Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Тверской государственный университет»

И.С. СОЛДАТЕНКО

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ

Учебное пособие

ТВЕРЬ 2017

УДК 004.438С++(075.8)
ББК 3973.2-018.1
С65

Рецензенты:

Доктор технических наук, доцент
С.В. Новикова
Кандидат технических наук, доцент
В.Л. Волушкова

Солдатенко И.С.
С65 Основы программирования на языке Си: учеб. пособие. – Тверь: Твер. гос. ун-т, 2017. – 159 с.

ISBN 978-5-7609-1229-9

Пособие посвящено основам программирования на языке Си. Рассмотрены такие темы, как: базовый синтаксис, логические и арифметические выражения, основные конструкции структурированных языков программирования (последовательное выполнение, ветвление, циклы), функции, массивы (одномерные и многомерные), символы и строки, типы данных, косвенный доступ к памяти через указатели, ввод/вывод, работа с файлами. В конце каждой главы приведены упражнения для закрепления материала.

Предназначено для студентов, изучающих программирование в качестве одной из профильных дисциплин, по направлениям укрупненной группы 02 «Компьютерные и информационные науки», а также по другим направлениям подготовки.

УДК 004.438С++(075.8)
ББК 3973.2-018.1

Печатается по решению научно-методического совета
Тверского государственного университета.

ISBN 978-5-7609-1229-9

Оглавление

Введение

В настоящее время все более востребованными становятся специалисты инженерных направлений подготовки, обладающие новым стилем научно-технического мышления. При этом в связи с проникновением техники и технологий во все сферы человеческой жизни, задачи, решаемые современным инженером, постоянно эволюционируют и усложняются. От современного специалиста требуется не просто освоить определенный объем материала, а, прежде всего, научиться им пользоваться для решения нетиповых задач, которые не разбирались в явном виде во время обучения и лежат на стыке различных областей.

В частности, бурное развитие в XX веке научных течений из области искусственного интеллекта породило целый класс новых задач, требующих от специалиста не только базового технического образования, но и глубокой математической подготовки, необходимой для понимания принципиально новых концепций, как например: интеллектуальное управление (например, в задачах проектирования так называемого «умного дома»), всевозможные вопросы из области искусственного интеллекта, программная инженерия, робототехника, нечеткие интеллектуальные системы, мягкие вычисления и т.д.

Первое, с чем требуется ознакомить будущего специалиста – это с основами инженерного моделирования, так как моделирование и конструирование являются базовыми навыками любого инженера, способствуют практическому познанию окружающего мира, развивают техническое мышление, мотивирует к творческому саморазвитию и в дальнейшем являются залогом профессионального роста. При этом одной из главных компонент инженерного моделирования становится математическое и тесно с ним связанное *компьютерное моделирование* для решения инженерных задач. Компьютерное моделирование, в свою очередь, требует знаний и навыков программирования на языке (а еще лучше языках) высокого уровня.

Язык Си в настоящее время является стандартным базовым языком, с которого начинают свое знакомство с программированием студенты первых курсов вузов. К тому же синтаксис многих современных языков берет свое начало в языке Си, поэтому, изучив

последний, в последствии не составит труда привыкнуть к синтаксису таких «мэйнстримовых» на сегодняшний день языков, как Java, C#, JavaScript и других.

В пособии рассмотрены такие темы, как: базовый синтаксис языка Си, логические и арифметические выражения, основные конструкции структурированных языков программирования (последовательное выполнение, ветвление, циклы), функции, одномерные и многомерные массивы, символы и строки, типы данных, косвенный доступ к памяти через указатели, ввод/вывод, работа с файлами. В конце каждой главы приведены упражнения для закрепления материала.

Настоящее учебное пособие предназначено для студентов направлений «02.03.02 Фундаментальная информатика и информационные технологии», «01.03.02 Прикладная математика и информатика», «09.03.03 Прикладная информатика», «38.03.05 Бизнес-информатика», а также любых других направлений подготовки и специальностей, в которых программирование является профильной дисциплиной. Материал пособия может быть использован для организации лабораторных практикумов по программированию (в частности, дисциплин «Практикум на ЭВМ», «Практикум по программированию» и др.).

Учебное пособие подготовлено в соответствии с ФГОС ВО по перечисленным выше направлениям и направлено на развитие следующих компетенций:

1. *по направлению подготовки «02.03.02 Фундаментальная информатика и информационные технологии»:*
   - способность применять в профессиональной деятельности современные языки программирования и языки баз данных, методологии системной инженерии, системы автоматизации проектирования, электронные библиотеки и коллекции, сетевые технологии, библиотеки и пакеты программ, современные профессиональные стандарты информационных технологий (ОПК-2);
   - способность к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей, созданию информационных ресурсов глобальных сетей, образовательного контента, прикладных баз данных, тестов и средств тестирования

систем и средств на соответствие стандартам и исходным требованиям (ОПК-3);
- способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учетом основных требований информационной безопасности (ОПК-4);
- способность использовать современные инструментальные и вычислительные средства (ПК-3);

2. *по направлению подготовки «01.03.02 Прикладная математика»:*
   - способность использовать базовые знания естественных наук, математики и информатики, основные факты, концепции, принципы теорий, связанных с прикладной математикой и информатикой (ОПК-1);
   - способность к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей, созданию информационных ресурсов глобальных сетей, образовательного контента, прикладных баз данных, тестов и средств тестирования систем и средств на соответствие стандартам и исходным требованиям (ОПК-3);
   - способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учетом основных требований информационной безопасности (ОПК-4);
   - способность к разработке и применению алгоритмических и программных решений в области системного и прикладного программного обеспечения (ПК-7);

3. *по направлению подготовки «09.03.03 Прикладная информатика»:*
   - способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учетом основных требований информационной безопасности (ОПК-4);

- способность разрабатывать, внедрять и адаптировать прикладное программное обеспечение (ПК-2);
- способность программировать приложения и создавать программные прототипы решения прикладных задач (ПК-8);

4. *по направлению подготовки «38.03.05 Бизнес-информатика»:*
   - умение разрабатывать контент и ИТ-сервисы предприятия и интернет-ресурсов (ПК-16).

Автор выражает признательность рецензентам за ценные критические замечания.

Тема 1

Введение в программирование на языке Си

Алгоритм. Программа. Исполнитель. Языки программирования. Уровни языков программирования. Работа в интегрированной среде разработки (IDE). Создание, компилирование, запуск программы. Базовые методы отладки. Пример простейшей программы. Операторы. Переменные. Константы: литеральные и символические. Присваивание. Выражения.

1.1 Алгоритм. Программа. Исполнитель

В чем разница между понятиями алгоритма и программы? *Алгоритм* – концептуальное описание шагов, которые необходимо выполнить для решения той или иной задачи, в то время как *программа* – это запись алгоритма на специальном языке (программирования), который понятен исполнителю. *Исполнитель* – нечто, что умеет шаг за шагом выполнять программу, написанную на понятному ему языке. Рассмотрим пример алгоритма нахождения минимального элемента в заданной последовательности чисел:

1. Запомним значение первого элемента последовательности как минимальное.
2. Перебираем все элементы последовательности, выполняя для каждого:
   а) если значение текущего элемента меньше минимального, то примем его за минимальное.

Данный алгоритм показывает идею (рецепт, описание) пошагового решения поставленной задачи. Исполнителем не обязательно должен быть компьютер. Им может быть человек, а инструкции, приведенные выше, можно рассматривать как программу для данного человека-исполнителя. Если же исполнителем является компьютер, то этот алгоритм можно записать на любом языке программирования: Си, Паскаль, Java, Бейсик, Питон и т.д. Программ получится много, но делать они будут одно и то же – *реализовывать алгоритм поиска минимального элемента в последовательности*.

Для того чтобы стать квалифицированным программистом, недостаточно выучить какой-либо из языков программирования – необходимо еще научиться составлять алгоритмы, позволяющие эффективно решать поставленную задачу. Искусство программирования состоит как минимум из двух частей: умения конструировать эффективные алгоритмы и знания инструментария, т.е. языков программирования и сред разработки. Причем первая часть является ничуть не менее важной, чем вторая, ведь одну и ту же задачу можно решить разными способами, так как могут существовать разные алгоритмы – некоторые из них могут работать быстрее других, а некоторые – существенно медленнее, какие-то будут требовать много памяти для своей работы, а какие-то смогут организовать все вычисления так, что дополнительной памяти не потребуется.

На Рис. 1.1 изображен пример, иллюстрирующий описанную идею. Задачу сортировки можно решить большим количеством концептуально разных алгоритмов, при этом каждый из них можно запрограммировать на одном из множества языков программирования. В итоге мы имеем огромное количество вариантов программного решения одной поставленной задачи. Поэтому хороший программист не просто должен уметь писать код на конкретном языке программирования – он еще должен уметь разрабатывать эффективные алгоритмы для решения любой

поставленной задачи. Точнее сказать – *почти* любой. На самом деле есть задачи, для которых не существует эффективных алгоритмов решения. При этом, как правило, написать код для них не составляет труда, только вот выполняться этот код будет дольше, чем осталось существовать нашей Вселенной. Любые попытки горе-разработчика «исправить» программу, чтобы она «не зависала», будут тщетны, и только настоящий квалифицированный программист будет понимать суть вещей и даже не станет приступать к решению подобной задачи в поставленном виде.

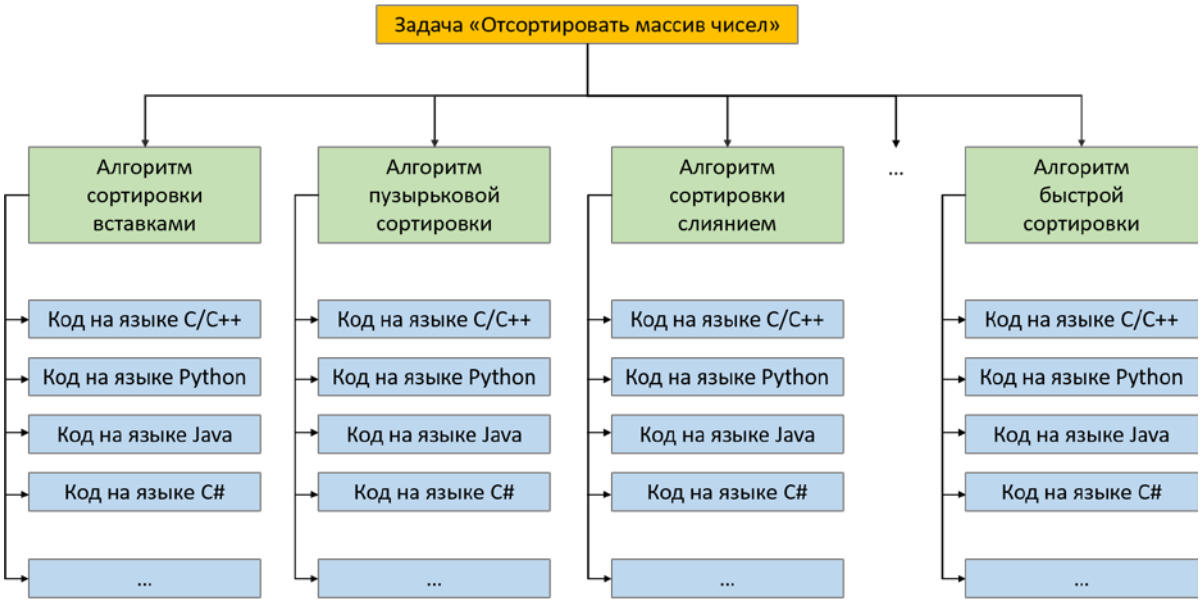


Рис. 1.1: Схема «задача – алгоритм – программа»

Знакомство с эффективными алгоритмами и структурами данных выходит за рамки настоящего учебного пособия, однако стоить помнить, что изучения одного лишь языка программирования недостаточно для разработки более-менее серьезных приложений.

1.2 Языки программирования

Цель настоящего учебного пособия – познакомить вас с языком программирования Си. Исполнителем в данном случае является центральный процессор компьютера. Для одного и того же компьютера можно написать программу на разных языках: Си, Бэйсик, Паскаль, Java, Питон и многих других. Означает ли это, что

компьютер «знает» их все? Нет. Компьютер знает только один язык – машинный, причем у каждой машины он свой. Это простейший язык, состоящий из команд типа: сложить, вычесть, сравнить, скопировать значение из одной ячейки памяти в другую, перейти к выполнению инструкции номер такой-то и т.д. Раньше все программы писались только на машинных языках (или, как еще говорят, в машинных кодах), но это было неудобно: во-первых, одну и ту же программу надо было переписывать под компьютеры с разными архитектурами, а во-вторых, программирование на таком примитивном языке – занятие утомительное и подверженное ошибкам. Поэтому человек придумал языки высокого уровня (как например, Си).

Уровень языка – его позиция в шкале «компьютер-человек» (см. Рис. 1.2). Чем ниже уровень – тем более он понятен компьютеру и менее понятен человеку, и наоборот.

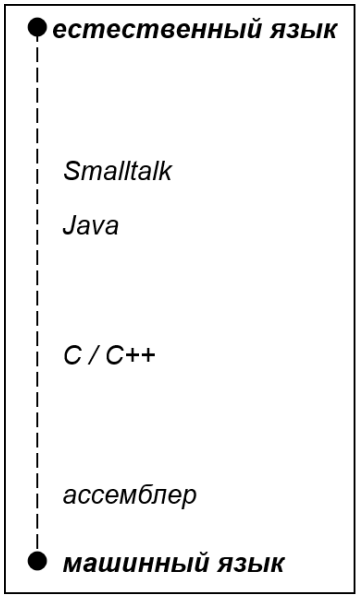


Рис. 1.2: Уровни языков программирования

Программы, написанные на языках высокого уровня, похожи на тексты на естественном (чаще всего английском) языке. Например, смысл строки кода «`if (a > b) then min = b`» несложно уловить даже не программисту.

Машинный же язык является языком низкого уровня. Машина не понимает языков высокого уровня, поэтому, прежде чем программу, написанную на языке Си (и любом другом не машинном), сможет

выполнить компьютер, ее необходимо перевести на язык, понятный компьютеру. Этим занимается *компилятор*.

Процесс создания программы на сегодняшний день в самом общем виде изображен на Рис. 1.3: сначала пишется код на высокоуровневом языке программирования, который сохраняется в обычном текстовом файле.

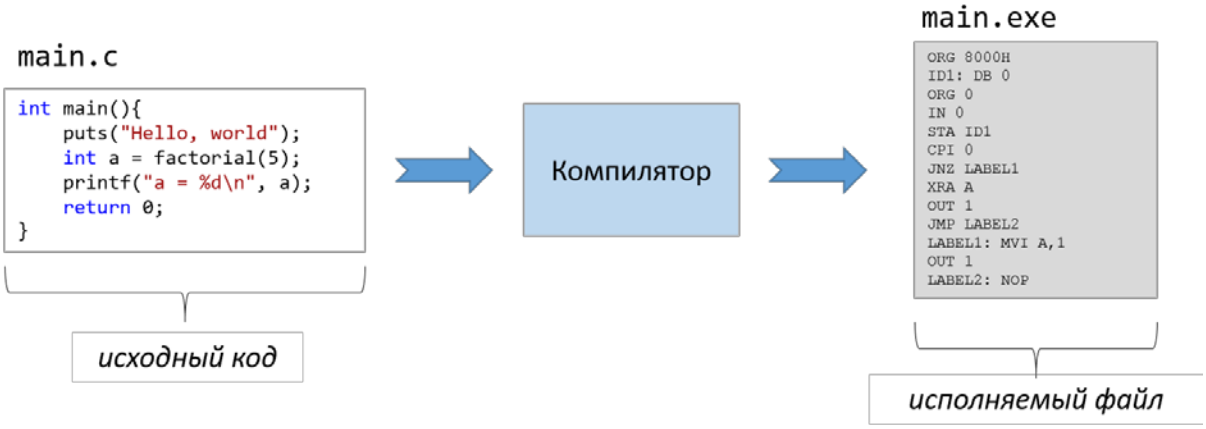


Рис. 1.3: Процесс компиляции программы

Затем запускается компилятор, транслирующий (переводящий) наш код на машинный язык и сохраняющий результат в виде исполняемого файла, который, как следует из его названия, можно запускать на выполнение.

Таким образом, для создания программы достаточно как минимум любого текстового редактора и компилятора с того языка, который мы используем.

1.3 Работа в IDE

Часто (в том числе и в нашем случае) компилятор не существует в одиночестве. Вместе с ним прилагается еще ряд программ, призванных упросить процесс разработки приложений – например, специальный текстовый редактор, который знает синтаксис языка Си и умеет подсвечивать разными цветами ключевые слова, имена переменных и другие элементы кода, средства для отладки программ, справочная документация и т.д. Весь этот комплекс программ называется *IDE* (Integrated Development Environment) –

интегрированная среда разработки. В отличие от набора дискретных (отдельных) приложений, как например, обычного текстового редактора, компилятора, файла со справочной информацией – IDE представляет собой совокупность очень тесно взаимосвязанных программ, которые с точки зрения программиста могут выглядеть как одно приложение, из которого доступны все необходимые функции. Современный IDE состоит из:

1. интеллектуального текстового редактора с функциями структурирования и подсветки кода, автодополнения и т.д.,
2. встроенной справки,
3. компилятора,
4. линковщика,
5. библиотек кода,
6. средств отладки,
7. средств автоматизированной сборки приложения,
8. средств для интеграции с системами управления версиями кода,
9. средств для совместной разработки,
10. средств для взаимодействия с базами данных,
11. всевозможных утилит, упрощающие разработку и отладку кода

и многого другого... Размер полностью установленной (за исключением справочного материала) современной IDE может измеряться несколькими гигабайтами, в то время как сам компилятор, являющийся по существу его центральной компонентой, занимает всего несколько мегабайт на жестком диске.

В рамках данного учебного пособия мы будем использовать бесплатно распространяемую среду разработки Microsoft Visual Studio Community, найти и установить которую не составляет никакого труда[1].

Создадим свое первое приложение. Для этого запускаем студию. Нас встречает стартовая страница с ссылками на последние проекты, с которыми мы работали, новости от Microsoft и другая информация справочного характера. Первое, что мы должны сделать – это создать *проект*. Файлы с исходным кодом в студии не существуют сами по себе – они всегда являются частью проекта. Проект объединяет в себе все файлы с исходными кодами и другими ресурсами, необходимыми

[1] На момент написания учебного пособия Visual Studio Community 2017 доступна по адресу https://www.visualstudio.com/downloads/.

для сборки одного приложения. Как мы скоро увидим, программа может собираться из нескольких файлов (крупные приложения могут собираться из сотен самых различных файлов). Например, если вы разрабатываете графическое приложение, то здесь будут собраны все оконные формы вашего приложения, все графические изображения, иконки, звуки и другие медиа-ресурсы.

Сложный программный продукт может состоять из нескольких отдельных программ, то есть являться *программным комплексом*. Примером такового является сама IDE. Для нас это одна программа, хотя в действительности их около сотни. Поэтому в студии все проекты объединяются в так называемые *решения* (solutions). Таким образом, решение – это совокупность проектов, где каждый проект представляет собой отдельную программу.

Нажимаем на ссылку «Создать проект...». Мы увидим мастер по созданию проектов (Рис. 1.4).

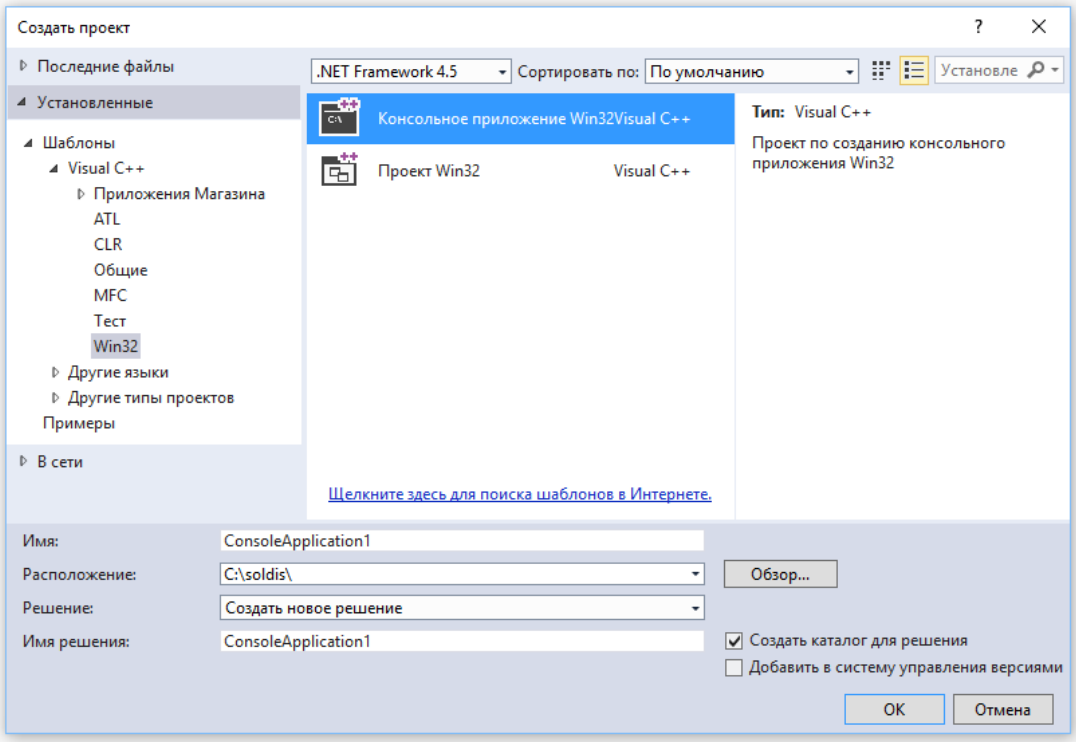


Рис. 1.4: Создание нового проекта

Студия позволяет выбрать язык программирования, на котором вы будете вести разработку – C++, C#, Бейсик, Питон и другие. Более того, в студии уже есть ряд шаблонов, например, для разработки веб-приложений, или приложений под мобильные платформы.

В проекте, созданном по шаблону, сразу будут подключены все необходимые библиотеки, созданы все требуемые файлы проекта с минимальным необходимым количеством кода и выполнен ряд других предварительных настроек. У каждого шаблона есть целый набор под-шаблонов.

Нас интересует шаблон Visual C++, подшаблон Win32, тип «Консольное приложение Win32». Теперь нам необходимо настроить параметры шаблона. Прежде всего, указываем место, где будет создан проект, имя проекта и имя решения. По умолчанию, имя решения совпадает с именем проекта. Нажимаем ОК и настраиваем дополнительные параметры. Нам необходимо выбрать *пустой проект* для *консольного приложения* (Рис. 1.5). Нажимаем «Готово».

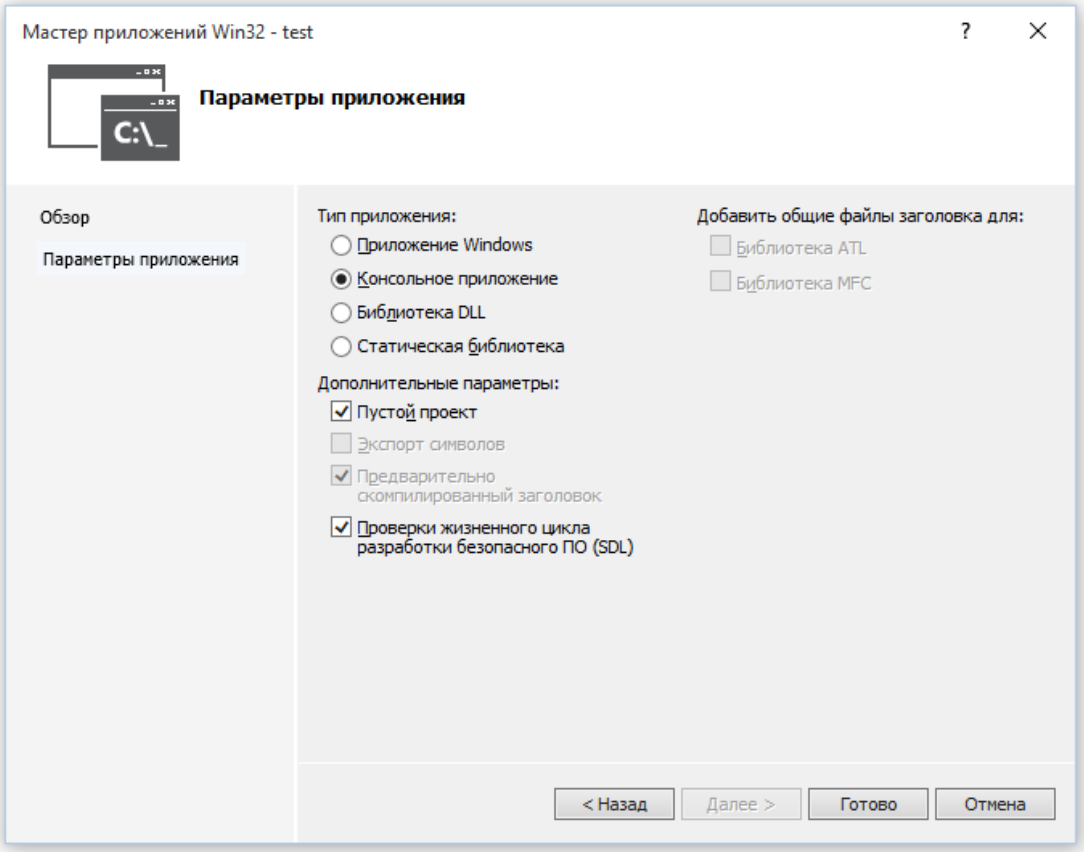


Рис. 1.5: Создание нового проекта. Параметры шаблона

На первый взгляд ничего не изменилось. В действительности, на диске по тому пути, который вы указали, было создано несколько папок с файлами для сопровождения вашего проекта, однако главный файл еще отсутствует – файл с исходным кодом.

Посмотрим на структуру проекта в *обозревателе решения* (Solution Explorer, если обозреватель решений не виден, его можно включить в меню «Вид»). Здесь изображена логическая структура нашего проекта (Рис. 1.6) – на верхнем уровне представлено решение, в нем перечислены проекты, в которых идут папки для заголовочных файлов, файлов с исходным кодом и файлами ресурсов.

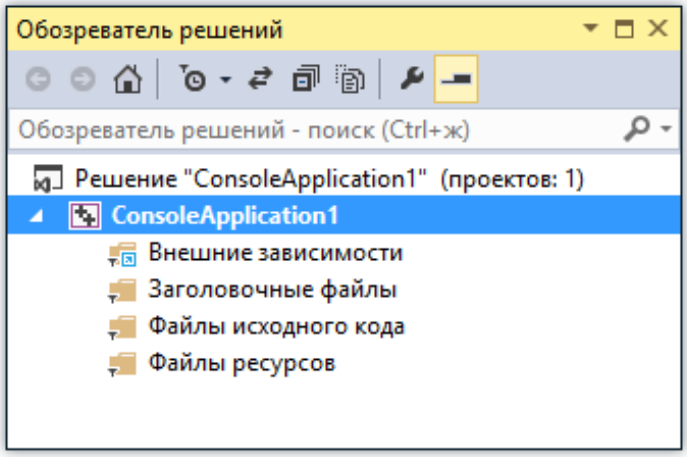


Рис. 1.6: Окно обозревателя решений (Solution Explorer)

Нажимаем правой кнопкой на папку «Файлы исходного кода» и выбираем пункт «Создать новый элемент» (Рис. 1.7).

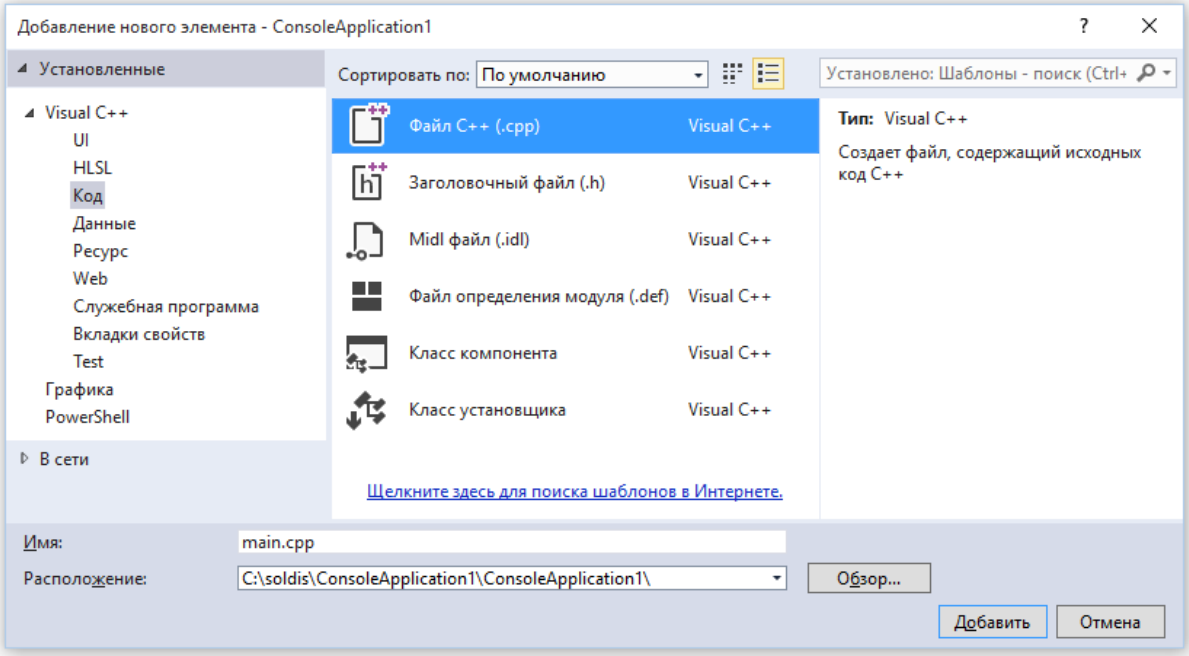


Рис. 1.7: Добавление файла в проект

Открывается окно с набором шаблонов файлов, которые мы можем создать/добавить в проект – от элементов графического интерфейса до файлов ресурса. Нам надо выбрать Код –> Файл C++[2]. Указываем его имя и нажимаем «Добавить». Перепишем в окно текстового редактора следующий код:

```
#include <stdio.h>
int main(){
     puts("Hello, world");
     int a = 2;
     printf("a = %d\n", a);
     return 0;
}
```

Прежде всего, обратим внимание, как студия выделяет цветом ключевые слова и автоматически выравнивает код, разбивая его на блоки, которые вы можете сворачивать и разворачивать по мере надобности. Также мы можем заметить многочисленные подсказки, «всплывающие» при наборе текста и при наведении мышкой на различные элементы кода. Все это – работа встроенного в студию интеллектуального текстового редактора.

Смысл написанного кода мы разберем в следующем разделе, а сейчас попробуем скомпилировать и запустить нашу первую программу, которая, при удачном стечении обстоятельств, должна вывести на экран строку приветствия и значение переменной `a`.

Компилируем решение. Для этого можно выбрать в меню «Сборка» пункт «Собрать решение» или просто нажать на клавиатуре <F7>. В окне «Вывод» снизу экрана мы видим сначала ход, а затем и результат компиляции (Рис. 1.8).

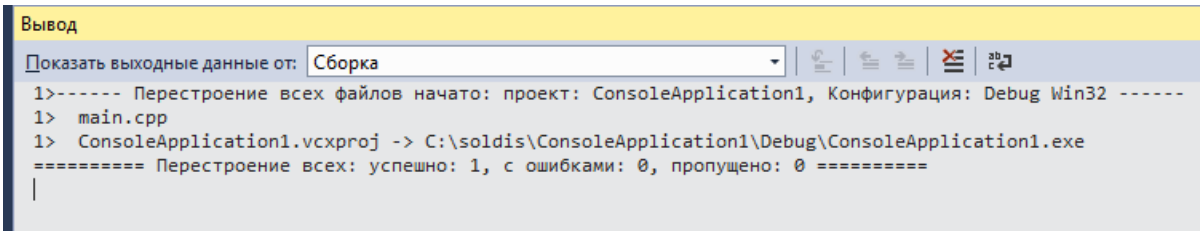


Рис. 1.8: Вывод процесса компиляции

[2] Почему мы выбрали шаблон C++, хотя изучаем язык Си? Дело в том, что язык Си является подмножеством языка C++, то есть компилятор C++ естественным образом транслирует код, написанный на языке Си.

Если все прошло без ошибок, то в последней строке будет написано «`успешно: 1, с ошибками: 0, пропущено: 0`». В этом окне мы также можем увидеть путь, по которому был создан исполняемый файл нашего приложения. Заметим, что исполняемый файл называется не так, как назван файл с исходным кодом, а по имени проекта.

Изучим содержимое каталога решения:

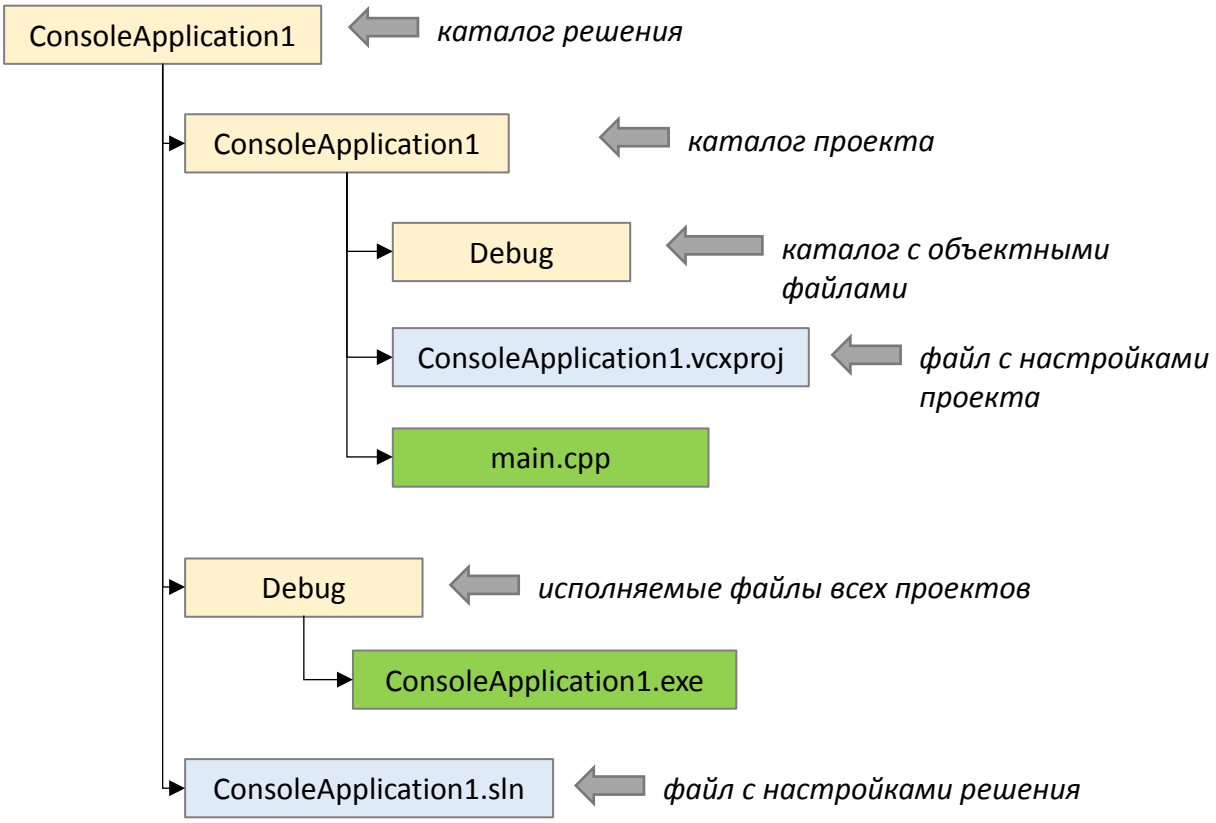


Рис. 1.9: Структура каталога решения

Прежде всего, мы найдем в нем файл с расширением `*.sln`, который содержит настройки решения. Для открытия нашего проекта в студии достаточно дважды щелкнуть мышью на этом файле. Помимо этого, в этой папке есть отдельный подкаталог для каждого проекта, а также каталоги Debug и Release, куда для удобства после компиляции выносятся исполняемые файлы всех проектов. Все пути можно изменить в настройках решения.

В каталоге проекта мы также видим подкаталог Debug, где хранятся промежуточные результаты компиляции, файл с расширением *.vcxproj с описанием проекта, а также все файлы, добавленные нами к проекту (в том числе и с исходным кодом).

Помимо вышеописанных в этих папках есть еще ряд служебных файлов, назначение которых нам пока не важно.

По умолчанию, студия собирает проекты в так называемом *отладочном режиме* (Debug mode), в котором после компиляции создается исполняемый файл, содержащий большое количество отладочной информации. Благодаря этому мы можем использовать предоставляемый студией мощный инструментарий отладки. Когда приложение будет готово, и мы захотим собрать его «начисто», нам надо будет выбрать в панели инструментов студии (см. Рис. 1.10) *режим релиза* (Release mode). После сборки решения в этом режиме рядом с каталогами Debug будут созданы каталоги Release с результатами компиляции. Мы можем заметить, что релиз-версия программы занимает существенно меньше места на диске, чем отладочная версия, потому что из нее убрана вся избыточная информация. Всюду далее будем предполагать, что используется отладочный режим.

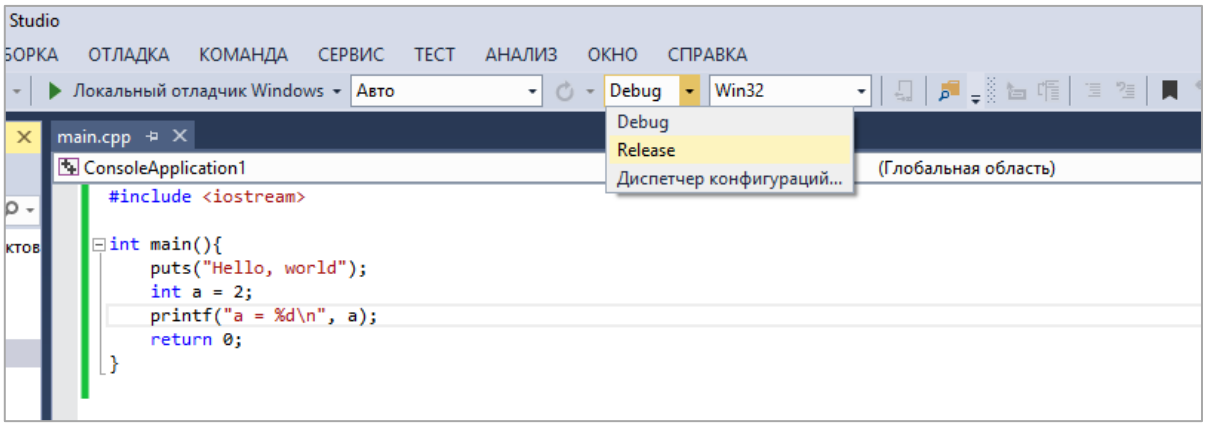


Рис. 1.10: Выбор режима сборки проекта

Осталось самое последнее – запустить наше приложение. Мы можем это сделать, нажав кнопку <F5>, однако в этом случае приложение запустится в новом консольном окне, отработает и исчезнет, не дав нам увидеть результат своей работы. Чтобы задержать консольное окно, приложение надо запускать с помощью комбинации клавиш <Ctrl>-<F5>. В этом случае после выполнения программы консольное окно будет держаться на экране до тех пор, пока мы не нажмем любую кнопку на клавиатуре (Рис. 1.11).

Если мы попытаемся вручную запустить exe-файл из каталога, то произойдет то же самое – приложение запустится, отработает

и исчезнет. Чтобы увидеть результаты его работы, необходимо в проводнике открыть папку с исполняемым файлом, в строке пути (см. Рис. 1.12) набрать команду `cmd`, нажать <Enter>, в открывшемся консольном окне набрать имя исполняемого файла и снова нажать <Enter>.

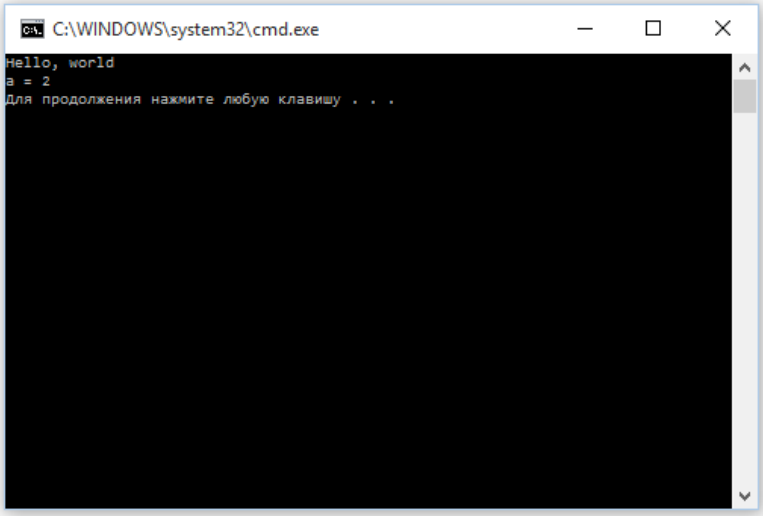


Рис. 1.11: Консольное окно приложения с результатом работы

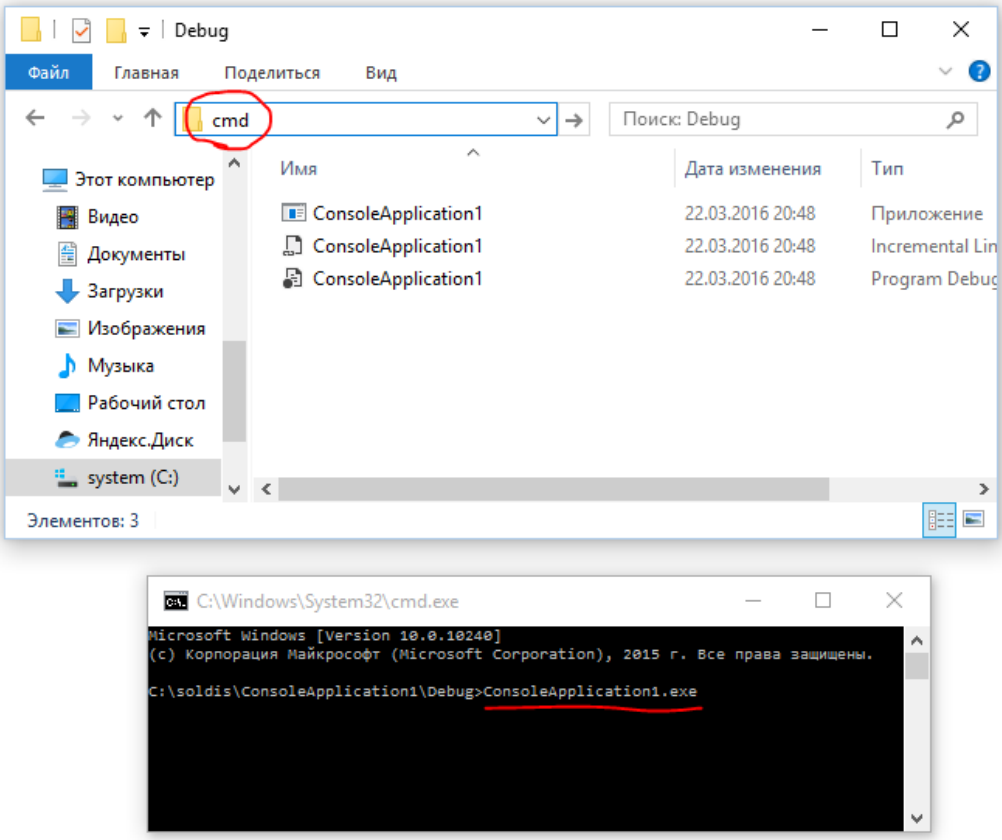


Рис. 1.12: Запуск приложения вручную

1.4 Отладка

Вкратце познакомимся с основными средствами и базовыми приемами отладки кода, предоставляемыми студией.

Во-первых, приложение может не скомпилироваться в виду наличия синтаксических ошибок. Соответствующая информация будет выведена в окне «Вывод» внизу рабочего экрана. Попробуем в качестве эксперимента стереть в нашей программе в шестой строке первую кавычку:

```
#include <iostream>

int main(){
    puts("Hello, world");
    int a = 2;
    printf(a = %d\n", a);
    return 0;
}
```

Обратим внимание, что еще до компиляции студия выделила красной волнистой чертой те фрагменты кода, которые вызывают у нее вопросы. Это происходит благодаря тому, что студия делает подробный анализ кода уже тогда, когда вы его только набираете в текстовом редакторе, практически осуществляя его «пробную» компиляцию на лету. Это позволяет заметить ошибки еще до процесса сборки решения, который для сложных программных продуктов может длиться часами.

Если же мы все-таки попытаемся откомпилировать приложение, то получим следующий результат в окне вывода:

```
1>------ Сборка начата: проект: ConsoleApplication1, Конфигурация: Debug Win32 ---
1>  main.cpp
1>…\main.cpp(6): error C2017: недопустимая escape-последовательность
1>…\main.cpp(6): error C2065: d: необъявленный идентификатор
1>…\main.cpp(6): error C2001: newline в константе
1>…\main.cpp(6): error C2146: синтаксическая ошибка: отсутствие ")" перед
идентификатором "n"
========== Сборка: успешно: 0, с ошибками: 1, без изменений: 0, пропущено: 0 ========
```

Мы стерли всего один символ, а получили целых четыре сообщения об ошибке! В этом нет ничего удивительного. Очень частое явление при компиляции кода с синтаксическими ошибками – одна ошибка в коде может порождать сразу несколько

диагностических сообщений от компилятора. В нашем пример их четыре, и они все разные. Причина этого заключается в том, что современный компилятор, встретив первую ошибку в коде, не завершает работу, а пытается продолжить компиляцию дальше. Для этого он делает «предположение», каким образом надо исправить код, чтобы он, по его мнению, не содержал ошибок. Однако, к сожалению, программным способом невозможно абсолютно точно определить, где и в чем именно была совершена ошибка, поэтому «предположение» компилятора часто оказывается неверным и его «исправление» порождает новые ошибки, которые компилятор также пытается исправить и т.д.

В большинстве случаев компилятору удается выйти на участок кода, не затрагиваемый хаосом, вызванным допущенной пользователем ошибкой и попытками компилятора «сделать как лучше», и завершить компиляцию. Этот процесс называется восстановлением после ошибки, и он позволяет локализовать некорректные фрагменты кода. Для чего же компилятору это нужно? Для того, чтобы за один проход попытаться найти по возможности большую часть ошибок в коде и сразу же предоставить эту информацию программисту. Ведь, как уже было сказано ранее, сборка серьезных сложных приложений может занимать по времени даже не минуты, а часы, поэтому перекомпилировать приложение после каждой исправленной одиночной ошибки не всегда удобно.

Для нас же главное – запомнить следующее правило: если в окне вывода мы видим сообщения о нескольких десятках ошибок, это вовсе не означает, что именно столько их присутствует в нашем коде. Скорее всего, после исправления одной-двух, количество диагностических сообщений существенно убавится, если не сведется к нулю. Для локализации предполагаемого места нахождения ошибки в коде, надо дважды щелкнуть мышью на самое первое диагностическое сообщение – курсор автоматически перейдет к нужной строке. Как правило, внимательный анализ строки сразу выявит, что нужно исправить.

Помимо ошибок, компилятор также будет иногда выводить *предупреждения*, которые можно распознать по ключевому слову «warning» в диагностическом сообщении. Предупреждения не мешают компиляции приложения, а предупреждают вас о возможных проблемах в коде: возможно вы используете устаревшую или

небезопасную функцию, которая в последней версии среды разработки была заменена, или компилятору «кажется», что вы написали не совсем корректный или безопасный код. В зависимости от ситуации, вы сами принимаете решение, как реагировать на эти предупреждения.

Если приложение скомпилировалось, это еще не значит, что оно корректно. Во-первых, во время работы приложения могут возникнуть так называемые *ошибки времени выполнения* (runtime errors), когда приложение аварийно завершается с сообщением, что оно где-то куда-то неправильно обратилось к памяти. Во-вторых, оно может отработать до конца без каких-либо сообщений об ошибках, но при этом выдать совсем не тот результат, который хотелось бы. В этих случаях можно прибегнуть к механизмам отладки, основным из которых является *пошаговое выполнение программы*.

Для этого мы нажимаем кнопку <F10> – приложение запускается, открывается и «зависает» консольное окно, а в текстовом редакторе студии напротив одной из строк появляется желтая стрелка:

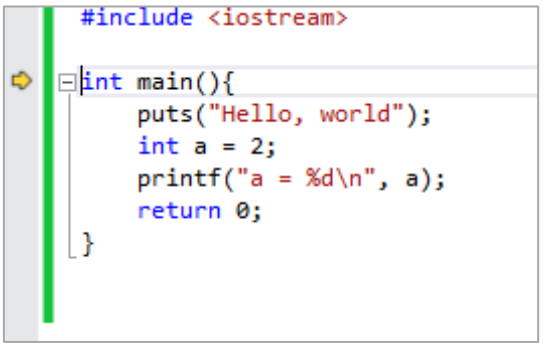


Рис. 1.13: Запуск пошагового выполнения приложения

Эта стрелка указывает на ту инструкцию, которая будет выполнена на следующем шаге. Чтобы сделать этот самый следующий шаг, надо снова нажать <F10> и т.д. Так, шаг за шагом, нажимая <F10>, можно «пройтись» по каждой строке кода, смотря на вывод в консольном окне. Если мы хотим закончить пошаговое выполнение, то можно нажать <F5>.

Во время отладки мы можем делать с кодом много интересных вещей. Можем навести на переменную и посмотреть ее текущее значение. Это значение также можно увидеть в специальном окне «Видимые» внизу рабочего экрана. Можем посмотреть на всю цепочку вызовов функций, если таковые были. Если желтая стрелка

указывает на строку, содержащую вызов функции, то, чтобы зайти «внутрь», нужно нажать <F11>. Таким образом, можно «ходить» по сколь угодно сложному коду, просматривая значения всех переменных в каждый момент времени. В результате, если логика приложения позволяет, можно найти то место в коде, где возникает ошибка.

Если код слишком большой и «шагать» до нужного места, которое хочется посмотреть повнимательнее, слишком долго, то можно воспользоваться механизмом *точек останова*. Для этого устанавливаем курсор на нужную строку, с которой мы хотели бы начать отладку, и нажимаем <F9>. Напротив этой строки появится красный маркер. Дальше нажимаем <F5> – приложение запускается, начинает работать и как только оно дойдет до помеченной красным кружком строки, оно перейдет в режим пошаговой отладки. Точек останова можно ставить произвольное количество. Чтобы снять точку останова, нужно на соответствующей строке снова нажать <F9>.

Этих простейших методов отладки нам будет достаточно для изучения языка Си на наших упражнениях.

1.5 Написание простейшей программы

Давайте теперь разберем код простейшей программы на языке Си. Изменим код приложения на следующий (нумерация строк приведена для удобства, в коде программы ее быть не должно):

```
1. #include <stdio.h>
2.
3. int main(){
4.   /* Выводим на экран строку */
5.   printf("Hello, World!");
6.
7.   return 0;
8. }
```

Строка `1` содержит команду препроцессора, подключающую заголовочный файл стандартной библиотеки Си, который дает нашей программе возможность печатать сообщения на экран. `stdio.h` – библиотека для работы с вводом/выводом информации (`standard input/output` – стандартный ввод/вывод). Если наша

программа будет что-то печатать на экран или считывать с клавиатуры, то в ней обязательно должна присутствовать данная строка.

Препроцессор (от англ. предварительная обработка) – это программа, которая запускается до этапа компиляции, просматривает код и выполняет все инструкции, начинающиеся с символа решетки. В частности, инструкция `include` копирует содержимое указанного в ней файла на место этой строки. Суть работы препроцессора описана в главе «Этапы сборки приложения», а сейчас достаточно запомнить, что первая строка необходима для того, чтобы мы могли пользоваться библиотечными функциями ввода/вывода, одна из которых – `printf`.

Строки 3-8 содержат определение главной функции программы. Функция языка Си, как и любого другого структурированного языка программирования, это именованный фрагмент кода, который можно вызывать по его имени (более подробно с функциями мы познакомимся в одной из следующих глав). Определение функции состоит из заголовка (строка 3) и тела функции (строки 4-7, заключенные в фигурные скобки). Главная функция `main` обязательна в любой программе на Си – это так называемая *точка входа* в программу. Именно она вызывается операционной системой в момент запуска нашей программы.

Код на языке Си состоит из *операторов*. Оператор – это отдельная инструкция, заканчивающаяся точкой с запятой. Точка с запятой – обязательный разделитель! Самая распространенная ошибка при изучении языка Си – забыть поставить точку с запятой. Строки 5 и 7 содержат два оператора. Их можно было бы написать на одной строке, однако, делать этого не следует – код получится более сложным для восприятия.

На строке 5 указан оператор вызова библиотечной функции печати. Вызов функции в языке Си выглядит следующим образом: сначала пишется имя функции, а затем в круглых скобках через запятую перечисляются передаваемые в функцию аргументы. Функция `printf` (от англ. print formatted, форматированный вывод) печатает на экран переданную ей строчку. Строки в языке Си выделяются двойными кавычками. Круглые скобки – обязательный синтаксис вызова функции. Вводу/выводу также посвящена отдельная глава учебного пособия.

7-я строка – это оператор возврата из функции. Под «возвратом» понимается завершение работы данной функции, а так как `main` – это главная функция, то данный оператор завершает выполнение всей нашей программы.

Строка 4 – это пример многострочного комментария (который, правда, записан всего на одной строке). Эта строчка игнорируется компилятором. Она необходима программистам, чтобы понимать, что же делает программа. Хороший код – это хорошо оформленный и достаточно подробно комментированный код. Многострочность означает, что мы можем закомментировать подобным образом несколько строк кода. Если мы хотим закомментировать только часть какой-то одной строки, то можно воспользоваться синтаксисом однострочного комментария, поставив перед комментируемой частью два слеша `//`.

Пробуем скомпилировать и выполнить. Результат очевиден – на экране будет напечатано приветствие. Теперь усложним программу. Напишем второй оператор печати строки после первого. Пусть компьютер представится:

```
printf("Hello, World!");
printf("My name is John");
```

Что получилось? Ерунда! Строчки склеились друг с другом. Для того чтобы этого не происходило, курсор необходимо перевести на новую строку после печати первой фразы. Делается это путем добавления в выводимую строку специальной последовательности символов - `\n`[3]:

```
printf("Hello, World!\n");
printf("My name is John");
```

Функция `printf`, встретив данную комбинацию символов, перенесет курсор на новую строку.

Усложним программу. Пусть она спрашивает, сколько нам лет и выводит свое мнение о возрасте на экран, а затем говорит, что она на два года старше. Для этого нам понадобятся две вещи: первое – оператор ввода информации с клавиатуры, второе – место, где бы мы могли сохранить возраст, введенный пользователем.

---

[3] Данная комбинация называется escape-последовательностью, или экранированным символом, и состоит из обратной косой черты и, чаще всего, одного дополнительного символа.

Для хранения информации программы обычно используют переменные – именованные области компьютерной памяти, в которых можно хранить различные значения. В языке Си, прежде чем использовать переменную, ее надо *объявить*. Сделать это нужно *до того, как переменная будет использована*. Оператор объявления выглядит следующим образом:

```
int age;
```

где `int` – тип переменной, говорящий компилятору, какую информацию можно будет в ней хранить. В нашем случае это `int` (integer) – целочисленный тип. После типа идет имя переменной (в нашем примере – `age`), которое может быть *любой комбинацией букв, цифр и символа нижнего подчеркивания, начинающейся с буквы или символа нижнего подчеркивания*.

Для ввода информации с клавиатуры используется еще одна функция из библиотеки `stdio` – `scanf_s`[4]:

```
scanf_s("%d", &age);
```

Первый параметр этой функции – форматная строка, которая описывает, что и какого типа нужно считать с клавиатуры. `%d` означает, что надо считать одно десятичное число (decimal). Второй параметр – имя переменной, куда функция должна сохранить введенное с клавиатуры значение, со специальным символом `&`, предназначение которого мы поймем чуть позже. Получаем следующий код:

```
1. #include <stdio.h>
2. #include <locale.h>
3. int main(){
4.   setlocale(LC_ALL, "Russian");
5.   int age, my_age;
6.   printf("Сколько вам лет?\n");
7.   scanf_s("%d", &age);
```

[4] Функция scanf_s не является обязательной частью стандарта языка Си и на настоящий момент полностью поддерживается лишь в Microsoft Visual Studio. По мнению Microsoft, scanf_s является безопасной версией функции scanf (отсюда суффикс _s – secure). В других средах разработки (не от Microsoft) вы должны использовать scanf вместо scanf_s, из-за того, что последняя в них не поддерживается. Так как в рамках данного учебного пособия предполагается использование бесплатной среды разработки MS Visual Studio Community Edition, то всюду далее будет применяться синтаксис «безопасных» функций: scanf_s, fscanf_s и т.д.

```
8.   my_age = age + 2;
9.   printf("Ваш возраст: %d. Но я старше! Мой возраст –
                                        %d!!!\n", age, my_age);
10.  return 0;
11. }
```

Что нового мы видим в этом листинге? Во-первых, мы добавили еще одну библиотеку `locale.h` и вызвали из нее функцию `setlocale()`, благодаря чему мы можем теперь использовать в наших программах кириллицу[5]. Во-вторых, мы в одном операторе объявили одновременно сразу несколько переменных (строка 5). В-третьих – мы познакомились с оператором *присваивания* (строка 8), который кладет (присваивает) значение правой части в переменную, имя которой указано в левой части. В этой же строке мы видим пример простейшего выражения, вычисляющего сумму. Ну и, наконец, в-пятых, вывод на печать значений переменных в строке 10. Разберемся со всем этим подробнее.

Переменные и константы – два механизма для работы с данными в любом языке программирования. *Переменная* – это именованная область памяти, используемая для хранения промежуточных результатов. Ее содержимое может изменяться, откуда и название – переменная. *Константа* же (лат. *constanta* – постоянная, неизменная) – это некоторое конкретное значение (числовое или строковое), которое, будучи определенным в программе, меняться уже не может.

Константы бывают двух типов: *литеральные* и *символические*. Литеральные константы (или просто литералы) – это *конкретные значения*, записанные непосредственно в тексте программы. Они бывают:

– числовыми (целочисленными и дробными), например: `0`, `-5`, `3.14`, `103` и т.д.,
– строковыми (заключенными в двойные кавычки) – `"Hello, world!\n"`, `"Input a number: "` и т.д.,
– символьными (заключенными в одинарные кавычки) – `'a'`, `'b'`, `'0'`, и т.д.

[5] Русификация приложений, особенно консольных, не является стандартной задачей, поэтому данная функция может у вас и не сработать. В этом случае удалите строки 2 и 4 и используйте латиницу в упражнениях.

Целочисленные константы в языке Си могут быть представлены в одной из следующих систем счисления (перед любой формой записи может идти знак плюс или минус):

– десятичные: последовательность цифр от `0` до `9`, которая начинается с цифры, отличной от `0` (исключение составляет лишь сам `0`). Пример: `2, -45, 23`;
– восьмеричные: последовательность цифр от `0` до `7`, которая всегда начинается с нуля. Пример: `00, 032, 063`;
– шестнадцатеричные: последовательность цифр от `0` до `9` и символов от `A` до `F` (без учета регистра), которая всегда начинается с комбинации `0x`. Пример: `0x0, 0x4, 0xFFFF, 0x2A4, 0x7fff`.

Символические константы – это своего рода «переменные», значения которых меняться не могут. Их также необходимо объявлять в коде программы, но перед типом надо ставить ключевое слово `const`. Например:

```
const double pi = 3.14;
```

При этом очень важно помнить, что присваивать значение символической константе необходимо *одновременно с ее объявлением*. Код вида:

```
const double pi;
pi = 3.14;
```

будет некорректным, потому что ключевое слово `const` говорит компилятору языка Си, что значение «переменной» `pi` после ее создания измениться уже не может.

Переменные бывают разных типов. Си – язык со *статической типизацией*. Это означает, что любая переменная должна иметь конкретный тип, и она сможет хранить в себе данные только этого типа (в отличие от, например, Бейсика, Питона и ряда других языков, где одно и то же имя можно использовать для поочередного хранения значений самых разных типов). В языке Си есть следующие базовые (включенные в синтаксис самого язык) типы данных:

– `char` – символ,
– `int` – целое число,
– `float` – дробное число одинарной точности,
– `double` – дробное число двойной точности.

Более подробно о переменных и типах данных речь пойдет в следующей главе.

Создаются переменные при помощи *оператора объявления*, который выглядит в общем виде так:

```
тип имя1 [= значение1], …, имяN [= значениеN];
```
[6]

Этот код позволяет нам *инициализировать* (присвоить начальное значение) переменным прямо во время создания. Если этого не сделать, то после создания переменная будет содержать «мусор» – значение, которое заранее неизвестно программисту и зависит от настроения компилятора и операционной системы.

Для записи в переменную какого-либо значения, используется *оператор присваивания* (=). Этот оператор работает следующим образом:

```
куда = что;
```

или, более правильно:

```
lvalue = rvalue;
```

где lvalue (left value) – то, что стоит слева, и обозначает то место, *куда надо копировать*, а rvalue (right value) – то, что стоит справа и что *можно вычислить*. Исходя из этой логики, становится понятно, что lvalue обязательно должно быть связано с какой-то изменяемой ячейкой памяти и, например, не может быть константой, так как в константу нельзя ничего записать.

Функция `printf` на строке 10 печатает фразу, в которую вставлены значения соответствующих переменных. В данном случае используются те же спецификаторы `%d`, что и в `scanf_s`. Напомним, что спецификатор состоит из двух символов, первый – `%`, второй – буква латинского алфавита, показывающая, со значением какого типа предстоит работать (`d` – decimal, десятичное). На места, занимаемые спецификаторами в форматной строке, вставляются по порядку значения переменных, идущих через запятую после форматной строки. Форматная строка позволяет нам *отформатировать* выводимый текст, вставив в него значения нужных переменных.

[6] Квадратные скобки в записи означают, что этот фрагмент кода является необязательным.

Давайте напишем программу, спрашивающую у пользователя радиус бассейна (r) и его глубину (h). Затем она вычислит объем бассейна по формуле $V = pi \cdot r^2 \cdot h$ и выведет его.

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

int main(){
     setlocale(LC_ALL, "Russian");
     const double pi = 3.14;
     double V, r, h;
     printf("Введите значения r и h через пробел: ");
     scanf_s("%lg %lg", &r, &h);
     V = pi * r * r * h;
     printf("Объем бассейна равен %lg\n", V);
     return 0;
}
```

Для работы со значениями других типов, отличных от `int`, используются другие спецификаторы. В частности, для дробного числа с двойной точностью это `%lg` (для типа данных `float` необходимо использовать спецификатор `%f`). Из данного фрагмента мы видим, что с помощью функции `scanf_s` можно считывать с клавиатуры сразу несколько значений.

В заключении первой главы скажем несколько слов о выражениях. *Выражения* – это основная рабочая сила языка Си. Выражения бывают двух типов – математические и логические. Математическое выражение – *это совокупность имен переменных, символических и литеральных констант, скобок и знаков математических операций (+, -, *, /, %)*[7], *представляющая собой корректную формулу*. Главная особенность выражений – это то, что они *вычислимы*, то есть их значение можно подсчитать. Вычисляются выражения интуитивно понятным образом: переменные и символические константы заменяются на их текущие значения и получившаяся математическая формула вычисляется в порядке приоритетов операций. Приоритеты рассмотренных операций знакомы нам еще

[7] Операция % – это взятие остатка от деления.

со школы: самый высокий приоритет у скобок, затем идут умножение, деление и взятие остатка от деления, а потом сложение и вычитание.

Выражения можно использовать почти везде, где можно использовать переменные, за редким исключением. Например, нельзя использовать выражение в левой части оператора присваивания, то есть оно не может быть lvalue.

Более подробно мы познакомимся с выражениями в следующей главе, а сейчас перейдем к упражнениям.

Упражнения

Попробуйте решить самостоятельно следующие упражнения.

1.1 Ввести с клавиатуры два числа, а затем вывести на экран результат вычисления суммы, разницы, умножения, деления и взятия остатка от деления первого на второе.

1.2 Ввести три числа, а затем вывести их среднее арифметическое.

1.3 Ввести с клавиатуры количество минут и напечатать количество целых часов и оставшихся минут. Например, 69 минут => 1 час и 9 минут, 45 минут => 0 часов и 45 минут, 254 минуты => 4 часа и 14 минут.

Решения упражнений

Всюду далее в учебном пособии в листингах кода инструкции подключения библиотек и объявления функции `main`:

```
#include <stdio.h>
#include <locale.h>
int main(){
     setlocale(LC_ALL, "Russian");
     /* Код упражнения */
}
```

могут быть пропущены для краткости.

1.1
```
int a, b;
printf("Введите два числа: ");
scanf_s("%d %d", &a, &b);
printf("%d + %d = %d\n", a, b, a + b);
printf("%d - %d = %d\n", a, b, a - b);
printf("%d * %d = %d\n", a, b, a * b);
printf("%d / %d = %d\n", a, b, a / b);
printf("%d %% %d = %d\n", a, b, a % b);
```

1.2
```
float a, b, c;
printf("Введите три числа: ");
scanf_s("%f %f %f", &a, &b, &c);
printf("Среднее = %f\n", (a + b + c) / 3);
```

1.3
```
int minutes;
printf("Введите минуты: ");
scanf_s("%d", &minutes);
printf("\n%d минут = %d часов, %d минут\n", minutes,
                              minutes / 60, minutes % 60);
```

Упражнения для самостоятельного решения

1.4 Ввести с клавиатуры радиус, а затем напечатать диаметр круга, длину его окружности и площадь. Для π используйте значение `3.14159`. Выполните каждое вычисление внутри оператора `printf` и используйте спецификатор преобразования `%f`.

1.5 Напечатайте узор шахматной доски при помощи символов пробела и *, используя восемь операторов `printf`. Затем нарисуйте ту же самую картинку, используя всего лишь один оператор `printf`.

1.6 Ввести с клавиатуры одно пятизначное число, а затем вывести все пять цифр на печать, отделяя их пробелами. Например, если пользователь введет `31221`, программа должна вывести `3 1 2 2 1`.

Тема 2

Сборка решений[8]

Эволюция языков программирования. Ассемблерные языки. Языки высокого уровня. Трансляция программ: компиляция и интерпретация. Фазы сборки решения. Препроцессоры. Заголовочные файлы. Раздельная компиляция. Компоновка, редактор связей. Таблица внешних ссылок. Разрешение зависимостей. Объектный модуль, исполняемый файл, статические и динамические библиотеки.

2.1 Краткая история языков программирования

Первые языки программирования были машинными. Каждый центральный процессор (CPU) имел (и до сих пор имеет) фиксированный набор сравнительно простых команд, которые он умеет выполнять, например, сложить или перемножить два числа, скопировать значение из одной ячейки памяти в другую, проверить

[8] Не весь материал данной главы может быть изначально понятным, поэтому рекомендую прочитать ее два раза: первый раз – перед изучением последующих глав, не обращая внимания на незнакомые термины и концепции, а второй раз – после.

некоторое условие и в случае его выполнения осуществить переход к определенной инструкции и т.д. Программирование велось в машинных кодах, то есть по сути в числах, так как каждая команда кодировалась с помощью определенной последовательности нулей и единиц. Затем последовало первое «упрощение» языков – вместо чисел появились *мнемонические названия команд*. Например, для обозначения операции сложения двух чисел использовалась не соответствующая ей числовая комбинация, а слово ADD. Логически программы не стали проще, но теперь хотя бы вместо одной большой последовательности чисел программист видел перед собой набор более-менее читаемых команд. Так появились *ассемблерные языки*. Они взаимно-однозначно отображались в машинные, не несли в себе принципиально новых идей и по сути служили лишь для удобства работы с машинным языком, так как человекочитаемые названия команд ADD, MUL, SUB, MOV, JMP гораздо проще запомнить, чем их числовые эквиваленты.

Затем задачи, решаемые с помощью компьютеров, еще больше усложнились, программы стали настолько большими и громоздкими, что эффективная работа с ними на ассемблерном уровне была практически невозможной. Начали появляться так называемые языки высокого уровня. Первый из них – Фортран – появился в 1957 году. Этот язык был создан группой программистов под руководством Джона Бэкуса из корпорации IBM. Название Fortran является сокращением от FORmula TRANslator (переводчик формул). Фортран широко используется в первую очередь для научных и инженерных вычислений.

Следующим языком был Лисп – 1958 год – функциональный язык, принципиально отличающийся своей парадигмой от структурированных языков программирования, с которыми мы, в основном, имеем дело сегодня, и к которым, в частности, относится язык Си.

В 1970 году появился Паскаль, ставший на то время одним из наиболее известных языков программирования. Паскаль использовался в основном для обучения программированию в старших классах школы и на первых курсах вузов.

В 1972 году сотрудник компании Bell Labs Денис Ритчи создал специально для разработки операционной системы UNIX язык Си, поэтому он и по сей день является языком системного программирования.

По мере эволюционного развития всех этих и других языков начали возникать различные *парадигмы* программирования: *структурированное, функциональное, объектно-ориентированное, событийное* и т.д. С 1960 по 1970 были разработаны все основные концепции языков программирования. Каждая новая парадигма все более и более «удаляет» нас от технических аспектов исполнителя и приближает к реальному миру. В идеале, компьютерные программы должны быть неотличимы от текстов на естественном языке. Примеры того, как это могло бы выглядеть на практике, мы уже видели в научно-фантастических фильмах. Собственно, искусственный интеллект, изображаемый в подобных художественных фильмах, это и есть образ идеального интерпретатора естественного языка. Мы просто описываем компьютеру нашу задачу на русском (английском или любом другом) языке, и он ее решает для нас.

2.2 Трансляция программ

Неизвестно, как будут обстоять дела в будущем, но сейчас все исполнители понимают только один язык – машинный, а все остальные языки программирования – виртуальные, в том смысле, что машине они не известны и более того – они ей не нужны. Они нужны только человеку для более удобного и эффективного написания программ. Как правило, они не делают *процесс вычислений* более эффективным, они делают таковым *процесс написания программ*.

В связи с этим любая программа прежде, чем быть выполненной, должна быть транслирована на машинный язык исполнителя. Под общим термином *трансляция* понимают либо *компиляцию*, либо *интерпретацию*. Компиляция – непосредственный *перевод* программы с языка высокого уровня в эквивалентную программу на низкоуровневом языке, близком машинному или им являющимся. Примеры компилируемых языков – Си/C++, Бейсик, Паскаль и другие. Откомпилированная программа выполняется на компьютере, как правило, без посторонней помощи. *Интерпретация, в свою очередь* – это покомандное *выполнение* исходного кода программы интерпретатором без полной предварительной компиляции.

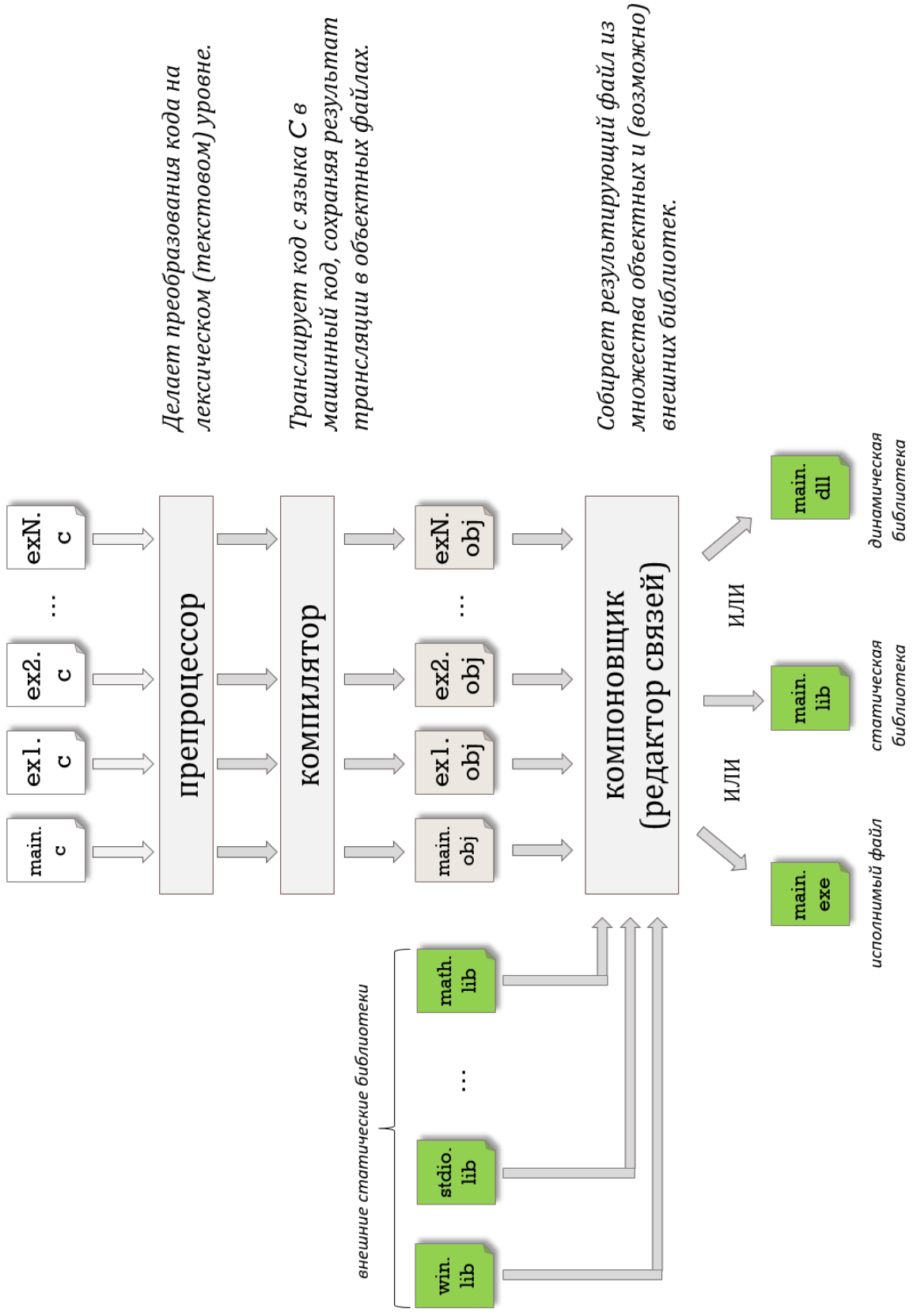


Рис. 2.1: Фазы сборки программного продукта

Пример – языки Питон, Руби, JavaScript и др. Интерпретируемую программу можно выполнить на компьютере только с помощью интерпретатора.

В виду сложности современных языков программирования процесс сборки программного продукта является многофазовым (Рис. 2.1), при этом фаза компиляции, являясь самой сложной из всех, в свою очередь, также состоит из ряда этапов (Рис. 2.2). Таким образом, наша программа проделывает огромный путь от кода в текстовом редакторе до исполняемого файла.

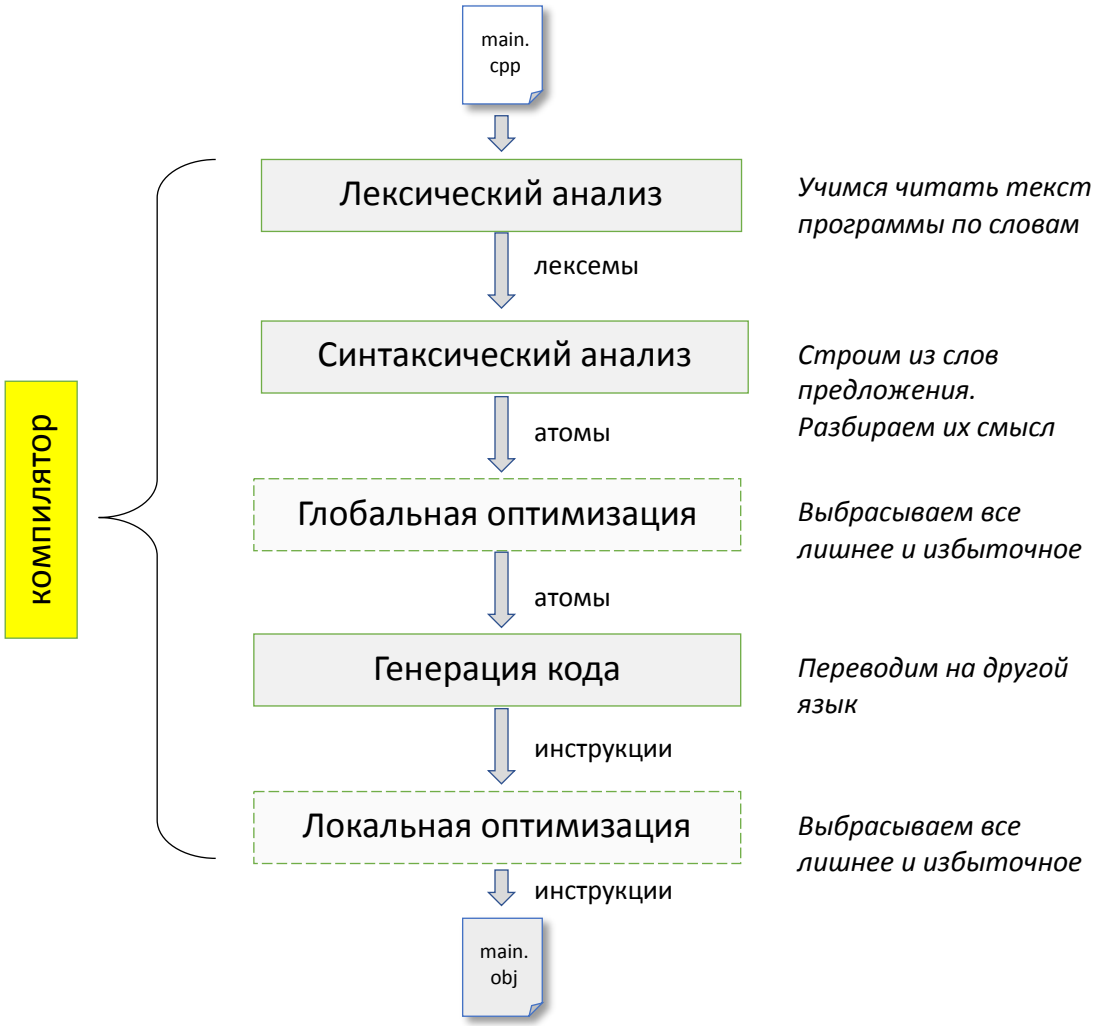


Рис. 2.2: Этапы компиляции

Рассмотрение этапов компиляции составляет предмет отдельного учебного пособия, поэтому пропустим его и кратко опишем каждую из фаз сборки решения.

2.3 Препроцессор

Препроцессор — это компьютерная программа, обрабатывающая данные, которые затем подаются на вход другой программе (например, компилятору). Отсюда и название – препроцессор (от англ. preprocess – *предварительная обработка*). О данных на выходе препроцессора говорят, что они находятся в *препроцессированной* форме, пригодной для обработки последующими программами. Результат и вид обработки зависят от вида препроцессора – так, некоторые препроцессоры могут выполнить только простой текстовый анализ, другие же способны по своим возможностям сравниться с самими компиляторами. Наиболее частый случай использования препроцессора — обработка исходного кода перед передачей его на следующий шаг компиляции.

Препроцессоры можно разделить на два класса – лексические и синтаксические. Лексическими называют низкоуровневые препроцессоры, потому что они требуют только лексического анализа, то есть они обрабатывают исходный текст, не вникая в его синтаксическую структуру и выполняя простую замену лексем и специальных символов заданными последовательностями символов в соответствии с некоторыми правилами. Обычно они выполняют замену макросов (или иногда говорят *раскрытие макросов*), вставляют содержимое других текстовых файлов, а также условную компиляцию.

Синтаксические препроцессоры разбирают синтаксическую структуру текста и вносят изменения уже в нее. Обычно они используются для уточнения синтаксиса языка, расширения языка путем добавления новых элементов. Одним из примеров является язык XSLT, с помощью которого можно преобразовать любой XML-подобный документ (например, HTML) в любой другой.

Препроцессор языка Си относится к классу лексических и работает с кодом программы на текстовом уровне. Данный препроцессор не знает синтаксиса языка Си – для него это просто последовательность символов, не более. Однако, несмотря на это, препроцессор существенно упрощает работу с исходным кодом. Зачем же он нужен? Препроцессор языка Си выполняет следующие операции:

1. Удаляет из текста все комментарии.

2. Выполняет директивы, начинающиеся с символа `#`.
3. Раскрывает макросы.
4. Делает еще кое-какую полезную работу...

Первая операция – простая и очевидная. Комментарии необходимы только программисту, поэтому препроцессор может их спокойно удалить.

Вторая операция является, в свою очередь, самой востребованной. Одна из наиболее часто используемых директив – `include`, которая вставляет содержимое одного файла в другой. Когда нам это можем потребоваться? Например, при использовании внешних (например, библиотечных) функций в коде программы. В языке Си все, что мы используем, должно быть объявлено заранее – в том числе и функции. Но как быть, если мы хотим использовать библиотечную функцию, код которой мы не то, что не хотим вставлять в нашу программу, он может быть даже нам недоступен – библиотека может храниться уже в скомпилированном виде. Например, мы хотим вызвать библиотечную функцию для вычисления синуса – `double sin(double x)`. Для этих целей в языке Си существует механизм разделяемых объявления и определения функции. *Объявление функции* – это ее заголовок, заканчивающийся точкой с запятой. В этом заголовке (который еще называется *сигнатурой функции*, если из него выкинуть имена переменных) есть вся информация, необходимая для вызова этой функции – тип возврата, количество и типы параметров, имя функции. *Определение* же *функции* помимо заголовка содержит еще и тело функции (код функции), которое определяет (задает) ее поведение. Соответственно, если мы хотим в нашей программе использовать стороннюю (библиотечную) функцию `sin`, то мы можем ее просто *объявить* перед `main`. После этого мы сможем вызывать ее так же, как и любую другую функцию:

```
double sin(double x);
int main(){
    // ...
    double x = sin(2 * Pi);
    // ...
    return 0;
}
```

Но как компилятор узнает, как работает функция `sin`? Никак. Но в этом нет ничего страшного, потому что ему это и не нужно. Компилятор же не *выполняет* функции, а занимается трансляцией кода. Код функции `sin` уже оттранслирован и лежит где-то во внешнем библиотечном файле. Всякий раз, когда компилятор встречает в программе вызов функции sin, он вставляет на это место специальную заглушку и в специальной *таблице внешних ссылок* делает пометку на местоположение этой заглушки вместе с описанием того, вызов какой функции здесь требуется (используя сигнатуру объявленной функции). Вся остальная работа будет предоставлена компоновщику, одной из задач которого, в частности, и является *разрешение* подобных *внешних ссылок*. Компоновщику на вход передается объектный файл нашей программы и библиотечный файл со скомпилированной функцией sin, из чего он и *скомпонует* одну работающую программу, заменив все заглушки на вызовы реальных функций.

С этим разобрались, но остается одна проблема. Объявление одной функции может не представлять трудностей, но если подобных библиотечных функций нам потребуется несколько десятков или сотен? Не стоит забывать, что наша программа не является единственным потребителем библиотечных функций – сами эти функции могут вызывать друг друга. Добавим к этому объявления специальных констант. Ну и наконец, зачем нам в каждой программе, в которой мы хотим использовать эти функции, таскать с собой все эти объявления. Именно для решения этой проблемы и существует одна из самых часто используемых директив препроцессора Си – `#include`. Как уже было ранее отмечено, она делает ровно то, что указано в ее названии – вставляет содержимое файла, имя которого идет после директивы, *вместо* этой директивы. Все, что нам осталось сделать, это вынести все объявления библиотечных функций и все связанные с ними константы в специальный файл (он называется *заголовочным*, так как содержит *заголовки* функций, и имеет расширение .h от слова `headers`), а затем подключить его с помощью директивы `#include`. При этом, возвращаясь к тому, с чего мы начали, еще раз уточним – препроцессору абсолютно все равно, что находится в файле, соответственно, это может быть любой файл с любым расширением – не обязательно заголовочный файл с заголовками функций.

Препроцессор просто открывает файл и вставляет его содержимое на место директивы.

Еще одна функция препроцессора – работа с константами и макросами. Константы и макросы препроцессора используются для именования небольших фрагментов кода. Например, объявление константы:

```
#define BUFFER_SIZE ( 1024 )
```

Всюду далее в программе мы можем использовать идентификатор `BUFFER_SIZE` вместо числа `1024`. Когда нам впоследствии понадобится поменять размер буфера, мы внесем изменения только в одну вышеуказанную строку.

Если мы посмотрим на оставшиеся возможности препроцессора Си, то увидим, что все они тоже работают на лексическом уровне программы, не углубляясь в синтаксис. Тем не менее, несмотря на кажущуюся примитивность и простоту этой работы, это исключительно важная фаза сборки программы, без которой написание кода было бы менее удобным, а в случае крупных проектов, трудноосуществимым.

2.4 Компиляция

Следующая стадия сборки приложения – компиляция, во время которой подготовленный препроцессором код переводится с языка Си на низкоуровневый машинный язык. Данная фаза является наиболее сложной из всех, и сама состоит из нескольких этапов, рассмотрение которых выходит за рамки данного учебного пособия.

Результатом компиляции является так называемый *объектный модуль* (слово «объектный» здесь не имеет ничего общего с объектно-ориентированным программированием, а является лишь историческим наследием). В операционной системе Windows объектные файлы имеют расширение «`obj`», в UNIX-подобных системах – расширение «`o`». Этот файл еще нельзя запустить на выполнение, так как он не содержит всей необходимой для этого информации. В нем находится исключительно результат компиляции какого-то одного конкретного файла с исходным кодом. Для того, чтобы объектный модуль превратился в исполняемый файл,

в него необходимо добавить (или его необходимо связать – «слинковать») код других объектных модулей, статических и динамических библиотек, а также некоторую дополнительную служебную информацию, необходимую операционной системе.

Откуда могут взяться другие объектные модули? Все упражнения, выполняемые в рамках данного пособия, представляют собой совсем небольшие программы, умещающиеся в одном файле, поэтому компилятор в нашем случае будет всегда компилировать один файл с исходным кодом. В то же время суммарный объем кода сложного программного продукта, не считая сторонних библиотек, может исчисляться десятками, а то и сотнями тысяч строк. Очевидно, что работать с кодом такого размера в одном файле невозможно, поэтому код реальных приложений обычно разбивается на многочисленные файлы, которые логически группируются в иерархическую модульную структуру. Для сборки подобных приложений на помощь приходит *раздельная компиляция*, смысл которой следует из ее названия – это *разделение кода программы на несколько файлов и компилирование каждого по отдельности*. Еще раз сформулируем причины, по которым это бывает необходимо:

– Неудобство работы с исходным кодом большого объема.
– Декомпозиция программы на отдельные модули, которые решают конкретные подзадачи. Таким образом весь код приобретает логическую структуру, работать с которой проще и эффективней.
– Разделение программы на отдельные модули с целью повторного использования этих модулей в других программах, то есть создание *библиотек кода*.

Когда часть программы выделяется в модуль (единицу компиляции), остальной части программы (а если быть точным, то компилятору, который будет обрабатывать остальную часть программы) надо каким-то образом объяснить, что имеется в этом модуле. Для этого как раз и служат *заголовочные файлы*, речь о которых шла в предыдущем разделе. Как правило, модуль состоит из двух файлов: *заголовочного* (*интерфейс*) и файла *реализации*. Заголовочный файл содержит все объявления, которые должны быть доступны программисту, использующему функциональность модуля.

Таким образом, мы можем разбить наше приложение на несколько, скажем, N файлов с исходным кодом и после их

компиляции получится N объектных файлов. Добавим к этому еще внешние библиотеки и файлы ресурсов. Из всего этого теперь необходимо собрать одно работающее приложение, чем и занимается на следующей стадии компоновщик.

2.5 Компоновка

Компоновщик (также *редактор связей* или *линкер*, от англ. *link editor*, *linker*) — это программа, которая осуществляет *компоновку* (*«линковку»*): принимает на вход один или несколько объектных модулей и файлов библиотек и собирает из них исполняемый файл (или файл динамической или статической библиотеки).

Как уже было упомянуто выше, результатом работы компилятора является не исполняемый модуль, который можно тут же запустить, а так называемый *объектный модуль*. Существуют различные форматы объектных файлов (например, COFF, ELF, PE и многие другие). В объектном файле помимо самого кода содержится много вспомогательной информации. В частности, можно выделить следующие часто встречающиеся разделы объектного модуля:

1. заголовок (описание структуры файла – размеры секций и их местоположение, значения различных параметров),
2. сегмент кода (результат компиляции),
3. сегмент инициализированных данных (инициализированные глобальные переменные),
4. сегмент данных только для чтения (rodata, инициализированные статические константы),
5. BSS сегмент (неинициализированные статические переменные),
6. таблицы внешних связей и ссылок для экспорта,
7. информация для перемещения кода,
8. отладочная информация,
9. ...

Исполняемый модуль (*executable file*) – файл, содержащий программу в том виде, в котором она может быть выполнена исполнителем (например, компьютером). Как правило, под операционной системой Windows, это файл с расширением .exe или .com. Перед исполнением программа загружается в память, и выпол-

няются некоторые подготовительные операции (настройка окружения, загрузка библиотек). Исполняемый модуль имеет схожий с объектным файлом формат. Принципиальная разница заключается в том, что первый содержит в себе всю информацию (код и данные) из всех объектных файлов и статических библиотек, из которых он собирался, и все зависимости между ними уже разрешены. При этом в исполняемый модуль берутся не все подряд содержимое объектных файлов и библиотек, а только то, которое используется. Например, если из библиотеки математических функций мы используем какую-то одну функцию, то именно ее код и будет включен в результирующий файл (см Рис. 2.3).

Второе отличие – в исполняемом файле есть так называемая точка входа, с которой начинается выполнение программы. В языке Си – это функция `main`. В библиотеках этой функции нет, так как библиотеку нельзя «выполнить».

Исполняемый файл – не единственный тип файла, который может сделать компоновщик. Помимо него, он может собрать из объектных модулей так называемую *библиотеку*. Библиотечный файл (с расширением `.lib`, `.dll` под Windows или `.a`, `.so` под UNIX) представляет собой упакованную коллекцию объектных файлов. Смысл библиотечного файла заключен в его названии – это просто хранилище кода, которое используется другими программами по необходимости.

Библиотеки кода бывают статическими и динамическими. Разница между ними – во времени, когда код библиотечной функции связывается с кодом основной программы. В случае статической библиотеки это происходит на этапе компоновки. При этом код библиотечной функции не просто связывается, а переписывается компоновщиком в результирующий исполняемый файл и становится, таким образом, его частью. Таким образом, в .exe файле присутствует как весь код, который был написан нами, так и весь код статических библиотек, который мы используем, или который необходим используемым нами функциям.

У статических библиотек есть свои достоинства и недостатки. К достоинствам можно отнести то, что весь необходимый код включается в один исполняемый файл, поэтому файлы библиотек «таскать» вместе с `.exe` файлом не надо. В качестве недостатков можно выделить следующее:

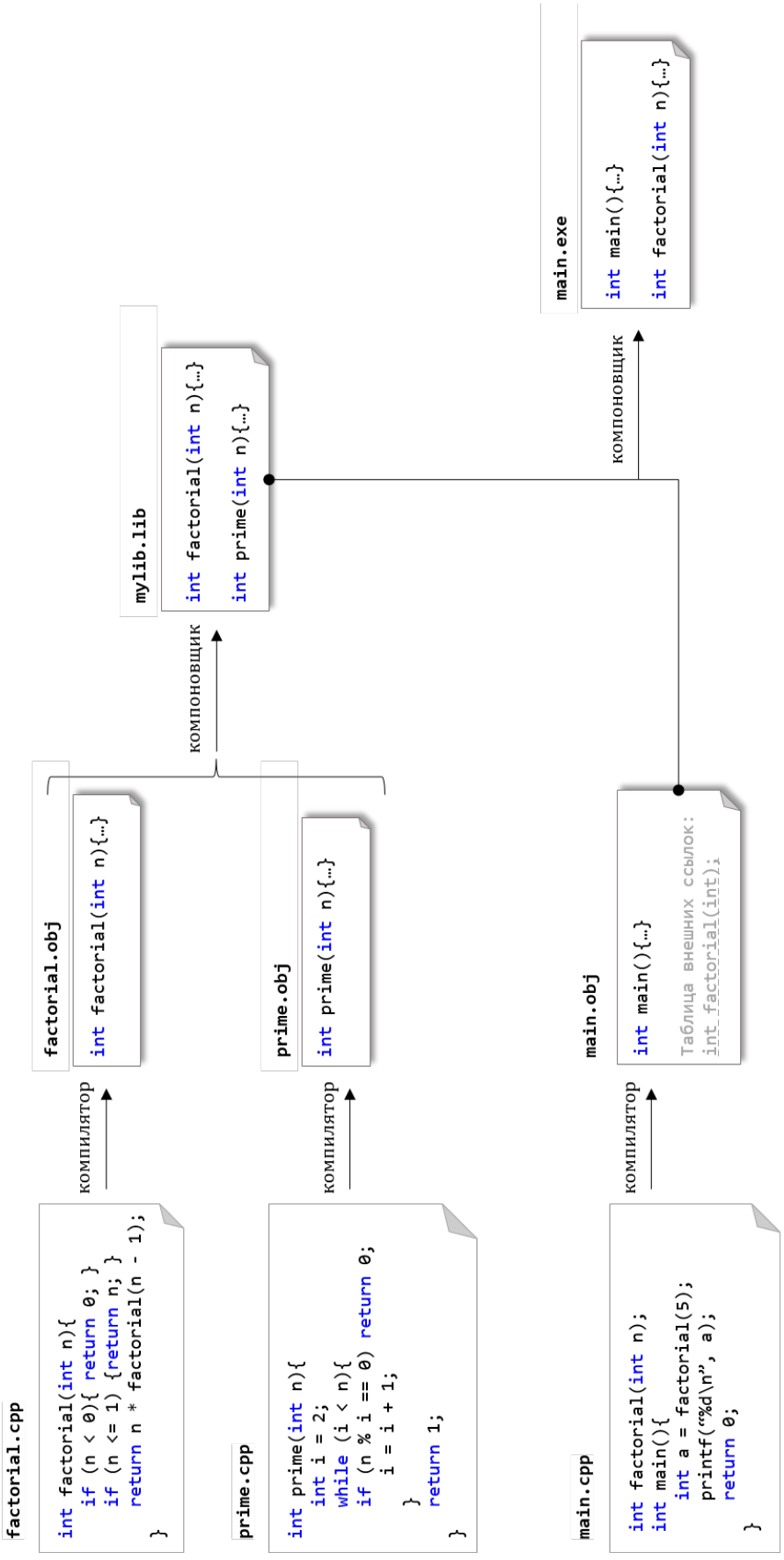


Рис. 2.3: Компоновка программы

– исполняемый файл занимает больше места на диске и в памяти;
– при обнаружении ошибок в библиотеке требуется повторная сборка всех программ.

Пример использования статических библиотек – стандартные библиотеки языков программирования.

Код из динамической библиотеки связывается с программой во время ее выполнения. Во время компоновки вместо кода функции вставляется специальная заглушка, вид и поведение которой зависит от настроек компилятора. Причем требуемая библиотека может загружаться как в момент загрузки самого приложения, так и в момент первого вызова любой внешней библиотечной функции. Второй случай имеет положительный момент – если функция ни разу не будет вызвана во время работы программы, то не будет тратиться время и память на загрузку библиотеки. Примеры использования динамических библиотек:
– библиотеки *плагинов*, расширяющие функциональность программы;
– библиотеки общего пользования операционной системы.

К достоинствам динамических библиотек можно отнести:
– экономия памяти за счет использования одной библиотеки несколькими выполняющимися программами;
– возможность быстрого исправления ошибок в библиотечном коде (достаточно заменить файл библиотеки и перезапустить работающие программы).

К недостаткам:
– при внесении некорректных изменений в библиотеку зависящие от нее программы могут перестать работать;
– конфликт версий динамических библиотек (разные программы могут ожидать разные версии библиотек);
– для работы программы необходимо иметь `.dll` файл библиотеки.

Тема 3

Основы структурированного программирования

Арифметические и логические выражения. Префиксный и постфиксный инкремент и декремент. Арифметические и логические операции. Приоритеты. Операторы: пустой, одиночный, составной. Основные конструкции языка структурированного программирования: последовательность операторов, ветвление, цикл. Циклы `while`, `do-while`, `for`. Тело цикла, итерация, вложенные циклы. Дерево программы.

3.1 Выражения

Выражения – основная рабочая единица языка Си, как, впрочем, и любого другого. В Си существуют выражения двух типов: арифметические и логические. Выражение по своей сути есть синоним слова «формула» в математике. Оно состоит из идентификаторов, констант, символов операций и скобок и обладает главной особенностью – *вычислимостью*, то есть, если мы подставим вместо идентификаторов их текущие значения, то сможем вычислить

образовавшуюся формулу, получив конкретный числовой результат: целочисленный или дробный.

С арифметическими выражениями мы познакомились в первой главе. Помимо перечисленных основных операций `+`, `-`, `*`, `/` и `%` язык Си также имеет операцию *унарного минуса*, который инвертирует знак своего операнда (переменной `a` будет присвоено значение переменной `b` с противоположным знаком):

```
a = -b;
```

унарного плюса (который, собственно, ничего не делает и существует только для полноты картины), а также специальные операции *префиксного и постфиксного инкремента* «++», увеличивающего значение аргумента на единицу, и *декремента* «--», уменьшающего значение аргумента на единицу. Поведение этих операций можно описать с помощью таблицы 3.1.

Таблица 3.1: Постфиксный и префиксный инкремент и декремент

Операция	Поведение	Пример

Постфиксный инкремент	1. Сохраняет значение b в промежуточной временной переменной
	2. Увеличивает на единицу значение переменной b
	3. Возвращает значение промежуточной временной переменной
		`int a, b = 1;`
		`a = b++;`
		`printf("%d %d", a, b);`
		Вывод на экран:
		`1 2`

Префиксный инкремент	1. Увеличивает на единицу значение переменной b
	2. Возвращает значение переменной b
		`int a, b = 1;`
		`a = ++b;`
		`printf("%d %d", a, b);`
		Вывод на экран:
		`2 2`

Декремент ведет себя так же, только не увеличивает, а уменьшает значения соответствующих переменных на единицу.

Так как описанные в таблице операции изменяют значения переменных, над которыми они применяются, то мы не можем

использовать их с константами. В частности, следующая строка вызовет ошибку компиляции:

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

потому что для работы инкременту и декременту нужно *леводопустимое* выражение (l-value), с которым связана ячейка памяти, а не просто литеральная константа.

С этими операциями необходимо обращаться осторожно, особенно используя их для одной и той же переменной в составе одного выражения, потому что результат не всегда может оказаться таким, как мы ожидали. Рассмотрим пример:

```
int a, b = 1, c;
a = b++ * b++;
printf("%d %d\n", a, b);
b = 1;
a = ++b * ++b;
printf("%d %d", a, b);
```

Вывод на экран:

```
1 3
9 3
```

Внимательно проанализировав код, приведенный выше, вы поймете, что в нем не все так очевидно. Самое удобное использование инкрементов и декрементов – это когда нам необходимо изменить на единицу какую-то одну переменную. В этом случае, вместо сравнительно длинной конструкции:

```
i = i + 1;
```

можно просто написать

```
++i;
```

При этом лучше (если это не принципиально) использовать именно префиксные версии операций, так как они не создают временных переменных для хранения старых значений операндов.

Для удобства программиста язык Си предоставляет ряд комбинированных операторов, являющихся связками арифметических операций и оператора присваивания:

```
a += 4;    то же, что и    a = a + 4;
```

```
a -= 1;   то же, что и   a = a - 1;
a *= 10;  то же, что и   a = a * 10;
a /= 10;  то же, что и   a = a / 10;
a %= 2;   то же, что и   a = a % 2;
```

Логическое выражение аналогично арифметическому – оно тоже представляет собой некоторую формулу, состоящую из идентификаторов, констант и символов операций; мы тоже можем его вычислить и получить конкретный результат. Основное отличие состоит в том, что в логических выражениях результат может быть лишь одним из двух: `0` – ложь, `1` – истина. Таким образом, логическое выражение представляет собой вопрос, на который можно ответить либо «да», либо «нет».

Еще одним отличием является возможность использования операций *сравнения* и *логических связок*, результатом применения которых также являются истина или ложь. Операций сравнения всего шесть: `<`, `>`, `<=`, `>=`, `!=`, `==`. Следует сразу запомнить, что в отличие от ряда других языков программирования, в языке Си **операция сравнения «==» отличается от операции присваивания «=»**. Одна из наиболее распространенных ошибок начинающих программистов, особенно тех, кто переходит с таких языков, как Basic или Pascal – использование одного символа равенства вместо двух. Плохая новость состоит в том, что это не является синтаксической ошибкой и ваше приложение скомпилируется и запустится, а ошибочный результат может проявить себя совсем нетривиальным образом.

Логические операции позволяют комбинировать отдельные логические выражения, получая таким образом более сложные «вопросы». Их всего три: `||` – «ИЛИ», `&&` – «И», `!` – «НЕ». Операции «И» и «ИЛИ» – бинарные, «НЕ» – унарная. Для описания работы этих операций обычно пользуются *таблицами истинности*, в которых перечислены все возможные комбинации значений аргументов и соответствующие им значения операций:

x	y	x && y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x || y
0	0	0
0	1	1
1	0	1
1	1	1

x	!x
0	1
1	0

Мы видим, что операция «И» истинна тогда и только тогда, когда оба ее аргумента истинны, операция «ИЛИ» истинна, когда хотя бы один из ее аргументов истинен, а операция «НЕ» просто инвертирует истинность своего аргумента.

Вычисление выражений идет согласно *приоритетам* операций, представленным в таблице 3.2, при этом самый высокий приоритет, очевидно, у скобок.

Таблица 3.2: Приоритеты операций

Операция	**Синтаксис**	**Приоритет** (`0` – самый высокий)
Скобки	`(a)`	`0`
Префиксный инкремент	`++a`	1
Постфиксный инкремент	`a++`	1
Префиксный декремент	`--a`	1
Постфиксный декремент	`a--`	1
Унарный минус	`-a`	1
Логическое «НЕ»	`!a`	1
Умножение	`a * b`	2
Деление	`a / b`	2
Взятие остатка от деления	`a % b`	2
Сложение	`a + b`	3
Вычитание	`a – b`	3
Больше	`a > b`	4
Меньше	`a < b`	4
Больше или равно	`a >= b`	4

Меньше или равно	`a <= b`	4
Равно	`a == b`	5
Неравно	`a != b`	5
Логическое «И»	`a && b`	6
Логическое «ИЛИ»	`a || b`	7

Рассмотрим несколько примеров логических выражений.

Пример 3.1. Пусть есть переменная `A`. Необходимо записать выражение, которое истинно, если значение `A` четное.

Решение: `(A % 2 == 1)`.

Пример 3.2. Для двух переменных `A` и `B` записать выражение, которое истинно, когда эти переменные положительны.

Решение: `(A > 0 && B > 0)`.

Пример 3.3. Даны две переменные `A` и `B`. Записать выражение, которое истинно, если эти переменные имеют разные значения.

Решение: `(A != B)`.

Пример 3.4. Даны три переменные `A`, `B` и `C`. Записать выражение, которое истинно, когда `C` равно максимальному значению из `A` и `B`.

Решение: `((A <= B && C == B) || (A >= B && C == A))`.

Во всех вышеперечисленных примерах знаки операций отделены слева и справа пробелами. С точки зрения синтаксиса это не обязательно: мы можем написать `A<=B` вместо `A <= B`, но в целях повышения удобочитаемости лучше этого не делать.

Любой квалифицированный программист должен стараться соблюдать три основных правила: писать *корректный*, *эффективный* и *удобочитаемый* код. Корректность означает отсутствие ошибок (логических и синтаксических), эффективность – способность выбрать из широкого класса применимых в данном конкретном случае алгоритмов наиболее эффективный (либо с точки зрения времени работы, либо требуемой памяти), ну а к удобочитаемости как раз и относятся способность ясного оформления кода (пробелы и отступы), понятное именование переменных и функций,

комментирование кода и другие вещи, который делают код читаемым и понятным как для других программистов, так и для самого автора в будущем.

3.2 Основные конструкции

Программа на языке Си состоит из операторов, заканчивающихся точкой с запятой. Самым простым из них является *пустой оператор*, состоящий только из точки с запятой:

```
;
```

который ничего не делает. Еще одним оператором, с которым мы уже познакомились, является *присваивание*. Идущие друг за другом операторы называются *последовательностью операторов*, что является первой базовой конструкцией любого структурированного языка программирования. При этом программа выполняется именно в этой последовательности – прямолинейно, оператор за оператором:

```
<оператор 1>;
<оператор 2>;
…
<оператор N>;
```

Язык Си допускает запись операторов в одну строку, однако лучше этого избегать, так как это приводит к плохо читаемому коду.

Операторы можно объединять в группы (блоки) с помощью фигурных скобок, создавая тем самым *сложные (составные) операторы*:

```
{
    <оператор 1>;
    <оператор 2>;
    …
    <оператор N>;
}
```

Ключевой особенностью блоков является то, что они могут быть сколь угодно вложенными друг в друга. В языке Си блоки могут существовать сами по себе, в то время как в некоторых других языках

они используются только вместе с ветвлениями и циклами. Сами блоки кода не добавляют ничего нового в логику исполнения программы – она по-прежнему выполняется последовательно: оператор за оператором, блок за блоком.

Если бы мы были ограничены возможностью писать только последовательно выполняющиеся программы, то это бы существенным образом сузило класс решаемых задач. По сути, это сделало бы такой язык практически бесполезным. Поэтому в языке Си, как и в любом другом структурированном языке программирования, помимо конструкции последовательного выполнения операторов, есть еще две фундаментальные конструкции: *ветвление* и *цикл*.

Ветвление используется тогда, когда возникает необходимость изменить ход выполнения программы добавлением развилки, разбивающей его на две и более альтернативных ветви. Синтаксис конструкции ветвления следующий:

```
if (условие) <оператор>
```

Условием является *любое* выражение. Как правило, применяется логическое выражение, но язык Си позволяет использовать и любое арифметическое, трактуя его нулевое значение как ложь, а любое отличное от нуля – как истину. Логика выполнения конструкции очень простая – *если условие истинно, то выполняется идущий следом <оператор>, в противном случае он пропускается*.

В качестве *<оператор>* может быть любой оператор: *одиночный* (если значение `a` отрицательно, то мы его делаем положительным):

```
if (a < 0) a = a * -1;
```

составной (если `a` больше `b`, то меняем значения переменных местами):

```
if (a > b) {
     int tmp = a;
     a = b;
     b = tmp;
}
```

и даже *пустой* (это допустимо, но вы должны хорошо понимать, зачем вам это нужно):

```
if (a % 2);
```

Всюду далее, если не оговорено иное, под `<оператор>` мы будем понимать любой из перечисленных выше типов операторов.

Обратите внимание на условие в последнем примере. Это арифметическое выражение, возвращающее `0`, если значение переменной `a` четное, и `1` – если нечетное. Таким образом, условие истинно, если переменная `a` нечетная.

Конструкция `if (условие)` «цепляется» к одному из операторов некоторой последовательности, делая его условным, то есть зависящим от выполнения определенного условия. Поэтому будет не совсем корректно называть саму конструкцию `if` оператором и ожидать, что после нее мы тоже должны поставить точку с запятой. Точка с запятой является частью того оператора, к которому «прицеплена» конструкция ветвления, именно поэтому она отсутствует после закрывающей фигурной скобки, потому что после фигурной скобки, ограничивающей блок кода, точка с запятой не ставится.

Иногда бывает необходимо указать альтернативный код, который будет выполнен тогда, когда условие ложно. Для этих целей служит else-секция ветвления:

```
if (условие)
    <оператор_1>
else
   <оператор_2>
```

Логика работы этой конструкции следующая: *если условие истинно, то выполняется <оператор_1>, а <оператор_2> пропускается, а если условие ложно, то все наоборот: <оператор_1> пропускается, а <оператор_2> выполняется*. Рассмотрим пример.

Пример 3.5. Написать программу, спрашивающую у пользователя два числа и выводящую «`TRUE`», если первое меньше второго. В противном случае вывести «`FALSE`».

Решение:

```
int a, b;
printf("Введите два числа: ");
scanf_s("%d %d", &a, &b);
if (a < b)
    printf("TRUE\n");
```

```
else
     printf("FALSE\n");
```

Пример 3.6. Написать программу, спрашивающую у пользователя три числа и выводящую наименьшее и наибольшее из них.

Решение:

```
int a, b, c, max, min;
printf("Введите три числа: ");
scanf_s("%d %d %d", &a, &b, &c);
if (a >= b && a >= c){
     max = a;
}
else {
     if (b >= a && b >= c)
          max = b;
     else
          max = c;
}
if (a <= b && a <= c){
     min = a;
}
else {
     if (b <= a && b <= c)
          min = b;
     else
          min = c;
}
printf("Max = %d, min = %d\n", max, min);
```

Иногда возникает необходимость повторить какой-то оператор некоторое количество раз. Для этих целей используется третья конструкция – цикл. Существуют две разновидности циклов – *управляемые условием*, и *управляемые счетчиком*. Первый используется тогда, когда мы не знаем точное количество раз, которое необходимо выполнить соответствующий оператор, но у нас есть условие, которое мы можем использовать как индикатор – пока оно истинно, мы должны выполнять цикл. Очевидно, что в этом случае переменные, входящие в условие, должны каким-то образом

меняться внутри цикла. Вторая разновидность цикла используется тогда, когда мы точно знаем, сколько раз надо выполнить фрагмент кода в цикле[9].

Язык Си имеет три конструкции для организации цикла: `for`, `while` и `do-while`. Рассмотрим каждую из них.

С синтаксической точки зрения цикл `while` является самым простым:

```
while (<условие>)
    <оператор>
```

`<оператор>` в данном случае называется *телом цикла*, а одно выполнение тела цикла называется *итерацией*. Логика работы следующая: *тело цикла будет выполняться до тех пор, пока условие является истинным*. Если условие ложно еще до выполнения цикла, то <оператор> не выполнится ни разу. Рассмотрим пример.

Пример 3.7. Программа спрашивает у пользователя число `N`, и проверяет, является ли число `N` степенью числа `3`, то есть можно ли найти такое `k`, что $3^k = N$. Если это так, то программа выводит фразу: «`N степень 3 с показателем k = K`», где вместо `N` и `K` подставляются найденные значения.

Решение:

```
int N, k = 0, power = 1;
printf("Введите число: ");
scanf_s("%d", &N);

while (power < N) {
    power = power * 3;
    k++;
}

if (power == N) {
    printf("%d степень 3 с показателем k = %d\n", N, k);
}
```

[9] В языке Си данное деление весьма условное, потому что при желании все конструкции цикла можно использовать как со счетчиком, так и с условием.

Оператор `while` еще называют *циклом с предусловием*, потому что в нем условие остановки проверяется до выполнения тела цикла по схеме:

– проверить условие: если он ложно, выйти из цикла,
– выполнить тело цикла,
– проверить условие: если он ложно, выйти из цикла,
– выполнить тело цикла,
– ...

Иногда бывает необходимо поменять эту логику и делать все наоборот – сначала выполнить тело цикла, а потом уже проверить условие:

– выполнить тело цикла,
– проверить условие: если он ложно, выйти из цикла,
– выполнить тело цикла,
– проверить условие: если он ложно, выйти из цикла,
– ...

Это нужно, например, если необходимо, чтобы тело цикла обязательно выполнилось как минимум один раз в самом начале, независимо от условия. Для этих целей в языке Си существует цикл `do-while`, которые еще называют *циклом с постусловием*:

```
do
    <оператор>
while (условие);
```

Он работает в точности так, как описано выше. Заметим, что `do-while` в отличие от остальных конструкций, меняющих логику работы программы, в силу своего синтаксиса вынужден оканчиваться точкой с запятой. Это единственное исключение в языке Си.

Пример 3.8. Напишите программу, которая запрашивает у пользователя целое число в диапазоне от `0` до `10`. Программа должна запрашивать число до тех пор, пока пользователь не введет его из нужного диапазона. После этого программа печатает на экране «`Спасибо`» и заканчивает работу.

Решение:

```
int i = 0;
do {
```

```
    printf("Введите целое число от 0 до 10: ");
    scanf_s("%d", &i);
} while (i < 0 || i > 10);
printf("Спасибо!\n");
```

Еще один оператор цикла в языке Си, это `for`, имеющий следующий синтаксис:

```
for (<инициализация>; <условие>; <инкремент>)
    <оператор>
```

Цикл `for` эквивалентен следующей записи цикла `while`:

```
<инициализация>;
while (<условие>) {
    <оператор>
    <инкремент>;
}
```

Блок инициализации выполняется ровно один раз – перед входом в цикл. В этом блоке обычно инициализируется счетчик. Условие – выражение, проверяемое каждый раз перед началом очередной итерации. Если условие истинно, то цикл выполняет `<оператор>`, если ложно – мы выходим из цикла. Блок инкремента выполняется каждый раз *после* выполнения очередной итерации. Как правило, в этом блоке счетчик увеличивается или уменьшается (в зависимости от решаемой задачи).

Пример 3.9. Напечатать все числа от `1` до `10`, разделенные пробелом.

Решение:

```
int i;
for (i = 1; i <= 10; i++)
    printf("%d ", i);
printf("\n");
```

Переменную `i` можно объявить в блоке инициализации цикла:

```
for (int i = 1; i <= 10; i++){...
```

тогда она будет *доступна только внутри этого цикла*, а за его рамками ее не будет существовать. Это связано с таким понятием как

область видимости переменной, которые мы рассмотрим в следующей главе.

Пример 3.10. Программа спрашивает у пользователя два числа A и B и выводит все числа между ними в порядке возрастания, включая и сами числа A и B. Числа выводятся через пробел в одной строке. На следующей строке выводится фраза «`Общее количество = `» и общее количество напечатанных чисел.

Решение:

```
int A, B, min, max, i;
scanf_s("%d %d", &A, &B);
if (A < B){
     max = B;
     min = A;
} else {
     max = A;
     min = B;
}
for (i = min; i <= max;)
     printf("%d ", i++);
printf("\nОбщее количество = %d\n", max - min + 1);
```

В данном примере блок инкремента в цикле `for` отсутствует, но заметим, что точка с запятой все равно ставится. Переменная `i` увеличивается в теле цикла. Побочный эффект инкремента, благодаря которому изменяется значение самой переменной, позволяет нам совместить его с передачей `i` в качестве аргумента в функцию `printf`.

Любая из трех компонент цикла `for`, записанных в скобках, может быть опущена, однако все точки с запятой должны быть на месте. Мы можем, в частности, записать цикл `for`, который будет эквивалентен циклу `while`:

```
for (;<условие>;)
     <оператор>
```

или можем написать «вечный» цикл, в котором нет условия:

```
for (;;)
     <оператор>
```

С циклами связаны два оператора, бывающие иногда полезными – это `break` и `continue`. Оператор `break` прерывает выполнение цикла, внутри которого он находится. Оператор `continue` переводит цикл на следующую итерацию, пропуская все инструкции, идущие в теле цикла после него. В этом случае `<инкремент>` в цикле `for` все равно будет выполнен.

Пример 3.11. Написать программу, которая определит, является ли введенное пользователем целое положительное число простым. Простое число – это такое, у которого есть только два делителя – `1` и оно само.

Решение:

```
int x;
scanf_s("%d", &x);
int simple = 1;
for (int i = 2; i < x; i++){
     if (x % i == 0){
          simple = 0;
          break;
     }
}
if (simple) printf("Число %d - простое\n", x);
else printf("Число %d - не простое\n", x);
```

В данном примере проиллюстрирована *флаговая техника*. Мы объявляем флаг `simple`, который равен `1`, если условие задачи выполняется, и `0` – иначе. Изначально предполагаем оптимистический сценарий – а именно, что введенное число является простым, поэтому инициализируем переменную `simple` единицей. Далее запускается цикл, который перебирает все потенциальные делители в диапазоне `[2, x-1]`, и если находится первое число, которое делит `x` нацело, то сбрасываем флаг `simple` и выходим из цикла с помощью оператора `break`. Можно было бы продолжить цикл – флаг уже не станет равным единице и ответ все равно будет корректным, но эти проходы цикла будут лишь тратить ресурсы центрального процессора впустую, особенно если `x` – большое число, а делитель был найден сразу.

В последнем примере мы также видим, что условный оператор *вложен* в цикл и, соответственно, выполняется столько раз, сколько итераций совершает цикл. Мы можем внутрь цикла вложить другой цикл, при этом внутренний цикл будет запускаться заново на каждой итерации внешнего. Рассмотрим вложенные циклы на примере.

Пример 3.12. Написать программу, которая считывает размер стороны квадрата и затем выводит его с помощью звездочек.

Решение:

```
1. int size, i, j;
2. printf("Введите размер стороны: ");
3. scanf_s("%d", &size);
4. for (i = 0; i < size; i++) {
5.   for (j = 0; j < size; j++)
6.       printf("*");
7.   printf("\n");
8. }
```

Цикл 5-6 выводит `size` звездочек друг за другом. После этого в 7-й строке печатается перенос (эта строка не является частью цикла 5-6, так как его телом является одиночный оператор на строке 6). Код 5-7, в свою очередь, являются телом внешнего цикла 4-8, поэтому будет выполняться `size` раз – таким образом у нас будет напечатано `size` строк по `size` звездочек в каждой. На Рис. 3.1 изображено дерево программы из примера 3.12.

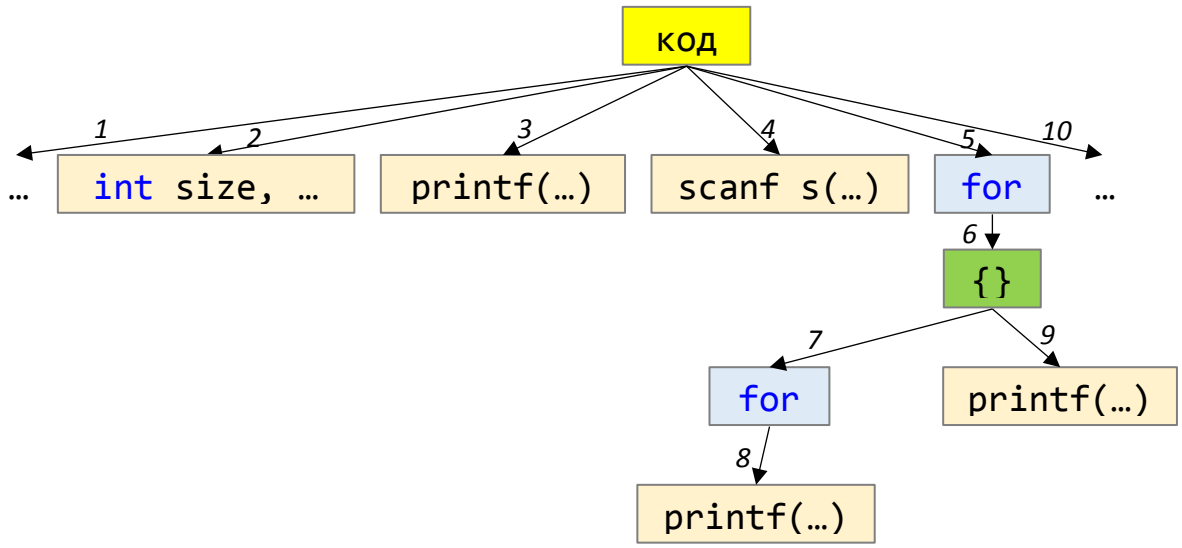


Рис. 3.1: Дерево программы из примера 3.12

Номера на стрелках соответствуют порядку выполнения инструкций. Из рисунка хорошо виден нелинейный путь выполнения программы – мы спускаемся вглубь *дерева*, либо пропуская какие-либо поддеревья (в случае условных конструкций), либо повторяя все инструкции какого-то поддерева заданное количество раз (для циклических конструкций). Из этого рисунка также хорошо видна первая конструкция структурированного программирования – последовательность операторов. Она задается операторами, являющимися непосредственными потомками какого-либо узла. Например: 1, 2, 3, 4, 5, 10, или: 7, 9.

Разобранные в данной главе три языковые конструкции: последовательное выполнение, ветвление и цикл – вместе с блочными операторами составляют основу любого языка структурированного программирования. На Рис. 3.2 проиллюстрирована разница между структурированным языком и линейным.

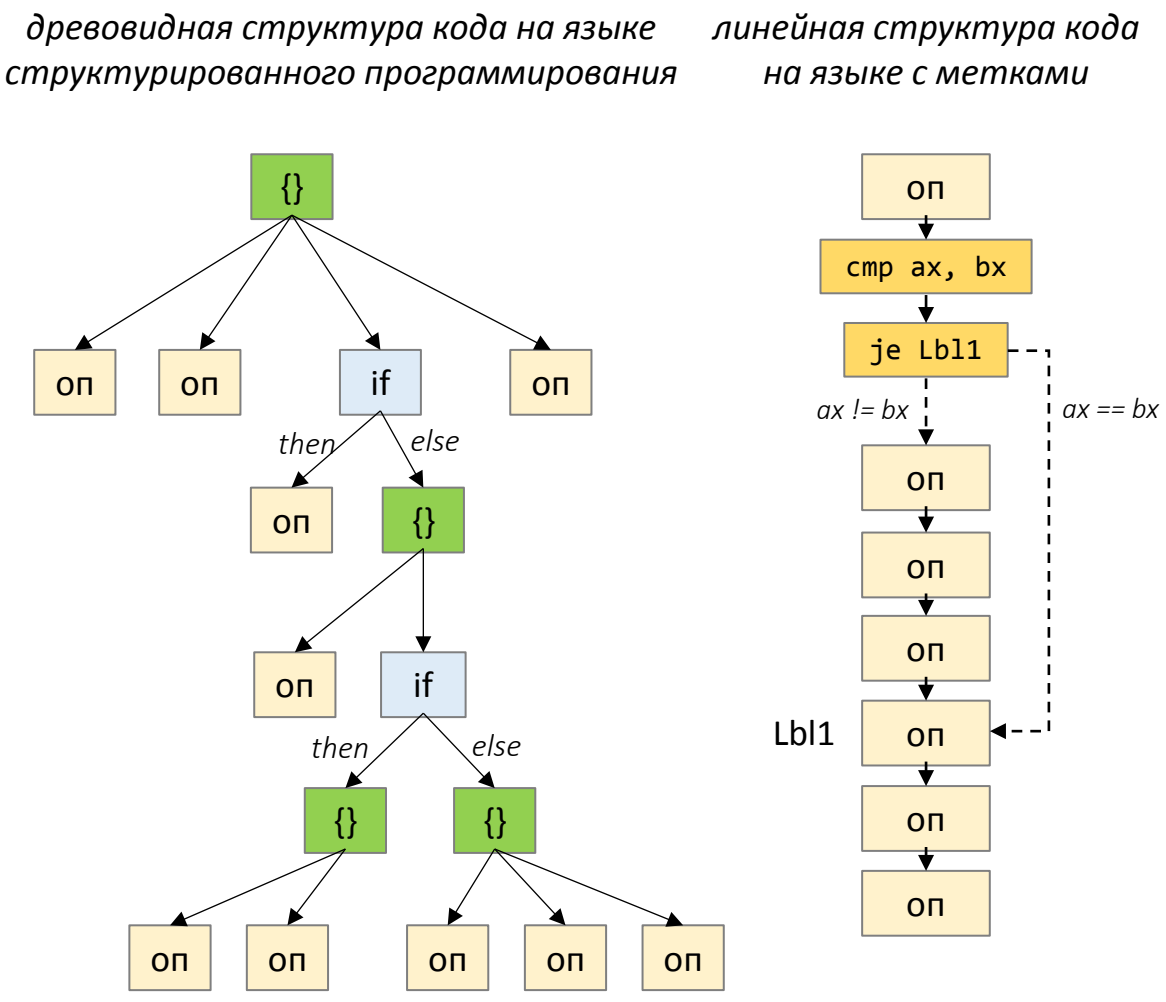


Рис. 3.2: Структурированный и линейный код

Благодаря составным операторам, которые могут содержать другие составные, а те, в свою очередь, опять могут содержать блочные операторы и т.д., мы можем *структурировать* программу в виде *дерева*, делая какие-то поддеревья условными, какие-то циклическими. Именно поэтому языки, поддерживающие использование подобного синтаксиса, называются языками *структурированного программирования*.

Упражнения

Попробуйте решить самостоятельно следующие упражнения.

3.1 Даны три целых числа: `A`, `B`, `C`. Напишите выражение, проверяющее истинность высказывания: «Каждое из чисел `A`, `B`, `C` положительное».

3.2 Дано целое положительное число. Напишите выражение, проверяющее истинность высказывания: «Данное число является четным двузначным».

3.3 Напишите выражение, проверяющее, что либо `A` четно, либо `B` четно (но не оба одновременно).

3.4 Напишите программу, спрашивающую у пользователя три числа и выводящую их в порядке возрастания.

3.5 Программа спрашивает у пользователя два числа `A` и `B` и выводит сумму всех чисел между ними (включительно).

3.6 Вычисление факториала. Программа спрашивает у пользователя число `N` ($1 \le N \le 10$, программа должна запрашивать число до тех пор, пока оно не будет введено корректно) и вычисляет значение произведения $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$. Результат выводится в формате «`N! = A`», где `N` – введенное число, `A` – значение факториала.

Решения упражнений

3.1 `(A > 0) && (B > 0) && (C > 0)`

3.2 `(A > 9) && (A < 100) && !(A % 2)`

3.3 `(A % 2) + (B % 2) == 1`

3.4
```
int a, b, c, tmp;
printf("Введите три числа: ");
scanf_s("%d %d %d", &a, &b, &c);
if (a > b) {
     tmp = a;
     a = b;
     b = tmp;
}
if (b > c) {
     tmp = b;
     b = c;
     c = tmp;
}
if (a > b) {
     tmp = a;
     a = b;
     b = tmp;
}
printf("%d %d %d\n", a, b, c);
```

3.5
```
int A, B, sum;
printf("Введите два числа: ");
scanf_s("%d %d", &A, &B);
int min, max;
if (A < B) {
     max = B; min = A;
} else {
     max = A; min = B;
}
sum = 0;
```

```
    for (int i = min; i <= max; i++)
        sum += i;
    printf("Сумма всех чисел от %d до %d = %d\n", min,
                                                max, sum);
```

3.6
```
    int N, i, result = 1;
    do {
        printf("Введите число (от 1 до 10): ");
        scanf_s("%d", &N);
    } while (N <= 0 || N > 10);

    for (i = 2; i <= N; i++)
        result = result * i;

    printf("%d! = %d\n", N, result);
```

Упражнения для самостоятельного решения

3.7 Даны три целых числа: `A`, `B`, `C`. Напишите выражение, проверяющее истинность высказывания: «Справедливо двойное неравенство `A < B < C`».

3.8 Даны два целых числа: `A` и `B`. Напишите выражение, проверяющее истинность высказывания: «Числа `A` и `B` имеют одинаковую четность».

3.9 Напишите выражение, проверяющее, что среди чисел `A`, `B`, `C` и `D` есть как минимум 2 нечетных.

3.10 Дано трехзначное число. Напишите выражение, проверяющее истинность высказывания: «Все цифры данного числа различны».

3.11 Напишите выражение, проверяющее, что `(A + B)` равно удвоенному остатку от деления `(C + D)` на `B`.

3.12 Напишите выражение, проверяющее, что `A` равно `B`, без использования оператора `==`.

3.13 Напишите выражение, проверяющее, что A меньше B, без использования оператора <.

3.14 Напишите выражение, проверяющее, что сумма последних цифр A и B больше C.

3.15 Напишите выражение, проверяющее, что из того, что A четно, следует, что B нечетно.

Логическое выражение «X влечет Y» (или «из X следует Y», обозначается X -> Y) имеет следующую таблицу истинности:

x	y	x -> y
0	0	1
0	1	1
1	0	0
1	1	1

То есть из лжи может следовать все что угодно, а из правды – только правда. Эта логическая операция называется *импликацией*.

3.16 Даны координаты двух различных полей шахматной доски x1, y1, x2, y2 (целые числа, лежащие в диапазоне 1–8). Напишите выражение, проверяющее истинность высказывания: «Ладья за один ход может перейти с поля (x1, y1) на поле (x2, y2)».

3.17 Напишите выражение, проверяющее, что из того, что A равняется B, следует, что B равняется A.

3.18 Даны два целых числа A и B (A < B). Вывести в порядке убывания все целые числа, расположенные между A и B (не включая числа A и B), а также количество напечатанных чисел.

3.19 Даны два целых числа A и B (A < B). Найти произведение всех целых чисел от A до B включительно.

3.20 Дано целое число N (> 1). Найти наименьшее целое число K, при котором выполняется неравенство 3K > N.

3.21 Палиндромом называется число или фраза текста, которая читается одинаково как слева направо, так и справа налево.

Например, каждое из следующих пятизначных целых чисел является палиндромом: `12321`, `55555`, `45554` и `11611`. Напишите программу, которая считывает с клавиатуры пятизначное целое число и определяет, является ли оно палиндромом.

3.22 Спросить у пользователя число `N` и вывести квадрат со стороной размера `N` следующим образом. Контур квадрата выводится символом `#`, а внутренности символом `+`. Например, для `N = 4` нужно вывести:

```
####
#++#
#++#
####
```

а для `N = 2`:

```
##
##
```

3.23 Подсчитать сумму всех нечетных чисел от `1` до `99`.

3.24 Подсчитать количество цифр вводимого с клавиатуры целого неотрицательного числа.

3.25 Шестизначный номер билета называют «счастливым», если в его записи сумма первых трех цифр равна сумме последних трех. Программа спрашивает у пользователя шестизначный номер билета и проверяет, является ли он счастливым. Если билет счастливый, то программа выводит на новой строке «`Ваш билет счастливый!`», иначе выводится сообщение «`Ваш билет не является счастливым.`»

Тема 4

Работа с памятью

Память как последовательность одинаковых пронумерованных ячеек. Байты, биты. Переменные: объявление, инициализация, правила именования. Типы данных: целочисленный и с плавающей запятой, знаковые и беззнаковые. Точность значения с плавающей запятой. Диапазоны значений. Приведение типов: явное и неявное. Прямая и косвенная адресация, указатели. Операторы взятия адреса и разыменования.

4.1 Переменные

В самом общем понимании работа любой программы сводится к манипуляциям с данными, а так как компьютер в основе своей понимает лишь числовые данные, то можно сказать, что жизнедеятельность любой программы сводится к манипуляциям с числами. Поэтому самым важным ресурсом компьютера и, соответственно, программы является память, в которой программы могут хранить все те данные, с которыми они работают.

Память компьютера представляет собой длинную *линейную* последовательность *ячеек одинакового размера*, называемых *байтами*. Каждый байт, в свою очередь, состоит из восьми *битов*,

где каждый бит может принимать одно из двух значений: `0` или `1`. Обычно программы работают с памятью на уровне байтов, но иногда опускаются и на уровень отдельных битов.

Все ячейки (байты) памяти пронумерованы от `0` до какого-то большого числа, зависящего от системы. Всего в однобайтной ячейке памяти можно хранить 256 различных значений. Общая формула «вместимости» N бит памяти есть 2^N. Так, с помощью четырех бит можно закодировать 16 различных значений, а с помощью 32 бит (4 байта) – 4 294 967 296 значений.

Прежде чем программа на языке Си сможет работать с памятью, она должна ее *зарезервировать* (*выделить*). Делается это при помощи *оператора объявления переменных*, который нам уже знаком по прошлым главам. Когда мы объявляем переменную, в памяти резервируется свободная ячейка (или, в зависимости от типа переменной, несколько подряд идущих ячеек – об этом далее). После этого мы можем использовать эту память для своих нужд.

Программа на языке Си имеет два способа обращения к ячейкам зарезервированной памяти: *прямой* и *косвенный*. Первый способ нам хорошо знаком – это доступ с помощью переменных, когда ячейкам памяти даются *имена*. В случае использования косвенного доступа, программа обращается к ячейкам памяти не по их *именам*, а по их *адресам*. Рассмотрим более подробно эти два механизма.

Как мы уже знаем, *переменная* – это *именованная область памяти*. Когда мы объявляем переменную, в памяти резервируется свободная ячейка (ячейки) и с ней ассоциируется данное имя. Далее, когда мы в программе используем эту переменную в выражениях или операторах, то все действия происходят как раз с тем самым зарезервированным фрагментом памяти.

Оператор объявления переменной выглядит следующим образом:

```
<тип> <имя₁> [= значение₁], …, <имяₙ> [= значениеₙ];
```

Напомним, что запись в квадратных скобках означает необязательную часть. Подытожим то, что мы уже знаем об этом операторе из предыдущих глав:

– переменные можно объявить в любом месте программы, но до момента их первого использования,
– в одном операторе можно сразу объявить несколько переменных,

– одновременно с объявлением можно указывать начальное значение переменных или, как еще говорят, *инициализировать* переменные,
– имя переменной – любая последовательность символов латинского алфавита, цифр и символа нижнего подчеркивания, которая начинается на любой допустимый символ кроме цифры.

Лучше всего переменным давать говорящие имена – `min`, `max`, `age`, `count` и т.д. – но при этом не переусердствовать, чтобы не получилось что-нибудь вроде `minAgeOfAllStudentsAmongFirstYearStudents`.

4.2 Типы данных

Язык Си – типизированный. Это означает, что каждая переменная имеет тип, который указывает, какие данные, могут в этой переменной храниться. После объявления переменной ее тип изменить уже нельзя. В языке Си существуют три большие группы базовых типов данных: целочисленные, с десятичной запятой (их еще называют с плавающей запятой) и указатели. Есть пять базовых целочисленных типов данных:

– `char`,
– `int`,
– `short int` (допускается сокращение `short`),
– `long int` (допускается сокращение `long`),
– `long long int` (допускается сокращение `long long`)

и два типа с плавающей запятой:

– `float`,
– `double`.

Целочисленные типы данных могут быть также *знаковыми* и *беззнаковыми*, что определяется с помощью спецификаторов `signed` и `unsigned` перед типом данных. Например строка

```
unsigned int uint;
```

объявляет беззнаковую переменную типа `int`.

В первом приближении тип говорит, что за значения могут храниться в соответствующих переменных. В частности, в целочисленных переменных могут храниться (как это следует из названия)

целые числа, а в переменных типа `float` и `double` – дробные (числа, в записи которых есть десятичная запятая). Однако, первое приближение не отвечает на следующие вопросы. Почему существует четыре целочисленных типа данных? И в чем разница между, например, переменной типа `int` и переменной типа `float`, если память, в конечном итоге, является последовательностью подряд идущих *одинаковых* ячеек, и какую переменную мы бы не создали, все равно ее значения будут храниться в этих ячейках? На что же именно влияет тип?

На самом деле тип переменной существенным образом влияет на две вещи: размер памяти, выделяемой под переменную, и формат, в котором значения хранятся в ячейках памяти. Тип `char` занимает в памяти ровно один байт, тип `short` – два подряд идущих байта, типы `int`, `long` и `float` – четыре байта, а типы `long long` и `double` – восемь.

Типы `char`, `short`, `int`, `long` и `long long` имеют простой формат хранения данных – числа хранятся в этих переменных в своем обычном *двоичном представлении*. Например, в восьмибитной ячейке памяти переменной типа `char` число `78` хранится как `01001110`, а число `5` хранится как `00000101`. В свою очередь, типы `float` и `double` имеют сложный внутренний формат. Существует целый стандарт (IEEE 754), который описывает, каким образом в четырех байтах типа `float` и восьми байтах типа `double` закодировано число с десятичной запятой. Изучение этого стандарта выходит за рамки учебного пособия, однако в интернете можно легко найти его описание. В связи с этим следует запомнить, что в целочисленных переменных каждый бит хранит соответствующий разряд двоичного представления числа, а для типов `float` и `double` каждый бит имеет свое собственное назначение.

Как следствие из всего вышесказанного, нетрудно заметить, что каждый тип имеет свой диапазон допустимых значений. Например, так как тип `char` занимает в памяти один байт, то хранить в такой переменной можно либо 2^8=256 различных неотрицательных целых чисел (от `0` до `255`), либо по 128 чисел разного знака (от `-128` до `127`). Для типа `int`, занимающего четыре байта, диапазон составит 2^{32} = 4 294 967 296 неотрицательных значений, либо числа в диапазоне от `-2 147 483 648` до `2 147 483 647`.

В таблице 4.1 приведены типичные характеристики базовых типов данных современных компиляторов языка Си для 32-битных систем.

Таблица 4.1: Типичные характеристики базовых типов данных

Тип	Размер в байтах	Спецификатор	Диапазон значений
`char`	1	`%c`	от -128 до 127
`unsigned char`	1	`%c`	от 0 до 255
`short`	2	`%hi`	от -32 768 до 32 767
`unsigned short`	2	`%hu`	от 0 до 65 535
`int, long`	4	`%d, %li`	от -2 147 483 648 до 2 147 483 647
`unsigned int, unsigned long`	4	`%u, %lu`	от 0 до 4 294 967 295
`long long`	8	`%lli`	от 0 до 18 446 744 073 709 551 615
`unsigned long long`	8	`%llu`	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
`float`	4	`%f`	$\pm(3{,}4 \cdot 10^{-38} \ldots 3{,}4 \cdot 10^{38})$, ~7 значащих цифр
`double`	8	`%lg`	$\pm(1{,}7 \cdot 10^{-308} \ldots 1{,}7 \cdot 10^{308})$ ~15 значащих цифр
указатели	4	`%p`	от 0 до 4 294 967 295 (адреса памяти программы)

Не стоит переоценивать величину диапазона чисел `double`. Здесь необходимо обратить внимание на количество *значащих цифр*. Их можно трактовать следующим образом. Пусть у нас есть некоторое дробное число, например, `3472,0913373`. Мы должны передвинуть десятичную запятую в самое начало, а далее действует следующее

правило: шесть цифр, идущих сразу после запятой, сохранятся в переменной, седьмая и восьмая – может да, а может и нет (в зависимости от двоичного представления), а все остальные – обнулятся: `0,34720913000`. Для типа `double` логика такая же, только граница значащих цифр проходит по 15-16 разрядам. Показатели степени говорят лишь о том, что мы можем сохранить в переменной соответствующий *порядок числа*, например, для массы электрона: $9{,}11 \cdot 10^{-31}$ кг, или числа Авогадро: $6{,}02214129 \cdot 10^{23}$ моль$^{-1}$, при этом точность хранимых значений ограничена.

4.3 Приведение типов

В языке Си не только переменные, но и константы имеют тип. Например, константы: `3` и `-7` – это целочисленные константы, а `3,14` и `-0,012` – константы с плавающей запятой. По умолчанию, тип целочисленной константы – `int`, а тип константы с плавающей запятой – `double`. Любое выражение тоже имеет тип – это тип результата, получающегося после вычисления этого выражения.

Что же произойдет, когда мы значение переменной одного типа присвоим переменной другого типа? Или если в одном выражении смешать переменные и константы разных типов? В языке Си, как и в других языках программирования, все двухместные операции умеют работать только в том случае, когда оба их операнда имеют одинаковый тип. Поэтому, отвечая на вопрос – произойдет то, что в языке Си называется *приведением типов данных* (или преобразованием типов).

Преобразования бывают *явными* и *неявными*. Неявные преобразования производит сам компилятор без помощи программиста. Они происходят везде и всюду в наших программах и их можно разделить на два класса:

1. когда мы присваиваем результат вычисления какого-либо выражения переменной – этот результат преобразуется к типу переменной;
2. при вычислении выражения операнды каждой операции преобразуются к какому-то одному типу (если они различаются), при этом результат операции будет того же типа.

Во втором случае основное правило неявного преобразования двух типов к какому-то одному: *компилятор всегда старается преобразовать «меньший» тип к «большему», чтобы минимизировать возможную потерю информации*. Выпишем все базовые типы в порядке возрастания их «размера»: `char` ≤ `short` ≤ `float` ≤ `int` ≤ `long` ≤ `double`. Если в результате операции возможна потеря точности, компилятор выдаст соответствующее предупреждение, однако останавливать процесс компиляции из-за этого не будет, так как, строго говоря, это не считается ошибкой.

Рассмотрим подробней два класса неявных преобразований. К первому классу, как уже было сказано, относятся преобразования, происходящие при присваивании. В этом случае правило преобразования «меньшего» типа к «большему» не действует. Результат вычисления выражения, стоящего справа, просто-напросто преобразуется к типу той переменной, которая стоит слева. Например:

```
int a = 2.3 + 4.5;
```

Очевидно, что результат вычисления выражения будет дробным числом, но вот переменная, в которой результат сохраняется – целочисленная. Поэтому происходит преобразование: дробный результат `6.8` преобразуется к типу `int`, причем преобразование будет заключаться в том, что дробная часть попросту отбросится. Таким образом, в предыдущем примере в переменной `a` сохранится значение `6`. Еще раз обратим внимание, что при преобразовании типа `float` или `double` к любому целочисленному типу *никакого округления не происходит* – дробная часть просто отбрасывается, что и называется компилятором *потерей данных,* о которой он сообщит вам в соответствующих предупреждениях. Во время преобразования целочисленного типа к дробному тоже может произойти потеря данных. Здесь играет роль количество значащих цифр дробного типа, которое обсуждалось выше. Например, тип `double` сохранит без потерь любое число типа `int`, а вот тип `float` – только часть диапазона.

Второй класс преобразований – преобразования, происходящие внутри выражения в момент его вычисления. Рассмотрим пример:

```
float a = 5 / 6;
```

Это выражение, состоящее из одной операции деления. Операция видит, что оба его операнда целочисленные, поэтому ничего преобразовывать не нужно и, как следствие, *результат тоже будет целочисленным*. В этом примере результатом целочисленного деления `5` на `6` будет ноль. Далее этот целочисленный ноль преобразуется к типу с плавающей запятой и записывается в переменную, но, увы, уже поздно – информация о дробной части потеряна. Для того чтобы исправить эту ситуацию, достаточно, например, изменить тип хотя бы одного из операндов на дробный путем добавления десятичной точки:

```
float a = 5.0 / 6;
```

В этом случае, как уже отмечалось ранее, должно будет выполниться преобразование типов, чтобы получились однотипные операнды. Тип константы `5.0` – `double`, тип константы `6` – `int`, и компилятор выбирает самый благоприятный для сохранения информации вариант – преобразует константу `6` к типу `double`. Соответственно, и результат вычисления выражения будет тоже дробным.

Схема, описанная выше, действует в любом выражении. Например:

```
int a, b;
double f1, f2;
f1 = a / (f2 + b) - b;
```

Выражение вычисляется согласно приоритетам операций шаг за шагом: от самой глубокой (приоритетной) операции к самой внешней, при этом операции одного приоритета выполняются слева направо. В примере выше сначала выполнится сложение, затем деление, и только потом – вычитание. Каждая из этих операций будет приводить оба своих аргумента к одинаковому типу в соответствии с описанными выше принципами. В примере выше:

1. сложение преобразует тип значения переменной `b` к `double` и выдаст результат типа `double`,
2. деление преобразует значение `a` тоже к типу `double`, так как результат только что вычисленного второго операнда будет типа `double`,
3. наконец, вычитание сделает ровно то же самое, так как первый операнд будет иметь тип `double`.

В результате значение всего выражения также будет иметь тип `double`.

Однако, здесь таится одна опасность. Например, чему будет равно значение выражения `2 / 4 * 3.14`? Казалось бы, `1.57`. Но нет! Так как деление и умножение – это операции одного приоритета, то выполняются они слева направо. Поэтому сначала выполняется деление, а потом уже умножение. Но оба операнда деления – целочисленные! Значит и деление будет тоже целочисленным с результатом `0`. Оператор же умножения преобразует свой первый операнд к типу `double`, но будет уже поздно. В результате вычисления всего выражения мы получим ноль. Решить эту проблему можно, например, записав вместо целочисленной `2` дробную `2.0`.

На все эти вопросы стоит обращать особое внимание, так как по невнимательности в результате неявных преобразований можно случайно потерять значимую информацию, что иногда приводит к трудноуловимым ошибкам.

Но что же нам делать, если подобная проблема возникает в том случае, когда в выражении содержатся переменные вместо констант? Например:

```
int a = 1, b = 2;
double result = a / b;
```

Написать `a.0`, как это мы делали в случае с константами, мы не можем! Здесь нам поможет *явное преобразование типов*, с помощью которого мы явно скажем компилятору, что мы хотим преобразовать значение одной из этих переменных к типу `double`:

```
int a = 1, b = 2;
double result = (double)a / b;
```

Значение переменной, идущей после скобок, будет преобразовано к типу, указанному в скобках. Явно можно преобразовывать тип не только значений переменных, но и целых выражений:

```
float a = 4.0, b = 2.0;
int result;
result = (int)((a * a) / (b * 6) + 4);
```

4.4 Адреса переменных и указатели

Теперь рассмотрим второй способ работы с памятью – *косвенный*, когда мы обращаемся к ячейке памяти не по ее имени, а по ее *адресу*. Необходимо сразу заметить, что это всего лишь *альтернативный способ обращения* к памяти. Чтобы с ней работать, нам по-прежнему необходимо ее сначала выделить (зарезервировать). Иными словами, мы сначала резервируем область памяти, а дальше у нас уже появляется выбор – мы можем обращаться к ней либо по имени, либо по адресу.

Для того чтобы обращаться к переменным по адресам, необходимы три вещи:

1. возможность находить адрес, по которому в памяти располагается та или иная переменная;
2. место, где этот адрес можно было бы сохранить для дальнейшей работы;
3. возможность обращаться к ячейке памяти по ее адресу.

Для нахождения адреса переменной используется *операция взятия адреса* – `&` (амперсанд). Это унарная операция, которая пишется непосредственно перед именем переменной.

Хранятся *адреса* переменных тоже в переменных. Но только для таких переменных существует специальный тип, который называется *указательным*, а сами переменные такого типа называются *указателями* (так эти переменные назвали из-за того, что они хранят в себе адреса других переменных, тем самым «указывая» на них). При этом существует столько указательных типов, сколько существует самих типов данных, потому что, когда мы создаем переменную-указатель, мы должны сразу решить, адрес переменной какого типа он будет хранить. Объявляется переменная-указатель (в примере ниже – `pInt`) следующим образом:

```
int *pInt, i;
```

Перед именем указателя ставится знак *. Теперь можно в переменной-указателе `pInt` сохранять значения адресов целочисленных переменных. Заметим, что в примере создаются две переменные: один указатель на целый тип `pInt` и одна обычная целочисленная переменная `i`. Соответственно, звездочку надо ставить не один раз

после типа, а *перед каждым именем переменной, которую мы хотим объявить, как указатель (см. Рис. 4.1)*:

```
int *a, b, *c;
```

Еще раз повторим, в чем разница между `pInt` и `i`. Разница в том, что `i` предназначена для хранения какого-либо *целого числа*, а `pInt` предназначена для хранения *адреса* какой-либо *целочисленной переменной* (например, той же переменной `i`).

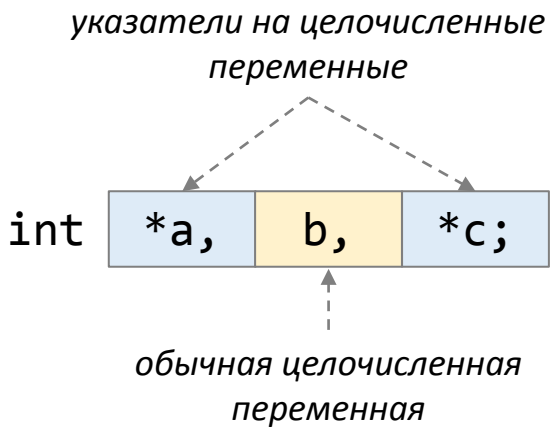


Рис. 4.1: Объявление обычных переменных и указателей

Продолжим. Пусть у нас есть переменная-указатель на тип `int` (всюду далее будем называть ее просто указателем), в которую мы записали адрес какой-то переменной типа `int`:

```
int *pInt, i;
pInt = &i;
```

Что дальше? Как, используя адрес, записанный в `pInt`, обратиться к ячейке памяти с этим адресом? Для этого используется еще один оператор языка Си – *оператор разыменования* *, который выглядит так же, как и звездочка из оператора объявления переменных. Однако этот символ ведет себя по-разному в зависимости от того, где он написан: **в** операторе объявления переменных или **вне** его. В первом случае звездочка является частью типа и говорит, что мы хотим создать переменную-указатель. Во втором случае звездочка является символом операции, при помощи которой осуществляется доступ к памяти по адресу, хранящемуся в указателе. Например:

```
1. char i = 0, *p;
2. p = &i;
```

```
3. *p = 2;
4. i = i + 2;
5. printf("%d %d", *p, i);
```

Что будет выведено на экран? Два одинаковых числа – 4. Значения соответствующих переменных изображены на Рис. 4.2.

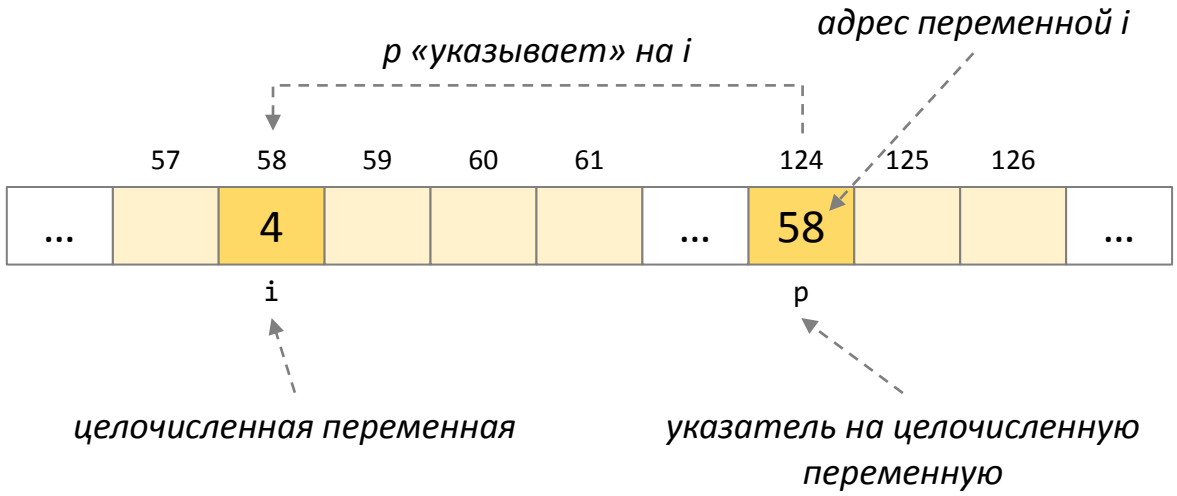


Рис. 4.2: Значения переменных после выполнения фрагмента кода

Переменная `i` – обычная целочисленная переменная, а `p` – указатель на тип `char`. В этот указатель во второй строке мы записываем *адрес* переменной `i`. Затем в третьей строке осуществляется *косвенный доступ* – с помощью оператора * мы обращаемся к той ячейке памяти, адрес которой находится в `p`, и записываем в нее число 2. Затем в строке 4 к той же самой ячейке осуществляется *прямой доступ* – по имени переменной `i`. В строке пять оба типа адресации используются для обращения к одной и той же области памяти, значение которой и выводится оператором `printf`.

Пусть теперь строка 4 вышеприведенной программы выглядит следующим образом:

```
p = p + 2;
```

Содержимое переменных в этом случае изображено на Рис. 4.3. Однако если мы попытаемся скомпилировать и выполнить эту программу, то получим сообщение об ошибке, потому что в пятой строке кода второй аргумент функции – `*p` – пытается получить значение переменной по адресу `60`, однако, в нашем случае этот адрес ни за какой нашей переменной не числится!

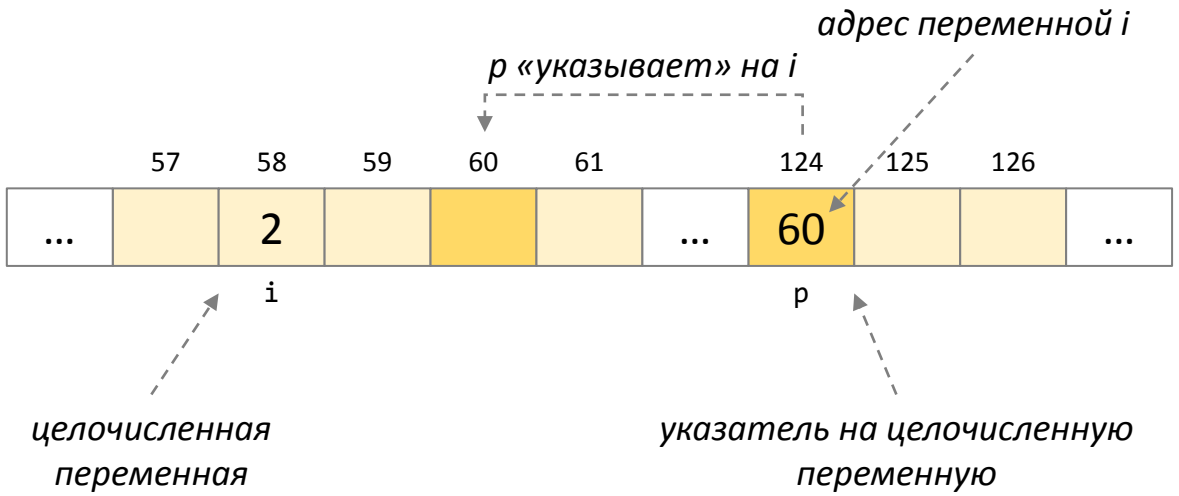


Рис. 4.3: Значения переменных после выполнения фрагмента кода

Разберемся теперь с типами указателей. Для того чтобы хранить адреса целочисленных переменных, мы должны использовать указатели на тип `int`:

```
int *p1;
```

Если же мы хотим работать с адресами переменных типа `double`, то нам потребуются указатели на тип `double`:

```
double *p2;
```

При этом хранить адрес целочисленной переменной в указателе на тип `double` нельзя, так же, как и хранить адрес переменной типа `double` в указателе на тип `int`. Почему это так? Зачем указателям нужно знать еще и типы тех переменных, на которые они указывают? Ответ на данный вопрос заключается в следующем. Дело в том, что указатели хранят в себе адреса памяти. При этом для переменных, занимающих несколько байт памяти, адресом считается адрес первого байта. Получается, что адрес однобайтовой переменной типа `char`, равно как и адрес четырехбайтной переменной типа `int` и восьмибайтной переменной типа `double` представляется лишь адресом **одного** первого байта. Теперь рассмотрим следующий оператор вывода на печать:

```
printf("%d", *p);
```

Что делает оператор разыменования в этом примере? Пусть в `p` содержится число `58`. Оператор разыменования интерпретирует это число как адрес и идет в пятьдесят восьмую ячейку памяти. Далее он

должен вывести значение, хранящееся по этому адресу. Но какой размер имеет это значение? Это однобайтный `char` и `58` – это и есть номер этого одного байта? Или же это четырехбайтный `int` и `58` – это адрес первого из четырех байт? Сколько байт надо взять оператору *, начиная с пятьдесят восьмого, чтобы вывести целочисленное значение? Именно для этого указателям и нужно знать тип той переменной, на которую они указывают. Зная тип, они знают размер и формат соответствующей переменной.

Зачем нужны указатели? Упомянем один из самых распространенных способов их применения, которым мы пользовались, начиная с самого первого упражнения. Вспомним, как мы передаем переменные в функцию `scanf_s`:

```
1. int a;
2. scanf_s("%d", &a);
3. printf("%d", a);
```

Тот амперсанд, который мы всегда писали перед именами переменных в функции `scanf_s`, есть не что иное, как оператор взятия адреса. Использование его в функции `scanf_s` в строке `2` означает, что мы в эту функцию передаем *адрес* переменной `a`, в то время как использование просто имени переменной в функции `printf` на строке `3` означает, что в нее мы передаем *значение* этой переменной. Это и логично, потому что функцию `scanf_s` интересует не значение переменной `a`, а ее адрес в памяти, чтобы записать туда вводимое с клавиатуры число. А функцию `printf`, наоборот, интересует именно значение переменной `a`, чтобы вывести его на экран.

Упражнения

Попробуйте решить самостоятельно следующие упражнения.

4.1 Для числа, введенного пользователем, напечатать в порядке возрастания все его делители.

4.2 Для двух положительных чисел, введенных пользователем, необходимо напечатать в порядке возрастания все их общие делители.

4.3 Составьте программу вывода на экран всех простых чисел, не превосходящих заданного `N`.

4.4 Алгоритм Евклида для нахождения наибольшего общего делителя двух чисел выглядит следующим образом. Даны два числа: `a` и `b`. Пока они не равны, вычитать из большего числа меньшее. Оставшееся значение `a` (или `b`) и будет наибольшим общим делителем двух чисел. Написать программу, вычисляющую наибольший общий делитель для двух введенных с клавиатуры чисел по алгоритму Евклида.

Решения упражнений

4.1
```
int N, i;
printf("Введите число: ");
scanf_s("%d", &N);
printf("Делители %d: ", N);
for (i = 1; i <= N; i++) {
     if (N % i == 0)
          printf("%d ", i);
}
printf("\n");
```

4.2
```
int N, M, i = 1;
printf("Введите два числа: ");
scanf_s("%d %d", &N, &M);
printf("Общие делители %d и %d: ", N, M);
while (i <= N && i <= M) {
     if ((N % i == 0) && (M % i == 0))
          printf("%d ", i);
     i++;
}
printf("\n");
```

4.3
```
int N, i, j, isPrime;
printf("Введите число: ");
scanf_s("%d", &N);
printf("Простые числа от 1 до %d: ", N);
for (i = 1; i <= N; i++) {
    isPrime = 1;
    for (j = 2; j < i; j++) {
        if (i % j == 0) isPrime = 0;
    }
    if (isPrime == 1) printf("%d ", i);
}
printf("\n");
```

4.4
```
int a, b;
printf("Введите два числа: ");
scanf_s("%d %d", &a, &b);
printf("NOD(%d, %d) = ", a, b);
while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
printf("%d\n", a);
```

Упражнения для самостоятельного решения

4.5 Напишите программу, которая по отдельности выводит один за другим следующие рисунки:

```
*          *****     *****           *
**         ****       ****          **
***        ***         ***         ***
****       **           **        ****
*****      *             *       *****
```

Для генерации рисунков используйте циклы `for`. Вы можете использовать операторы `printf`, которые выводят либо одну

звездочку, либо один пробел. Высота треугольника отдельно считывается с клавиатуры.

4.6 Напишите программу, которая выводит на экран фигуру в виде ромба. Вы можете использовать операторы `printf`, которые выводят либо одну звездочку, либо один пробел. Высота ромба вводится отдельно с клавиатуры. Например, для высоты `8` ромб будет выглядеть следующим образом:

```
   *
  ***
 *****
*******
*******
 *****
  ***
   *
```

4.7 Начальный вклад в банке равен `1000` руб. Через каждый месяц размер вклада увеличивается на `P` процентов от имеющейся суммы (`P` – вещественное число, `0 < P < 25`). По данному `P` определить, через сколько месяцев размер вклада превысит `1100` руб., и вывести найденное количество месяцев `K` (целое число) и итоговый размер вклада `S` (вещественное число).

4.8 Целое число называется совершенным, если сумма его делителей, включая `1` (но не само число), равна этому числу. Например, `6` является совершенным числом, поскольку `6=1+2+3`. Напишите программу, которая определяет, является ли введенное пользователем число совершенным.

4.9 Распечатать все совершенные числа в диапазоне от `1` до `1000`. Напечатайте все делители для каждого совершенного числа.

4.10 Найти наибольшее и наименьшее значения функции $y=3x^2+x-4$ на заданном интервале `[a,b]`. `X` изменяется с шагом `0.1`.

4.11 В 1202 году Итальянский математик Леонард Пизанский (Фибоначчи) предложил такую задачу: пара кроликов каждый месяц дает приплод – двух кроликов (самца и самку), от которых через два

месяца уже получается новый приплод. Согласно условию задачи, числа, соответствующие количеству пар кроликов, которые появляются через каждый месяц, составляют последовательность 1, 1, 2, 3, 5, 8, 13, 21, ..., в которой каждый элемент последовательности равен сумме двух предыдущих: $y_n = y_{n-1} + y_{n-2}$. Эту последовательность назвали числами Фибоначчи. Составьте программу, выводящую все числа Фибоначчи, меньшие заданного числа N.

4.12 Известно, что сумма N первых нечетных чисел равна квадрату числа N, например, 1 + 3 + 5 = 3^2, 1 + 3 + 5 + 7 = 4^2 и т.д. Ввести натуральное K и распечатать таблицу всех натуральных чисел от 1 до K и их квадратов, вычисленных с помощью указанного соотношения (использовать операцию умножения нельзя).

Тема 5

Функции

Общая структура программ на языке Си. Главная функция `main`. Использование функций: декомпозиция задач, библиотеки кода. Проектирование сверху-вниз и снизу-вверх. Объявление, определение и вызов функции. Аргументы и параметры функции. Возвращаемое значение, `void`-функции. Локальные переменные, область видимости. Оператор `return`. Передача параметров по указателю.

5.1 Кому, когда и зачем нужны функции

Функция – именованный обособленный фрагмент кода. В отличие от ряда других языков программирования любая программа на Си состоит исключительно из функций – в ней нет такого кода, кроме объявлений переменных, который бы содержался «во вне». Есть как минимум одна функция – `main`, которая так и называется – главная, а все остальные делятся на две категории: те, которые вы создаете сами, и библиотечные.

Основная идея использования функций, проиллюстрированная на Рис. 5.1, является очевидной и заключается в том, что если какой-

то фрагмент кода используется в программе несколько раз, то было бы удобно его оформить в виде подпрограммы, вызываемой всякий раз, когда это необходимо.

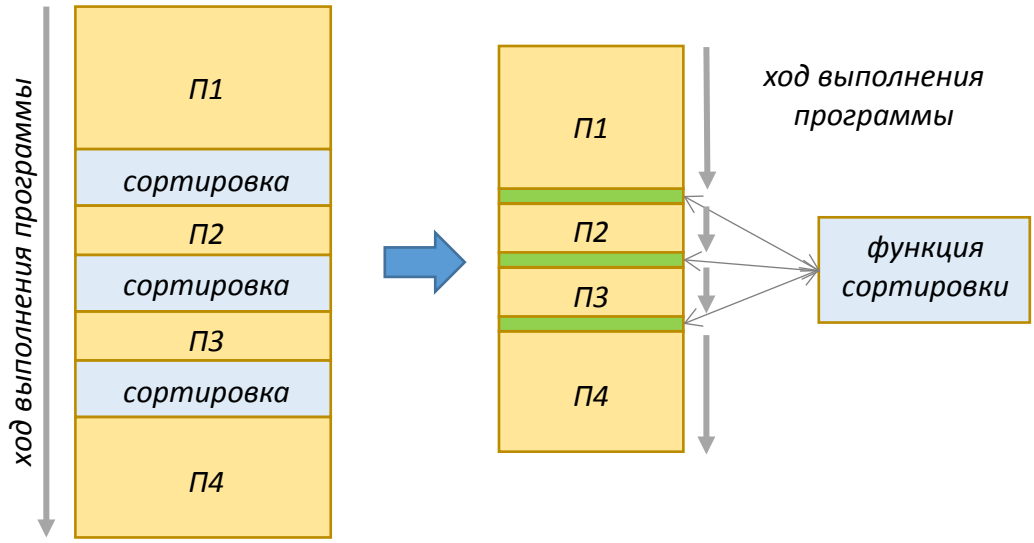


Рис. 5.1: Обособление повторяющегося фрагмента кода в виде функции

В момент вызова функции управление передается первому ее оператору, и она выполняется до тех пор, пока не кончатся операторы, или пока не будет выполнен оператор `return` (которых в коде функции может быть несколько). Преимущества использования подпрограмм заключаются в:

– объеме используемой памяти – во время выполнения программы код функции находится в памяти всего лишь в единственном экземпляре;
– поддержке кода – если мы нашли в нашей функции ошибку, то ее нужно исправить всего в одном месте;
– многократном использовании – если код не является сугубо специфическим для нашей программы и решает распространенную задачу, то мы можем его использовать и в других наших программах, или выложить в сеть, чтобы им могли воспользоваться другие программисты;
– возможности декомпозиции сложной задачи.

Из предпоследнего пункта следует еще одно очевидное и широко распространенное использование функций – создание библиотек кода, основывающихся на принципе – зачем постоянно заново

изобретать велосипед? Если мы один раз написали хороший (на наш взгляд) код, выполняющий что-то полезное и что может нам или кому-нибудь еще в дальнейшем пригодиться, то зачем его выбрасывать? Мы можем оформить его в виде функции, положить в библиотеку и нам больше не придется его переписывать заново.

В действительности, любой современный язык программирования состоит как минимум из двух частей – компилятора и библиотеки кода, где уже есть множество эффективных отлаженных функций, доступных нам для использования в наших программах. Например, вместе с языком Си идет *стандартная библиотека Си*, в которой можно найти уже готовые математические функции, функции по работе с датами, вводом/выводом, строками и т.д. По сути мы уже с самой первой программы вызывали библиотечные функции `printf` и `scanf_s` из стандартной библиотеки Си. Есть также огромное количество сторонних библиотек, которые мы можем использовать в нашей программе: для работы с сетью, графикой, базами данных, текстом, звуком и т.д.

Скажем еще отдельно несколько слово о последнем преимуществе из списка. Почти все практически полезные задачи решаются с помощью программ, размер которых существенно превосходит размер кода для упражнений, выполняемых в рамках данного или любого другого учебного пособия по программированию. Очевидно, что писать код компьютерной игры или финансовой программы в виде одной единственной функции `main`, которая делает все – теоретически возможно, но практически неосуществимо. Наше мышление устроено таким образом, что любую сложную задачу мы пытаемся разбить на более мелкие и, соответственно, простые подзадачи и уже решать их. При этом если сами подзадачи по-прежнему остаются сложными, то процесс повторяется – и так до тех пор, пока мы не дойдем до элементарных подзадач.

С процессом декомпозиции связаны два основных метода разработки программного обеспечения: *сверху-вниз* и *снизу-вверх* (см. Рис. 5.2). При проектировании сверху-вниз, мы сначала пишем главную функцию нашей программы, заменяя сложные фрагменты кода на вызовы пустых функций-заглушек. После этого мы расписываем каждую из этих функций, прибегая к тому же методу – если в коде функции встречается сравнительно сложный фрагмент логики, который легко обособляется в виде функции, мы вставляем вместо

него вызов еще одной «заглушки», определение которой оставляем на потом и т.д.

При написании кода снизу-вверх мы поступаем ровно наоборот: сначала пишем наиболее простые «примитивы», которые нам понадобятся в дальнейшем. Затем, на их основе собираем уже более сложные программные модули, на основе которых – еще более сложные и так далее, пока не получим такую ситуацию, что главная функция программы собирается из уже имеющихся модулей как конструктор.

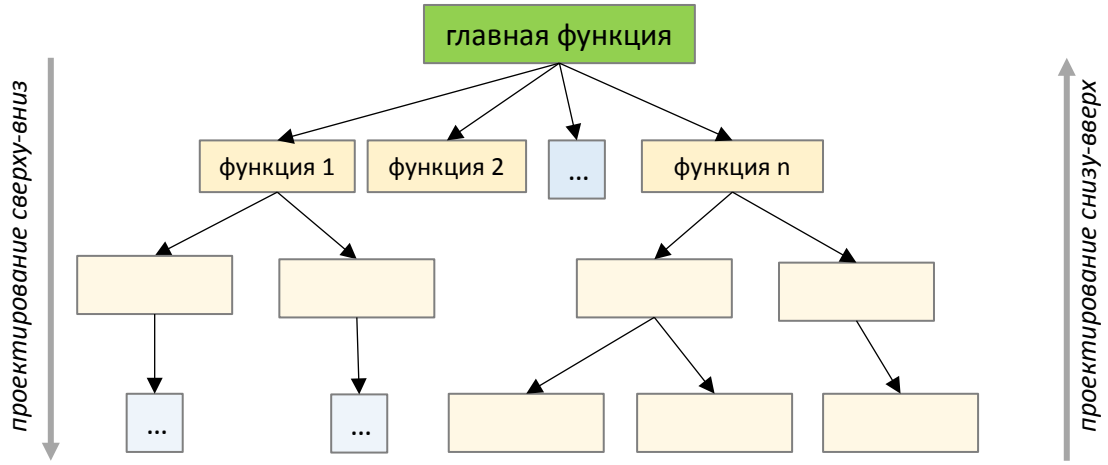


Рис. 5.2: Проектирование кода

С самых первых упражнений необходимо учиться *разбивать программу на компоненты*. Одна из распространенных ошибок начинающих программистов заключается в том, что они «лепят» весь код внутри `main`. Существует правило хорошего стиля: *хорошая функция должна полностью умещаться на экране.*

5.2 Объявление, определение и вызов функции

Разберемся теперь с тем, как все это работает и, самое главное, как нам с этим работать в языке Си.

Для *вызова функции* надо написать ее имя, затем круглые скобки, внутри которых должны быть перечислены через запятую передаваемые в нее аргументы. Например,

```
printf("Hello, world!\n");
```

есть вызов функции `printf` с одной строковой литеральной константой в качестве аргумента. *Аргументом функции* может быть любое выражение на языке Си. Сколько параметров надо передавать функции зависит от нее самой. Будучи выполненной, функция возвращает какое-то значение – единственное. А может и ничего не возвращать – такие подпрограммы иногда называют *процедурами*. Принципиальным моментом является то, что *функция, возвращающая значение, может быть частью любого выражения*, например:

```
c = (max(a, b) / min(a, b)) * 100;
```

Здесь в выражении наряду с константами и переменными также участвуют вызовы функций `max` и `min`. Вычисляется все это следующим образом: сначала поочередно вызываются все функции, затем возвращенные ими значения подставляются в выражение на место соответствующих вызовов и после этого все выражение вычисляется в соответствии с обычными правилами.

Для создания функции используют две синтаксические конструкции: *объявление* и *определение*. Объявление функции состоит из заголовка в следующем формате:

```
<тип_возврата> имя_функции(<тип1> <парам1>, …, <типN> <парамN>);
```

В конце объявления обязательно ставится точка с запятой. В определенном смысле этот оператор аналогичен оператору объявления переменных. Тип возврата – это тип значения, возвращаемого функцией. Если мы не хотим, чтобы она возвращала какое-либо значение, то надо вместо типа возврата написать ключевое слово `void`, а если функция не принимает никаких параметров, то вместо списка в круглых скобках тоже надо поставить `void`. Функции, которые ничего не возвращают, будем называть `void`-функциями. Правила для именования функций такие же, как и для переменных.

Определение функции выглядит следующим образом:

```
<тип_возврата> имя_функции(<тип1> <парам1>, …, <типN> <парамN>){
    // тело функции
}
```

и отличается от объявления только наличием *тела*, содержащего, собственно, сам код. Точка с запятой здесь уже не ставится.

Пример 5.1. Найти максимум из двух целых чисел.

Решение:

```
// объявление:
int max(int a, int b);

// определение:
int max(int a, int b){
     if (a > b) return a;
     else return b;
}
```

Определение является обязательным при создании функции, а вот объявление используется тогда, когда это необходимо. В частности, в примере 1 можно было обойтись и без отдельного объявления. Так зачем же нужно разбивать функцию на объявление и определение? Дело в том, что, как и с переменными, прежде чем использовать функцию, мы должны ее объявить, чтобы компилятор «знал», как ее вызывать, то есть сколько параметров и какого типа ей надо передать на вход и значение какого типа она возвращает. В отличие от ряда других языков, компилятор языка Си руководствуется принципом одного прохода: *все, что мы используем сейчас, должно было уже ранее быть объявлено*.

Ничего не мешает нам *определять* функции именно в том порядке, в котором они используются, однако иногда это невозможно. Первый пример – предположим, что функция `A` вызывает функцию `B`, а та, в свою очередь, вызывает функцию `A`:

```
void A(int x, int y){
     // ...
     B();
     // ...
}

void B(int x, int y){
     // ...
     A();
     // ...
}
```

Это называется *косвенной рекурсией*. Неважно, какую функцию мы определим первой – все равно оставшаяся из двух будет не определена в момент ее вызова. Для решения этой проблемы как раз и используется механизм объявлений:

```
void B(int x, int y);

void A(int x, int y){
    // ...
    B();
    // ...
}

void B(int x, int y){
    // ...
    A();
    // ...
}
```

Определение функции подразумевает ее объявление (собственно, объявление – это «усеченное» определение), поэтому функцию `A` объявлять нам не требуется.

Второй пример – раздельная компиляция. Предположим, что мы вынесли код функции `max` в отдельный файл и компилируем его независимо. Теперь, если мы захотим использовать `max` в нашей программе, то перед нами встанет проблема: не будем же мы для этого снова полностью переписывать ее код? Тогда полностью теряется смысл раздельной компиляции. В этом случае опять на помощь приходит механизм объявлений – мы попросту пишем в нашей программе объявление функции `max` и этого компилятору будет достаточно. Действительно, функция `max` уже скомпилирована и ее код будет связан с нашим кодом на следующем шаге сборки приложения – линковке, а сейчас компилятору важно лишь проверить соблюдение синтаксической корректности вызова функции.

Как мы помним, именно объявления (заголовки) библиотечных функций и содержатся в заголовочных файлах, которые мы подключаем с помощью директивы препроцессора `include`. Сам код

этих функций лежит на диске в скомпилированном виде и стыкуется с нашей программой во время линковки.

5.3 Аргументы, параметры и возврат значения

Функцию можно рассматривать как черный ящик с несколькими входами и одним выходом. Получив на вход какие-то данные, она выполняет свою работу и возвращает определенное значение. Входных параметров может быть сколько угодно (их может даже и не быть вовсе), а возвращаемое значение только одно (и его тоже может и не быть, хотя такие функции – редкость).

Поговорим о *передаче параметров* на примере функции `add`.

Пример 5.2. Написать функцию для сложения двух чисел.

Решение:

```
1.  int main(){
2.      int a = 1, b = 2;
3.      int c = add(a, 2 + b);
4.      return c;
5.  }
6.  int add(int a, int b){
7.      int result = a + b;
8.      return result;
9.  }
```

Назначение функции `add` – сложить два переданных ей числа и вернуть результат. Мы вызываем ее в строке 3 и передаем ей два *значения* – переменной `a` и выражения `2 + b`. В программировании для обозначения этих конкретных передаваемых значений принято использовать термин *аргументы*. Таким образом, в строке 3 мы в функцию `add` передаем два аргумента – значение переменной `a` в качестве первого аргумента и значение выражения `2 + b` в качестве второго. Как теперь получить к ним доступ *внутри* самой функции? Ровно так же, как и к любым другим переменным, используя те имена, которые мы дали им в заголовке функции. В нашем примере это два целочисленных параметра `a` и `b`. Таким образом, *параметр* – это *переменная*, в которую *копируется передаваемое значение*

аргумента и с которой функция дальше работает. В примере выше переменные `a` и `b` в строках 6 и 7 – это параметры функции `add`.

Еще раз заметим один принципиальный момент, а именно, что в строках 2, 3, 6 и 7 используются одинаковые имена переменных. Например, переменная с именем `a` объявляется в строке 2, используется в строке 3, затем объявляется (снова?!) в строке 6 и используется в строке 7. На самом деле, как уже было сказано выше, здесь присутствуют **две разные переменные с одинаковыми именами**. В строках 2 и 3 –переменные функции `main`, а в строках 6 и 7 – параметры функции `add`.

Параметры функции – это обычные переменные, которые видны только внутри этой функции, и этим переменным *присваиваются значения аргументов, переданных в функцию в момент вызова* (см. Рис. 5.3). В переменную `a` функции `add` копируется значение переменной `a` функции `main`, а в переменную `b` функции `add` копируется значение выражения `(2 + b)`.

Внутри любой функции мы точно так же, как и в `main`, можем объявлять любые переменные (в нашем примере – `result`), при этом важно помнить, что они будут доступны *только внутри* этой функции. Это так называемые *локальные переменные*.

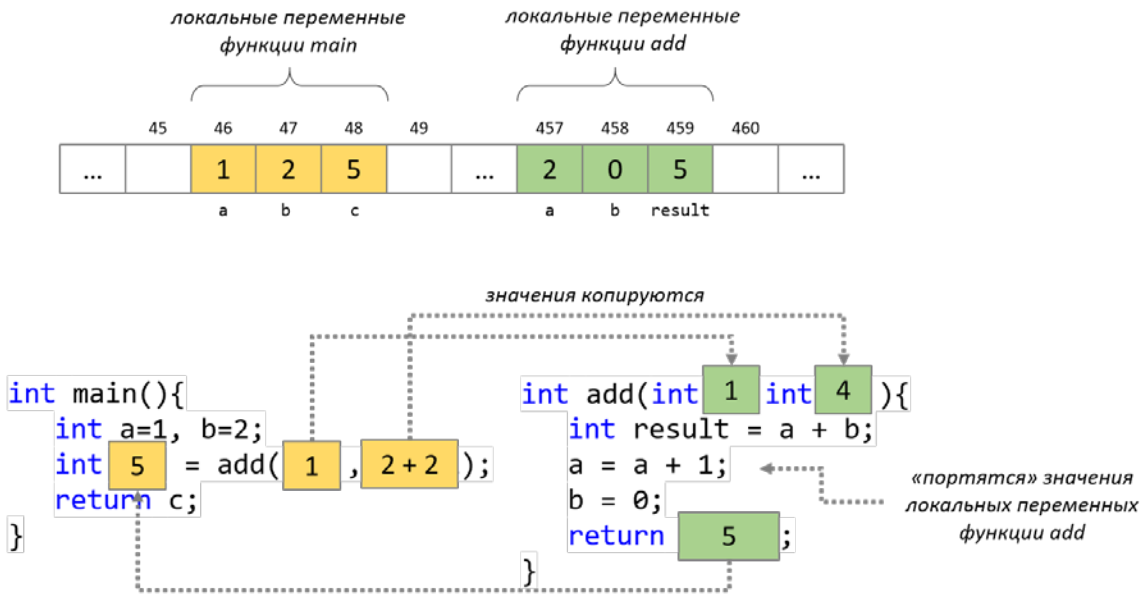


Рис. 5.3: Передача параметров в функцию

Если говорить более точно, то с переменными связано такое понятие, как *область видимости*. Каждая функция задает свою

область видимости и *все переменные, объявленные внутри некоторой функции «существуют» только внутри нее.*

Поговорим теперь о возврате результата из функции. Функция заканчивает свою работу в двух случаях:

– выполнив оператор `return`,
– если функция не возвращает никакого значения (то есть тип возврата – `void`), то она может завершить работу после выполнения своего самого последнего оператора.

Синтаксис оператора `return` следующий:

```
return выражение;
```

Выполнение данного оператора *завершает работу функции* и возвращает вызывавшему ее коду значение `выражения`. `void`-функцию можно завершить с помощью `return` без операнда:

```
return;
```

Оператор `return` можно образно сравнить с кнопкой катапультирования – нажав ее, мы тут же «вылетаем» из функции. Одна из распространенных ошибок у начинающих программистов связана с тем, что они не воспринимают этот оператор как оператор, *завершающий* работу функции. Например, следующий код хоть синтаксически и корректен:

```
1. scanf_s("%d", &a);
2. return a;
3. printf("Вы ввели %d\n", a);
```

но имеет бесполезную инструкцию на строке 3, которая *никогда* не будет выполнена. Данное свойство `return` можно использовать так же для того, чтобы вместо

```
if (a > b)
     return a;
else
     return b;
```

писать

```
if (a > b)
     return a;
return b;
```

Вызов любой не `void`-функции можно использовать внутри выражения и, например, передавать ее в качестве входного аргумента в другую функцию, например:

```
int main(){
     int a, b;
     scanf_s("%d %d", &a, &b);
     printf("%d + %d = %d\n", a, b, sum(a, b));
     return 0;
}
```

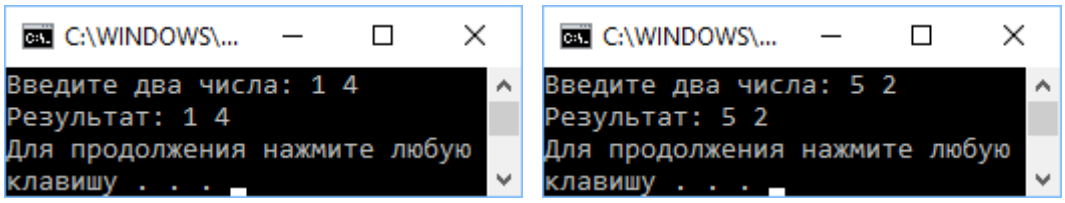
5.4 Передача параметров по указателю

Что делать, если требуется, чтобы функция *изменяла* значения переменных, находящихся в области видимости *другой* функции? Рассмотрим в качестве примера функцию `sortArgs`, которая принимает на вход два параметра и сортирует их, то есть после вызова `sortArgs(a,b)` *значения исходных переменных* `a` и `b` должны поменяться местами, если первое больше второго. Рассмотрим следующий код:

```
1. #include <stdio.h>
2. #include <locale.h>
3.
4. void sortArgs(int a, int b);
5. int main() {
6.    setlocale(LC_ALL, "Russian");
7.
8.    int a, b;
9.    printf("Введите два числа: ");
10.   scanf_s("%d %d", &a, &b);
11.
12.   sortArgs(a, b);
13.
14.   printf("Результат: %d %d\n", a, b);
15.   return 0;
16. }
```

```
17.
18. void sortArgs(int a, int b) {
19.   int tmp;
20.   if (a > b) {
21.       tmp = a; a = b; b = tmp;
22.   }
23.   return;
24. }
```

Попробуем ее запустить с разными входами:



Как видим, ничего не происходит. Значения переменных как были неотсортированными до вызова функции `sortArgs`, так и остаются таковыми. Прежде чем мы поймем, почему функция не делает то, чего от нее ожидают, и как это исправить, заметим несколько вещей относительно приведенного кода:

– в строке 4 записано объявление функции `sortArgs`, а в строках 18-24 – ее определение. Это обычная практика – самой первой определять функцию `main`, а уже после нее – все вспомогательные. При этом до `main` размещаются объявления всех вспомогательных функций;
– функция `sortArgs` является примером `void`-функции, которая ничего не возвращает. От нее требуется отсортировать значения переменных, объявленных в строке 8 (с чем она в данном примере, пока, увы, не справляется);
– в строке 23 мы видим пример пустого оператора `return`, который завершает работу функции без возврата какого-либо значения. Так как этот оператор последний, то в данном конкретном примере его можно было опустить;
– обратите внимание на то, как студия выделяет имена *параметров* в строках 18, 20 и 21 серым цветом. Это сделано для удобства (хотя параметры ничем не отличаются от обычных переменных);
– в строке 15 мы видим знакомый нам оператор `return 0`, который мы писали с самого первого упражнения. Мы всегда объявляли

главную функцию нашей программы `main` как возвращающую целочисленное значение и всегда писали `return 0` в качестве завершающей инструкции. Кому и зачем нужен этот ноль? Функцию `main` вызывает операционная система в момент запуска нашей программы и, соответственно, значение `0` получает она же. Значение, возвращаемое `main`, называется *кодом возврата* программы, по которому операционная система понимает, успешно ли она отработала. Принято, что если программа отработала корректно, то она должна вернуть нулевой код возврата, поэтому строку `return 0` в функции `main` можно считать своеобразным посланием операционной системе «Все прошло в штатном режиме».

Теперь разберемся, почему же `sortArgs` не делает того, что мы хотим. Ответ, думаю, уже очевиден – даже подсветка переменных нам об этом говорит – в строке 21 меняются значения параметров, а не исходных переменных из строки 8. Вспомним, что происходит в строке 12 – значения переменных `a` и `b`, объявленных в 8, копируются в переменные `a` и `b`, объявленные в 18 как параметры функции. В результате мы сортируем уже совсем другие переменные, которые не имеют ничего общего с теми, которые надо было отсортировать.

Как же исправить эту ситуацию? Ответ нам тоже должен быть уже очевиден – передать в функцию `sortArgs` не *значения* переменных `a` и `b`, а их *адреса*, и, используя косвенную адресацию, работать с исходными переменными `a` и `b`. Заметим, что в этом нет ничего нового, ведь именно так и работает, например, функция `scanf_s`. Обратите внимание, что мы передаем в нее в качестве аргументов адреса, взятые с помощью оператора `&`. Это и логично – ведь функции `scanf_s` не нужны значения переменных `a` и `b` – ей нужны адреса этих переменных, чтобы записать в них то, что пользователь введет на клавиатуре.

Корректный код будет выглядеть следующим образом.

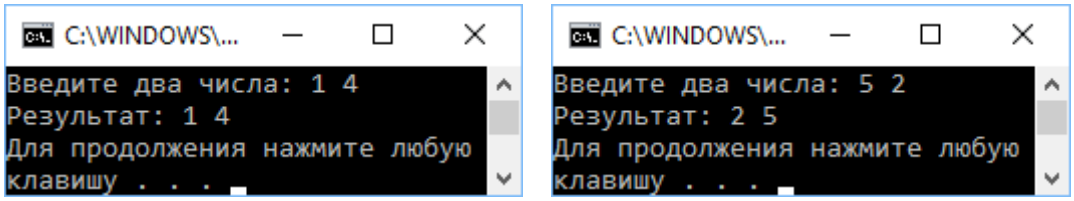
```
1. #include <stdio.h>
2. #include <locale.h>
3.
4. void sortArgs(int *a, int *b);

5. int main() {
```

```
6.    setlocale(LC_ALL, "Russian");
7.
8.    int a, b;
9.    printf("Введите два числа: ");
10.   scanf_s("%d %d", &a, &b);
11.
12.   sortArgs(&a, &b);
13.
14.   printf("Результат: %d %d\n", a, b);
15.   return 0;
16. }
17.
18. void sortArgs(int *a, int *b) {
19.   int tmp;
20.   if (*a > *b) {
21.        tmp = *a; *a = *b; *b = tmp;
22.   }
23.   return;
24. }
```

Мы объявили параметры функции `sortArgs` как указатели на целочисленные переменные (в строках 4 и 18!), а в строке 12 передаем в качестве аргументов адреса переменных `a` и `b`, объявленных в 8. Так же мы изменили код на строках 20-21, добавив оператор разыменования – обращения к ячейкам памяти по указателю.

Попробуем теперь вызвать эту программу:



Как видим, все прошло успешно, и наша функция `sortArgs`, выйдя за рамки своей области видимости, «залезла» в область видимости функции `main` и выполнила свою работу.

Упражнения

Попробуйте решить самостоятельно следующие упражнения.

5.1 Написать функцию `void swap(int *a, int *b)`, меняющую значения переменных `a` и `b`. Продемонстрировать ее работу на двух числах, введенных с клавиатуры.

5.2 Написать функцию `double circle_length(double radius)`. С помощью этой функции написать программу, запрашивающую у пользователя радиус и вычисляющую длину окружности.

5.3 Написать функцию `int factorial(int n)`, вычисляющую факториал переданного целого числа. Используя данную функцию вывести факториалы чисел от `1` до `10`.

5.4 Написать функцию `int is_prime(int n)`, проверяющую, что переданное ей целое число – простое. Вывести все простые числа от `1` до `100`.

Решения упражнений

5.1
```c
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int a, b;
    scanf_s("%d %d", &a, &b);
    printf("swap(%d,%d) = %d\n", a, b, swap(&a, &b));
    return 0;
}
```

5.2
```c
double circle_length(double radius) {
    return 2 * 3.14 * radius;
}
```

```
int main() {
    setlocale(LC_ALL, "Russian");
    double radius;
    scanf_s("%lg", &radius);
    printf("Длина окружности радиуса %lg = %lg\n",
                        radius, circle_length(radius));
    return 0;
}
```

5.3

```
int factorial(int N) {
    int result = 1;
    for (int i = 1; i <= N; i++) {
        result *= i;
    }
    return result;
}

int main() {
    for (int i = 1; i <= 10; i++) {
        printf("%d! = %d\n", i, factorial(i));
    }
    return 0;
}
```

5.4

```
int is_prime(int n) {
    for (int i = 2; i < n; i++) {
        if (n % i == 0)
        // Нашли делитель - число непростое,
        // можем выходить из функции и возвращать 0
            return 0;
    }
    // Просмотрели всех кандидатов,
    // делителей не нашли, возвращаем 1
    return 1;
}
```

```
int main() {
    int i;
    for (i = 1; i <= 100; i++) {
        if (is_prime(i) == 1)
            printf("%d ", i);
    }
    printf("\n");
}
```

Упражнения для самостоятельного решения

5.5 Написать функцию `double hypotenuse (double a, double b)`, которая вычисляет длину гипотенузы прямоугольного треугольника по двум другим сторонам. Используйте эту функцию в программе для определения длины гипотенузы треугольников, стороны которых пользователь вводит с клавиатуры.

5.6 Написать функцию `double power(double x, int n)`, возводящую число `x` в степень `n`. Использовать эту функцию в программе, возводящей в степень действительные числа. Число `x` и показатель степени `n` вводятся с клавиатуры.

5.7 Написать функцию `double minVal(double a, double b, double c)`, которая возвращает наименьшее из трех чисел с плавающей точкой.

5.8 Написать функцию `void mark(int val, int *markVal)`, которая принимает на вход балл в диапазоне от `0` до `100` и возвращает через параметр-указатель `markVal` оценку, соответствующую этому баллу согласно следующим правилам: `0-49` – `2`, `50-69` – `3`, `70-84` – `4`, `85-100` – `5`. Реализуйте программу, которая просит ввести балл с клавиатуры, а затем выводит соответствующую ему оценку.

5.9 Написать функцию `int fib(int n)`, вычисляющую `n`-е число Фибоначчи. С ее помощью вывести все числа Фибоначчи от `1` до `10`.

5.10 Написать функцию `int get_prime(int i)`, возвращающую `i`-е по счету простое число. Для определения, является ли число

простым, использовать функцию `is_prime()`. Написать программу, которая вводит с клавиатуры число `n` и печатает `n`-ое простое число.

5.11 Написать функцию `void prime_divisors(int n)`, печатающую все простые делители числа `n`. Для определения, является ли число простым, использовать `is_prime()`. Написать программу, которая вводит с клавиатуры число `n` и печатает все его простые делители.

5.12 Написать функцию `int factorial_2(int n)`, вычисляющую двойной факториал:

- `N!! = 1·3·5·…·N`, если `N` — нечетное;
- `N!! = 2·4·6·…·N`, если `N` — четное.

С помощью этой функции найти двойной факториал числа, вводимого пользователем с клавиатуры.

5.13 Написать функцию `int nod(int a, int b)`, вычисляющую наибольший общий делитель чисел с помощью алгоритма Евклида. Используя функцию `nod`, написать функцию `int nok(int a, int b)`, вычисляющую наименьшее общее кратное по формуле:

$$\text{НОК}(a, b) = \frac{a \cdot b}{\text{НОД}(a, b)}.$$

Наименьшим общим кратным (НОК) двух чисел `a` и `b` называется наименьшее число, которое делится и на `a`, и на `b` без остатка. Используя эти функции, написать программу, выводящую НОД и НОК введенных пользователем чисел.

5.14 Метод «переверни и сложи» заключается в следующем: берется целое положительное число `n`, все его цифры переворачиваются в обратном порядке (например, `123` становится `321`) и полученное число прибавляется к `n`. Если получился палиндром, то процесс останавливается, иначе проделывается то же самое с полученной суммой. Процесс повторяется до тех пор, пока не получится палиндром. Ваша задача написать программу, которая спрашивает у пользователя число `n` и вычисляет по данному числу палиндром, используя метод «переверни и сложи». Программа должна напечатать получившийся палиндром и количество шагов метода «переверни и сложи», за которое он был получен.

Тема 6

Ввод/вывод

Консоль. Буфер ввода. Форматированный ввод/вывод, форматная строка, спецификаторы преобразования, ширина и точность ввода/вывода. Файловый ввод/вывод. Открытие и закрытие файлов. Режимы работы с файлами: чтение, запись, дозапись. Тип `FILE` и макроконстанта `EOF`.

6.1 Буфер ввода

Операции ввода и вывода данных – неотъемлемая составляющая абсолютно любой программы. Любая программа либо получает какую-то информацию из вне, либо выдает какую-то информацию наружу, либо (что чаще всего и бывает) делает и то, и другое.

До сих пор мы использовали так называемые функции форматированного ввода/вывода: `printf`, печатающую на экран, и `scanf_s` – считывающую с клавиатуры. Экран и клавиатура вместе называются *консолью*, поэтому эти функции еще называют консольным вводом/выводом.

Рассмотрим принципы работы функции чтения – `scanf_s`. Данные, которые мы печатаем на клавиатуре, попадают сначала в *буфер*

ввода – специальную область системной памяти, за которой следит операционная система, и из которого они уже считываются функцией `scanf_s`. Данные всегда дописываются в конец буфера. Рассмотрим этот процесс на примере. Пусть мы имеем следующий код:

```
1. int main() {
2.   int a, b;
3.   scanf_s("%d", &a);
4.   scanf_s("%d", &b);
5.   printf("%d + %d = %d", a, b, a + b);
6.   return 0;
7. }
```

Состояния буфера ввода изображены на Рис. 6.1.

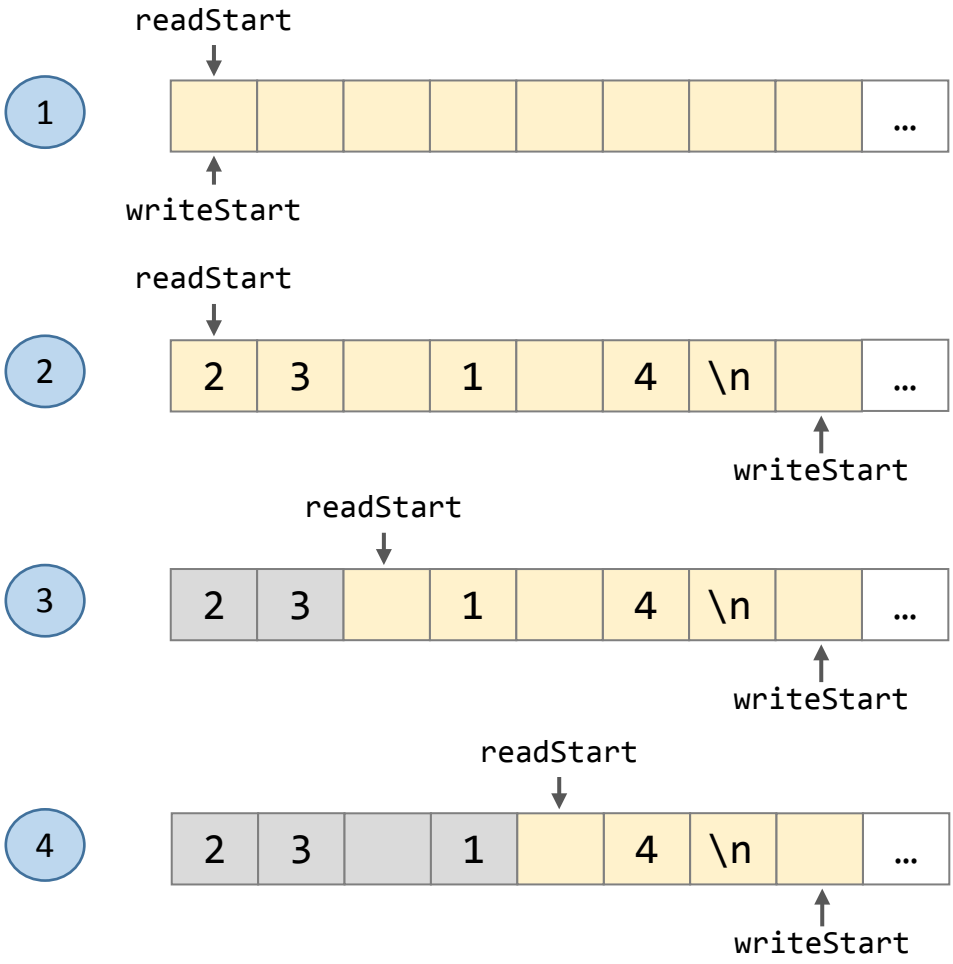


Рис. 6.1: Состояния буфера чтения

Здесь `readStart` – это стартовая позиция, с которой функция `scanf_s` начинает считывать из буфера данные, `writeStart` – позиция, начиная с которой в буфер записываются данные при вводе их с клавиатуры. Если оба указателя ссылаются на одну и ту же ячейку памяти, то это означает, что буфер пуст.

В момент выполнения 3-й строки программы вызывается функция `scanf_s`. Она анализирует спецификатор `%d`, понимает, что ее просят считать целое число, и обращается в системный буфер. Так как он пуст (состояние 1 на Рис. 6.1), то программа приостанавливает свою работу и запрашивает данные у пользователя – в консольном окне появляется мигающий курсор ввода, приглашающий нас ввести данные. Окончание ввода данных мы сигнализируем нажатием кнопки <Enter>.

Предположим, что мы ввели строку `23 1 4` – очевидно, никто не запрещает нам этого сделать, даже несмотря на то, что программе от нас нужно сейчас всего одно число. Состояние буфера в этот момент изображено под номером 2 на Рис. 6.1.

Так как данные появились, программа возобновляет свою работу, и функция `scanf_s` считывает одно целое число, останавливаясь на первом символе, который не может являться частью числа – пробеле. Буфер переходит в состояние №3, функция `scanf_s` преобразует строку `"23"` в число `23` (благодаря спецификатору `%d`) и записывает его в переменную `a`.

При выполнении четвертой строки кода снова вызывается функция `scanf_s`, однако теперь буфер не пуст, поэтому программа не останавливает свой работы, а считывает данные из буфера, пока это возможно в соответствии со спецификатором. При этом функция `scanf_s` пропускает все пробельные символы, к числу которых относятся пробел, табуляция и перевод строки. Мы могли бы разделить наши числа не одним, а двумя, тремя, каким угодно количеством пробелов – `scanf_s` их пропустит. Состояние буфера после выполнения четвертой строки кода изображено под номером 4. Обратим внимание, что хоть мы и считали два числа, которые нам были нужны, в буфере по-прежнему еще остаются данные. Что с ними произойдет? Ничего – они будут ждать своей очереди, когда в очередной раз вызванная `scanf_s` их считает.

А что произойдет, если в буфере ничего не будет? В этом случае `scanf_s` опять остановится и будет ждать, пока там что-нибудь

появится. Внешне в наших программах это выглядит как приостановка выполнения программы и появление мигающего курсора, предлагающего ввести какие-то данные. Текст, которые мы вводим в консольном окне, не сразу попадают в буфер по той причине, что мы еще можем их редактировать, воспользовавшись кнопкой <Backspace>. Данные попадают в буфер в тот момент, когда мы нажимаем <Enter>, завершая тем самым ввод одной строки. При этом важно помнить, что нажатие на <Enter> не только заносит набранные данные в буфер, но еще и добавляет символ переноса строки в этот буфер (символ `\n` на Рис. 6.1).

Осталось сделать только одно замечание в этой связи – у каждой выполняющейся программы есть свой системный буфер чтения, поэтому, когда наша программа завершит своей выполнение на строке 6, вместе с ней свое существование и прекратит буфер чтения.

Внимательный читатель может заметить еще одну проблему – многоточие в конце буфера, при этом указатели `writeStart` и `readStart` всегда продвигаются вперед. Буфер, как подсказывает нам интуиция, не может быть бесконечным. Означает ли это, что объем данных, которые может считать программа, ограничен размером буфера, ведь рано или поздно оба указателя «наткнутся» на его границы? Нет, не означает. В действительности работа с этим буфером устроена более хитрым образом – циклическим. Когда указатели доходят до границы, они «перепрыгивают» в самое начало. Да, сам размер буфера ограничен, но если наша программа будет вовремя считывать поступающие в него данные, то переполнения не произойдет. К тому же большинство программ считывают данные из файлов, где данная проблема не актуальна.

6.2 Форматированный ввод/вывод

Рассмотрим теперь подробнее синтаксис самих функций. `printf` и `scanf_s` называются функциями *форматированного вода/вывода,* этим объясняется буква ‘f’ (formatted) в их названии. Данные функции не просто позволяют вводить и выводить информацию, но и осуществлять при этом ее форматирование.

Рассмотрим синтаксис форматированного вывода:

```
int printf(const char* str, ...);
```

Мы об этом никогда не задумывались, но на самом деле функция `printf` возвращает целочисленное значение, а именно – количество напечатанных символов или отрицательное значение в случае ошибки. Ее первый параметр называется *форматной строкой*[10]. Она состоит из элементов двух типов: обычных символов, которые выводятся на печать, и *спецификаторов преобразования*, начинающихся на знак процента. Спецификатор – это своего рода «метка», на место которой будет подставлено значение соответствующего выражения, указываемого после форматной строки в вызове функции `printf`:

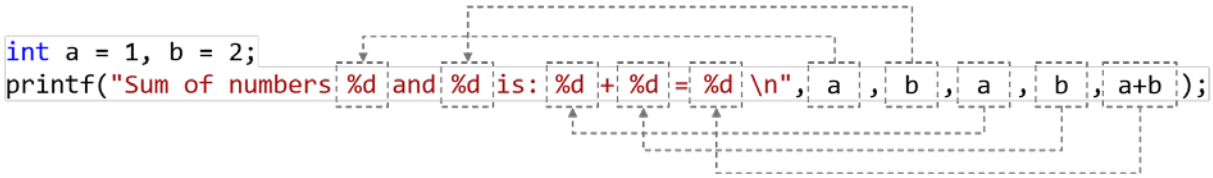


Рис. 6.2: Работа функции printf

Вывод на экран результатов работы кода, изображенного на Рис. 6.2, следующий:

```
Sum of numbers 1 and 2 is: 1 + 2 = 3
```

Многоточие в заголовке функции означает, что количество параметров этой функции неограниченно. Функция определит их количество по форматной строке – сколько спецификаторов в ней записано, столько и параметров она будет искать.

Помимо указания на место, куда надо вставить значение, спецификатор выполняет еще одну важную роль, а именно, говорит, в каком формате необходимо вывести (или считать) значение. До сих пор мы рассматривали лишь простую форму спецификатора: `%d` или `%lg` – то есть знак процента и тип. Полный формат спецификатора выглядит следующим образом:

```
%[<флаг>][<ширина>][<.точность>]<тип>
```

Квадратные скобки означают, что поле является опциональным. Предназначение полей следующее:

[10] Почему типом первого параметра является указатель на переменную типа `char`, мы поговорим в следующей главе, а сейчас достаточно считать, что это строка – например, литеральная строковая константа, заключенная в двойные кавычки.

– `<ширина>` – это целое положительное число, задающее минимальную ширину поля, в котором будет напечатано значение. Значение выравнивается по правому краю поля. Если надо выровнять его по левому краю, используется флаг `'-'`.
– `<точность>` – целое положительное число, задающее точность, с которой будет выведено дробное значение. При этом происходит округление по обычным правилам, а нули в конце числа не пишутся.
– `<флаг>` – это либо символ минус `'-'`, использующийся при заданной ширине поля и означающий, что значение будет выровнено по левому краю поля; либо символ `'+'`, говорящий, что будет выведен знак числа, даже если оно положительное; либо и то, и другое вместе (их действия комбинируются).

Пример 6.1.

```
1. printf("%5d%3d\n", 4, 3);
2. printf("%-5d%-3d\n", 4, 3);
3. printf("%.3lg\n", 1.0 / 3.0);
4. printf("%.2lg\n", 1.0 / 3.0);
5. printf("%.1lg\n", 1.0 / 3.0);
6. printf("%3.3lg\n", 1.0 / 3.0);
7. printf("%10.3lg\n", 1.0 / 3.0);
8. printf("%+d%-+5d\n", 4, -1);
```

Вывод на экран:

```
    4  3
4    3
0.333
0.33
0.3
0.333
     0.333
+4-1
```

1) Первый вызов `printf` печатает два числа – первое в поле шириной пять символов, второе – в поле шириной три символа, выравнивание осуществляется по правому краю.
2) Второй вызов отличается от первого наличием флага `'-'`, благодаря чему выравнивание осуществляется по левому краю.

3) Третий, четвертый и пятый вызовы выводят дробное число `1/3` с точностью до, соответственно, трех, двух и одного знаков.
4) Шестой и седьмой вызовы также выводят `1/3` с точностью до трех знаков после запятой в полях шириной, соответственно, `3` и `10` символов. Данные примеры показывают, что если размер поля недостаточен, то оно увеличивается.
5) Восьмой пример показывает использование флага `'+'` на первом аргументе и результат комбинирования флагов `'+'` и `'–'` на втором.

Разберемся теперь с функцией форматированного ввода. Ее синтаксис выглядит так же, как и для `printf`:

```
int scanf_s(const char* format, ...);
```

Первое поле – форматная строка, указывающая, что и в каком формате надо считать, а дальше идут адреса переменных, в которые нужно записать считанные значения. Возвращает функция количество успешно считанных и преобразованных полей – возвращаемое значение `0` указывает на отсутствие таковых. Как уже было упомянуто в одной из предыдущих глав, в эту функцию переменные-приемники необходимо передавать с амперсандом, так как функцию `scanf_s` интересуют их *адреса*, а не значения.

В форматной строке `scanf_s` следует использовать только тип и ширину поля. Ширина поля в `scanf_s` определяет максимальное количество цифр в считываемом числе[11]. Если мы ввели число с бóльшим количеством цифр, то оставшиеся знаки остаются лежать в буфере. Например:

```
int n;
scanf_s("%1d", &n);
```

Если мы введем на клавиатуре число `573`, то в `n` попадет только `5`, а `7` и `3` останутся в буфере. Благодаря данной возможности, решение задачи определения счастливого билета из предыдущей главы[12] становится исключительно простым:

```
int a, b, c, d, e, f;
scanf_s("%1d%1d%1d%1d%1d%1d", &a, &b, &c, &d, &e, &f);
```

[11] Или символов в считываемой строке – о строках речь пойдет в следующей главе.
[12] Напомним: шестизначный номер билета называется счастливым, если сумма первых трех цифр равна сумме четвертой, пятой и шестой.

```
if (a + b + c == d + e + f){
  // билет счастливый
}
```

Отсутствие пробелов в форматной строке совершенно не принципиально. Как мы помним, `scanf_s` их просто игнорирует, поэтому мы могли бы записать и так:

```
scanf_s("%1d %1d %1d %1d %1d %1d", &a, &b, &c, &d, &e, &f);
```

До сих пор мы в форматной строке `scanf_s` писали только спецификаторы и не писали никакого дополнительного текста (разве что кроме пробелов). Это в общем и целом правильно, так как `scanf_s` ожидает команды-спецификаторы, описывающие, что ей нужно считать. Но если мы твердо уверены в формате вводимой информации, то мы можем воспользоваться еще одной особенностью форматированного ввода – мы можем писать в форматной строке `scanf_s` произвольные символы. В этом случае, если вводимая строка выглядит соответствующим образом, `scanf_s` будет эти символы просто игнорировать во входной строке. Например, с помощью следующего кода мы можем ввести дату, разбив ее сразу же на три поля (год, месяц, день):

```
int d, m, y;
scanf_s("%d-%d-%d", &d, &m, &y);
```

Если мы теперь введем на клавиатуре строку `01-07-2009`, то `scanf_s` считает все три значения сразу. Если мы введем дату по-другому, то `scanf_s` считает то, что согласуется с форматной строкой, а остальное оставит в буфере. Как мы помним, `scanf_s` возвращает количество успешно считанных и сохраненных значений, поэтому понять, удалось ли функции распознать то, что ввел пользователь, можно следующим образом:

```
if (scanf_s("%d-%d-%d", &d, &m, &y) == 3) // все ok
```

6.3 Работа с файлами

Посмотрим теперь, как работать с файлами, а не консолью. К счастью, делается это очень просто. Сначала мы создаем указатель:

```
FILE *pfile;
```

`FILE` – это специальный тип данных, определенный в файле `stdio.h`, который можно использоваться точно также, как и, например, тип `int`, для создания переменных или указателей на переменные данного типа. Мы можем считать `pfile` указателем на файл. `FILE` обязательно пишется большими буквами.

Перед тем, как начать работу с файлом, его необходимо *открыть*, а в конце работы с ним – *закрыть*. Открывает файл вызов библиотечной функции `fopen_s`[13]:

```
fopen_s(&pfile, "file.txt", "r");
```

Первый параметр функции `fopen_s` – это указатель[14] на объявленную нами ранее переменную типа `FILE*`, второй параметр – имя открываемого файла, а третий параметр – так называемый *режим* работы:

- `"r"` (от слова `read`) – означает, что мы открываем файл на чтение. Если файла не существует, то `fopen_s` возвратит ненулевое значение[15].
- `"w"` (от слова write) – означает, что мы открываем файл на запись. Если файл с таким именем уже существует, его содержимое будет удалено.
- `"a"` (от слова `append`) – файл открывается на запись, но если он уже существует, то данные будут записываться в конец файла.

Как пользоваться открытым файлом? В языке Си это делается исключительно просто: точно так же, как осуществляется консольный ввод и вывод. Разница заключается лишь в том, что мы должны будем использовать функции `fscanf_s` и `fprintf`, которые отличаются от `scanf_s` и `printf` наличием еще одного параметра типа `FILE*`. В остальном работа происходит точно так же: функция

---

[13] В других средах вместо этого надо использовать следующий синтаксис:

```
pFile = fopen("file.txt", "r");
```

Студия на него будет выдавать большое количество предупреждение (`warning`), так как она считает функцию `fopen` небезопасной.

[14] Все верно – это *указатель на указатель*. Указатели ведь тоже являются переменными, хранящимися где-то в памяти, поэтому и к ним мы можем обратиться косвенно. Вы также можете создать указатель на указатель на указатель и т.д.

[15] `fopen_s`, как и большое количество других библиотечных функций, возвращает числовое значение, называемое *кодом ошибки*. Как правило, нулевой код означает, что ошибок не было и операция выполнена успешно.

`fscanf_s` вместо системного буфера чтения использует файл, а `fprintf` то, что простая функция `printf` напечатала бы на экран, печатает в файл. Например, если мы хотим считать целое число из файла `file.txt` в переменную `n`, то мы должны выполнить следующий код:

```
int n;
FILE *myFile;
fopen_s(&myFile, "file.txt", "r");
fscanf_s(myFile, "%d", &n);
```

Разница, как мы видим, только в имени функции и новом аргументе – указателе на открытый файл. Смысл и назначение форматной строки и оставшихся аргументов – тот же самый.

Осталось выяснить еще один вопрос. Если мы считываем данные с клавиатуры и буфер чтения пуст, то программа приостанавливает свою работу и ожидает данные от пользователя. А что будет, если попытаться считать данные из файла, который закончился? В этом случае выполнение программы останавливаться не будет, а функция `fscanf_s` вернет значение специальной макроконстанты `EOF`, определенной в заголовочном файле как

```
#define EOF (-1)
```

Подытожим смысл значений, возвращаемых функциями `scanf_s` и `fscanf_s`, в таблице 6.1.

Таблица 6.1: Смысл возвращаемых `scanf_s` и `fscanf_s` значений

Значение	Что обозначает
`EOF`	Файл закончился
`0`	Функции не удалось считать ничего, что соответствовало бы форматной строке
`n > 0`	Функция считала `n` значений, при этом `n` может быть и меньше количества спецификаторов в форматной строке – это означает, что что-то считать не удалось.

После того, как файл больше не нужен, его необходимо закрыть. Делается это с помощью следующего вызова: `fclose(pFile);`

Упражнения

Попробуйте решить самостоятельно следующие упражнения.

6.1 Спросить у пользователя число и записать его в файл `output.txt`. Найти этот файл на диске и проверить, что число записалось.

6.2 Там, где мы нашли выходной файл предыдущей задачи, создать в блокноте файл `input.txt` и записать в него целое число. Написать программу, которая считывает данное число `N` и печатает в выходном файле `output.txt` квадрат из звездочек, высотой `N`.

6.3 Во входном файле записано несколько чисел. Считать все числа и вывести в выходной файл и на консоль максимальное и минимальное из данных чисел (для определения, все ли числа считаны, использовать функцию `feof`[16] и понимание того, что `fscanf_s` возвращает значение `-1` при попытке чтения из закончившегося файла).

Решения упражнений

6.1
```
int main() {
    int n;
    FILE *pFile;
    scanf_s("%d", &n);
    fopen_s(&pFile, "output.txt", "w");
    fprintf(pFile, "%d", n);
    fclose(pFile);
    return 0;
}
```

6.2
```
int main() {
    FILE *inFile, *outFile;
    int size, i, j;
    fopen_s(&inFile, "input.txt", "r");
```

[16] Функция `feof(pFile)` принимает на вход указатель на файл и возвращает истину, если он закончился, и ложь в противном случае. Используется в условии цикла `while` при чтении всего содержимого файла.

```
        fopen_s(&outFile, "output.txt", "w");
        fscanf_s(inFile, "%d", &size);
        for (i = 0; i < size; i++) {
            for (j = 0; j < size; j++) {
                fprintf(outFile, "*");
            }
            fprintf(outFile, "\n");
        }
        fclose(inFile);
        fclose(outFile);
        return 0;
}
```

6.3

```
int main() {
    FILE *inFile, *outFile;
    int max, min, tmp;
    fopen_s(&inFile, "input.txt", "r");
    fopen_s(&outFile, "output.txt", "w");

    fscanf_s(inFile, "%d", &min);
    max = min;
    while (!feof(inFile)) {
        if (fscanf_s(inFile, "%d", &tmp) != -1) {
            if (max < tmp) max = tmp;
            if (min > tmp) min = tmp;
        }
    }

    printf("Min = %d, max = %d\n", min, max);
    fprintf(outFile, "Min = %d, max = %d\n",min,max);

    fclose(inFile);
    fclose(outFile);
    return 0;
}
```

Упражнения для самостоятельного решения

6.4 Во входном файле записаны два числа на одной строке, разделенные пробелом. В выходной файл вывести их сумму, произведение и разность – каждое значение на своей строке. Например, для `10` и `20`, вывести:

```
Sum = 30
Mul = 200
Sub = -10
```

6.5 Ввести с клавиатуры два числа, а затем вывести результат деления первого на второе с точностью до ближайшего целого, до десятых, сотых, тысячных и до десятитысячных. Для округления использовать соответствующее поле форматной строки.

6.6 Напишите программу, которая напечатает таблицу целых значений температуры в шкале Фаренгейта в диапазоне от `0` до `212` градусов, и соответствующие им значения в шкале Цельсия (с точностью до третьего знака после запятой). Для вычислений используйте формулу `celcius = 5.0 / 9.0 * (fahrenheit - 32)`. Результат должен быть напечатан в два столбца шириной `10` символов с выравниванием по правому краю. Перед значением температуры в шкале Цельсия должен выводиться знак как для отрицательных температур, так и для положительных.

6.7 В России дату и время принято представлять в виде: `04:12:57 14.03.2009` (14 мая 2009 года 4 часа 12 минут 57 секунд), в Америке та же дата и время представляются в виде: `04:12:57 AM 03/14/2009`. AM используется для указания времени до полудня, PM – после, таким образом, количество часов никогда не превосходит `12`. Во входном файле вам дана дата в российском формате, ваша задача записать в выходной файл ее американское представление.

6.8 Даны два файла с одинаковым количеством целых чисел. Написать программу, которая создает текстовый файл, содержащий эти числа, расположенные в два столбца шириной по `10` символов (в первом столбце содержатся числа из первого исходного файла, во втором – из второго файла). Столбцы разделить друг от друга при

помощи разделителя «|». Числа выравниваются по правому краю столбца и отступают от него на один пробел. Например:

```
|     12 |     42 |
|   -123 |      4 |
|      0 |  12345 |
```

6.9 Дан файл, в котором написано несколько целых положительных чисел, каждое на своей строке. Известно, что количество цифр в каждом числе нечетно. Написать программу, которая выведет данные числа в выходной файл выровненные по центру. Например:

```
  2
 123
  3
12345
 456
```

6.10 Задача LC-дисплей. Ваш друг купил новый компьютер. Наиболее мощный компьютер, который он использовал до данного момента, был карманный калькулятор. И сейчас, смотря на свой новый компьютер, он выглядит очень расстроенным, ведь ему так нравился дисплей на его калькуляторе. Вы решили помочь своему другу и написать для него программу, выводящую числа так же, как они выглядели на его калькуляторе. Во входном файле дано целое положительное число, ваша программа должна напечатать его в выходном файле (располагая цифры на одной «строке»), используя цифры в следующем формате:

```
   --   --        --   --   --   --   --   --
  |    |    | |  | |    |      | |  | |  | |  |
  |    |    | |  | |    |      | |  | |  | |  |
     --   --   --   --   --        --   --
  | |       |    |    | |  |    | |  |    | |  |
  | |       |    |    | |  |    | |  |    | |  |
     --   --        --   --        --   --   --
```

Тема 7

Массивы

Массив как агрегатный тип данных. Использование массивов. Объявление и инициализация массивов. Индексирование элементов. Размещение в памяти. Заполнение случайными данными. Передача массивов в функцию. Многомерные массивы: объявление и работа с ними. Размещение многомерных массивов в памяти, передача в функцию.

7.1 Массив как агрегатный тип данных

Сравнительно часто нам надо обрабатывать большие объемы данных. Например, программа должна считать из файла возраст учащихся в группе из 14 человек, отсортировать их в порядке возрастания, а затем вывести на экран. Для этого необходимо, чтобы все данные одновременно находились в памяти, иначе отсортировать их будет проблематично. Использовать для этих целей 14 различных целочисленных переменных крайне неудобно хотя бы по одной, вполне очевидной причине – а что если учащихся станет больше или меньше? Не будем же мы под каждый размер входных данных писать отдельную программу? Именно для таких целей в языке Си наряду

с базовыми существуют еще и так называемые *агрегатные* (сложные, составные) типы данных. Одним из таких типов данных является *массив*. Массивы хранят элементы одного и того же базового типа (например, *массив целых чисел*, или *массив действительных чисел*), доступ к которым осуществляется по их *порядковым номерам* (см. Рис. 7.1).

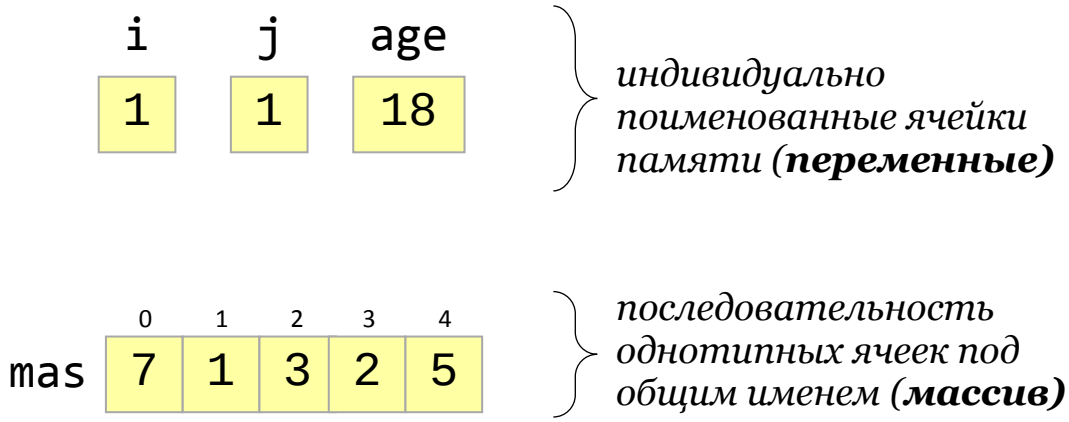


Рис. 7.1: Два способа работы с однотипными данными

Как работать с массивом? Так как это область памяти, пусть и сложного типа, то прежде всего ее надо зарезервировать (объявить). Делается это точно так же, как и для обычных переменных, с использованием следующего синтаксиса:

```
<тип> <имя>[<размер>];
```

В данном случае квадратные скобки являются частью синтаксиса. С этим оператором мы уже знакомы – это оператор объявления переменных. Разница в синтаксисе заключается в том, что мы после имени в квадратных скобках записываем размер создаваемого массива. Например:

```
int mas[14]; // массив из 14 целочисленных переменных
double degrees[100]; // массив из 100 переменных типа double
```

Мы также можем смешивать в одном операторе объявление обычных переменных и массивов:

```
int i, j = 1, buf[100], buf2[100], count = 0;
```

Чтобы получить доступ к элементу массива, необходимо сначала записать имя массива, а затем в квадратных скобках – порядковый номер того элемента, к которому мы хотим обратиться. Например:

```
mas[2] = 4;
degrees[28] = 4.5;
printf("%d %lg", mas[2], degrees[28]);
```

Записи вида `mas[2]` и `degrees[28]` можно использовать там же, где используются переменные по той простой причине, что это и есть обычные переменные, просто они не имеют своих собственных имен, а имеют лишь общее имя массива и свой порядковый номер.

Так же, как и в случае указателей, не стоит путать смысл значений, стоящих в квадратных скобках в операторе объявления массива и при обращении к элементу массива. В операторе объявления массива в квадратных скобках стоит *размер* создаваемого массива, а в операторе обращения к элементу – *порядковый номер* этого элемента.

В языке Си, в отличие от, например, языка Паскаль, *нумерация элементов в массиве начинается с нуля*, поэтому в примере выше первый элемент массива `mas` – `mas[0]`, последний – `mas[13]`, а `mas[2]` обращается к *третьему элементу по порядку*. Как правило, номера элементов массива называются *индексами*, а процесс доступа к ним – *индексированием*.

7.2 Размещение в памяти и инициализация

Массивы, с которыми мы сейчас работаем, называются *статическими*. Это означает, что их размер задается один раз во время объявления и дальше уже не изменяется. При указании размера создаваемого массива можно использовать только целочисленные константы, поэтому следующий код будет неверен:

```
int n = 10;
char mas[n]; // в квадратных скобках должна быть константа!
```

Зато данный код корректен:

```
const int size = 10;
char mas2[size];
char mas3[50];
```

В памяти компьютера массив представляет собой последовательность подряд идущих переменных заданного типа. Например, `char str[10]` – это 10 подряд идущих байт, а `int mas[3]`

– это три подряд идущие целочисленные переменные, каждая из которых занимает по 4 байта, то есть всего этот массив занимает 12 байт (Рис. 7.2).

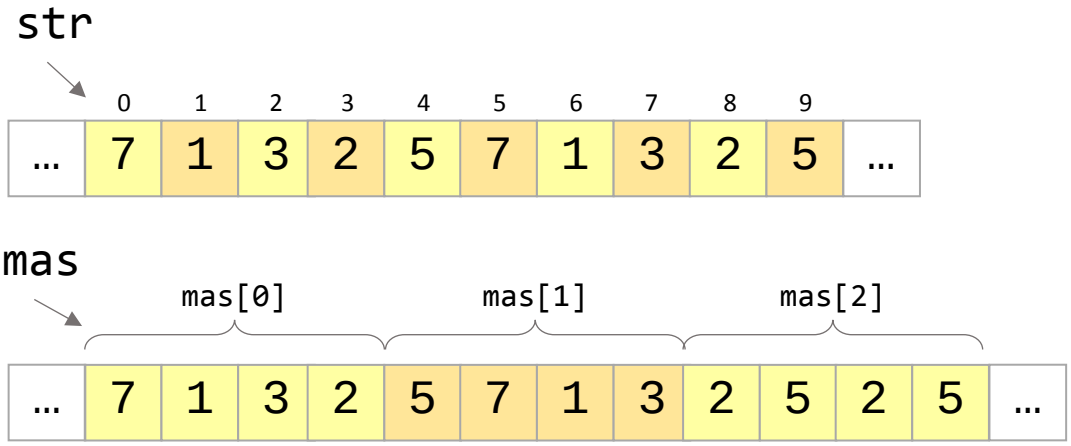


Рис. 7.2: Массивы `char str[10]` *и* `int mas[3]` *в памяти*

А что находится слева и справа от массива? Что такое `mas[-1]` или `str[11]`? Это тоже память, но только скорее всего уже нам не принадлежащая. Если мы попробуем к ней обратиться, то может случиться одно из двух: либо это вызовет сбой нашей программы, так как мы попытаемся влезть в чужую память, а операционная система не позволит нам этого сделать, либо ничего не произойдет, так как мы попадем в память, уже выделенную под одну из наших собственных переменных, что тоже нехорошо, так как вполне вероятно, что программа станет выдавать неверные результаты. Это одна из самых частых ошибок программ – когда, например, цикл, перебирающий элементы массива, промахивается и выходит за его границы.

В ячейках только что созданного массива хранится то же, что и в только что созданных переменных – мусор. Поэтому любой массив после его создания необходимо инициализировать начальным значением. Делать это можно также прямо в операторе объявления – для этого мы после имени и размера массива ставим знак равенства и в фигурных скобах через запятую перечисляем все элементы, которыми мы хотим инициализировать наш массив. Например:

```
int mas[5] = { 1, 2, 3, 4, 5 };
```

Мы просто перечислили значения всех пяти элементов. Что произойдет, если размер списка инициализации будет не совпадать с размером массива? Первый пример:

```
int mas[5] = { 1, 2, 3, 4, 5, 6, 7, 8 };
```

Эта запись вызовет ошибку компиляции, так как значений в списке инициализации больше, чем элементов массива. Второй пример:

```
int mas[5] = { 1, 2 };
```

Эта запись ошибки не вызовет. В первый элемент будет записана единица, во второй – двойка, а в три оставшихся – ноль. Отсюда следует простой способ обнуления массива любого размера:

```
int mas[256] = { 0 };
```

7.3 Заполнение массива случайными данными

Для многих упражнений нам понадобятся массивы, заполненные какими-либо произвольными случайными данными. Чтобы не забивать их вручную, удобно сгенерировать случайные значения программным способом. Сделать это позволяет библиотечная функция `rand()`, возвращающая случайно выбранное число на отрезке `[0…RAND_MAX]`, где `RAND_MAX`[17] – некоторая константа. В основе функции `rand()` лежит специальный алгоритм генерирования псевдослучайных чисел. Для того, чтобы он работал корректно, его необходимо инициализировать в самом начале работы программы с помощью функции `srand(число)`, причем `число` должно быть разным при каждом запуске программы. Чаще всего для этих целей используют текущее системное время, а именно – количество секунд, прошедших с 1-го января 1970 года[18], которое можно получить с помощью вызова еще одной библиотечной функции `time(NULL`[19]`)`. Это число от запуска к запуску программы будет разным. Для использования этих

[17] Определена в `stdlib.h` как `#define RAND_MAX 0x7fff`, где `0x7fff` – число `32767` в шестнадцатеричном формате.

[18] С этого времени начинается так называемая «эра юникса» (UNIX Epoch), где дата 01.01.1970 – время начала эпохи (Epoch Time). Это связано с представлением времени в операционных системах на заре развития вычислительной техники. В последствие эта дата стала стандартом для отсчета времени в приложениях, написанных на языке Си.

[19] NULL – еще одна макроконстанта, определенная как `#define NULL 0`

функций, необходимо подключить заголовочные файлы `stdlib.h` и `time.h`. Весь механизм заполнения массивов случайными числами иллюстрируют следующие два примера.

Пример 7.1. Заполнить массив из `10` элементов случайными числами на отрезке `[0…RAND_MAX]`.

Решение:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
     const int N = 10;
     int mas[N], i;
     srand(time(NULL));
     for (i = 0; i < N; i++)
          mas[i] = rand();
     for (i = 0; i < N; i++)
          printf("%d ", mas[i]);
     return 0;
}
```

Пример 7.2. Заполнить массив из `10` элементов случайными числами на отрезке `[1…100]`.

Решение:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
     const int N = 10;
     int mas[N], i;
     srand(time(NULL));
     for (i = 0; i < N; i++)
          mas[i] = 1 + rand() % 100;
     for (i = 0; i < N; i++)
          printf("%d ", mas[i]);
     return 0;
}
```

Обратите внимание на способ генерации случайных чисел из конкретного диапазона, в частности `[1,100]`. Сначала с помощью операции нахождения остатка от деления `rand() % 100` мы отображаем все получаемые случайные числа на отрезок `[0,99]`, а затем, прибавляя `1`, получаем то, что требовалось. В общем случае, если нам нужно получить числа из диапазона `[a,b]`, формула будет выглядеть следующим образом:

```
int x = a + (rand() % (b - a + 1));
```

А если, например, нам нужно получить случайное вещественное число на единичном отрезке `[0,1]`, то можно воспользоваться следующей формулой:

```
double x = (double)rand() / RAND_MAX;
```

7.4 Передача массивов в функцию

А как передавать массивы в функцию? Здесь нам на помощь опять приходят указатели. Массивы передаются в функцию *через указатель на первый элемент массива*. Более того, в языке Си *массив* как раз и *является указателем на его первый элемент*. Рассмотрим пример.

Пример 7.3. Написать функцию, которая принимает на вход массив и печатает его на экран. Сгенерировать массив из `10` случайных чисел и распечатать его с помощью данной функции.

Решение:

```
void printMas(int* mas, int num);
int main(){
     const int N = 10;
     int mas[N];
     srand(time(NULL));
     for (int i = 0; i < N; i++)
          mas[i] = rand();
     printMas(mas, N);
     return 0;
}
```

```
void printMas(int* mas, int num){
     for (int i = 0; i < num; i++)
          printf("%d ", mas[i]);
     printf("\n");
}
```

Обратим внимание на объявление функции `printMas`. Ее первый параметр имеет тип указателя на `int` – через него функция получит доступ к массиву. Так как никакой другой информации кроме адреса первого элемента указатель в себе не несет, то размер массива мы должны передать отдельно – через второй параметр. Сразу же напрашивается вопрос – а как функция различает, что ему передано на вход – массив или указатель на одну единственную переменную? Ответ, увы, никак. Об этом должен помнить программист, который пишет, а затем использует функцию.

В функции `main`, вызывая `printMas`, мы передаем в качестве первого аргумента сам массив – `mas` (без каких-либо квадратных скобок!).

7.5 Многомерные массивы

До сих пор мы изучали лишь одномерные массивы, то есть массивы, которые образно можно представить в виде линейной последовательности ячеек, пронумерованных от `0` до какого-то `N`. Предположим, нам необходимо в нашей программе создать таблицу умножения. Для этих целей мы, конечно, можем использовать и одномерный массив размером в `100` ячеек, при этом первые десять ячеек (с номерами от `0` до `9`) отвести под первую строку таблицы, вторые десять (с номерами от `10` до `19`) – под вторую и т.д. Но это неудобно, так как нам всякий раз придется вычислять номер нужной ячейки по двум индексам. В языке Си есть возможность явным образом создавать двумерные массивы (их еще называют *матрицами*). Делается это следующим образом:

```
int mas[10][10];
```

Как мы видим, добавляется вторая пара квадратных скобок для указания второй размерности массива.

Работа с двумерными массивами практически ничем не отличается от работы с одномерными массивами. Надо только вместо одной пары квадратных скобок писать две и указывать значения, соответственно, двух координат. Нумерация в обоих измерениях идет с нуля, таким образом, `mas[2][5]` – это значение, находящееся на пересечении третьей строки и шестого столбца (по порядку).

Точно также можно объявить и использовать трехмерный, четырехмерный и массив любой другой размерности:

```
int array3d[10][20][30];
```

Всего в массиве, объявленном выше – 6000 ячеек, а в памяти он занимает 24 000 байт. Возникает вопрос. Если память компьютера представляет собой линейную последовательность ячеек, то как в ней располагается нелинейная структура вроде матрицы (куба и любого другого n-мерного массива)? Тоже линейно! Матрица записывается в память строка за строкой: сначала первая строка, потом вторая, следом – третья и т.д. Например,

```
char M[3][4];
```

размещается в памяти следующим образом:

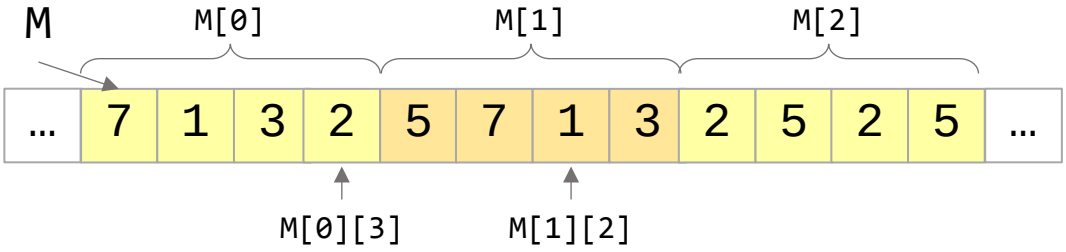


Рис. 7.3: Матрица `char M[3][4]` *в памяти*

а трехмерный массив

```
char M[3][3][2];
```

выглядит в памяти так:

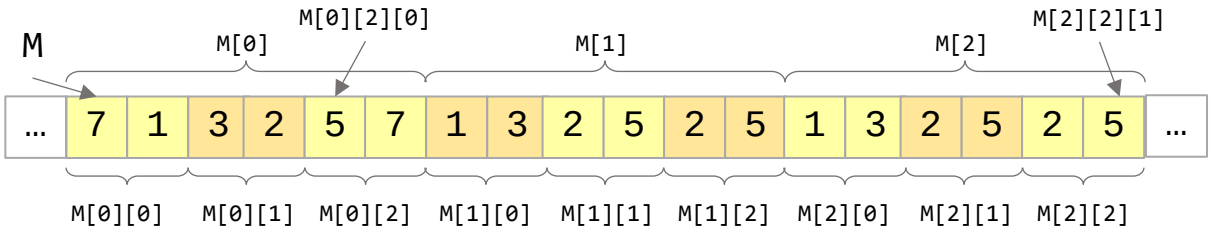


Рис. 7.4: Трехмерный массив `char M[3][3][2]` *в памяти*

Передача многомерных массивов в функцию происходит следующим образом. При объявлении функции мы должны указать тип массива так же, как и при объявлении самого массива:

```
int func(int mas[10][10]);
```

Так же можно передавать и одномерные массивы:

```
int func2(int mas[10]);
```

хотя мы всегда пользовались передачей одномерных массивов в функцию через указатель на первый элемент:

```
int func3(int *mas);
```

Почему же нельзя, например, и матрицы передавать через указатель на первый элемент? Потому что когда внутри функции мы напишем `mas[3][3]`, функция не сможет понять, к какой именно ячейке матрицы надо обратиться, так как это зависит от ее второй размерности. Если исходный массив был размером `10` на `10`, то `mas[3][3]` обращается к `34`-й, а если `10` на `5`, то к `19`-й ячейке.

Рассмотрим теперь пример с таблицей умножения.

Пример 7.4. Создать матрицу `10x10` и заполнить ее таблицей умножения, а затем вывести на экран.

Решение:

```
int main() {
     const int N = 11;
     int mas[N][N], i, j;
     for (i = 0; i < N; i++) {
          for (j = 0; j < N; j++)
               mas[i][j] = i * j;
     }
     for (i = 1; i < N; i++) {
          for (j = 1; j < N; j++)
               printf("4d", mas[i][j]);
          printf("\n");
     }
     return 0;
}
```

Упражнения

7.1 Сгенерировать случайным образом массив из `10` целых чисел в диапазоне от `10` до `30`. Затем вывести этот массив в прямом и обратном порядке.

7.2 Сгенерировать случайным образом массив из `30` целых чисел в диапазоне от `1` до `500`. Затем найти минимальный и максимальный элементы. Вывести весь массив, а также min и max.

7.3 Сгенерировать случайным образом массив из 1o целых чисел в диапазоне от `-500` до `500`. Вывести сначала исходный массив, а затем тот же самый массив, только выводить необходимо модули чисел (без знака).

7.4 Создать и заполнить случайным образом две матрицы `10x10`. После этого поэлементно сложить их и вывести на экран.

7.5 Создать и заполнить случайным образом матрицу `10x10`. После этого найти и вывести минимальные значения в каждой строке и минимальные значений в каждом столбце.

Решения упражнений

7.1
```c
int main() {
    const int N = 10;
    int mas[N], i;
    srand(time(0));
    for (i = 0; i < N; i++) {
        mas[i] = 10 + rand() % 21;
        printf("%d ", mas[i]);
    }
    printf("\n");
    for (i = N - 1; i >= 0; i--) {
        printf("%d ", mas[i]);
    }
    printf("\n");
    return 0;
}
```

7.2
```
int main() {
    const int N = 30;
    int mas[N], i, min, max;
    srand(time(0));
    for (i = 0; i < N; i++) {
        mas[i] = 1 + rand() % 500;
    }
    min = mas[0];
    max = mas[0];
    for (i = 0; i < N; i++) {
        if (min > mas[i]) { min = mas[i]; }
        if (max < mas[i]) { max = mas[i]; }
    }
    for (i = 0; i < N; i++)
        printf("%d ", mas[i]);
    printf("\nmax = %d, min = %d\n", max, min);
    return 0;
}
```

7.3
```
int main() {
    const int N = 10;
    int mas[N], i;
    srand(time(0));
    for (i = 0; i < N; i++)
        mas[i] = -500 + rand() % 1001;
    for (i = 0; i < N; i++)
        printf("%d ", mas[i]);
    printf("\n");
    for (i = 0; i < N; i++) {
        if (mas[i] >= 0)
            printf("%d ", mas[i]);
        else
            printf("%d ", -1 * mas[i]);
    }
    printf("\n");
    return 0;
}
```

Процедуры печати и заполнения случайными числами матрицы 10x10, используемые в упражнениях 7.4 и 7.5:

```
const int N = 10;
void printMas(int mas[N][N]) {
     for (int i = 0; i < N; i++) {
          for (int j = 0; j < N; j++)
               printf("%4d", mas[i][j]);
          printf("\n");
     }
     printf("\n");
}
void generateRandomMas(int mas[N][N]){
     for (int i = 0; i < N; i++)
          for (int j = 0; j < N; j++)
               mas[i][j] = rand() % 101;
}
```

7.4

```
int main() {
     const int N = 10;
     int mas1[N][N], mas2[N][N], result[N][N];
     srand(time(0));

     generateRandomMas(mas1);
     generateRandomMas(mas2);
     printMas(mas1);
     printMas(mas2);

     for (int i = 0; i < N; i++)
          for (int j = 0; j < N; j++)
               result[i][j] = mas1[i][j] + mas2[i][j];

     printMas(result);
     return 0;
}
```

7.5

```
int main() {
     setlocale(LC_ALL, "Russian");
```

```
    const int N = 10;
    int mas[N][N], i, j, min;
    srand(time(0));
    generateRandomMas(mas);
    printMas(mas);

    printf("Минимумы в строках:  ");
    for (i = 0; i < N; i++) {
        min = mas[i][0];
        for (j = 0; j < N; j++)
            if (min > mas[i][j])
                min = mas[i][j];
        printf("%3d", min);
    }
    printf("\n");

    printf("Минимумы в столбцах: ");
    for (i = 0; i < N; i++) {
        min = mas[0][i];
        for (j = 0; j < N; j++)
            if (min > mas[j][i])
                min = mas[j][i];
        printf("%3d", min);
    }
    printf("\n");

    return 0;
}
```

Упражнения для самостоятельного решения

7.6 Напишите программу, которая спрашивает у пользователя число `N`. Если `N < 1` или `N > 100`, то скажите пользователю, что он ввел неверное число, и спросите заново. Так делайте до тех пор, пока пользователь не введет число из заданного диапазона.

Затем ваша программа должна создать файл `input.txt`, в который на первую строку она запишет число `N`, а на вторую строку – `N` целых случайных чисел, разделенных пробелами. Этот файл можно будет использовать как входной файл для дальнейших упражнений.

7.7 Создайте массив заведомо большого размера (например, из `1000` элементов), затем считайте в него все числа из файла `input.txt`. После этого найдите количества чисел в этом массиве, которые меньше последнего элемента, бóльше последнего элемента и равны последнему элементу последовательности. Сохраните в выходной файл сам массив, а также найденные три числа. Информацию, выводимую в файл, продублируйте на экран.

7.8 Используя программу, написанную в 7.6, сгенерируйте два файла, содержащих по `N` случайных чисел. Считайте содержимое обоих файлов в два массива. Затем в третий массив того же размера, что и первые два, запишите суммы чисел, стоящих на соответствующих позициях в двух исходных массивах.

Т.е. в первой ячейке результирующего массива должна стоять сумма первых ячеек исходных массивов, и т.д. Выведите все три массива на экран и в файл.

7.9 Напишите программу, которая считывает все числа из файла `input.txt` в массив заведомо бóльшего размера, спрашивает у пользователя произвольное число и вставляет его в массив.

Для этого она спрашивает сначала число `a` – то, которое надо вставить, а затем число `i` – номер той ячейки, в которую его надо вставить. При этом программа должна спрашивать у пользователя число `i` до тех пор, пока оно не будет принадлежать отрезку `[0, N-1]`. Затем она должна вставить число `a` в ячейку `i`, *сдвинув* при этом подмассив (a_i, …, a_{N-1}) на одну ячейку вправо.

Например, из файла мы считали массив `(1, 4, 2, 10, 3)`. Затем пользователь ввел `a = 7`, `i = 3`. Мы должны получить массив `(1, 4, 2, 7, 10, 3)`. Выведите результат на экран и в файл.

7.10 Модифицируйте программу из 7.9 так, чтобы она удаляла элемент из массива. Программа должна спрашивать только число `i`,

следя, чтобы оно было из нужного диапазона. Затем она должна удалить `i`-тое число из массива, сдвинув подмассив (a_{i+1}, …, a_{N-1}) на одну ячейку влево. Например, из файла мы считали массив `(1, 4, 2, 10, 3)`. Затем пользователь ввел `i = 1`. Мы должны получить массив `(1, 2, 10, 3)`. Выведите результат на экран и в файл.

7.11 Модифицируйте программу из 7.10 так, чтобы она удаляла из массива не один элемент, а целый блок элементов. Программа должна спрашивать два числа i и j, следя при этом, чтобы `0 <= i <= j <= N - 1`. Затем она должна удалить из массива все числа, начиная с `i`-го и заканчивая `j`-тым, сдвинув подмассив (a_{j+1}, …, a_{N-1}) на нужное количество ячеек влево.

Например, из файла мы считали массив `(1, 4, 2, 10, 3)`. Затем пользователь ввел `i = 1`, `j = 3`. Мы должны получить массив `(1, 3)`. Выведите результат на экран и в файл.

7.12 Напишите программу, которая считает из файла в массив числа следующим образом: если очередное считываемое из файла число уже было считано до этого (т.е. уже есть в массиве), то оно игнорируется, в противном случае оно записывается в массив. Выведите исходный и получившийся массивы на экран.

7.13 Решето Эратосфена – простой алгоритм нахождения всех простых чисел до некоторого целого числа `n`, созданный древнегреческим математиком Эратосфеном.

Рассмотрим пример для `n = 20`. Запишем натуральные числа от `2` до `20`, в ряд:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Первое число в списке `2` – простое. Пройдем по ряду чисел, вычеркивая все числа, кратные `2` (кроме самой двойки):

```
2 3 5 7 9 11 13 15 17 19
```

Следующее невычеркнутое число `3` – простое. Пройдем по ряду чисел, вычеркивая все числа, кратные `3` (кроме самой тройки):

```
2 3 5 7 11 13 17 19
```

И так далее пока квадрат очередного невычеркнутого числа меньше n.

Напишите программу, которая спрашивает у пользователя число N, заполняет заранее созданный массив N натуральными числами, а затем использует вышеизложенный алгоритм для нахождения всех простых чисел на отрезке [2,N], выводя при этом все шаги работы данного алгоритма. Т.е. сначала надо вывести весь массив, затем вывести содержимое массива, когда из него удалены числа, кратные 2, и т.д. Операцию вычеркивания можно заменить обнулением соответствующих элементов массива.

7.12 Дана квадратная матрица A размерности MxM. Найти сумму элементов ее главной диагонали, то есть диагонали, содержащей следующие элементы: $A_{0,0}$, $A_{1,1}$, $A_{2,2}$, …, $A_{M-1,M-1}$.

7.13 В матрице A поменять местами наибольший и наименьший элементы (можно считать, что такие элементы в матрице уникальны). Полученную матрицу вывести на экран.

7.14 В каждой строке матрицы A, размером n x m программа находит наименьший элемент, затем среди этих чисел выбирает наибольшее. Нужно напечатать индексы элемента с найденным значением.

7.15 Дана квадратная матрица A порядка M. Начиная с элемента $A_{0,0}$, вывести ее элементы следующим образом («уголками»): все элементы первой строки; элементы последнего столбца, кроме первого (уже выведенного) элемента; оставшиеся элементы второй строки; оставшиеся элементы предпоследнего столбца и т.д. Последним выводится элемент $A_{M-1,0}$.

7.16 Дана квадратная матрица A порядка M (M – нечетное число). Начиная с элемента $A_{0,0}$ и перемещаясь по часовой стрелке, вывести все ее элементы по спирали: первая строка, последний столбец, последняя строка в обратном порядке, первый столбец в обратном порядке, оставшиеся элементы второй строки и т. д.; последним выводится центральный элемент матрицы.

7.17 Дана целочисленная матрица размера M × N. Найти количество ее строк, все элементы которых различны.

7.18 Дана матрица размера `M × N`. Элемент матрицы называется ее локальным минимумом, если он меньше всех окружающих его элементов. Заменить все локальные минимумы данной матрицы на нули.

Подсказка: для решения задачи лучше использовать вспомогательную матрицу.

Тема 8

Символы и строки

Символ, кодировка, ASCII. Управляющие символы. Работа с символами. Символ как числовая константа. Ввод и вывод символов. Escape-последовательности. Строки как массивы символов. Маркер конца строки. Массивы в качестве глобальных переменных. Объявление и инициализация строк, чтение с клавиатуры и из файла. Передача строк в функцию. Библиотечные функции для работы со строками.

8.1 Символы и кодировка ASCII

До сих пор мы работали преимущественно с числовыми данными. Рассмотрим теперь, как в языке Си можно работать с символами и строками.

Для начала разберемся с понятием *символа*. Символ – это просто печатный знак, который можно «нарисовать» или «напечатать». Буква, цифра, знак пунктуации или математической операции и т.п. – все это примеры символов, которые постоянно используются нами при работе за компьютером. Текст, который мы набираем в MS Word'е состоит из символов, текст программы, которую мы пишем – тоже состоит из символов. Мы можем эти тексты сохранять в файлах,

а затем снова загружать и работать с ними. Однако мы знаем, что память компьютера состоит из битов, каждый из которых хранит либо ноль, либо один. Каким же образом в ней умещаются еще и всевозможные символы из разных алфавитов? Как они хранятся в памяти? Все как всегда просто. Если мы выпишем подряд все символы, которые хотим использовать в нашей программе, а затем их пронумеруем, то тогда вместо символов в памяти компьютера можно хранить их номера в этом пронумерованном алфавите. Программа, которая должна что-то напечатать на экране, зная данную нумерацию, будет рисовать соответствующие символы.

Таблица, устанавливающая соответствие между символами и целыми числами, называется *кодировкой*. Самой распространенной и самой старой стандартной кодировкой является ASCII[20], с помощью которой кодируются 256 различных символов[21]. В эти символы входят символы латиницы, цифр, математические знаки, символы местного алфавита и некоторые другие. Для хранения одного символа в этой кодировке нужен всего один байт. Когда создавался язык Си, компьютеры не были такими «полиглотами» как сейчас, поэтому 256 символов хватало, чтобы работать с различными текстами в компьютере. Именно поэтому для такого важного объекта как символ был создан отдельный тип данных `char` (от английского – символ), который как раз и занимает в памяти ровно один байт.

Сегодня программы оперируют гораздо большим количеством символов: это и символы всевозможных национальных алфавитов, и иероглифы, и математически знаки, и символы мертвых языков, и множество других. Очевидно, что перенумеровать их все числами от 0 до 255 невозможно, поэтому сейчас применяют более объемные и хитрые кодировки, в которых под один символ требуется не один байт, а несколько. Однако во всех этих кодировках первые 256 номеров все равно отводятся под кодировку ASCII, а в языке Си тип

[20] От англ. American Standard Code for Information Interchange (американский стандартный код для обмена информацией).

[21] В действительности, стандарт ASCII описывает лишь первую половину таблицы, в которую входят 127 различных символов, потому что на заре развития вычислительной техники 7-ми бит вполне хватало для представления символов латиницы – о других языках тогда еще никто не думал. В последствие вторую половину таблицы стали использовать для национальных кодировок. Все 256 символов таблицы называются расширенным ASCII.

данных `char` по-прежнему занимает ровно один байт. Первая половина ASCII-таблицы представлена ниже.

Таблица 8.1: Первая половина таблицы ASCII

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
0	NUL	16	DLE	32	(sp)	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	TAB	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Номера от `0` до `31` отведены под *служебные символы* (их еще называют *управляющими*), которым не соответствуют никакие печатные знаки. Из них мы чаще всего пользуемся только `9` (табуляция) и сочетанием `13` (Carriage Return – перенос каретки) и `10` (Line Feed – новая строка) – перенос строки[22]. Символ с кодом `32` – пробел, а с кодом `8` – Backspace. Печатая этот символ на экран, вы эмулируете нажатие кнопки Backspace, стирая только что введенную букву. С его помощью можно, например, в консольных приложениях анимировать строку с «бегущими» процентами, показывающими прогресс выполнения какой-нибудь операции.

Внимательный читатель опять заметит одно несоответствие – в сноске ниже сказано, что в ОС Windows для обозначения переноса строки используется комбинация из двух специальных символов –

[22] В UNIX системах перенос строки обозначается одним символом с кодом `10` (`\n`), а в Windows-системах двумя – `13 10` (`\r\n`). Данная двухсимвольная комбинация берет свое начало еще от ручных печатающих машинок, в которых переход на новую строку осуществлялся с помощью двух действий – возврата «каретки» в начальную позицию и прокрутки барабана с бумагой для перехода на новую строку. Так, собственно, и действовали первые принтеры.

\r\n, в то время как мы всегда использовали только один – \n. Дело все в том, что язык Си изначально проектировался как *портабельный*, то есть одна и та же программа (если она не использует специальных вызовов ОС) должна без каких-либо изменений компилироваться в любой операционной системе. Это касается и такой мелочи, как символ переноса строки. Ведь не будем же мы переписывать программу под Windows только для того, чтобы поменять “Hello, World!\n” на “Hello, World!\r\n”? Поэтому в программах на языке Си для обозначения переноса строки всегда используется только один символ – \n – а уже сами библиотечные функции, знающие на какой платформе они выполняются, либо заменяет его на два, либо оставляют как есть. То же самое происходит и при чтении данных: независимо от того, что находится в файле – \r\n или \n – считываться обе эти комбинации будут как \n.

Таким образом, тип char создан специально для хранения ASCII-кодов (номеров) символов. Важно понимать, что с точки зрения кода *переменная типа char отличается от переменной типа int только объемом* – первая занимает в памяти один байт, а вторая – четыре. Поэтому мы можем использовать ее в том числе и для работы с обычными числовыми данными, если нам достаточно диапазонов [0…255] (для unsigned char) и [-128…127] (для обычного char).

8.2 Работа с символьными данными

Как же нам присвоить переменной типа char какой-либо символ[23]? Это можно сделать одним из трех способов. Первый – использовать *символьные константы*:

```
char c1 = 'a';
```

Необходимо помнить, что символьная константа, в отличие от строковой, заключается в *одинарные* кавычки. Если мы заключим a в двойные кавычки, то наша программа не станет компилироваться

[23] Несмотря на то, что в переменных типа char хранятся числа и правильно будет говорить «присвоить переменной типа char код такого-то символа», всюду далее для простоты мы будем говорить «присвоить переменной типа char такой-то символ» или просто «присвоить переменной x такой-то символ».

по причинам, описанным позже в этой главе. Рассмотрим еще один пример:

```
char x = '0', y = 0;
printf("%d %d\n", x, y);
```

В результате выполнения этого кода на экран будет выведено `48 0`. Первое число – это ASCII-код *символа* `'0'`, а второе – это *просто число ноль*.

Второй способ – если мы знаем ASCII-код какого-либо символа, мы можем прямо его и записать в переменную:

```
char c2 = 56;
```

Как его теперь вывести? Если использовать спецификатор `%d`, то на экране мы получим код (целое число) и в первом, и во втором случае:

```
printf("%d %d", c1, c2); // выведет 97 56
```

Для печати именно *символа*, а не его кода, надо использовать спецификатор `%c`:

```
printf("%c %c", c1, c2); // выведет a 8
```

Обратите внимание: *выводя значение переменной `c2`, функция `printf` напечатала не число `8`, а символ `'8'` с кодом `56`.*

Третий способ – мы можем воспользоваться Escape-последовательностями, начинающимися с обратной косой черты `'\'`. Их, как правило, используют в строковых константах, но никто не мешает нам присвоить escape-последовательность символьной переменной. С одной такой последовательностью мы уже познакомились – `\n`, которая соответствует символу с кодом `10`. Наиболее часто используемые последовательности представлены в таблице 8.2.

Таблица 8.2: Escape-последовательности

Escape-последовательность	Код	Описание
`\b`	`8`	удаление предыдущего символа (Backspace)
`\n`	`10`	новая строка
`\r`	`13`	возврат каретки
`\t`	`9`	табуляция
`\’`	`39`	одинарная кавычка

\”	34	двойная кавычка
\\	92	обратная косая черта

В частности, если мы захотим в символьной переменной сохранить символ одинарной кавычки, и мы не помним его ASCII-кода, то самым простым способом будет написать:

```
char c = '\'';
```

Если необходимо использовать двойные кавычки в строковой константе, то для этих целей также используют экранирование[24]:

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

Есть две полезные вещи, которые нужно знать о символах. Во-первых, язык Си интерпретирует символьные константы как целые числа (как коды, которые соответствуют этим константам). То есть мы можем, например, написать следующее:

```
// в c попадает 114 (сумма кодов 'a' и 'b')
unsigned char c = 'a' + 'b';
// 114 увеличивается на 1 и становится 115
c++;
// выводятся число 115 и символ 's'
printf("%d %c", c, c);
```

Во-вторых, символы латинского алфавита, а также цифры идут в ASCII таблице по порядку (`'a'`, `'b'`, `'c'` и т.д.). Это дает нам возможность определять, что за символ у нас в переменной – буква, цифра или что-то другое; а также дает возможность менять регистр букв и легко конвертировать символы цифр `'0'`, `'1'`, `'2'`, ... в числа `0`, `1`, `2` и обратно.

Пример 8.1. Определить, является ли символ, хранящийся в переменной `c`, маленькой латинской буквой.

Решение: `if (c >= 'a' && c <= 'z') { // является }`

Пример 8.2. Определить, является ли символ, хранящийся в переменной `c`, символом цифры.

Решение: `if (c >= '0' && c <= '9') { // является }`

[24] Escape-последовательности иногда также называют экранированными символами.

Пример 8.3. Если в переменной c лежит символ цифры, то преобразовать ее к соответствующему числу.

Решение:

```
int a;
if (c >= '0' && c <= '9')
    a = c – ‘0’;
```

Пример 8.4. Преобразовать маленькую латинскую букву в заглавную.

Решение:

```
if (c >= 'a' && c <= 'z')
    c = c - ('a' - 'A');
```

8.3 Строки

С отдельными символами разобрались. Поговорим теперь о *строках*. В языке Си строка – это массив символов (чем она, по сути, и является). К сожалению, в Си нет встроенного строкового типа данных, как в других языках программирования, поэтому вся работа со строками в языке Си является работой с массивами (символов).

Первая особенность работы со строками, которую необходимо усвоить с самого начала, это то, что *длина строки* и *размер массива символов* – это разные вещи. Массив может состоять из 250 ячеек (байт), а строка, записанная в этом массиве, может иметь длину в 16 символов. Как определить длину строки? В языке Си это делается с помощью специального *маркера конца строки*, а именно – *нулевого байта* (число `0` или символ `'\0'`). Допустим, в массиве из 15 символов записана строка “`Hello, world!`”. На Рис. 8.1 изображено, как это будет выглядеть в памяти.

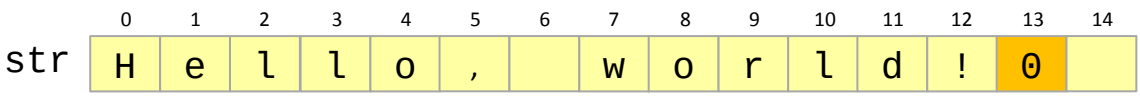


Рис. 8.1: Строка “Hello, world!” в пятнадцати-символьном массиве

Ноль в ячейке 13 – это не символ нуля, а нулевое значение! Во всех остальных ячейках для удобства изображены символы, хотя

в действительности там будут идти их коды, как изображено на Рис. 8.2.

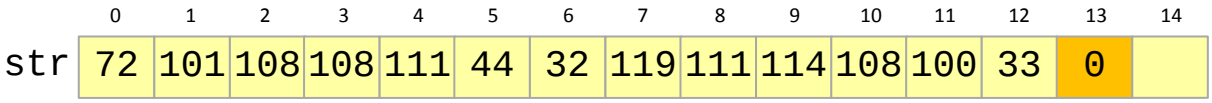


Рис. 8.2: Строка "Hello, world!" в пятнадцати-символьном массиве (на самом деле)

Как записать строку в массив? Есть три способа – инициализация массива строкой, чтение строки с клавиатуры (из файла) и использование библиотечных функций для работы со строками.

Точно так же, как мы можем инициализировать массив целых чисел, мы можем сразу инициализировать и строку группой символов. Например:

```
char str[50] = { 'H', 'e', 'l', 'l', 'o', 0 };
```

Не забываем про нулевой маркер в конце строки! В противном случае мы просто не сможем понять, где в массиве заканчивается строка. Количество ячеек массива, занимаемое строкой из 5 букв – шесть, так как последняя отводится под нулевой маркер. Это, в частности, ответ на вопрос, почему следующий код некорректен:

```
char q = "A";
```

Строка, состоящая даже из одного символа, в действительности содержит два – сам символ и нулевой байт.

В языке Си для инициализации строк есть более простая запись:

```
char str[6] = "Hello";
```

То есть вместо того, чтобы перечислять в фигурных скобках отдельные символы, мы можем инициализировать массив строковой константой. Компилятор сам разобьет эту строку на отдельные символы. Нужно только помнить, что в этой форме записи нулевой маркер конца строки не указывается – компилятор самостоятельного его вставляет в массив. Длина массива должна это учитывать!

Есть еще более удобная форма записи, позволяющая не подсчитывать количество элементов в строке:

```
char str[] = "Hello World!";
```

Компилятор сам создаст массив нужного размера (в данном примере 13 байт) и инициализирует его соответствующей строкой.

Еще один способ записать в массив строку – считать ее с клавиатуры или из файла:

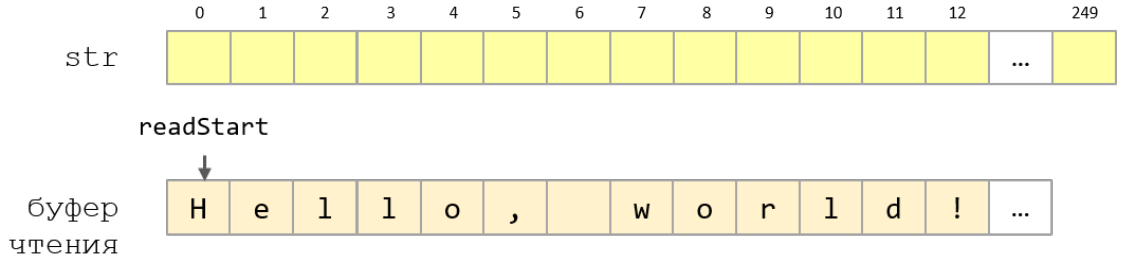
```
1. char str[250];
2. scanf_s("%s", str, 250);
3. gets_s(str, sizeof(str));
```

В строке 2 мы использовали уже знакомую нам функцию `scanf_s`, однако с двумя нововведениями: во-первых, мы указали спецификатор `%s`, благодаря которому функция понимает, что ее просят считать строку; во-вторых, мы использовали числовой аргумент – максимальное количество символов, которое функция может считать. Последнее как раз и отличает функцию `scanf_s` от ее «небезопасной» версии `scanf`, не отслеживающей выход за границы массива и оставляя это на усмотрение программиста, который, как правило, не всегда обращает на это внимание. В результате получается код, *уязвимый* к так называемым *атакам переполнения буфера* (buffer overflow attacks), когда злоумышленник имеет возможность записать в память за пределами отведенного массива вредоносный код. Компания Microsoft попыталась исправить эту ситуацию, запретив[25] в последних версиях своей среды разработки все стандартные функции ввода/вывода, и включив свои «защищенные» версии с суффиксом `_s` на конце, которые, работая со строками, в качестве дополнительного параметра всегда принимают размер массива.

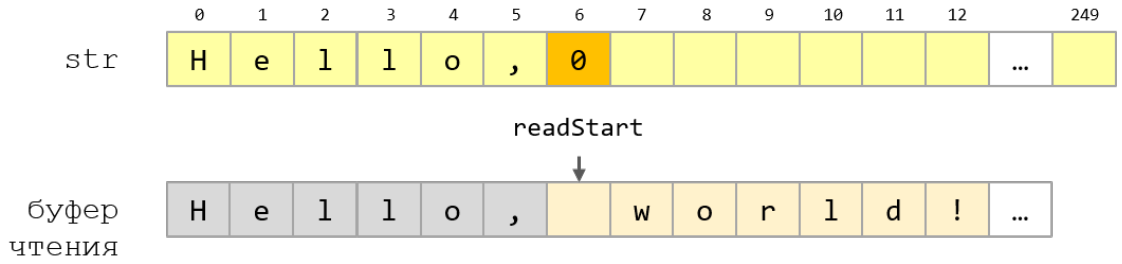
Функция `scanf_s` считывает символы до первого пробела или конца строки (что встретится раньше). В третьей строке мы воспользовались еще одной библиотечной функцией – `gets_s` (от get string), которая отличается от `scanf_s` тем, что считывает строку целиком вместе со всеми пробелами. В качестве второго аргумента мы ей передали не числовую константу, а значение встроенного в язык Си оператора `sizeof`, который возвращает размер

[25] В зависимости от версии студии использование стандартных функций может выдавать либо предупреждения, либо ошибки компиляции. И то, и другое вы можете запретить в настройках компилятора. Подобные нестандартные функции для работы с вводом/выводом безусловно рушат идею портабельности программ на Си, но если вы в принципе не планируете когда-либо портировать вашу программу под другие операционные системы, то лучше придерживаться правил игры от Microsoft и не использовать небезопасные версии функций. В конце концов, учитывая опыт компании, надо полагать, что у Microsoft был веский повод для подобного шага.

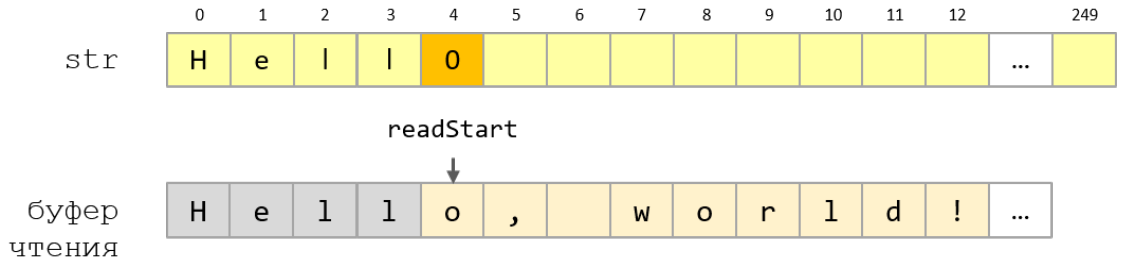
переменной/массива/типа данных[26] в байтах. Если количество считанных символов больше указанного максимального размера, то данные функции ведут себя по-разному. Функция `scanf_s` весь считанный фрагмент строки просто выбрасывает, а в массив ничего не записывает. Функция `gets_s` завершает работу программы с ошибкой. Если это поведение нас не устраивает, мы можем в спецификаторе функции `scanf_s` указать ширину поля, тогда она будет считывать ровно указанное количество символов (или меньше). На Рис. 8.3 показано состояние буфера чтения и массива `str` при различных вариантах вызова `scanf_s`, когда пользователь ввел с клавиатуры строку “`Hello, world!`”.



(а) исходное состояние

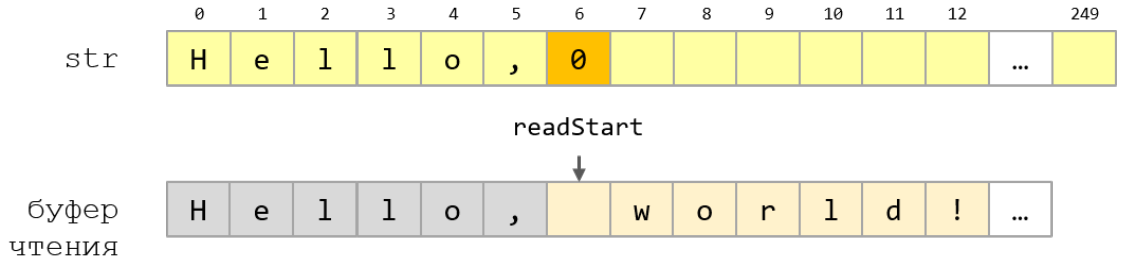


(б) после `scanf_s("%s", str, 250);`

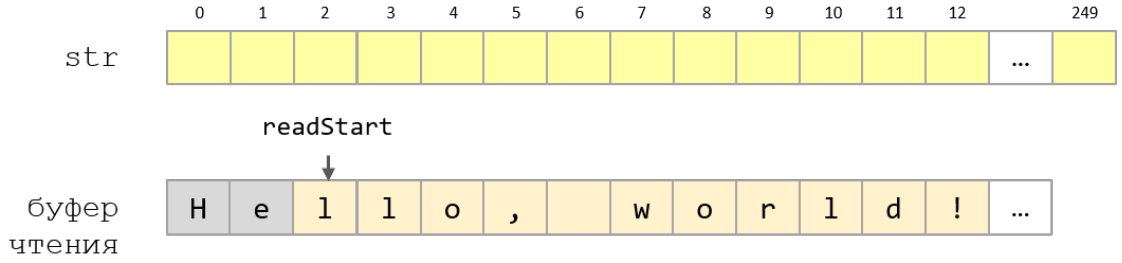


(в) после `scanf_s("%4s", str, 250);`

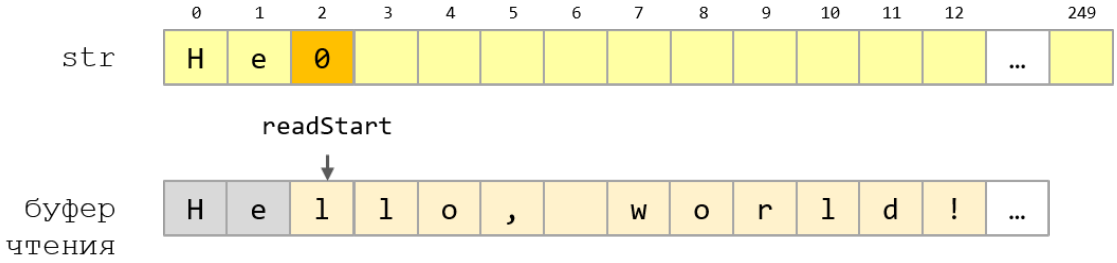
[26] Мы можем передать на вход sizeof как имя переменной/массива – `sizeof(i)` или `sizeof(mas)`, так и имя типа данных – `sizeof(int)` или `sizeof(char)`.



(г) после `scanf_s("%9s", str, 250);`



(д) после `scanf_s("%s", str, 3);`



(е) после `scanf_s("%2s", str, 3);`

Рис. 8.3: Различные варианты вызова функции `scanf_s`

Наконец, третий способ работы со строковыми данными – это использование библиотечных функций для работы со строками. Это, к сожалению, единственный способ, чтобы, например, скопировать значение одной строки в другую. В частности, если массив уже создан, то вариант с инициализацией не пройдет:

```
char str[250];
str = "Hello, world!";
```

Как передавать строки в функцию? Так же, как и любой массив – через указатель на первый элемент. Рассмотрим пример:

```
int startsWithDigit(char *str){
    if (str[0] >= '0' && str[0] <= '9')
        return 1;
```

```
    return 0;
}

char myString[250];

int main(){
    gets_s(myString, 250);
    printf("%d", startsWithDigit(myString));
    return 0;
}
```

В этой программе мы объявили массив `myString` вне какой-либо функции. Это способ создания так называемых *глобальных переменных*, которые видны из любой функции, поэтому мы можем обратиться к ней из `main`. Мы могли бы точно так же обратиться к строке и из функции `startsWithDigit()`, но мы не стали этого делать, чтобы получить универсальную функцию, которую можно использовать с любыми строками, объявленными где угодно.

Если мы не хотим, чтобы функция `startsWithDigit` изменяла нашу строку, нужно объявить параметр со спецификатором `const`:

```
int startsWithDigit(const char *str){
// str[0] = 'x'; - ошибка!
}
```

Тогда любая попытка записать что-нибудь в `str` приведет к ошибке компиляции. В функцию, объявленную таким образом (с параметром `const char* str`) можно передавать строковые константы, потому что именно такой тип они имеют:

```
printf("%d", startsWithDigit("Some string that does not
                                    start with digit."));
```

В следующем списке перечислены некоторые наиболее часто используемые функции из стандартной библиотеки. Для их использования необходимо подключить заголовочный файл `<string.h>`.

```
1) size_t strlen(const char *str);
```

Возвращает длину строки[27] в массиве `str`[28].

```
2) errno_t strcpy_s(char *strDst, size_t num, const char
*strSrc);
```

Копирует строку из `strSrc` в `strDst`, включая нулевой маркер. Переменная `num` содержит размер буфера `strDst`. Если копируемое значение плюс нулевой маркер больше `num`, функция возвращает код ошибки[29]. Если все прошло успешно, функция возвращает `0`.

```
3) errno_t strncpy_s(char *strDst, size_t num, const char
*strSrc, size_t count);
```

То же, что и `strcpy_s`, но только копирует не более `count` символов, дописывая в конец нулевой маркер.

```
4) errno_t strcat_s(char *strDst, size_t num, const char
*strSrc);
```

Дописывает `strSrc` в конец строки `strDst`, начиная с нулевого маркера в `strSrc`. Является функцией *конкатенации* (объединения) двух строк.

```
5) errno_t strncat_s(char *strDst, size_t num, const char
*strSrc, size_t count);
```

То же, что и `strcat_s`, только берется не более `count` символов строки `strSrc`.

```
6) int strcmp(const char *str1, const char *str2);
```

Сравнивает лексикографически строки `str1` и `str2`. Если первая строка меньше, возвращает отрицательное значение, если больше – положительное. Если строки полностью совпадают, возвращает `0`.

---

[27] Название типа `size_t` создано с помощью оператора `typedef`:

```
typedef unsigned int size_t;
```

Оператор `typedef` создает не новый тип данных, а еще одно имя для уже существующего, и используется для упрощения написания длинных имен типов, а также для создания «говорящих» имен, как в данном примере: size type – size_t.

[28] Данная функция не имеет суффикса _s, потому что является стандартной. В студии безопасные аналоги есть только у тех функций, которые что-то записывают в память. Функция strlen и ей подобные, конечно, тоже могут выйти за границы отведенного массива (например, если программист забыл поставить нулевой маркер в конце строки), но это не приводит к потенциальной возможности *записи* вредоносного кода.

[29] Еще одно имя-синоним:

```
typedef int errno_t;
```

```
7) int strncmp(const char *str1, const char *str2, size_t
count);
```

Сравнивает первые `count` символов строк `str1` и `str2`.

```
8) char* strchr(char *str, int ch);
```

Возвращает указатель на первое вхождение символа `ch` в строку `str1` или `NULL`, если вхождений не найдено.

```
9) char* strrchr(char *str, int ch);
```

Возвращает указатель на последнее вхождение символа `ch` в строку `str1` или `NULL`, если вхождений не найдено.

```
10) char* strstr(char *str, const char *substr);
```

Возвращает указатель на первое вхождение подстроки `substr` в строку `str` или `NULL`, если вхождений не найдено.

```
11) char* strpbrk(char *str, const char *control);
```

Возвращает указатель на первое вхождение любого символа из строки `control` в строку `str` или `NULL`, если вхождений не найдено.

Упражнения

Попробуйте решить самостоятельно следующие упражнения, *не прибегая к библиотечным функциям для работы со строками*.

8.1 Написать свою собственную функцию `strln()` для определения длины строки.

8.2 Написать свою функцию определения первого вхождения указанного символа в заданную строку. Функция должна возвращать адрес найденного символа или `NULL`, если найти его не удалось.

8.3 Написать свою функцию поиска первого вхождения подстроки в строку. Функция должна возвращать номер найденной позиции или `-1`, если найти вхождение не удалось.

Решения упражнений

8.1

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

int strln(const char* str) {
    int len = 0;
    while (str[len] != 0) len++;
    return len;
}

char myString[250];
int main() {
    setlocale(LC_ALL, "Russian");
    gets_s(myString, 250);
    printf("Длина введенной строки - %d\n",
                                    strln(myString));
    return 0;
}
```

8.2

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

char* findSymbol(char* str, char c) {
    int i = 0;
    while (str[i] != 0) {
        if (str[i] == c) return &str[i];
        i++;
    }
    return NULL;
}

char myString[250];
int main() {
    setlocale(LC_ALL, "Russian");
    char c;
```

```
    gets_s(myString, 250);
    scanf_s("%c", &c, 1);
    char *res = findSymbol(myString, c);
    if (res != NULL)
        printf("Символ %c вход в строку на позиции
                                %d\n", c, res - myString);
    else
        printf("Символ %c не найден в строке\n", c);
    return 0;
}
```

8.3

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

int findStr(char* where, char* what) {
    int where_len = strlen(where), what_len =
                                strlen(what), flag, i, j;
    if (where_len < what_len) return -1;
    for (i = 0; i < where_len - what_len + 1; i++) {
        flag = 1;
        for (j = 0; (j < what_len) && flag; j++) {
            if (where[i + j] != what[j]) flag = 0;
        }
        if (flag) return i;
    }
    return -1;
}

int main() {
    setlocale(LC_ALL, "Russian");
    char myString[250], myString2[250];
    gets_s(myString, 250);
    gets_s(myString2, 250);
    printf("%d\n", findStr(myString, myString2));
    return 0;
}
```

Упражнения для самостоятельного решения

8.4 Написать функцию, которая вычисляет количество вхождений переданного символа в строке. Например, для строки `"a simple string"` и символа `'s'` функция должна вернуть `2`. Программа спрашивает у пользователя строчку и символ и печатает количество вхождений этого символа в строку.

8.5 Написать функцию, удаляющую первый символ из строки.

8.6 Написать функцию, проверяющую, что строка является палиндромом.

8.7 Написать функцию, вычисляющую количество маленьких и больших букв в строке. Например, в строке `“Now I see bees, I won”` больших букв – `3`, маленьких – `10`. Пробелы и запятые буквами не являются.

8.8 Написать функцию, вычисляющую количество цифр в строке. Например, в строке `"sdf 2q43 jdsfhas3434"` содержится `7` цифр.

8.9 Написать функцию, вычисляющую длину первого слова в строке. Разделителем слов является пробел (строка может состоять из одного слова – в этом случае разделителей не будет, а также может быть вообще пустой).

8.10 Написать функцию, которая принимает на вход строчку и символ. Функцию должна выдать позицию самого последнего вхождения указанного символа в строке.

8.11 Написать две функции. Первая преобразует все символы строки к заглавным буквам, вторая преобразует все символы строки к строчным буквам.

8.12 Написать собственную функцию сравнения двух строк. Функция возвращает `1`, если строки одинаковые, `0` – иначе.

8.13 Написать функцию сравнения двух строк без учета регистра.

8.14 Написать функцию, переворачивающую переданную строку задом наперед. Например, из строки `"gateman"` должно получиться `"nametag"`.

8.15 Написать функцию, которая принимает на вход строчку и символ. Функция возвращает максимальное количество подряд идущих переданных символов в строке. Например, для строки `"asdaajhys as asaaafg r"` и символа `'a'`, нужно выдать `3`.

8.16 Написать функцию, которая принимает на вход строчку и два символа. Функция заменяет все вхождения первого символа в строке на второй символ. Например, для строки `"No lemon, no melon"` и символов `'m'` и `'d'` должна получиться строка `"No ledon, no delon"`.

8.17 Написать функцию, проверяющую, что переданную строку можно разбить на две одинаковые подстроки. Например, строку `"abcabc"` можно разбить на две одинаковые половины `"abc"`.

8.18 Напишите функцию, которая записывает строковое представление числа в массив. Число и массив передаются ей в качестве параметров.

8.19 Написать программу, которая выводит на экран все буквы, встречающиеся в строке, и их количество. Строку пользователь вводит с клавиатуры. Например, для строки `"A simple string"` нужно вывести:

```
A: 1
 : 2
s: 2
i: 2
m: 1
p: 1
l: 1
e: 1
t: 1
r: 1
n: 1
g: 1
```

Для решение этого и следующего упражнения можно использовать таблицу частот символов:

```
int freq[257] = { 0 };
char str[] = "Some string";
i = 0;
while (str[i])
    freq[str[i]]++;
```

Так как символы представляются числами, то мы можем использовать их для индексирования ячеек массива, в которых мы храним частоту появления соответствующих символов. После выполнения цикла ячейка `freq[i]` содержит частоту появления в строке символа с кодом `i`.

8.20 Написать программу, которая выводит на экран символ, встречающийся в введенной пользователем строке чаще всего.

8.21 Написать программу, которая выводит на экран кратчайшее слово в строке. Слова отделяются пробелами.

8.22 Написать программу, которая выполняет выравнивание строки, введенной пользователем, по ширине. То есть пользователь вводит строку `S` и желаемую ширину `N`. Программа добавляет в `S` необходимое число пробелов между словами, так, чтобы длина строки стала равна `N`. Причем:

– перед первым словом все пробелы удалить;
– после последнего слова все пробелы удалить;
– добавленные пробелы равномерно распределить между словами.

Если длина строки `S` изначально превосходит `N`, то удалить из `S` все лишние пробелы. Если после этого длина получившейся строки по-прежнему больше `N`, то оставить строку в таком состоянии.

8.23 Написать программу, которая выводит на экран слово `"yes"`, если из букв введенной строки `X` можно составить введенную строку `Y`, при условии, что каждую букву строки `X` можно использовать один раз. Если это невозможно сделать, вывести `"no"`.

Подсказка: можно сравнить частотные таблицы первой и второй строки.

Литература

[1] Керниган Б., Ритчи Д. Язык программирования С. — Москва: Вильямс, 2015. — 304 с.

[2] Страница загрузки Microsoft Visual Studio [Электронный ресурс] // Официальный сайт компании Майкрософт. — URL: https://www.visualstudio.com/ru/downloads/

[3] Документация по Visual C++ [Электронный ресурс] // Единая платформа размещения технической документации Майкрософт. — URL: https://docs.microsoft.com/ru-ru/cpp/

[4] Описание стандарта IEEE 754-2008 [Электронный ресурс] // Свободная энциклопедия Википедия. — URL: https://ru.wikipedia.org/wiki/IEEE_754-2008

Учебное издание

СОЛДАТЕНКО Илья Сергеевич

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ

Учебное пособие
(программирование)

По направлениям:
фундаментальная информатика и информационные технологии, прикладная математика и информатика, прикладная информатика, бизнес-информатика.

Компьютерная верстка И.С. Солдатенко
Подписано в печать 11.05.2017. Формат 60×84 1/16.
Усл.печ.л. 9,94. Тираж 200 экз. Заказ № 236.
Редакционно-издательское управление
Тверского государственного университета.
Адрес: 170100, г. Тверь, Студенческий пер. 12, корпус Б.
Тел РИУ: (4822) 35-60-63.