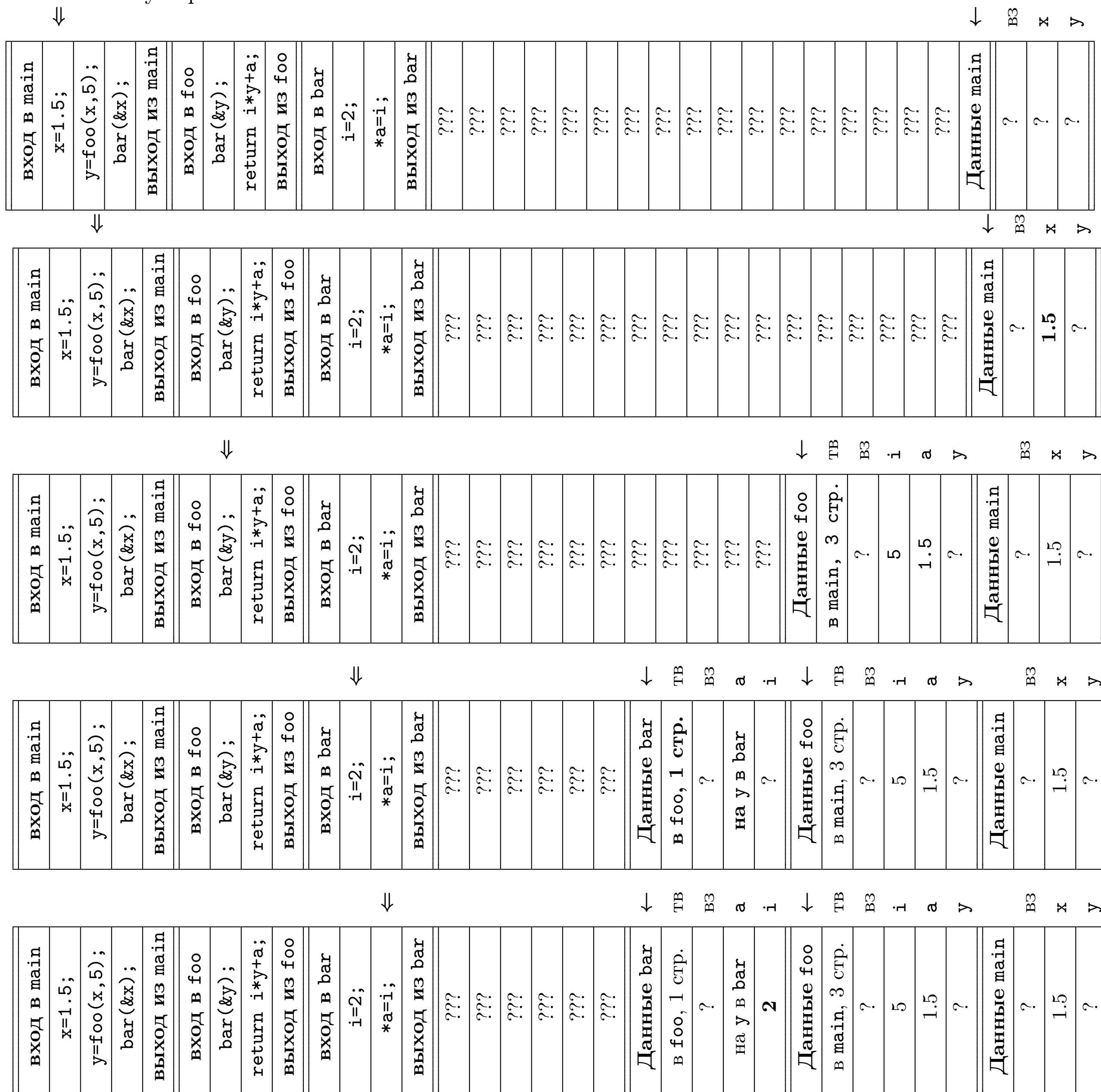
данных функции показана как актуальная ячейка памяти, но это не так: указатель стека указывает на начало данных функции.

Ячейки слева от (ниже) указателя стека считаются свободными: на практике но там могут находиться случайные (мусорные) данные. В частности, именно так при вызове новой функции неинициализированные переменные приобретают значения сформированные из данных ранее завершенной функции непредсказуемым образом.

Имеющиеся функции:

прототип	локальные переменные
`int main();`	`double x, y;`
`double foo(double a, int i);`	`double x; int j;`
`void bar(double *a);`	`int i;`

Над стеком указаны имена локальных параметров и переменных. Обозначения: вз — возвращаемое значение, тв — точка возврата. Двойной стрелкой показан указатель кода. Практическая реализация может отличаться, например, возвращаемое значение может быть реализовано в виде указателя на временную ячейку в вызывающей функции. Также на схеме не показано хранение временных значений при вычислении выражений, однако они тоже могут хранятся в стеке.



вход в main	
x=1.5;	
y=foo(x,5);	
bar(&x);	
выход из main	
вход в foo	
bar(&y);	
return i*y+a;	
выход из foo	
вход в bar	
i=2;	
*a=i;	
выход из bar	⇐
???	
???	
???	
???	
???	
???	
Данные bar	←
в foo, 1 стр.	тв
?	вз
на y в bar	a
2	i
Данные foo	←
в main, 3 стр.	тв
?	вз
5	i
1.5	a
2	y
Данные main	
?	вз
1.5	x
?	y

вход в main	
x=1.5;	
y=foo(x,5);	
bar(&x);	
выход из main	
вход в foo	
bar(&y);	
return i*y+a;	⇐
выход из foo	
вход в bar	
i=2;	
*a=i;	
выход из bar	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
Данные foo	←
в main, 3 стр.	тв
?	вз
5	i
1.5	a
2	y
Данные main	
?	вз
1.5	x
?	y

вход в main	
x=1.5;	
y=foo(x,5);	
bar(&x);	
выход из main	
вход в foo	
bar(&y);	
return i*y+a;	
выход из foo	⇐
вход в bar	
i=2;	
*a=i;	
выход из bar	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
Данные foo	←
в main, 3 стр.	тв
8.5	вз
5	i
1.5	a
2	y
Данные main	
?	вз
1.5	x
?	y

вход в main	
x=1.5;	
y=foo(x,5);	
bar(&x);	⇐
выход из main	
вход в foo	
bar(&y);	
return i*y+a;	
выход из foo	
вход в bar	
i=2;	
*a=i;	
выход из bar	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
Данные main	
?	вз
1.5	x
8.5	y

вход в main	
x=1.5;	
y=foo(x,5);	
bar(&x);	
выход из main	
вход в foo	
bar(&y);	
return i*y+a;	
выход из foo	
вход в bar	
i=2;	⇐
*a=i;	
выход из bar	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
Данные bar	←
в main, 4 стр.	тв
?	вз
на x в main	a
?	i
Данные main	
?	вз
1.5	x
8.5	y

вход в main	
x=1.5;	
y=foo(x,5);	
bar(&x);	
выход из main	
вход в foo	
bar(&y);	
return i*y+a;	
выход из foo	
вход в bar	
i=2;	
*a=i;	⇐
выход из bar	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
Данные bar	←
в main, 4 стр.	тв
?	вз
на x в main	a
2	i
Данные main	
?	вз
1.5	x
8.5	y

вход в main	
x=1.5;	
y=foo(x,5);	
bar(&x);	
выход из main	
вход в foo	
bar(&y);	
return i*y+a;	
выход из foo	
вход в bar	
i=2;	
*a=i;	
выход из bar	⇐
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
???	
Данные bar	←
в main, 4 стр.	тв
?	вз
на x в main	a
2	i
Данные main	
?	вз
2	x
8.5	y

				⇓																											вз	x	y
вход в main	x=1.5;	y=foo(x,5);	bar(&x);	выход из main	вход в foo	bar(&y);	return i*y+a;	выход из foo	вход в bar	i=2;	*a=i;	выход из bar	???	???	???	???	???	???	???	???	???	???	???	???	???	???	???	???	???	Данные main	?	2	8.5

D. Дополнительные возможности языка Си

Целью данного курса является изучение фундаментальных основ программирования и методологии парадигм структурного и процедурно-модульного программирования на примере языка Си. Это не предполагает изложение всех возможностей разных стандартов языка, включая описание типов данных и функций стандартной библиотеки. Детальное описание всех типов данных и функций стандартной библиотеки, как и любой другой библиотеки, следует рассматривать как справочный материал (документацию). Поэтому ряд элементов языка, такие как объединения, регистровые переменные и различные возможности новых стандартов лежат за рамками данного пособия. В этом приложении описываются упомянутые, но не описанные в основном тексте, возможности языка Си.

§D.1. Неполный тип данных

В Си имеется возможность объявлять идентификатор типа данных “структура” без указания полей структуры — то есть объявлять т.н. **неполный тип**, например:

```
struct some_struct;
```

Сделать это можно, например, в заголовочном файле некого модуля (как и всякое объявление), тогда в данном заголовочном файле можно использовать указатель на структуру `some_struct` в качестве параметров функции, например:

Листинг D.1: some_struct.h

```
/* Объявляем структуру как неполный тип */

struct some_struct;

/* Функция создает структуру */
struct some_struct * some_struct_alloc ();

/* Функция модифицирует структуру s */
void foo (struct some_struct *s);

/* Функция использует структуру s как входной параметр */
void bar (const struct some_struct *s);

/* Освобождение памяти структуры
void some_struct_free (const struct some_struct * s);
```

Подобного заголовочного файла достаточно для использования типа данных `some_struct`, так как появляется возможность объявлять и передавать в функции указатель на данную структуру. Действительно, все указатели имеют одинаковый размер, поэтому количество и тип полей для объявления и копирования данной переменной не важны.

Листинг D.2: file.c

```
#include "some_struct.h"

/* ... */

/* Создаем структуру */
```

```
struct some_struct *s = some_struct_alloc ();

/* Проверяем, что структура действительно выделилась в памяти */

if (!s) return ALLOC_ERROR_CODE;

/* Работаем со структурой */

foo(s);
bar(s);
/* ... */

/* Освобождаем память */

some_struct_free (s);
```

Заметим, что работа с указателем на `some_struct` выглядит почти как работа с обычной переменной — инициализация и передача в качестве параметра функций лишь дополняется необходимостью освобождения памяти в конце, операции взятия адреса и разыменования данного указателя не используются. Действительно, такой указатель нельзя разыменовывать (ни с помощью операции разыменования `*`, ни с помощью операции `->`), так как получение самой структуры требует информацию о полях.

В файле исходного кода модуля необходимо объявить поля данной структуры — только после этого становится возможной реализация функций обработки структуры, например:

Листинг D.3: some_struct.c

```
struct some_struct
{
    int n;
    double d;
    /* ... */
}

struct some_struct * some_struct_alloc
{
    return malloc (sizeof (some_struct));
}

void some_struct_free (const struct some_struct * s)
{
    free(s);
}


void foo (struct some_struct *s)
{
        s->n = 0;
        s->d = 0;
        /* ... */
}

void bar (const struct some_struct *s)
{
    printf ("%i␣%le\n", s->n, s->d);
    /* ... */
}

/* ... */
```

При таком подходе детали реализации структуры становятся скрыты от пользователя модуля, у которого нет возможности чтения и изменения полей структуры способами, отличными от функций, предоставляемых модулем. При модификации модуля количество и тип данных полей могут быть изменены без изменения интерфейса модуля.

Заметим, что тип данных FILE стандартной библиотеки может быть реализован как неполный тип: роль функций выделения и освобождения памяти играют, соответственно, fopen и fclose.

§D.2. Макросы

Директива препроцессора #define определяет символический псевдоним не только для произвольной константы, но и вообще для любого программного кода. Строго говоря, препроцессор заменяет все вхождения идентификатора от момента объявления до конца единицы трансляции на заданную последовательность символов, которая уже в дальнейшем обрабатывается компилятором.

Кроме того, данная последовательность символов может иметь параметры — идентификаторы, которые будут заменены препроцессором на указанные значения. Поэтому, в общем случае define объявляет **макрос** — шаблон, в соответствии с которым его вызов заменяется (**"раскрывается"**) препроцессором на код, передаваемый компилятору.

Например, можно объявить макрос возведения в квадрат:

```
#define SQR(x) ((x) * (x))
```

тогда вызов SQR от любого выражения приведет к получению квадрата от этого выражения, например SQR(2), SQR(x+3) и т.д. — идентификатор макроса SQR можно использовать как функцию с одним аргументом.

При этом, в отличие от функций, макросы не имеют привязку к типам данных, соответственно результат SQR(2) будет целым, SQR(2.0) — типа double и т.д. Дополнительным преимуществом макросов над функциями является отсутствие траты времени на вызов функции, в то время как недостатком подобного макроса — двукратное вычисление значения аргумента. Для современных компиляторов с развитыми технологиями оптимизации последние два обстоятельства обычно считаются несущественными.

Следует обратить внимание на отсутствие пробела после имени макроса перед списком аргументов: все, что находится после пробела, считается значением макроса, поэтому директива

```
#define SQR (x) ((x) * (x))
```

является некорректной: вызов SQR (y+2) раскроется в

```
(x) ((x) * (x)) (y+2)
```

вместо

```
((y+2) * (y+2))
```

Наличие или отсутствие пробелов при вызове макроса значения не имеет.

Скобки вокруг аргумента и операции в теле макроса в данном примере также являются существенными. Сравните:

```
#define SQR1(x) x * x

#define SQR2(x) (x) * (x)

#define SQR3(x) ((x) * (x))
```

Выражения

```
n1 = (int) SQR1 (x + y) * 2;
n2 = (int) SQR2 (x + y) * 2;
n3 = (int) SQR3 (x + y) * 2;
```

раскроются, соответственно в

```
n1 = (int) x + y * x + y * 2;
n2 = (int) (x + y) * (x + y) * 2;
n3 = (int) ((x + y) * (x + y)) * 2;
```

В первом случае получен совершенно не тот порядок действий, во втором — приведение типа относится только к первой скобке, а не ко всему квадрату выражения, что также может дать отличный результат (при x=2.5 и y=2 получится n2=36, а n3=40).

Отсутствие точки с запятой в конце объявления также критично: в противном случае она будет частью раскрытия макроса и такую "функцию" нельзя будет использовать в выражении корректно.

Возможно объявление многопараметрических макросов (опять же важно отсутствие пробела в списке формальных параметров макроса), например — типонезависимый максимум двух значений может быть объявлен следующим образом:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

В Си `assert` является именно макросом с единственным параметром-условием, а не функцией, т.к. функция не смогла бы получить информацию о коде условия, имени файла и строке, выводимыми на экран при получении ЛОЖЬ в ходе проверки результата условия: в препроцессор языка встроены макросы `__FILE__` и `__LINE__`, заменяемые, соответственно, на имя файла и номер строки, также в макросе возможно "превращение" параметра в строковую константу, содержащую указанный код с помощью операции `#`. Одной из возможных реализаций макроса `assert` является следующая:

```
#ifdef NDEBUG
#define assert(COND)
#else
#define assert(COND) if (!COND)                                              \
{                                                                            \
   fprintf (stderr, "Assertion␣'%s'␣failed␣at␣'%s':%u", #COND, __FILE__, __LINE__);  \
   exit(1);                                                                  \
}
#endif
```

Здесь сначала проверяется заданность директивы `NDEBUG`: если она задана, макрос определяется как пустой (но с одним аргументом!) и не генерирует исполняемый код, в противном случае макрос заменяется на код проверки условия и вывода сообщения об ошибке, если условие ложное.

Все директивы препроцессора должны быть записаны в одну строку. Символ `\` может быть использован для "отмены" перевода строки и превращения его в обычный пробел. Это позволяет записывать визуально многострочные директивы, в частности макросы.

Также в макросе можно использовать конкатенацию `##`, позволяющую объединить значение токена-параметр и другого токена в единый токен. Например, следующий макрос

```
#define DEF3(type,var) \
    type var##1;       \
    type var##2;       \
    type var##3;
```

Объявляет 3 переменных с префиксом `var` и индексами 1, 2 и 3, так как

`DEF3(int, x)`

раскрывается в

```
    int x1;
    int x2;
    int x3;
```

В некоторых случаях макросы являются незаменим инструментом, но обычно их использование не рекомендуется из-за сложности прочтения кода и отладки при наличии ошибок (компилятор сообщит о синтаксической ошибке в строке вызова макроса, но не в самом макросе; поиск семантических ошибок же может быть вообще крайне затруднителен). Большинство компиляторов имеют возможность применить препроцессор к файлу исходного кода без компиляции — это позволяет увидеть в какой именно код раскрывается макрос, что может облегчить поиск ошибок.

§D.3. Классы хранения: `extern` и `static`

Как уже было отмечено, определение переменной и функции — создание ячейки памяти, содержащей значение переменной или код функции — автоматически является объявлением идентификатора переменной или функции с заданными свойствами (типом данных или возвращаемым значение и количеством и типом аргументов соответственно).

Прототипы функций являются объявлением идентификатора функций без определения — тело функции должно быть включено в программу на стадии компоновки (линкования). При этом одна и та же функция может быть объявлена неограниченное количество раз в различных единицах трансляции (модулях), обычно это достигается путем включения прототипа в заголовочном файле модуля или библиотеки. Тело функции должно встречаться в программе ровно один раз.

Аналогом "прототипа" глобальной переменной является является объявление переменной с ключевым словом `extern`, например

```
extern int n;
```

является объявлением переменной `n` типа `int`, но не является определением, т.е. не выделяет ячейку памяти. Такую конструкцию можно включать, например, в заголовочный файл для создания глобальной переменной, разделяемой между модулями. При этом в исходном файле модуля та же самая переменная должна быть определена (уже без слова `extern`) и ровно один раз — аналогично телу функции. Например,

```
/* Объявление */

extern int module_global_var;
```

```
#include "module.h"

/* ... */

/* Определение */

int  module_global_var;
```

Данную переменную можно использовать во всех файлах исходного кода, после подключения соответствующего заголовочного файла

```
#include "module.h"

/* Использование */

module_global_var = /* ... */;
```

Разумеется, без острой необходимости использование глобальных переменных категорически не рекомендуется. Объявление глобальной переменной без ключевого слова `extern` в разделяемом между несколькими единицами трансляции заголовочном файле приведет к ошибке многократного определения одного и того же символа (идентификатора, с которым связана ячейка памяти на уровне исполняемого файла), как и включение в такой заголовочный файл тела функции.

Переменные, определяемые внутри функции без каких-либо ключевых слов, являются локальными для каждого вызова данной функции — ее значение не сохраняется между вызовами функции. Альтернативой является объявление переменной внутри функции с ключевым словом `static`: такая переменная сохраняет свое значение между вызовами функции. Ее начальное значение должно быть обязательно проинициализировано, оно будет получено при первом вызове функции. Например,

```
#include <stdio.h>

void foo()
{
        int n1 = 0;
        static int n2 = 0;

        ++n1;
        ++n2;

        printf("n1␣=␣%i␣n2␣=␣%i\n", n1, n2);
}


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

выведет

```
n1 = 1, n2 = 1
n1 = 1, n2 = 2
n1 = 1, n2 = 3
```

В глобальной области видимости ключевое слово `static` имеет несколько другой смысл: объявляемая функция (или глобальная переменная) становится недоступной извне данной единицы трансляции. Таким образом рекомендуется объявлять внутренние (неотражаемые в интерфейсе) функции модуля или библиотеки, чтобы их нельзя было вызвать (случайно или со злым умыслом) пользователю данного модуля или библиотеки. Например,

```
static void foo ();
```

или сразу с определением

```
static void foo ()
{
    /* ... */
}
```

Это не лишено смысла: действительно, даже если в заголовочный файл модуля или библиотеки прототип данной функции не включен, ничто не мешает добавить этот прототип в любое место глобальной области видимости единицы трансляции, например строку вида

```
void foo ();
```

и вызвать данную функцию (или некорректно изменить значение внутренней глобальной переменной) — заголовочные файлы являются инструментом удобства программирования, а не средством изоляции модулей. Кроме того, использование `static` позволяет избежать конфликта имен символов при компоновке, если внутренние функции различных модулей имеют одинаковые имена. *Поэтому, хорошей практикой является обязательное использование* `static` *для всех внутренних функций и глобальных переменных модуля, если таковые имеются.*

Ключевые слова `extern` и `static` определяют классы хранения символов — внешний, по отношению к данному модулю или внутренний по отношению к данному модулю (неразделяемый для глобальных идентификаторов, общий для всех вызовов функции для локальных идентификаторов) соответственно и являются взаимоисключающими.

§D.4. Встраиваемые функции

Начиная со стандарта C99 в языке возможно использование **встраиваемых функций** — функций, не вызываемых путем передачи управления коду функции, а чей код встраивается непосредственно в место вызова функции. Это позволяет сократить время вызова функций в т.ч. за счет отказа от передачи управления и копирования входных и выходных данных.

Для создании встраиваемых функций можно использовать следующую схему.

- В *заголовочном файле* указать тело (*определение*) функции, предварив ключевым словом `inline`, например:

```
inline foo ()
{
/* код функции */
}
```

- В *одном из файлов исходного кода* указать прототип (*ообъявление*) функции, предварив парой ключевых словом `extern inline`, например:

```
extern inline void foo ();
```

Действительно, код встраиваемой функции действительно должен быть доступен всем функциям всех единиц трансляции, которые ее используют.

Встраиваемые функции похожи на макросы в том смысле, что встраивают код в тело функции. Однако между ними есть существенные отличия:

- Формальные параметры и локальные переменные встраиваемых функций подчиняются правилам для области видимости этой функции, что сокращает число возможных ошибок.
- Формальные параметры и возвращаемые значение встраиваемых функций имеют тип данных, поэтому в отличие от, например, одного макроса нахождения наибольшего из двух значений, для всех типов данных, для которых определена операция сравнения,

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

потребуется определить ряд встраиваемых функций, например

```
extern inline int max (int x, int y)
{
    return x > y ? x : y;
}
extern inline double fmax (double x, double y);
{
    return x > y ? x : y;
}

extern inline int max (int x, int y);
extern inline double fmax (double x, double y);
```

и т.д.

- Во встраиваемых функциях неприменимы средства метапрограммирования (конкатенация, преобразования кода в строку).
- Встраиваемые функции могут быть рекурсивными.

Заметим, что не во всех случаях (например, в случае рекурсивной функции) компилятор имеет возможность и вообще обязан делать встраиваемый код. Скорее, объявление и определение встраиваемых функций следует воспринимать как совет и предоставление возможности компилятору создать более оптимальный код за счет встраивания тела функции в код другой функции.

Встраиваемые функции не могут заменить макросы, но т.к. их написание, восприятие и отладка ничем не отличается от обычных функций и не сопряжена с высоким риском появления труднообнаружимых ошибок, считается, что при наличии возможности, следует использовать встраиваемые функции, а не макросы.

В то же время не следует стремиться сделать все функции встраиваемыми, ограничиваясь лишь небольшими и часто вызываемыми функциями, для которых время работы может быть сопоставимо со временем вызова функции. Злоупотребление встраиваемыми функциями увеличивает время компиляции программы (в т.ч. при модификации и исправлении ошибок), усложняя разработку, а также увеличивает размер исполняемого файла, что может, наоборот, привести к замедлению скорости работы программы. Встраивание долго работающих, в т.ч. за счет вызова операций ввода-вывода, функций, не рекомендуется.

§D.5. Функции с переменным числом аргументов

Такие функции стандартной библиотеки, как `printf` и `scanf`, являются функциям с переменным числом аргументов, называемые также **вариативными функциями**. При задании прототипа таких функций используется пунктуатор '`...`', например:

```
int fprintf (FILE *stream, const char *format, ...);
```

У данной функции два обязательных аргумента — типа `FILE *` и типа `const char *` соответственно, после которых может указываться произвольное (в том числе и нулевое) количество аргументов любых (в том числе различных) типов.

Для создания таких функций следует использовать средства, предоставляемые в заголовочном файле стандартной библиотеки `stdarg.h`. Это

- тип данных `va_list`, который позволяет хранить и извлекать аргументы из списка;
- макрос `va_start`, инициализирующий начало извлечения аргументов, имеет два аргумента: список типа `va_list` и последний обязательный параметр функции;
- макрос `va_arg`, извлекающий очередной аргумент из списка, имеет два аргумента: список типа `va_list` и тип извлекаемого параметра;
- макрос `va_end`, завершающий извлечение аргументов, имеет один аргумент: список типа `va_list`.

Таким образом, процесс доступа к аргументам списка состоит в требуемом количестве раз последовательных вызовов `va_arg`. Заметим, что

- вызов `va_start` и `va_end` являются обязательными, без вызова `va_end` результат дальнейшей работы программы непредсказуем;
- функция с переменным числом аргументов должна иметь хотя бы один обязательный аргумент;
- определить тип и количество переданных аргументов не представляется возможным (эту информацию приходится передавать при использовании функции с переменным числом аргументов "вручную").

Именно из-за последнего обстоятельства использование функций с переменным числом аргументов не рекомендуется: если для функций стандартной библиотеки в компилятор можно встроить проверку соблюдения количества и типа аргументов (для функций ввода-вывода они определяются количеством и парой длина-спецификатор полей — в современных компиляторах имеется необходимый режим предупреждений), то для пользовательских функций остается лишь “надеяться” на правильность кода и отслеживать случаи некорректного и неопределенного поведения во время тестовых запусков.

Пример реализации функции с переменным числом аргументов: вычисление среднего арифметического списка.

```
#include <stdarg.h>

/* Вычисляет среднее арифметическое из n чисел типа double */

double dbl_average (size_t n, ...)
{
        double sum = 0.0;
        size_t i;

        /* список аргументов */
        va_list va;

        /* инициализация списка аргументов */
        va_start (va, n);

        /* сумма последовательности */
        for(i = 0; i < n; ++i)
                sum += va_arg(va, double);

        /* закрытие списка аргументов */
        va_end(va);

        return sum / n;
}

/* ... */

int main (void)
{
        double mean;

        mean = dbl_average(5, -1.0, 2.5, 3.5, -5.1, 1e-5);

        printf("Average␣is␣%lf\n", mean);

        return 0;
}
```

Заметим, что важно не только строгое соответствие количества, но строгое соответствие типа аргументов: если в качестве аргумента данной функции передать значение типа `float` или `int` (например, указав в первом аргументе списка `-1f` или `-1` вместо `-1.0`), результат будет непредсказуем.

E. Полный список вопросов для самопроверки

1. В чем различие понятий “информация” и “данные”? 8
2. Приведите примеры данных. 8
3. Что такое информатика? 8
4. Что такое компьютер? 8
5. Что такое алгоритм (интуитивно)? 8
6. Приведите примеры алгоритмов, отличных от упомянутых в тексте. 8
7. Какими свойствами должен обладать “хороший” алгоритм (интуитивно)? 8
8. Что такое входные и выходные данные алгоритма? 8

9. Приведите примеры алгоритмов, незавершающихся за конечное число шагов для некоторых, но не всех входных данных. 8
10. Что такое черный ящик? 8
11. В чем различие понятий “алгоритм” и “компьютерная программа”? 8
12. Что такое алгоритмически неразрешимая задача? 9
13. В чем разница между алгоритмически неразрешимой и трансвычислительной задачей? 9
14. Какое значение для науки и практики имеет существование алгоритмически неразрешимых и трансвычислительных задач? 9
15. Что такое цифровой компьютер? 11
16. Дайте определение цифры в позиционной системе счисления. 11
17. Как именно зависит значение цифры от позиции в записи? 11
18. Приведите пример перевода числа из восьмеричной системы счисления в троичную, как его бы записывал человек, для которого троичная система счисления — основная (число ||| записывается как 10). 11
19. Что такое бит? Что такое байт? 11
20. Постройте и поясните схему ЭВМ. 13
21. Приведите примеры компьютерных комплектующих и периферийных устройств (принтер, жесткий диск, карта памяти, монитор, мышь, центральный процессор, оперативная память). Поясните, к каким узлам схемы ЭВМ они относятся. Замечание. Материнская плата содержит в себе различные элементы, включая шину и ПЗУ. 13
22. Каковы положения архитектуры фон Неймана? 13
23. Каким образом кодируются команды и данные в архитектуре фон Неймана? 13
24. Каковы свойства памяти в архитектуре фон Неймана? 13
25. Что такое архитектура компьютера? 13
26. Опишите, что именно делает программа, изображенная на рис. 1.3: какое именно преобразование входных данных в выходные производит алгоритм. 13
27. Что такое процесс? 16
28. Что такое виртуальное адресное пространство процесса? 16
29. Зачем нужно виртуальное адресное пространство процесса? 16
30. Что такое функция? 16
31. Как осуществляется вызов функции? 17
32. Какую задачу решает аппарат функций? 17
33. Что такое библиотеки функций? 17
34. Кто является поставщиком библиотек функций? 17
35. Что такое тип данных? 22
36. Зачем нужны типы данных? 22
37. Что определяет тип данных? 22
38. Что такое интерфейс? 22
39. Что такое абстракция, уровень абстракции? 22
40. Какие уровни абстракции можно выделить? 22
41. С какими проблемами сталкивается программист при написании программы на машинных кодах? 23
42. Что такое язык ассемблера и какие задачи он решает? 23
43. Что такое язык программирования высокого уровня? 23
44. Что такое транслятор и какие виды трансляторов бывают? 23
45. Чем отличается компилятор от интерпретатора? 23
46. Чем компиляция отличается от компоновки? 23
47. Какие виды языков по их происхождению бывают? 23
48. Что такое синтаксис? 23
49. Что такое семантика? 23
50. Как классифицируются языки программирования? 23
51. Что такое отладчик и зачем он нужен? 23
52. Что такое интегрированная среда разработки? 23
53. Какие этапы разработки программного обеспечения выделяют? 23
54. Что такое технология программирования? 23
55. Как представляются целые числа в компьютере? 29
56. Как моделируются вещественные числа в компьютере? 29
57. Что такое числа с плавающей точкой? 30
58. Какие ошибки могут возникнуть при работе с целыми числами? 30
59. Какие ошибки могут возникнуть при работе с числами с плавающей точкой? 30

60. Что такое код символа? . . . 30
61. Что такое таблица символов (кодовая страница)? Что от нее зависит? . . . 30
62. Что такое шрифт? . . . 30
63. Чем отличаются Си-строки от паскалевских строк? Какие преимущества и недостатки у этих способов представления строк? . . . 30
64. Поясните различие между строкой, содержащий символ “цифра 0”, символом “цифра 0” и числом 0. 30
65. Перебирая всевозможные комбинации значений логических переменных, докажите приведенные логические законы. . . . 30
66. Каково место языка Си в общей классификации языков программирования? . . . 31
67. Каковы преимущества и недостатки языка Си? . . . 31
68. Какие базовые типы данных есть в Си? . . . 33
69. Каковы свойства типа `char`? . . . 33
70. Как представляются логические значения в Си? . . . 33
71. Из чего состоит алфавит языка Си? . . . 33
72. Перечислите элементы синтаксиса языка Си. . . . 33
73. Зачем нужно понимание норм синтаксиса языка программирования? . . . 33
74. Что такое токен? . . . 41
75. Как разделяются токены в языке Си? . . . 41
76. В чем отличие понятий “токен” и “слово”? . . . 41
77. Какие виды токенов есть в Си? . . . 41
78. Что такое ключевые слова? . . . 41
79. Что такое идентификатор? . . . 41
80. Что такое форма Бэкуса-Наура и зачем она нужна? . . . 41
81. Различает ли язык Си строчные и заглавные буквы алфавита? . . . 41
82. Что такое константы? . . . 41
83. Чем отличаются константы от переменных? . . . 41
84. Какие виды констант в Си есть? . . . 41
85. Как задаются символьные константы в Си? . . . 41
86. Как задаются целочисленные константы в Си? . . . 41
87. Как задаются константы-числа с плавающей точной в Си? . . . 41
88. Почему шестнадцатеричные константы начинаются в `0x`, а не, например, просто `x` или с числа 16? 41
89. Как задаются строковые константы в Си? . . . 41
90. В чем различие между токенами `0`, `'0'`, `"0"`, `'\0'`, , `0l`, `0.0`? . . . 41
91. В чем различие между токенами `1`, `'1'`, `"1"`, `'\1'`, `1l`, `1.`, `.1`, `1f`, `1.0l`? . . . 41
92. Что такое операция? . . . 43
93. Какие виды операций с точки зрения синтаксиса есть в Си? . . . 43
94. Что такое выражение? . . . 43
95. Всегда ли выражение представляет собой значение? . . . 43
96. Как определяется тип данных выражения? . . . 43
97. Что такое void-выражение? . . . 43
98. Что такое приоритет и порядок операций? . . . 43
99. Приведите примеры операций с разным приоритетом, с разным порядком ассоциативности. . . . 43
100. Что задает объявление переменой? . . . 48
101. Что такое инициализатор переменных? . . . 48
102. Что такое прототип функции, каков его синтаксис? . . . 48
103. Как компилятор использует прототип функции? . . . 48
104. Как программист использует прототипы чужих и библиотечных функции? . . . 48
105. Что такое void-функция? . . . 48
106. Подумайте, каков может быть смысл создания void-функций? . . . 48
107. Что такое оператор? . . . 54
108. Что такое оператор-выражение, какие его особенности? . . . 54
109. Что такое оператор return? . . . 54
110. Что такое пустой оператор? . . . 54
111. Что такое блок операторов . . . 54
112. Что представляет собой список объявлений переменных в операторном блоке? Где он может располагаться? . . . 54
113. Каков синтаксис условного оператора? . . . 55
114. Что может выступать в качестве условия в Си? . . . 55
115. Как Си трактует значение выражения-условия? . . . 55

116. Как работает условный оператор? 55
117. Какие есть операторы цикла в Си? Каков синтаксис каждого из них? 55
118. Как работают операторы цикла в Си? 55
119. Что такое тело цикла? 55
120. Что будет, если в цикле `for` оставить пустым инициализатор? Условие? Итератор? 55
121. Как прервать исполнение текущей итерации цикла? 55
122. Как прервать исполнение цикла? 55
123. Почему использование операторов `continue` и `break` не рекомендуется? 55
124. Что такое бесконечный цикл? 55
125. Чем чреват бесконечный цикл? 55
126. Что такое недостижимый код? 56
127. Приведите примеры недостижимого кода. Придумайте ситуацию, когда код является недостижим, отличную от кода, находящегося за `return`. 56
128. Как синтаксически записывается функция? 56
129. Чем отличаются заголовок и прототип функции? Почему это отличие существенно? 56
130. Является ли прототип функции объявлением? Определением? 56
131. Является ли заголовок функции с ее телом объявлением? Определением? 56
132. Является ли объявление переменных в деклараторе, определением? Объявлением? 56
133. Что делать, чтобы функция могла изменить данные вызывающей функции? 56
134. Как написать функцию, имеющую более одного выходного параметра? 56
135. Что такое неявное декларирование функции? 56
136. Что такое глобальные переменные? 56
137. Что такое объявление и определение идентификаторов? 56
138. Что такое комментарий? 57
139. Как записывается комментарий? 57
140. Как используется комментарий? 57
141. Что такое область видимости идентификатора? 58
142. Как определяется область видимости идентификатора? 58
143. Что такое перекрытие идентификатора? 58
144. Как связаны одноименные идентификаторы в разных областях видимости? 58
145. Зачем нужны заголовочные файлы библиотек? 59
146. Что представляют собой заголовочные файлы? 59
147. Что такое препроцессор и его директивы? 59
148. Что делает директива `#include` и какой у нее синтаксис? 59
149. Что делает директива `#define` и какой у нее синтаксис? 59
150. Что такое символическая константа? 60
151. Какими буквами принято записывать идентификаторы переменных? Функций? Символических констант? 60
152. Что такое перечисляемый тип? 61
153. Что декларирует `enum`? 61
154. Что такое анонимные перечисления и как они могут использоваться? 61
155. Чем отличаются константы-перечисления от символических констант? 61
156. Какие арифметические операции есть в Си? 64
157. Чем отличается постфиксный и префиксный инкремент (декремент)? 64
158. Как определяется тип результата арифметических операций? 64
159. Какую особенность имеет операция деления в Си? 64
160. Зачем нужен унарный плюс? 64
161. Что такое побочный эффект операции? 64
162. Как записывается операция присваивания в Си и что она делает? 65
163. Чем отличается присваивание от инициализатора? 65
164. Что такое операции арифметического присваивания, как они записываются, как работают, каковы их свойства (порядок, приоритет, возвращаемое значение)? 65
165. Как представляются логические значения в Си? 67
166. Какие операции сравнения есть в Си и как они работают? 67
167. Чем отличается операция присваивания от операции равенство? 67
168. Почему нельзя сравнивать знаковые и беззнаковые целые числа? 67
169. Какие логические операции есть в Си? 67
170. Как записывается и работает тернарная условная операция? 67
171. Как корректно проверить равенство значений трех переменных? 67

172. Что такое операции битовой арифметики? 68
173. Чем отличается битовая конъюнкция от логической? 69
174. Что такое битовый сдвиг? 69
175. Как проверить значение отдельного бита? 69
176. Как установить значение отдельного бита? Объясните как это работает на примере. 69
177. Какие операции обеспечивают присваивание совмещенное с битовыми операциями? 69
178. Что такое указатель? 75
179. Как объявляется указатель? 75
180. Какой смысл имеет тип указателя? 75
181. Как объявляется нетипизированный указатель? 75
182. Какие операции можно выполнять с указателями? 75
183. Как получить адрес переменной (указатель на переменную)? 75
184. Как изменить значение данных по адресу (указателю)? 75
185. Что такое нулевой указатель? 75
186. Какие операции допустимы для нетипизированного указателя? 75
187. Какие ошибки можно допустить при работе с нетипизированными указателями? 75
188. Какие типичные для указателей операции недопустимы для нетипизированного указателя? . . . 75
189. Что делает ключевое слово const? 76
190. Чем отличаются неизменяемые данные от констант? 77
191. Чем отличается указатель на константу от константного указателя? 77
192. Как объявить константный указатель на константу? 77
193. Что такое главная функция? 77
194. Какой прототип может быть у главное функции? 77
195. Каковы недостатки функций форматированного ввода и вывода стандартной библиотеки Си? . . 82
196. Какая функция отвечает за форматированный вывод? Ввод? 82
197. Что такое формат и поле? 82
198. Как определяется тип данных поля? 82
199. За что в строке формата отвечают спецификатор? Длина? Флаги? Точность? Ширина? 82
200. Почему в `scanf` необходимо использовать указатели? 83
201. Что будет, если в `scanf` указать переменные, а не указатели? 83
202. Какие ошибки можно допустить при работе с `printf` и `scanf`? 83
203. Что такое неявное определение типа? 85
204. Когда происходит неявное приведение типа? 85
205. Как осуществляется явное определение типа? 85
206. Что делает операция sizeof? 85
207. Как определить размер (длину) типа данных? Конкретных данных? 85
208. Что такое void-выражение, R-value и L-value? 85
209. Приведите примеры выражений L-value. 86
210. Как вычисляются выражения? 88
211. Какие два процесса производятся при вычислении значения выражения? 88
212. Как регламентирован порядок при вычислении выражения? 88
213. Что такое укороченная оценка логических выражений? 88
214. Применима ли укороченная оценка к битовой арифметике? 88
215. Что такое побочный эффект? 88
216. Что такое неопределенное поведение? 90
217. Приведите примеры неопределенного поведения. 90
218. Чем чревато использование значений неинциализированных переменных? 90
219. Что такое стек как способ организации данных? 92
220. Что такое очередь как способ организации данных? 92
221. Что такое стек вызовов? 92
222. Что содержится в стеке вызовов? 92
223. Когда функция попадает в стек вызовов? 92
224. Когда функция исключается из стека вызовов? 92
225. Что такое стек программы? 92
226. Что хранится в стеке программы? 92
227. Как организовано хранение данных функций в стеке программы? 92
228. Опишите, как происходит процесс вызова функции и ее завершение в Си. 92
229. Что в программировании понимается понимается под рекурсией? 95
230. Что такое рекурсивная функция? 95

231. Как классифицируют рекурсивные алгоритмы? Рекурсивные функции? 95
232. Каковы преимущества и недостатки рекурсивных функций? 95
233. Как организуется выход из рекурсии? 95
234. Что произойдет, если будет запущенна функция с бесконечной рекурсией? 95
235. Какая разница между бесконечным циклом и бесконечной рекурсией? 95
236. Что такое ошибка переполнения стека? 95
237. В каких ситуациях может возникнуть ошибка переполнения стека? Какая ситуация более вероятна? 95
238. В каких случаях следует реализовывать рекурсивный алгоритм? 95
239. Что такое массив? 98
240. Что такое элемент массива? 98
241. Что такое индекс элемента массива? 98
242. Как объявляется одномерный массив в Си? 98
243. Как инициализировать значение элементов массива в Си? 98
244. Как осуществляется доступ к элементам массива в Си? 98
245. Что возвращает операция `[]`? 98
246. Как нумеруются элементы массива в Си? 98
247. Как проверяются границы массива при доступе к элементам в Си? 98
248. Что может произойти при попытке доступа к элементу за границами массива в Си? 98
249. Что общего и в чем разница между указателем и одномерным массивом? 99
250. Указатель какого типа является совместимым с одномерным массивом элементов заданного типа? 99
251. Как вычисляется адрес элемента массива? 99
252. Как объявляются и инициализируются многомерные массивы в Си? 100
253. Как располагаются в памяти элементы многомерных массивов в Си? 100
254. Какой тип указателя совместим с многомерным массивом в Си? 100
255. Чем отличается указатель на массив от массива указателей? 100
256. Как объявить двумерный массив указателей на `int`? 100
257. Как объявить указатель на двумерный массив указателей на `int`? 100
258. Как передать массив в качестве аргумента функции? 102
259. Какая разница между передачей массива и указателя на нулевой элемент массива в качестве аргумента функции? 102
260. Как узнать число элементов массива-аргумента в функции? 102
261. В каких случаях при передаче массива (указателя на нулевой элемент) в функцию следует использовать `const`? 102
262. Как хранятся строки в Си? 104
263. Какие функции для работы со строками существуют в стандартной библиотеке языка Си? 104
264. Чем отличаются строки и массивы символов? 104
265. Что такое строковый буфер? 104
266. Что такое переполнение строкового буфера и к каким последствиям оно может привести? 104
267. Что такое нуль-терминированная строка? 104
268. К каким последствиям может привести отсутствие символа с кодом 0 в строке? 104
269. Что делает функция `puts`? 105
270. Как выводить строки с помощью `printf`? 105
271. Что делает функция `gets`? 105
272. Почему функция `gets` считается “запрещенной”? 105
273. Как вводить строки с помощью `scanf`? 105
274. Что следует указывать в качестве строкового поля в `scanf`? 105
275. Чем лучше использование `scanf` вместо `gets`? 105
276. Можно ли использовать `%s` без ограничения числа вводимых байтов в `scanf`? 105
277. Что делает функция `memcpy`? 107
278. Какие ошибки могут быть допущены при работе с функцией `memcpy`? 107
279. Что такое структура (тип данных) в Си? 109
280. Для чего используются структуры (тип данных)? 109
281. Как объявляются структуры в Си? 109
282. Что такое поля структуры в Си? 109
283. Как осуществляется доступ к полям структур в Си? 109
284. Как можно и как нужно передавать структуры в функции и почему? 109
285. Может ли существовать массив структур? 109
286. Как объявить структуру так, чтобы при объявлении переменных не нужно было указывать ключевое слово `struct`? 109

287. Перечислите операции, результатом которых являются значения L-value. 110
288. Что такое функциональный указатель? 112
289. Какую информацию несет тип данных “функциональный указатель”? Сам функциональный указатель? 112
290. Каков синтаксис объявления функционального указателя? 112
291. Чем отличается объявление функционального указателя от прототипа функции, возвращающей указатель? 112
292. Какие операции допустимы для функциональных указателей? 112
293. Как записать `typedef`-псевдоним функционального указателя? 112
294. Что является результатом разыменования функционального указателя? 112
295. Что является результатом взятия адреса идентификатора функции? 112
296. Что такое стандартные потоки ввода и вывода? 113
297. Откуда по умолчанию осуществляется стандартный ввод? 113
298. Куда по умолчанию осуществляется стандартный вывод? 113
299. Как перенаправить стандартный ввод из файла? 113
300. Как перенаправить стандартный вывод в файл? 113
301. С какими рисками сопряжен стандартный вывод в файл? 113
302. Зачем нужно открытие файла? 114
303. Какой функцией осуществляется открытие файла? 115
304. Зачем нужно закрытие файла? 115
305. Какой функцией осуществляется закрытие файла? 115
306. Что такое тип данных `FILE`? 115
307. Какие есть режимы открытия файла? 115
308. Что такое константа `EOF`? 115
309. С какими рисками сопряжено открытие файла для записи? 115
310. В каких режимах открываются стандартные потоки ввода и вывода? 115
311. Как перенаправить форматированный ввод и вывод в файл? 115
312. Как корректно проверить достижение конца файла при форматированном вводе? 115
313. Что делает функция `feof`? 115
314. Каким файловым переменным соответствуют стандартные потоки ввода, вывода и сообщений об ошибках? 117
315. Какой тип данных у `stdin`? `stdout`? `stderr`? 117
316. Зачем нужен отдельный поток сообщений об ошибках? 117
317. Как перенаправить поток сообщений об ошибках и поток вывода в разные файлы? 117
318. Как перенаправить поток сообщений об ошибках и поток вывода в один файл? 117
319. Что делает функция `perror`? 117
320. Что такое `errno`? 117
321. Какие есть источники ошибок при файловых операциях? 117
322. Какая функции осуществляют ввод символа (байта) из файла? 118
323. Какая функции осуществляют вывод символа (байта) файл? 118
324. Почему функция `fgetc` возвращает `int`, а `char`? 118
325. Может ли константа `EOF` иметь значение 0? 1? 118
326. Какое значение может быть у константы `EOF`? 118
327. Как корректно разделить достижение конца файла и ошибку чтения при символьном вводе? 118
328. Какие функции осуществляют ввод и вывод строк? 119
329. Чем отличаются функции `fgets` и `gets`? 119
330. Чем отличаются функции `fputs` и `puts`? 119
331. Какие функции осуществляют файловый ввод и вывод блоков памяти? 120
332. Какие функции читают и перемещают указатель позиции файла? 120
333. Почему использовать файл для произвольного доступа к данным не рекомендуется? 120
334. Какие сложности имеются при работе с двоичными файлами? 120
335. С помощью каких функций можно проводить преобразование числовых данных в строковые и наоборот? 121
336. Почему функция `sprinf` небезопасна? 121
337. Имеет ли функция `sprinf` безопасный аналог? 121
338. Что такое параметры командной строки? 122
339. Как разделяются параметры командой стоки? 122
340. Как используются параметры командной строки? 122
341. Как прочитать параметры командой строки в функции `main`? 122

342. Какие параметры может иметь функция `main`? . . . 122
343. Какой критике подвергается оператор перехода? . . . 123
344. Какова альтернатива оператору перехода? . . . 123
345. Что такое парадигма программирования? . . . 124
346. Зачем нужны парадигмы программирования? . . . 124
347. Что такое блок кода? . . . 124
348. Что такое структура управления? . . . 124
349. Что такое структурное программирование? . . . 124
350. В чем состоит теорема о структурном программировании? . . . 124
351. Что такое процедурное программирование? . . . 125
352. Какие цели преследует процедурное программирование? . . . 125
353. Что такое модуль? . . . 131
354. Каковы цели и преимущества разделения программы на модули? . . . 131
355. Чем отличаются модули и библиотеки? . . . 131
356. Из каких двух частей (сторон) концептуально состоит модуль? . . . 132
357. Что такое сокрытие деталей реализации? . . . 132
358. Что такое модульное программирование? . . . 132
359. Как реализуется концепция модульного программирования в Си? . . . 132
360. Что такое раздельная компиляция? . . . 132
361. Что такое единица трансляции? . . . 132
362. Что такое объектный модуль? . . . 132
363. Как связаны заголовочные файлы и раздельная компиляция? . . . 132
364. Какие преимущества дает раздельная компиляция? . . . 132
365. Как связаны заголовочные файлы и модули? . . . 132
366. Какую задачу решает компоновщик? . . . 132
367. Что может и что не должно находиться в заголовочном файле? . . . 132
368. Что следует помещать в заголовочный файл? . . . 132
369. Какие ошибки компоновки могут возникнуть при неправильном составлении заголовочного файла и как их избежать? . . . 132
370. Что такое защита от двойного подключения заголовочного файла и как она организуется? . . . 132
371. Что делают директивы препроцессора `#ifdef`/`#ifndef`? . . . 132
372. Какими правилами следует руководствоваться при разделении программы отдельные файлы исходного кода и при составлении заголовочных файлов? . . . 132
373. Какими принципами следует руководствоваться при создании функции? . . . 139
374. Чем следует руководствоваться при написании прототипа функции и документации к нему? . . . 139
375. Чем отличаются глобальные переменные, локальные переменные и формальные параметры? В смысле синтаксиса? В смысле семантики? . . . 139
376. Укажите базовые правила форматирования программ. . . . 139
377. Куда следует писать более короткий блок условного оператора и почему? . . . 139
378. Приведите примеры избыточных и ничего не делающих конструкций в языке Си. . . . 139
379. Что такое тестирование программного обеспечения? . . . 142
380. Чем определяется качество программного продукта? . . . 142
381. Что такое модульное тестирование? . . . 142
382. Что такое тестирование черного, белого и серого ящика? Какие у них преимущества и недостатки? 142
383. Что такое тестирование производительности? . . . 142
384. Что такое профилирование кода? . . . 142
385. Что обеспечивает и что гарантирует правильно проведенное тестирование программного обеспечения? . . . 142
386. Что такое верификация программного обеспечения? . . . 142
387. Какова идея логики Хоара? . . . 142
388. Что такое тройка Хоара? . . . 142
389. Для чего и как применяется логика Хоара? . . . 142
390. Какой аппарат языка программирования Си предназначен для проверки условий логики Хоара? 142
391. Каков синтаксис макроса `assert`? . . . 142
392. Как отключить проверку условий `assert` во всей программе? . . . 142
393. Какие ограничения должны быть наложены на выражения, применяемые в качестве аргумента макроса `assert` . . . 142
394. Каковы недостатки статических массивов? . . . 146
395. Каковы преимущества динамических массивов? . . . 146

396. Каковы недостатки динамических массивов? 146
397. С помощью каких функций стандартной библиотеки выделяются блоки памяти из кучи? 146
398. Что общего между блоком памяти, выделенным с помощью `malloc`, и статическим массивом? . . 146
399. Каковы различия между блоком памяти, выделенным с помощью `malloc`, и статическим массивом? 146
400. Почему важно освобождение выделенного блока памяти по окончании работы с ним? 146
401. Что такое “утечка памяти”? 146
402. Как работает функция `realloc`? 146
403. Какие арифметические операции применимы к указателям? 149
404. Каков результат применения арифметических операций к указателям? 149
405. Что означает прибавление целого числа к указателю? 149
406. Что означает разность двух указателей? 149
407. Что показывают операции сравнения двух указателей? 149
408. Какие ограничения накладываются на применение арифметики и сравнения указателей? 149
409. Как осуществить проход массива с помощью арифметики указателей? 149
410. Почему арифметика указателей считается более быстрой, чем работа с индексами? 149