

Министерство науки и высшего образования Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Г. В. САБЛИНА, О. Д. ЯДРЫШНИКОВ

ПРОГРАММИРОВАНИЕ ЯЗЫК СИ

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2023

УДК 004.432(075.8)
С122

Рецензенты

А. В. Гунько, канд. тех. наук, доцент

Г. П. Голодных, канд. техн. наук, главный инженер АО «Синетик»

Саблина Г. В.

С122 Программирование. Язык СИ: учебное пособие / Г. В. Саблина, О. Д. Ядрышников. – Новосибирск: Изд-во НГТУ, 2023. – 134 с.

ISBN 978-5-7782-4964-6

Учебное пособие посвящено основам программирования на языке Си. Рассмотрены современные методы и средства разработки программного обеспечения, базовый синтаксис языка Си, типы данных, логические и арифметические выражения, основные конструкции структурированных языков программирования (последовательное выполнение, ветвление, циклы), косвенный доступ к памяти через указатели, функции, структуры, ввод-вывод и работа с файлами. В конце учебного пособия приведены задания для закрепления материала.

Учебное пособие предназначено для студентов, обучающихся по направлениям 27.03.04 «Управление в технических системах» и 09.03.01 «Информатика и вычислительная техника», а также может быть полезно студентам других направлений подготовки, изучающим программирование на языке Си.

Работа подготовлена на кафедре автоматике

УДК 004.432(075.8)

ISBN 978-5-7782-4964-6

© Саблина Г. В., Ядрышников О. Д., 2023
© Новосибирский государственный
технический университет, 2023

ВВЕДЕНИЕ

В настоящее время всё 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 постоянно эволюционируют и усложняются задачи, решаемые современным инженером. От современного специалиста требуется не просто освоить определённый объем материала, а прежде всего научиться им пользоваться для решения нетиповых задач, которые не разбирались в явном виде во время обучения и находятся на стыке различных областей знаний.

В частности, бурное развитие в XX в. 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. Примерами могут служить интеллектуальное управление (в задачах проектирования так называемого умного дома), всевозможные вопросы из области искусственного интеллекта, программная инженерия, робототехника, нечеткие интеллектуальные системы, мягкие вычисления и др. Всё это, в свою очередь, требует знаний и навыков программирования на языках высокого уровня.

Язык Си в настоящее время является стандартным базовым языком, с которого начинают своё знакомство с программированием студенты первых курсов вузов. Синтаксис многих современных языков берёт начало в языке Си. Поэтому, изучив его, впоследствии не составит труда освоить синтаксис таких популярных на сегодняшний день языков, как Java, C#, Java Script и др.

Настоящее учебное пособие подготовлено в соответствии с ФГОСЗ++ ВО и предназначено для студентов направлений 27.03.04 «Управление в технических системах» и 09.03.01 «Информатика и вычислительная техника», а также других направлений подготовки и специальностей, в которых программирование является профильной дисциплиной.

Учебное пособие состоит из 10 глав и содержит варианты практических заданий для закрепления материала.

1. СОВРЕМЕННЫЕ МЕТОДЫ И СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ (ПО)

1.1. ТЕХНОЛОГИИ РАЗРАБОТКИ ПО

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Технология программирования представляет собой набор технологических инструкций, включающих в себя следующее:

- указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой из них определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии, методы оценки и т. п.

Кроме набора операций и их последовательности технология также определяет способ описания проектируемой системы, точнее – модели, используемой на конкретном этапе разработки. Различают технологии, применяемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый метод, позволяющий решить конкретную задачу. В основе вторых – базовый метод или подход, который определяет совокупность методов, используемых на разных этапах разработки.

Чтобы разобраться в существующих технологиях программирования и определить основные тенденции их развития, целесообразно рассматривать эти технологии в историческом контексте, выделяя основные этапы развития программирования как науки.

1.2. ОСНОВНЫЕ ЭТАПЫ РАЗВИТИЯ ТЕХНОЛОГИИ РАЗРАБОТКИ ПО

Этап 1. «Стихийное» программирование (от момента появления первых вычислительных машин до середины 60-х гг. XX в.).

Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части. Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван структурным.

Этап 2. Структурный подход к программированию (60–70-е гг. XX в.).

Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приёмов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции (объектного, логического и пр.) такой способ получил название «процедурная декомпозиция».

В отличие от используемого ранее процедурного подхода к декомпозиции структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху вниз» и подразумевало реализацию общей идеи с обеспечением проработки интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов – метод пошаговой детализации.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных. Как следствие этого, в языках появилась возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы для того, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно

компилируемые модули – библиотеки подпрограмм (например, модуль графических ресурсов, модуль подпрограмм вывода на принтер). Связи между модулями при использовании рассматриваемой технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещён. Для разработки программного обеспечения большого объёма было предложено использовать объектный подход.

Этап 3. Объектный подход к программированию (с середины 1980-х гг. до настоящего времени).

Объектно-ориентированное программирование основывается на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путём передачи сообщений. При использовании этого подхода сохраняется зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь к ресурсам друг друга. Связи модулей нельзя разорвать, но можно попробовать стандартизировать их взаимодействие, на этом и основан компонентный подход к программированию.

Этап 4. Компонентный подход и CASE-технологии (с середины 1990-х гг. до настоящего времени).

Компонентный подход предполагает построение программного обеспечения из отдельных компонентов – физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию.

Этап 5. Разработка, ориентированная на архитектуру и CASE-технологии (с начала XXI в. до настоящего времени).

До конца XX в. все возможные проектные модели использовали исключительно для документирования промежуточных и заключительных этапов разработки программного обеспечения. Для этого применялись различные графические нотации и технологии, на основе которых

впоследствии был создан стандарт объектного моделирования. В ноябре 1997 г. после продолжительного процесса объединения различных методик группа OMG (Object Management Group) приняла получившийся в результате унифицированный язык моделирования (Unified Model Language, UML) в качестве стандарта.

Идея создания языка UML включала в себя не только реализацию стандарта для документирования и общения разработчиков, но и реализацию возможности использования UML как языка программирования. Тенденции развития средств разработки программных систем заключаются в создании таких средств, которые обеспечили бы не только автоматизацию всех этапов и процессов разработки программных систем, но и связь между результатами этапов. Одним из ключевых соединительных узлов является связь между проектными моделями и программным кодом. Когда разработка программной системы включает в себя проектирование её структуры и последующее кодирование, а все изменения в функциях разрабатываемой системы реализуются (начиная с перепроектирования архитектуры), то такая технология называется ориентированной на архитектуру (Model Driven Architecture, MDA).

Рассмотрим применяемые подходы на примере написания ПО на языке Си – одном из наиболее популярных и получивших широкое применение языков программирования.

2. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ

2.1. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

В настоящее время в мире существует несколько сотен реально используемых языков программирования, у каждого есть своя область применения. Для одного и того же компьютера можно написать программу на разных языках: Java, C#, Python и многих других. Исполнителем в данном случае является центральный процессор компьютера.

Итак, любой алгоритм есть последовательность предписаний, выполнив которые можно за конечное число шагов перейти от исходных данных к результату.

В зависимости от степени детализации предписаний обычно определяется уровень языка программирования: чем меньше детализация, тем выше уровень языка. По этому критерию можно выделить следующие уровни языков программирования: машинные, машинно-ориентированные (ассемблеры) и машинно-независимые (языки высокого уровня).

Машинные и машинно-ориентированные языки являются языками низкого уровня, требующими указания мелких деталей процесса обработки данных. Программирование на таких примитивных языках – занятие утомительное и подверженное ошибкам. Поэтому человек придумал языки высокого уровня, например, такой как Си.

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 язык является языком низкого уровня. Машина не понимает языков высокого уровня, поэтому прежде чем программу,

написанную на языке Си (или любом другом немашинном), сможет выполнить компьютер, её необходимо перевести на язык, понятный компьютеру. Этим занимается **компилятор**.

Процесс создания программы в самом общем виде изображён на рис. 1. Сначала пишется исходный код на высокоуровневом языке программирования, который сохраняется в обычном текстовом файле. Затем запускается компилятор, транслирующий (переводящий) исходный код (файл с расширением *.c) на машинный язык. Однако компилятор создаёт не готовую к исполнению программу, а только объектный код (файл с расширением *.obj). Этот код является промежуточным этапом при создании готовой программы. Дело в том, что создаваемая программа может содержать функции стандартных библиотек языка Си, реализации которых описаны в объектных файлах библиотек. Например, если в программе используется предназначенная для вывода информации на экран функция `printf()` стандартной библиотеки `stdio.h`, то это означает, что объектный файл *.obj будет содержать лишь инструкции по вызову данной функции, но код самой функции в нем будет отсутствовать. Для того чтобы итоговая исполняемая программа содержала все необходимые реализации функций, используется компоновщик объектных кодов.

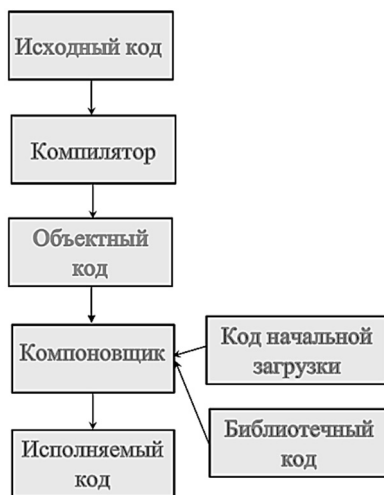


Рис. 1. Процесс создания программы

Компоновщик – это программа, которая объединяет в единый исполняемый файл объектные коды создаваемой программы, объектные коды реализаций библиотечных функций и стандартный код начальной загрузки для заданной операционной системы. В итоге и объектный файл, и исполняемый файл состоят из инструкций машинного кода. Однако объектный файл содержит только результат перевода на машинный язык текста программы, созданной программистом, а исполняемый файл – также и машинный код для используемых стандартных библиотечных подпрограмм и для кода начальной загрузки. Результат сохраняется в виде исполняемого кода (файл с расширением *.exe), который можно запускать на выполнение.

2.2. РАБОТА В IDE

Часто к компилятору прилагается ряд программ, призванных упростить 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 др.). Весь этот комплекс программ называется IDE (Integrated Development Environment) – интегрированная среда разработки. В отличие от набора дискретных (отдельных) приложений (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. Современная IDE состоит из следующих частей:

- 1) интеллектуального текстового редактора с функциями структурирования и подсветки кода, автозаполнения и др.;
- 2) встроенной справки;
- 3) компилятора;
- 4) компоновщика;
- 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, упрощающих разработку и отладку кода, и др.

Размер полностью установленной (за исключением справочного материала) современной IDE может измеряться несколькими гигабайтами, в то время как сам компилятор, являющийся по существу её центральным компонентом, занимает всего несколько мегабайт на жёстком диске.

В рамках настоящего учебного пособия мы будем использовать среду разработки Microsoft Visual Studio 2015, установить которую не составляет большого труда.

Создадим своё первое приложение. Для этого запустим Visual Studio. Откроется стартовая страница со ссылками на последние проекты, с которыми проводилась работа, новостями от компании Microsoft и другой информацией справочного характера. Первое, что необходимо сделать, – это создать проект.

Файлы с исходным кодом в Visual Studio не существуют сами по себе: они всегда являются частью проекта. Проект объединяет в себе все файлы с исходными кодами и другими ресурсами, необходимыми для сборки одного приложения. Программа может собираться из нескольких файлов (крупные приложения – из сотен различных файлов). Например, если разрабатывается графическое приложение, то будут собраны все оконные формы этого приложения, все графические изображения, иконки, звуки и другие медиаресурсы. Сложный программный продукт может состоять из нескольких отдельных программ, т. е. являться программным комплексом. Примером такового является сама IDE: для нас это одна программа, хотя в действительности их около сотни. Поэтому в Visual Studio все проекты объединяются в так называемые решения (solutions). Таким образом, решение – это совокупность проектов, где каждый проект представляет собой отдельную программу.

Далее нажимаем на ссылку «Создать проект...». Откроется окно мастера по созданию проектов (рис. 2).

Visual Studio позволяет выбрать язык программирования, на котором будет вестись разработка: Java, C#, Python и др. Более того, в среде разработки уже имеется 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.

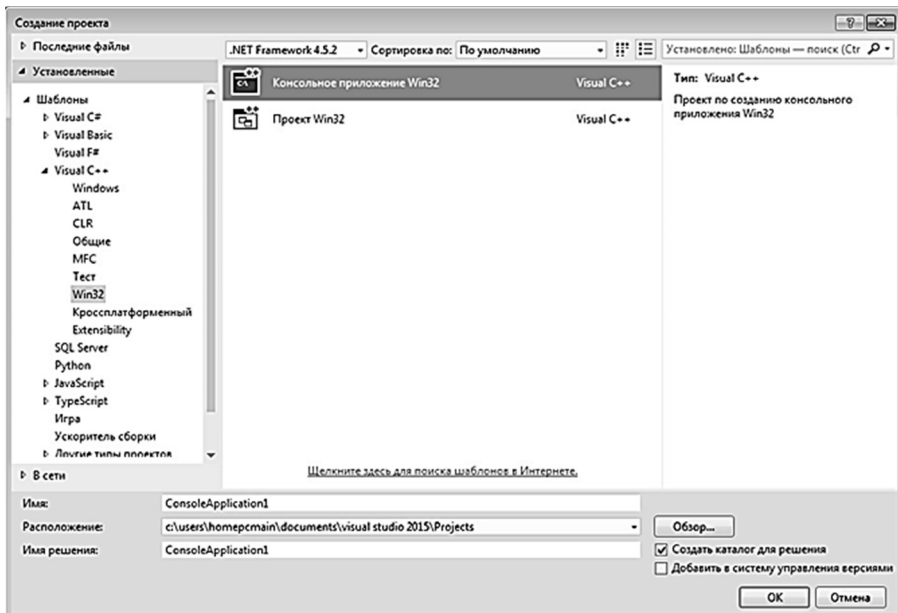


Рис. 2. Создание нового проекта

У каждого шаблона есть целый набор подшаблонов. Нас интересует шаблон Visual C++, подшаблон Win32, тип «Консольное приложение Win32».

Теперь 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 кнопку «ОК» и настраиваем дополнительные параметры. Необходимо выбрать пустой проект для консольного приложения (рис. 3).

После нажатия кнопки «Готово» на диске по указанному пути будет создано несколько папок с файлами для сопровождения проекта, однако главный файл с исходным кодом будет создан позже.

При этом появится окно, содержащее следующий код:

```
#include <stdio.h>
int main()
{ printf("Hello, World!");
  return 0;
}
```

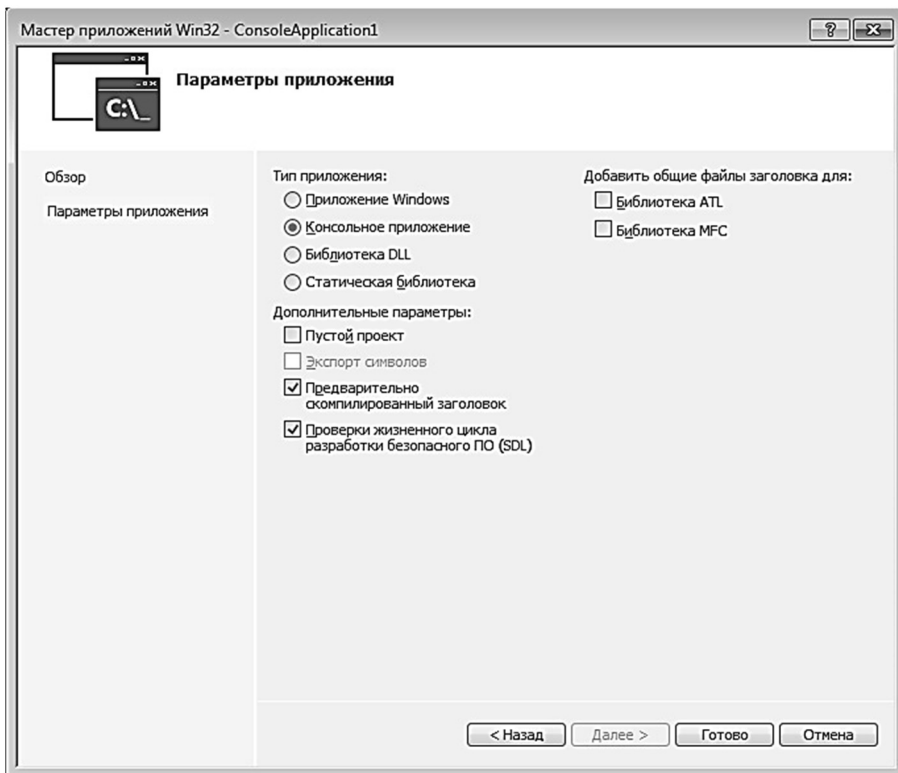


Рис. 3. Создание нового проекта. Параметры шаблона

Посмотрим на структуру проекта в *обозревателе решения* (Solution Explorer). Если обозреватель решений не виден, то его можно включить в меню «Вид»). На рис. 4 изображена логическая структура нашего проекта: на верхнем уровне представлено решение, в нем перечислены проекты, в которых имеются папки для заголовочных файлов, файлов с исходным кодом и файлами ресурсов.

Нажимаем правой кнопкой на папку «Файлы исходного кода» и выбираем пункт «Создать новый элемент». Откроется окно с набором шаблонов файлов, которые мы можем создать либо добавить в проект, – от элементов графического интерфейса до файлов ресурса (рис. 5).

Нам надо выбрать Код → Файл → C++. Указываем его имя и нажимаем кнопку «Добавить».

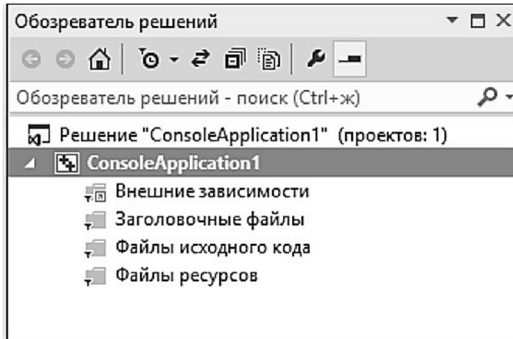


Рис. 4. Окно обозревателя решений (Solution Explorer)

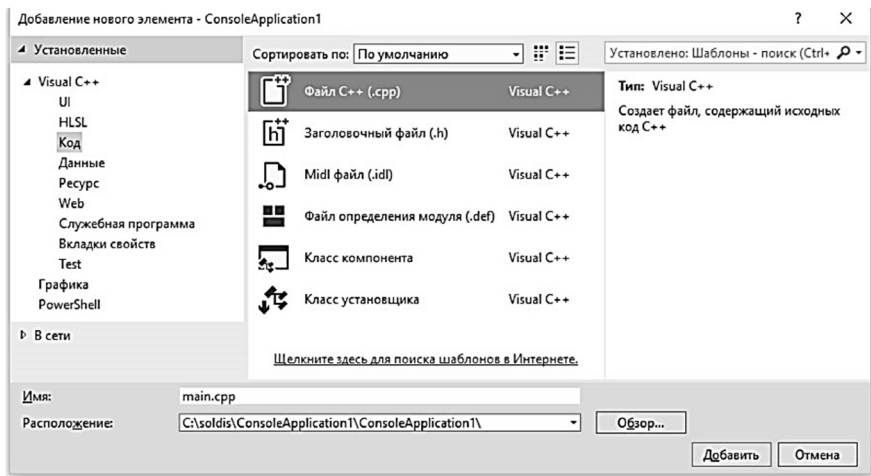


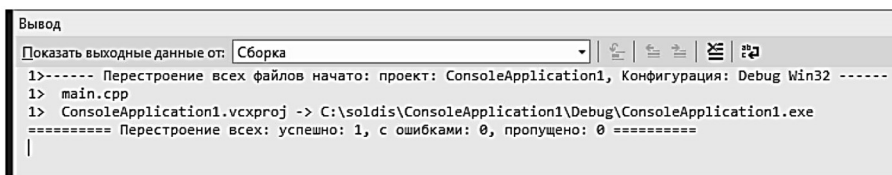
Рис. 5. Добавление файла в проект

Перепишем в окно текстового редактора следующий код:

```
#include <stdio.h>
#include <conio.h>
int main()
{ printf("Hello, World!");
  _getch();
  return 0;
}
```

Visual Studio выделяет цветом ключевые слова и автоматически выравнивает код, разбивая его на блоки, которые можно сворачивать и разворачивать по мере надобности. Можно также заметить многочисленные подсказки, «всплывающие» при наборе текста и при наведении мышью на различные элементы кода. Всё это – работа встроенного в Visual Studio интеллектуального текстового редактора. Смысл написанного кода мы разберём позже, а сейчас скомпилируем и запустим нашу первую программу, которая при удачном стечении обстоятельств должна вывести на экран строку приветствия.

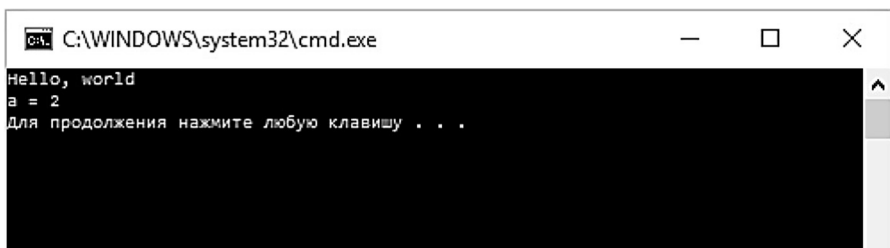
Компилируем решение. Для этого можно выбрать в меню «Сборка» пункт «Собрать решение» или просто нажать на клавиатуре клавишу <F7>. В окне «Вывод» внизу экрана мы видим сначала ход компиляции, а затем и её результат (рис. 6).



```
Вывод
Показать выходные данные от: Сборка
1>----- Перестроение всех файлов начато: проект: ConsoleApplication1, Конфигурация: Debug Win32 -----
1>  main.cpp
1>  ConsoleApplication1.vcxproj -> C:\soldis\ConsoleApplication1\Debug\ConsoleApplication1.exe
===== Перестроение всех: успешно: 1, с ошибками: 0, пропущено: 0 =====
|
```

Рис. 6. Вывод процесса компиляции

Если всё прошло без ошибок, то в последней строке будет написано: «успешно: 1, с ошибками: 0, пропущено: 0». В этом окне также видно 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.



```
C:\WINDOWS\system32\cmd.exe
Hello, world
a = 2
Для продолжения нажмите любую клавишу . . .
```

Рис. 7. Консольное окно приложения с результатами работы

Осталось запустить приложение. Это можно сделать, нажав клавишу <F5>, однако в этом случае приложение запустится в новом консольном окне, отработает и исчезнет, не дав увидеть результат работы.

Чтобы задержать консольное окно, приложение надо запускать с помощью комбинации клавиш <Ctrl>-<F5>, тогда после выполнения программы консольное окно будет держаться на экране до тех пор, пока не будет нажата любая клавиша на клавиатуре (рис. 7).

Задержать консольное окно на экране также можно, добавив в программу специальную функцию задержки экрана `_getch()` и подключив для неё библиотечный файл `conio.h`.

2.3. ОТЛАДКА

Рассмотрим основные средства и базовые приёмы отладки кода, которые предоставляет Visual Studio.

Приложение может не скомпилироваться из-за наличия синтаксических ошибок. Соответствующая информация будет выведена в окне «Вывод» внизу рабочего экрана.

Необходимо запомнить следующее правило: если в окне вывода отображаются сообщения о большом количестве ошибок, это вовсе не означает, что именно столько их присутствует в коде. Скорее всего, после исправления одной-двух ошибок количество диагностических сообщений существенно убавится либо сведётся к нулю. Для локализации предполагаемого места нахождения ошибки в коде надо дважды щёлкнуть мышью на самое первое диагностическое сообщение – курсор автоматически перейдёт 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 о возможных проблемах в коде (например, что используется устаревшая или небезопасная функция).

Если приложение скомпилировалось, это ещё не значит, что оно корректно. Во-первых, во время его работы могут возникнуть так называемые ошибки времени выполнения (`runtime errors`), когда приложение аварийно завершается с сообщением, что оно неправильно обратилось к памяти. Во-вторых, оно может отработать до конца без каких-либо сообщений, но при этом выдать совсем не тот результат, который должен быть. Такие ошибки называются логическими.

В этих случаях можно прибегнуть к механизмам отладки, основным из которых является пошаговое выполнение программы. Для перехода

в этот режим надо нажать клавишу <F10>, тогда приложение запускается, открывается консольное окно, а в текстовом редакторе напротив одной из строк появляется жёлтая стрелка. Эта стрелка указывает на ту инструкцию, которая будет выполнена следующей. Далее необходимо снова нажать <F10>. Так, последовательно нажимая <F10>, можно пройти по каждой строке кода, наблюдая за выводом в консольном окне. Если необходимо закончить пошаговое выполнение, то следует нажать клавишу <F5>. Во время отладки можно навести курсор на переменную и посмотреть её текущее значение. Это значение также можно увидеть в специальном окне «Видимые» внизу рабочего экрана. Можно посмотреть на всю цепочку вызовов функций, если таковые были. Если жёлтая стрелка указывает на строку, содержащую вызов функции, то для того чтобы перейти к этой функции, нужно нажать клавишу <F11>. В результате с большой степенью вероятности можно найти то место в коде, где возникает ошибка.

Если код слишком большой и ошибку найти не удаётся, то можно воспользоваться механизмом точек останова. Для этого нужно установить курсор на строку, с которой необходимо начать отладку, и нажать клавишу <F9>. Рядом с выбранной строкой появится красный маркер. Далее следует нажать клавишу <F5>, тогда приложение запустится, начнёт работать и, как только дойдёт до помеченной строки, перейдёт в режим пошаговой отладки. Можно устанавливать произвольное количество точек останова. Чтобы снять точку останова, нужно на соответствующей строке снова нажать <F9>. Этих простейших методов отладки будет достаточно для изучения языка Си и при выполнении упражнений.

3. СБОРКА РЕШЕНИЙ

3.1. ПРЕПРОЦЕССОР

Препроцессор (от англ. «предварительная обработка») – это программа, которая запускается до этапа компиляции, просматривает код и выполняет все инструкции, начинающиеся с символа «решётка». В частности, инструкция `include` копирует содержимое указанного в ней файла на место этой строки. Данная строка необходима для того, чтобы можно было пользоваться библиотечными функциями ввода-вывода, например `printf()`.

Разберём код простейшей программы на языке Си (нумерация строк приведена для удобства, в коде программы её быть не должно):

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

Строки 1 и 2 содержат команду препроцессора, подключающую заголовочные файлы стандартной библиотеки Си, которые дают программе возможность печатать сообщения на экран и использовать функции для работы с консолью.

`stdio.h` – библиотека для работы с вводом-выводом информации (standard input/output, стандартный ввод-вывод). Если программа будет что-то печатать на экран или считывать с клавиатуры, то в ней обязательно должна присутствовать строка 1.

`conio.h` – библиотека для работы с вводом-выводом информации с консоли (`concol input/output`, консольный ввод-вывод). Если программа будет что-то вводить на консоль или выводить информацию с консоли, то в ней обязательно должна присутствовать строка 2.

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 одном из следующих разделов). Определение функции состоит из заголовка (строка 3) и тела функции (строки 4–8), заключённого в фигурные скобки.

Главная функция `main()` обязательна в любой программе на языке Си, это так называемая точка входа в программу. Именно она вызывается операционной системой в момент запуска программы.

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

В 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.

Строка 6 с функцией `_getch()` предназначена для задержки экрана, она позволяет просмотреть его содержимое, пока не нажата любая клавиша на клавиатуре.

Строка 7 – это возврат параметра из функции, тип которого определён в строке 3. В данном примере это тип `integer`.

Вводу-выводу также посвящён отдельный раздел учебного пособия.

3.2. КОМПИАЦИЯ

Следующая стадия сборки приложения – компиляция, во время которой подготовленный препроцессором код переводится с языка Си на низкоуровневый машинный язык. Эта фаза является наиболее сложной из всех и сама состоит из нескольких этапов, рассмотрение которых выходит за рамки настоящего 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 ещё нельзя запустить на выполнение, так как он не содержит всей необходимой информации: в нём 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 код других объектных модулей, статических и динамических библиотек, а также некоторую дополнительную служебную информацию, необходимую операционной системе.

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. Из всего перечисленного необходимо собрать одно работающее приложение, чем и занимается на следующей стадии компоновщик.

3.3. КОМПОНОВКА

Компоновщик – это программа, которая осуществляет компоновку («связывание»): принимает на вход один или несколько объектных модулей и файлов библиотек и собирает из них исполняемый файл (либо файл динамической или статической библиотеки).

Исполняемый модуль (англ. executable file) – это файл, содержащий программу в том виде, в котором она может быть выполнена исполнителем (например, компьютером). Как правило, в случае операционной системы Windows это файл с расширением .exe или .com. Перед исполнением программа загружается в память и выполняются некоторые подготовительные операции (настройка окружения, загрузка библиотек).

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` для Windows или `.a`, `.so` для UNIX) представляет собой упакованную коллекцию объектных файлов. Смысл библиотечного файла заключён в его названии: это просто хранилище кода, которое используется другими программами по необходимости.

4. РАБОТА С ПАМЯТЬЮ

4.1. КОНСТАНТЫ И ПЕРЕМЕННЫЕ

Для хранения информации программы обычно используют переменные – именованные области компьютерной памяти, в которых можно хранить различные значения. В языке Си, прежде чем использовать переменную, её 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 записывается имя переменной (в нашем примере – `age`), которое может быть любой комбинацией букв, цифр и символа нижнего подчёркивания, начинающейся с буквы или символа нижнего подчёркивания. Для ввода информации с клавиатуры используется ещё одна функция из библиотеки `stdio.h` – `scanf_s`:

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

Первый параметр этой функции – форматная строка, которая описывает, что и какого типа нужно считать с клавиатуры. Комбинация символов `%d` означает, что надо считать одно десятичное число (`decimal`). Второй параметр – имя переменной, куда функция должна сохранить введённое с клавиатуры значение, со специальным символом `&`, предназначение которого будет рассмотрено 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);
8. my_age = age + 2;
9. printf("Ваш возраст: %d. Но я старше! Мой возраст – %d!!!\n",
age, my_age);
10. return 0;
11.}

```

Функция `scanf_s` не является обязательной частью стандарта языка Си и в настоящее время 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) необходимо использовать `scanf` вместо `scanf_s` потому, что последняя в них не поддерживается. Поскольку в рамках настоящего учебного пособия предполагается использование среды разработки MS Visual Studio 2015, то далее везде будет применяться синтаксис «безопасных» функций: `scanf_s`, `fscanf_s` и др.

Заметим, что, во-первых, в исходный код добавлена ещё одна библиотека – `locale.h`, где описан прототип функции `setlocale()`, благодаря которой теперь можно использовать в программах кириллицу. Во-вторых, в одном операторе объявлено одновременно сразу несколько переменных (строка 5). В-третьих, можно увидеть, что в коде используется оператор присваивания (строка 8), который присваивает значение правой части переменной, чьё имя указано в левой части. В-четвертых, в этой же строке присутствует пример простейшего выражения, вычисляющего сумму. В-пятых, в строке 9 кода осуществляется вывод на печать значений сразу двух переменных: `age` и `my_age`.

Переменные и константы используются для работы с данными в любом языке программирования. Переменная – это именованная область памяти, используемая для хранения промежуточных результатов. Её содержимое может изменяться, откуда и название: переменная. Константа – это некоторое конкретное значение (числовое или строковое), которое, будучи определённым 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 следующих видов:

– числовые (целочисленные и дробные), например, 0, -5, 3.14, 103 и др.;

– строковые (заклѳючѳенные в двойные кавычки) – "Hello, world! \n", "Input a number: " и др.

– символьные (заклѳючѳенные в одинарные кавычки) – 'a', 'b', '0' и др.

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 её 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 её создания измениться уже не может.

4.2. ТИПЫ ДАННЫХ

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, BASIC, Python и ряда других языков, где одно и то же имя можно использовать для поочерѳдного хранения значений самых разных типов). В языке Си есть следующие базовые (включѳенные в синтаксис самого языка) типы данных:

- **char** – символ;
- **int** – целое число;
- **float** – дробное число одинарной точности;
- **double** – дробное число двойной точности.

Напишем программу, спрашивающую у пользователя радиус бассейна (*r*) и его глубину (*h*). Затем она вычислит объѳм бассейна по формуле

$$V = \pi r^2 h,$$

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 иные спецификаторы. Например, для типа данных `float` необходимо использовать спецификатор `%f`.

Из приведённого фрагмента видно, что с помощью функции `scanf_s` можно считывать с клавиатуры сразу несколько значений.

4.3. КОММЕНТАРИИ В ПРОГРАММЕ НА ЯЗЫКЕ СИ

Комментарий – это набор символов, которые игнорируются компилятором. В Си все комментарии начинаются с пары символов `/*` и заканчиваются парой `*/`. Между косой чертой и звёздочкой не должно быть пробелов. Компилятор игнорирует любой текст между этими парами символов. Например, следующая программа выводит на экран только текст «hello»:

```
#include <stdio.h>
int main(void)
{printf("hello");
/* printf("there"); */
return 0;}
```

Комментарии следует использовать, когда необходимо объяснить какую-либо операцию кода. Все функции, за исключением самых очевидных, должны содержать комментарии в начале их объявления, где следует писать, что функция делает, какие параметры она получает

и что возвращает. Например, можно закомментировать целую строку, которая поясняет, что делает данная программа:

```
// Эта функция выводит на печать слово «hello»  
#include <stdio.h>  
int main(void)  
{printf("hello");  
/* printf("there"); */  
return 0;}
```

Комментарии в текстовом редакторе выделяются зелёным цветом.

4.4. 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 произойдёт, когда значение переменной одного типа присваивается переменной другого типа или если в одном выражении смешать переменные и константы разных типов? В языке Си, как и в других языках программирования, все бинарные операции работают только в том случае, если оба их операнда имеют одинаковый тип. Поэтому произойдёт 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 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 следующее: компилятор всегда старается преобразовать «меньший» тип к «большему», чтобы минимизировать возможную потерю информации.

Выпишем все базовые типы в порядке возрастания их «размера»: $\text{char} \leq \text{short} \leq \text{float} \leq \text{int} \leq \text{long} \leq \text{double}$. Если в результате операции возможна потеря точности, то компилятор выдаст соответствующее предупреждение, однако останавливать процесс компиляции из-за этого не будет, так как, строго говоря, это не считается ошибкой.

Что же 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 происходило в случае с константами, невозможно. Здесь поможет явное преобразование типов, с помощью которого мы явно скажем компилятору, что 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);
```

4.5. АДРЕСА ПЕРЕМЕННЫХ И УКАЗАТЕЛИ

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

Для того чтобы обращаться к переменным по адресам, необходимо следующее:

- 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 её адресу.

Для нахождения адреса переменной используется операция получения адреса – `&` (амперсанд). Это унарная операция, которая пишется непосредственно перед именем переменной. Хранятся адреса переменных тоже в переменных. Для таких переменных существует специальный тип, который называется указательным, а сами переменные такого типа называются указателями (так эти переменные называли 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 создаётся переменная-указатель, то необходимо сразу решить, адрес переменной какого типа он будет хранить. Объявляется переменная-указатель (в примере ниже – `plnt`) следующим образом:

```
int *plnt, i;
```

Перед именем указателя ставится знак `*`. Теперь можно в переменной-указателе `plnt` сохранять значения адресов целочисленных переменных. Заметим, что в примере создаются две переменные: один указатель на целый тип `plnt` и одна обычная целочисленная переменная `i`. Соответственно звёздочку надо ставить не один раз после типа, а перед каждым именем переменной, которую мы хотим объявить как указатель: `int *a, b, *c;`

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

5. ОСНОВЫ СТРУКТУРИРОВАННОГО ПРОГРАММИРОВАНИЯ

5.1. ОПЕРАНДЫ, ОПЕРАЦИИ, ВЫРАЖЕНИЯ

Операнд – это константа, литерал, идентификатор, вызов функции, индексное выражение, выражение выбора элемента или более сложное выражение, сформированное комбинацией операндов, знаков операций и круглых скобок. Любой операнд, который имеет константное значение, называется константным выражением. Каждый операнд имеет тип.

Любое **выражение** языка состоит из операндов, соединённых знаками **операций**. Знак операции – это символ или группа символов, которые сообщают компилятору о необходимости выполнения определённых арифметических, логических или других действий.

Операции выполняются в строгой последовательности. Величина, определяющая преимущественное право на выполнение той или иной операции, называется приоритетом. В табл. 1 перечислены различные операции языка Си. Их приоритеты для каждой группы одинаковы. Чем бóльшим преимуществом пользуется соответствующая группа операций, тем выше она расположена в таблице. Порядок выполнения операций может регулироваться с помощью круглых скобок.

Таблица 1

| Знак операции | Операция | Группа операций |
|---------------|--------------------|-------------------|
| * | Умножение | Мультипликативные |
| / | Деление | |
| % | Остаток от деления | |
| + | Сложение | Аддитивные |
| - | Вычитание | |

Окончание табл. 1

| Знак операции | Операция | Группа операций |
|---------------|---|------------------------------|
| << | Сдвиг влево | Операции сдвига |
| >> | Сдвиг вправо | |
| < | Меньше | Операции отношения |
| <= | Меньше или равно | |
| >= | Больше или равно | |
| == | Равно | |
| != | Не равно | |
| & | Поразрядное И | Поразрядные операции |
| | Поразрядное ИЛИ | |
| ^ | Поразрядное исключающее ИЛИ | |
| && | Логическое И | Логические операции |
| | Логическое ИЛИ | |
| , | Последовательное вычисление | Последовательного вычисления |
| = | Присваивание | Операции присваивания |
| *= | Умножение с присваиванием | |
| /= | Деление с присваиванием | |
| %= | Остаток от деления с присваиванием | |
| -= | Вычитание с присваиванием | |
| += | Сложение с присваиванием | |
| <<= | Сдвиг влево с присваиванием | |
| >>= | Сдвиг вправо с присваиванием | |
| &= | Поразрядное И с присваиванием | |
| = | Поразрядное ИЛИ с присваиванием | |
| ^= | Поразрядное исключающее ИЛИ с присваиванием | |

В языке Си присваивание также является выражением, и значением такого выражения является величина, которая присваивается.

Константное выражение – это выражение, результатом которого является константа. Операндом константного выражения могут быть

целые константы, символьные константы, константы с плавающей точкой, константы перечисления, выражения приведения типов, выражения с операцией `sizeof` и другие константные выражения. Однако на использование знаков операций в константных выражениях налагаются следующие ограничения:

1) в константных выражениях нельзя использовать операции присваивания и последовательного вычисления (,);

2) операция «адрес» (&) может быть использована только при некоторых инициализациях.

Выражения со знаками операций могут участвовать в выражениях как операнды. Выражения со знаками операций могут быть унарными (с одним операндом), бинарными (с двумя операндами) и тернарными (с тремя операндами).

Унарное выражение состоит из операнда и предшествующего ему знака унарной операции и имеет следующий формат:

знак-унарной-операции операнд;

Бинарное выражение состоит из двух операндов, разделённых знаком бинарной операции:

операнд1 знак-бинарной-операции операнд2.

Тернарное выражение состоит из трёх операндов, разделённых знаками тернарной операции (?) и (:), и имеет формат

операнд1? операнд2: операнд3;

В языке Си имеются следующие унарные операции:

- арифметическое отрицание (отрицание и дополнение);

~ побитовое логическое отрицание (дополнение);

! логическое отрицание;

* разадресация (косвенная адресация);

& вычисление адреса;

+ унарный плюс;

++ увеличение (инкремент);

-- уменьшение (декремент);

sizeof размер.

Унарные операции выполняются справа налево.

Операция арифметического отрицания (-) вырабатывает отрицание своего операнда. Операнд должен быть целой или плавающей величиной. При выполнении осуществляются обычные арифметические преобразования.

Пример:

```
double u = 5;
```

```
u = -u; /* переменной u присваивается ее отрицание,  
т. е. u принимает значение -5 */
```

Операция двоичного дополнения (~) вырабатывает двоичное дополнение своего операнда. Операнд должен быть целого типа. Осуществляется обычное арифметическое преобразование, результат имеет тип операнда после преобразования.

Пример:

```
char b = '9';
```

```
unsigned char f;
```

```
b = ~f;
```

Шестнадцатеричное значение символа '9' равно 39. В результате операции ~f будет получено шестнадцатеричное значение C6, что соответствует символу 'ц'.

Операция логического отрицания НЕ (!) вырабатывает значение 0, если операнд есть истина (не ноль), и значение 1, если операнд равен нулю. Результат имеет тип int. Операнд должен быть целого или плавающего типа, или типа указателя.

Пример:

```
int t, z=0;
```

```
t=!z;
```

Переменная t получит значение, равное единице, так как переменная z имела значение, равное нулю (ложь).

Операции разадресации и адреса используются для работы с переменными типа указатель.

Операция разадресации (*) осуществляет косвенный доступ к адресуемой величине через указатель. Операнд должен быть указателем. Результатом операции является величина, на которую указывает операнд.

Типом результата является тип величины, адресуемой указателем. Результат не определён, если указатель содержит недопустимый адрес.

Рассмотрим типичные ситуации, когда указатель содержит недопустимый адрес:

- указатель является нулевым;
- указатель определяет адрес такого объекта, который не является активным в момент ссылки;
- указатель определяет адрес, который не выровнен до типа объекта, на который он указывает;
- указатель определяет адрес, не используемый выполняющейся программой.

Операция «адрес» (&) даёт адрес своего операнда. Операндом может быть любое именуемое выражение. Имя функции или массива также может быть операндом операции «адрес», хотя в этом случае знак операции является лишним, потому что имена массивов и функций являются адресами. Результатом операции «адрес» является указатель на операнд. Тип, адресуемый указателем, является типом операнда.

Операция «адрес» не может применяться к элементам структуры, являющимся полями битов, и к объектам с классом памяти `register`.

Примеры:

```
int t, f=0, * address;
address = &t; /* переменной address, объявляемой как
              указатель, присваивается адрес переменной t */
* address = f; /* переменной, находящейся по адресу,
               содержащемуся в переменной address,
               присваивается значение переменной f, т.е. 0,
               что эквивалентно t=f; т. е. t=0; */
```

Помимо перечисленных основных операций язык Си также имеет операцию унарного минуса, который инвертирует знак своего операнда (переменной `a` будет присвоено значение переменной `b` с противоположным знаком): `a = -b`, и унарного плюса, который, собственно, ничего не делает и существует только для полноты картины.

Операции инкремента (`++`) и декремента (`--`) увеличивают или уменьшают значение операнда на единицу и могут быть записаны как справа, так и слева от операнда. Если знак операции записан перед операндом (префиксная форма), то изменение операнда происходит до его

использования в выражении. Если знак операции записан после операнда (постфиксная форма), то операнд вначале используется в выражении, а затем происходит его изменение (табл. 2).

Т а б л и ц а 2

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

С помощью операции **sizeof** можно определить размер памяти, которая соответствует идентификатору или типу. Операция **sizeof** имеет следующий формат: **sizeof(выражение)**;

В качестве выражения может использоваться любой идентификатор либо имя типа, заключённое в скобки. Отметим, что не может быть использовано имя типа **void**, а идентификатор не может относиться к полю битов или быть именем функции.

В отличие от унарных операций бинарные выполняются слева направо. Список бинарных операций приведён в табл. 3.

Т а б л и ц а 3

| Приоритет | Знак операции | Типы операции | Порядок выполнения |
|-----------|---|-----------------------|--------------------|
| 1 | () [] . -> | Выражение | Слева направо |
| 2 | - ~ ! * & ++ -- sizeof приведение типов | Унарные | Справа налево |
| 3 | * / % | Мультипликативные | Слева направо |
| 4 | + - | Аддитивные | |
| 5 | << >> | Сдвиг | |
| 6 | < > <= >= | Отношение | |
| 7 | == != | Отношение (равенство) | |

| Приоритет | Знак операции | Типы операции | Порядок выполнения |
|-----------|-----------------------------------|----------------------------------|--------------------|
| 8 | & | Поразрядное И | |
| 9 | ^ | Поразрядное исключающее ИЛИ | |
| 10 | | Поразрядное ИЛИ | |
| 11 | && | Логическое И | |
| 12 | | Логическое ИЛИ | |
| 13 | ? : | Условная | |
| 14 | = *= /= %= += -= &= = >>= <<= ^= | Простое и составное присваивание | Справа налево |
| 15 | , | Последовательное вычисление | Слева направо |

Все бинарные операции можно разделить на мультипликативные, аддитивные, сравнения (отношения), поразрядные, логические, присваивания, последовательного вычисления.

К классу мультипликативных относятся операции умножения (*), деления (/) и получение остатка от деления (%). Операндами операции (%) должны быть целые числа. Отметим, что типы операндов операций умножения и деления могут отличаться и для них справедливы правила преобразования типов. Типом результата является тип операндов после преобразования.

Операция умножения (*) выполняет умножение операндов:

```
int i=5;
float f=0.2;
double g,z;
g=f*i;
```

Тип произведения `i` и `f` преобразуется к типу `double`, затем результат присваивается переменной `g`.

Операция деления (/) выполняет деление первого операнда на второй. Если две целые величины не делятся нацело, то результат округляется в меньшую сторону.

При попытке деления на ноль выдаётся сообщение во время выполнения.

```
int i=49, j=10, n, m;
n = i/j;          /* результат  4  */
m = i/(-j);       /* результат -4  */
```

Операция «остаток от деления (%)» даёт остаток от деления первого операнда на второй.

Знак результата зависит от конкретной реализации операции. В данной реализации знак результата совпадает со знаком делимого. Если второй операнд равен нулю, то выдаётся сообщение

```
int n = 49, m = 10, i, j, k, l;
i = n % m;          /*  9  */
j = n % (-m);       /*  9  */
k = (-n) % m;       /* -9  */
l = (-n) % (-m);    /* -9  */
```

К аддитивным операциям относятся сложение (+) и вычитание (-). Операнды могут быть целого или плавающего типов. В некоторых случаях над операндами аддитивных операций выполняются общие арифметические преобразования. Однако преобразования, выполняемые при аддитивных операциях, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат аддитивной операции не может быть представлен типом операндов после преобразования. При этом сообщение об ошибке не выдаётся.

Пример:

```
int i=30000, j=30000, k;
k=i+j;
```

В результате сложения k получит значение, равное -5536.

Результатом выполнения операции сложения является сумма двух операндов. Операнды могут быть целого или плавающего типа, или один операнд может быть указателем, а второй – целой величиной.

Если целая величина складывается с указателем, то целая величина преобразуется путём её умножения на размер памяти, занимаемой величиной, которая адресуется указателем. Когда преобразованная целая величина складывается с величиной указателя, то результатом является указатель, адресующий ячейку памяти, расположенную на целую величину дальше от исходного адреса. Новое значение указателя адресует тот же самый тип данных, что и исходный указатель.

Операция вычитания (-) вычитает второй операнд из первого. Возможна следующая комбинация операндов:

- 1) оба операнда целого или плавающего типа;
- 2) оба операнда являются указателями на один и тот же тип;
- 3) первый операнд является указателем, а второй – целым.

Отметим, что операции сложения и вычитания над адресами в единицах, отличных от длины типа, могут привести к непредсказуемым результатам.

Пример:

```
double d[10], * u;  
int i;  
u = d+2; /* u указывает на третий элемент массива */  
i = u-d; /* i принимает значение равное 2 */
```

Операции сдвига осуществляют смещение операнда влево (<<) или вправо (>>) на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами. Выполняются обычные арифметические преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в ноль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип `unsigned`, то свободные левые биты устанавливаются в ноль, в противном случае они заполняются копией знакового бита. Результат операции сдвига не определён, если второй операнд отрицательный.

К поразрядным операциям относятся операции поразрядного логического И (&), поразрядного логического ИЛИ (|) и поразрядного исключающего ИЛИ (^).

Операнды поразрядных операций могут быть любого целого типа. При необходимости над операндами выполняются преобразования по умолчанию, тип результата – это тип операндов после преобразования.

Операция поразрядного логического И (&) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита – единицы, то соответствующий бит результата устанавливается в единицу, в противном случае – в ноль.

Операция поразрядного логического ИЛИ (|) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой из сравниваемых битов равен единице (или оба), то соответствующий бит результата устанавливается в единицу, в противном случае результирующий бит равен нулю.

Операция поразрядного исключающего ИЛИ (^) сравнивает каждый бит первого операнда с соответствующими битами второго операнда. Если один из сравниваемых битов равен нулю, а второй бит – единице, то соответствующий бит результата устанавливается в единицу, в противном случае (т. е. когда оба бита равны единице или нулю) бит результата устанавливается в ноль.

Пример.

```
int i=0x45FF,    /* i= 0100 0101 1111 1111 */
    j=0x00FF;     /* j= 0000 0000 1111 1111 */
char r;
r = i^j;          /* r=0x4500 = 0100 0101 0000 0000 */
r = i|j;          /* r=0x45FF = 0100 0101 0000 0000 */
r = i&j;          /* r=0x00FF = 0000 0000 1111 1111 */
```

К логическим операциям относятся операции логического И (&&) и логического ИЛИ (||). Операнды логических операций могут быть целого типа, плавающего типа или типа указателя, при этом в каждой операции могут участвовать операнды различных типов.

Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, то второй операнд не вычисляется.

Логические операции не вызывают стандартных арифметических преобразований. Они оценивают каждый операнд с точки зрения его эквивалентности нулю. Результатом логической операции является ноль или единица, тип результата – int.

Операция логического И (&&) вырабатывает значение «1», если оба операнда имеют нулевые значения. Если один из операндов равен нулю, то результат также равен нулю. Если значение первого операнда равно нулю, то второй операнд не вычисляется.

Операция логического ИЛИ (||) выполняет над операндами операцию включающего ИЛИ. Она вырабатывает значение «0», если оба операнда имеют значение «0»; если какой-либо из операндов имеет ненулевое значение, то результат операции равен единице. Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

Операция простого присваивания используется для замены значения левого операнда значением правого операнда. При присваивании

производится преобразование типа правого операнда к типу левого операнда по правилам, упомянутым раньше. Левый операнд должен быть модифицируемым.

Пример:

```
int t;  
char f;  
long z;  
t=f+z;
```

В приведённом примере значение переменной `f` преобразуется к типу `long`, вычисляется `f+z`, результат преобразуется к типу `int` и затем присваивается переменной `t`.

Кроме простого присваивания имеется целая группа операций, которые объединяют простое присваивание с одной из бинарных операций. Такие операции называются составными операциями присваивания и имеют вид

$$(\text{операнд-1}) (\text{бинарная операция}) = (\text{операнд-2}).$$

Составное присваивание по результату эквивалентно следующему простому присваиванию:

$$(\text{операнд-1}) = (\text{операнд-1}) (\text{бинарная операция}) (\text{операнд-2}).$$

Отметим, что выражение составного присваивания с точки зрения реализации не эквивалентно простому присваиванию, так как в последнем операнд-1 вычисляется дважды.

Каждая операция составного присваивания выполняет преобразование, которые осуществляются соответствующей бинарной операцией. Левым операндом операций `(+=)` `(-=)` может быть указатель, в то время как правый операнд должен быть целым числом.

Примеры:

```
double arr[4]={ 2.0, 3.3, 5.2, 7.5 } ;  
double b=3.0;  
b+=arr[2]; /* эквивалентно b=b+arr[2] */  
arr[3]/=b+1; /* эквивалентно arr[3]=arr[3]/(b+1) */
```


Заметим, что при втором присваивании использование составного присваивания даёт более заметный выигрыш во времени выполнения операции, так как левый операнд является индексным выражением.

В языке Си имеется одна тернарная операция – условная операция, которая имеет следующий формат:

операнд-1 ? Операнд-2 : операнд-3;

Операнд-1 должен быть целого или плавающего типа либо быть указателем. Он оценивается с точки зрения его эквивалентности нулю. Если операнд-1 не равен нулю, то вычисляется операнд-2 и его значение является результатом операции. Если операнд-1 равен нулю, то вычисляется операнд-3 и его значение является результатом операции. Следует отметить, что вычисляется либо операнд-2, либо операнд-3, но не оба. Тип результата зависит от типов операнда-2 и операнда-3 следующим образом.

1. Если операнд-2 или операнд-3 имеет целый или плавающий тип (отметим, что их типы могут отличаться), то выполняются обычные арифметические преобразования. Типом результата является тип операнда после преобразования.

2. Если операнд-2 и операнд-3 имеют один и тот же тип структуры, объединения или указателя, то тип результата будет тем же самым типом структуры, объединения или указателя.

3. Если оба операнда имеют тип `void` (ранее не описан), то результат имеет тип `void`.

4. Если один операнд является указателем на объект любого типа, а другой – указателем на `void`, то указатель на объект преобразуется к указателю на `void`, который и будет типом результата.

5. Если один из операндов является указателем, а другой – константным выражением со значением «0», то типом результата будет тип указателя.

Пример:

`max = (d<=b) ? b : d;`

Здесь переменной `max` присваивается максимальное значение переменных `d` и `b`.

В языке Си операции с высшими приоритетами вычисляются первыми. Наивысшим приоритетом является приоритет, равный единице. Приоритеты и порядок операций приведены в табл. 3.

Поскольку описанные в таблице операции изменяют значения переменных, над которыми они применяются, то невозможно использовать их с константами. В частности, следующая строка вызовет ошибку компиляции:

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

Это происходит, потому что для работы инкременту и декременту требуется *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 ожидается.

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, приведённый выше, вы поймёте, что в нём не всё так очевидно. Удобнее использовать инкременты и декременты, когда 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 и представляет собой некоторую формулу, состоящую из идентификаторов, констант и символов операций; его тоже можно 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».

Ещё 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 распространённых ошибок начинающих программистов (особенно тех, кто переходит с таких языков, как BASIC или Pascal) – использование одного символа равенства вместо двух. Следует знать, что это не является синтаксической ошибкой и приложение скомпилируется и запустится, но ошибочный результат может проявить себя нетривиальным образом.

Логические операции позволяют комбинировать отдельные логические выражения, таким образом получаются более сложные «вопросы». Их всего три:

\parallel – 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.

Логические операторы – это операторы, которые принимают в качестве аргументов логические значения (ложь или истина) и возвращают логическое значение. Как и обычные операторы, они могут быть одноместными (унарными, т. е. принимать один аргумент), двуместными (бинарными, принимать два аргумента), трёхместными и т. д.

Особенностью языка Си является то, что в нём нет типа, хранящего булево значение (ложь или истина). В языке Си ложью (логическим нулем) считается целочисленный ноль, а любое ненулевое целое будет логической истиной. Например:

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

void main() {
    char boolValue = -71;
    if (boolValue) {
        printf("boolValue is true");
    } else }
```

Логические значения обычно порождаются операторами сравнения (`==`, `!=`, `>`, `<`, `>=`, `<=`).

Главная особенность выражений – это то, что они вычислимы, т. е. их значение можно подсчитать. Вычисляются выражения интуитивно понятным образом: переменные и символические константы заменяются на их текущие значения, затем получившаяся математическая формула вычисляется в порядке приоритетов операций. Приоритеты рассмотренных операций знакомы нам из школьной программы: самый высокий приоритет имеют скобки, затем операции умножения, деления и взятия остатка от деления, а потом сложение и вычитание.

5.2. ОСНОВНЫЕ ОПЕРАТОРЫ ЯЗЫКА СИ

Все операторы языка Си могут быть разделены на следующие категории:

- условные операторы, к которым относятся оператор условия `if` и оператор выбора `switch`;
- операторы цикла (`for`, `while`, `do-while`);
- операторы перехода (`break`, `continue`, `return`, `goto`);
- другие операторы (оператор «выражение», пустой оператор).

Операторы в программе могут объединяться в составные операторы с помощью фигурных скобок. Любой оператор в программе может быть помечен меткой, состоящей из имени и следующего за ним двоеточия. Все операторы языка Си, кроме составных операторов, заканчиваются точкой с запятой (;).

Оператор «выражение»

Любое выражение, которое заканчивается точкой с запятой, является оператором.

Выполнение оператора «выражение» заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызвать функцию, не возвращающую значения, можно только при помощи оператора «выражение». Правила вычисления выражений были сформулированы выше.

Примеры:

`++i`; – этот оператор представляет выражение, которое увеличивает значение переменной `i` на единицу.

`a=cos(b * 5)`; – этот оператор представляет выражение, включающее в себя операции присваивания и вызова функции.

`a(x,y)`; – этот оператор представляет выражение, состоящее из вызова функции.

Пустой оператор

Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Он обычно используется в следующих случаях:

- в операторах `for`, `while`, `do-while` и `if` в строках, когда место оператора не требуется, но по синтаксису требуется хотя бы один оператор;
- при необходимости пометить фигурную скобку.

Синтаксис языка Си требует, чтобы после метки обязательно следовал оператор, но фигурная скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.

Пример:

```
int main ( )
{
    :
    { if (...) goto a; /* переход на скобку */
      { ...
        }
    a;; }
    return 0;
}
```

Составной оператор

Составной оператор представляет собой несколько операторов и объявлений, заключённых в фигурные скобки:

```
{ [объявление]
  :
  оператор; [оператор];
  :
}
```

Заметим, что в конце составного оператора точка с запятой не ставится. Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов.

Пример:

```
int main ( )
{
    int q,b;
    double t,d;
    :
    if (...)
    {
```

```

    int  e,g;
    double f,q;
    :
    }
    :
    return (0);
}

```

Переменные *e*, *g*, *f* и *q* будут уничтожены после выполнения составного оператора. Отметим, что переменная *q* является локальной в составном операторе, т. е. она никоим образом не связана с переменной *q*, объявленной в начале функции *main* с типом *int*. Отметим также, что выражение, стоящее после *return*, может быть заключено в круглые скобки, хотя наличие последних необязательно.

Оператор **if**

Формат оператора:

if (выражение) оператор 1; [**else** оператор 2;]

Выполнение оператора **if** начинается с вычисления выражения. Далее выполнение осуществляется по следующей схеме:

- если выражение истинно (т. е. отлично от нуля), то выполняется оператор 1;
- если выражение ложно (т. е. равно нулю), то выполняется оператор 2;
- если выражение ложно и отсутствует оператор 2 (в квадратные скобки заключена необязательная конструкция), то выполняется следующий за **if** оператор.

После выполнения оператора **if** значение передаётся на следующий оператор программы, если последовательность выполнения операторов программы не будет принудительно нарушена использованием операторов перехода.

Пример:

```

if (i < j) i++;
else { j = i-3; i++; }

```

Этот пример иллюстрирует также и тот факт, что на месте оператора 1, так же как и на месте оператора 2, могут находиться сложные конструкции.

Допускается использование вложенных операторов `if`. Оператор `if` может быть включён в конструкцию `if` или в конструкцию `else` другого оператора `if`. Чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах `if` с использованием фигурных скобок. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово `else` с наиболее близким `if`, для которого нет `else`.

Пример:

```
int main ( )
{
    int t=2, b=7, r=3;
    if (t>b)
    {
        if (b < r) r=b;
    }
    else r=t;
    return (0);}

```

В результате выполнения этой программы $r = 2$.

Если же в программе опустить фигурные скобки, стоящие после оператора `if`, то программа будет иметь следующий вид:

```
int main ( )
{
    int t=2,b=7,r=3;
    if ( a>b )
        if ( b < c ) t=b;
    else    r=t;
    return (0);}

```

В этом случае $r = 3$, так как `else` относится ко второму оператору `if`, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе `if`.

Следующий фрагмент иллюстрирует вложенные операторы `if`:

```
char ZNAC;
int x,y,z;
:

```



```

if (ZNAC == '-') x = y - z;
else if (ZNAC == '+') x = y + z;
    else if (ZNAC == '*') x = y * z;
        else if (ZNAC == '/') x = y / z;
            else ...

```

Из рассмотрения этого примера можно сделать вывод, что конструкции, использующие вложенные операторы `if`, являются довольно громоздкими и не всегда достаточно надёжными. Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора `switch`.

Оператор `switch`

Оператор `switch` предназначен для организации выбора из множества различных вариантов. Формат оператора следующий:

```

switch (выражение)
{ [объявление]
  :
  [ case константное-выражение 1]: [ список операторов 1]
  [ case константное-выражение 2]: [ список операторов 2]
  :
  [ default: [ список операторов n ]]
}

```

Выражение, следующее за ключевым словом `switch` в круглых скобках, может быть любым выражением, допустимым в языке Си, и его значение должно быть целым. Отметим, что можно использовать явное приведение к целому типу, однако необходимо помнить о тех ограничениях и рекомендациях, о которых говорилось выше.

Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора `switch` состоит из нескольких операторов, помеченных ключевым словом `case` с последующим константным выражением. Следует отметить, что использование целого константного выражения является существенным недостатком, присущим рассматриваемому оператору.

Поскольку константное выражение вычисляется во время трансляции, то оно не может содержать переменные или вызовы функций.

Обычно в качестве константного выражения используются целые или символьные константы.

Все константные выражения в операторе **switch** должны быть уникальны. Кроме операторов, помеченных ключевым словом **case**, может быть (но обязательно один) фрагмент, помеченный ключевым словом **default**.

Список операторов может быть пустым либо содержать один оператор или более. Причём в операторе **switch** не требуется заключать последовательность операторов в фигурные скобки.

Отметим также, что в операторе **switch** можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом **case**, однако в объявлениях не должна использоваться инициализация.

Схема выполнения оператора **switch** следующая:

- вычисляется выражение в круглых скобках;
- вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами **case**;
- если одно из константных выражений совпадает со значением выражения, то управление передаётся оператору, помеченному соответствующим ключевым словом **case**;
- если ни одно из константных выражений не равно выражению, то управление передаётся оператору, помеченному ключевым словом **default**, а в случае его отсутствия управление передаётся на следующий после **switch** оператор.

Укажем интересную особенность использования оператора **switch**: конструкция со словом **default** может быть не последней в теле оператора **switch**. Ключевые слова **case** и **default** в теле оператора **switch** существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора **switch**. Все операторы между начальным оператором и концом тела выполняются вне зависимости от ключевых слов, если только какой-то из операторов не передаст управления из тела оператора **switch**. Таким образом, программист должен сам позаботиться о выходе из **case**, если это необходимо. Чаще всего для этого используется оператор **break**.

Для того чтобы выполнить одни и те же действия для различных значений выражения, можно пометить один и тот же оператор несколькими ключевыми словами **case**.

Пример:

```
int i=2;
switch (i)
{
    case 1: i += 2;
    case 2: i *= 3;
    case 0: i /= 2;
    case 4: i -= 5;
    default: ;
}
```

Выполнение оператора **switch** начинается с оператора, помеченного **case 2**. Таким образом, переменная *i* получает значение «6»; далее выполняется оператор, помеченный ключевым словом **case 0**, а затем **case 4**. Переменная *i* при этом примет значение «3», а затем значение «-2». Оператор, помеченный ключевым словом **default**, не изменяет значения переменной.

Рассмотрим ранее приведённый пример, в котором иллюстрировалось использование вложенных операторов **if**, переписанный теперь с использованием оператора **switch**.

```
char ZNAC;
int x,y,z;
switch (ZNAC)
{
    case '+': x = y + z; break;
    case '-': x = y - z; break;
    case '*': x = y * z; break;
    case '/': x = u / z; break;
    default : ;
}
```

Использование оператора **break** позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора **switch** путём передачи управления оператору, следующему за **switch**.

Отметим, что в теле оператора **switch** можно использовать вложенные операторы **switch**, при этом в ключевых словах **case** можно использовать одинаковые константные выражения.

Пример:

```
switch (a)
{
    case 1: b=c; break;
    case 2:
        switch (d)
        {
            case 0: f=s; break;
            case 1: f=9; break;
            case 2: f-=9; break;
        }
    case 3: b-=c; break;
}
```

Оператор **break**

Оператор **break** обеспечивает прекращение выполнения самого внутреннего из объединяющих его операторов **switch**, **for**, **while** и **do-while**. После выполнения оператора **break** управление передаётся оператору, следующему за прерванным.

Оператор **for**

Оператор **for** – это наиболее общий способ организации цикла. Он имеет следующий формат:

for (выражение 1; выражение 2; выражение 3) {тело}

Выражение 1 обычно используется для установления начального значения переменных, управляющих циклом. Выражение 2 определяет условие, при котором тело цикла будет выполняться. Выражение 3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора **for**:

- 1) вычисляется выражение 1;
- 2) вычисляется выражение 2;
- 3) если значения выражения 2 отлично от нуля (истина), то выполняется тело цикла, вычисляется выражение 3 и осуществляется переход

к пункту 2. Если выражение 2 равно нулю (ложь), то управление передаётся оператору, следующему за оператором **for**.

Существенно то, что проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

Пример:

```
int main()
{ int i,b;
  for (i=1; i<10; i++)
    b=i*i;
  return 0;
}
```

В этом примере вычисляются квадраты чисел от 1 до 9.

Некоторые варианты использования оператора **for** повышают его гибкость за счет возможности использования нескольких переменных, управляющих циклом.

Пример:

```
int main()
{ int top, bot;
  char string[100], temp;
  for ( top=0, bot=100 ; top < bot ; top++, bot--)
  { temp=string[top];
    string[bot]=temp;
  }
  return 0;
}
```

В этом примере, реализующем запись строки символов в обратном порядке, для управления циклом используются две переменные: **top** и **bot**. Отметим, что на месте выражения 1 и выражения 3 здесь используются несколько выражений, записанных через запятую и выполняемых последовательно.

Другим вариантом использования оператора **for** является бесконечный цикл. Для организации такого цикла можно использовать пустое условное выражение, а для выхода из цикла обычно используют дополнительное условие и оператор **break**.

Пример:

```
for (;;)
{ ...
  ... break;
  ...
}
```

Так как согласно синтаксису языка Си оператор может быть пустым, тело оператора **for** также может быть пустым. Такая форма оператора используется для организации поиска.

Пример:

```
for (i=0; t[i]<10 ; i++);
```

В этом примере переменная цикла *i* принимает значение номера первого элемента массива *t*, значение которого больше чем 10.

Оператор **while**

Оператор цикла **while** называется оператором цикла с предусловием и имеет следующий формат:

while (выражение) тело;

В качестве выражения допускается использовать любое выражение языка Си, а в качестве тела – любой оператор, в том числе пустой или составной. Схема выполнения оператора **while** следующая:

- 1) вычисляется выражение;
- 2) если выражение ложно, то выполнение оператора **while** заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора **while**;
- 3) процесс повторяется с пункта 1.

Оператор цикла вида

for (выражение 1; выражение 2; выражение 3) тело;

может быть заменён оператором **while** следующим образом:

```
выражение 1;
while (выражение 2)
{тело
  выражение 3;
}
```

При выполнении оператора **while** (так же как и **for**) вначале происходит проверка условия. Поэтому оператор **while** удобно использовать в ситуациях, когда тело оператора не всегда нужно выполнять.

Внутри операторов **for** и **while** можно использовать локальные переменные, которые должны быть объявлены с определением соответствующих типов.

Оператор **do-while**

Оператор цикла **do-while** называется оператором цикла с постусловием и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз. Формат оператора имеет следующий вид:

do тело **while** (выражение);

Схема выполнения оператора **do-while**:

- 1) выполняется тело цикла (которое может быть составным оператором);
- 2) вычисляется выражение;
- 3) если выражение ложно, то выполнение оператора **do-while** заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с пункта 1.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор **break**.

Операторы **while** и **do-while** могут быть вложенными.

Пример:

```
int i,j,k;
...
i=0; j=0; k=0;
do { i++;
    j--;
    while (a[k] < i) k++;
}
while (i<30 && j<-30);
```

Оператор **continue**

Оператор **continue**, как и оператор **break**, используется только внутри операторов цикла, но в этом случае выполнение программы продолжается не с оператора, следующего за прерванным оператором, а с начала прерванного оператора. Формат оператора следующий:

continue;

Пример:

```
int main()
{   int a,b;
    for (a=1,b=0; a<100; b+=a,a++)
    {   if (b%2) continue;
        ... /* обработка чётных сумм */
    }
    return 0;
}
```

Когда сумма чисел от единицы до **a** становится нечётной, оператор **continue** передаёт управление на очередную итерацию цикла **for**, не выполняя операторы обработки чётных сумм.

Оператор **continue**, как и оператор **break**, прерывает самый внутренний из объёмлющих его циклов.

Оператор **return**

Оператор **return** завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию в точку, непосредственно следующую за вызовом. Функция **main** передаёт управление операционной системе. Формат оператора:

return [выражение];

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие необязательно.

Если в какой-либо функции отсутствует оператор **return**, то передача управления в вызывающую функцию происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое

значение не определено. Если функция не должна иметь возвращаемого значения, то её нужно объявлять с типом `void`.

Таким образом, использование оператора `return` необходимо либо для немедленного выхода из функции, либо для передачи возвращаемого значения.

Пример:

```
int sum (int a, int b)
{ return (a+b); }
```

Функция `sum` имеет два формальных параметра `a` и `b` типа `int` и возвращает значение типа `int`, о чём говорит описатель, стоящий перед именем функции. Возвращаемое оператором `return` значение равно сумме фактических параметров.

Пример:

```
void prov (int a, double b)
{ double c;
  if (a<3) return;
  else if (b>10) return;
  else { c=a+b;
        if ((2*c-b)==11) return;
      }
}
```

В этом примере оператор `return` используется для выхода из функции в случае выполнения одного из проверяемых условий.

Оператор `goto`

Использование оператора безусловного перехода `goto` в практике программирования на языке Си настоятельно не рекомендуется, так как он затрудняет понимание программ и возможность их модификаций.

Формат этого оператора следующий:

```
goto имя метки;

...

имя метки: оператор;
```

Оператор **goto** передаёт управление оператору, помеченному меткой «имя метки». Помеченный оператор должен находиться в той же функции, что и оператор **goto**, а используемая метка должна быть уникальной, т. е. одно имя метки не может быть использовано для разных операторов программы. «Имя метки» – это идентификатор.

Любой оператор в составном операторе может иметь свою метку. Используя оператор **goto**, можно передавать управление внутрь составного оператора. Но нужно быть осторожным при входе в составной оператор, содержащий объявления переменных с инициализацией, так как объявления располагаются перед выполняемыми операторами, а значения объявленных переменных при таком переходе будут не определены.

6. ФУНКЦИИ

6.1. НАЗНАЧЕНИЕ ФУНКЦИЙ

Функция – это 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 «вовне». Есть как минимум одна функция – `main`, которая так и называется – главная, а все остальные делятся на две категории: те, которые вы создаёте сами, и библиотечные.

Основная идея использования функций, проиллюстрированная на рис. 8, 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.

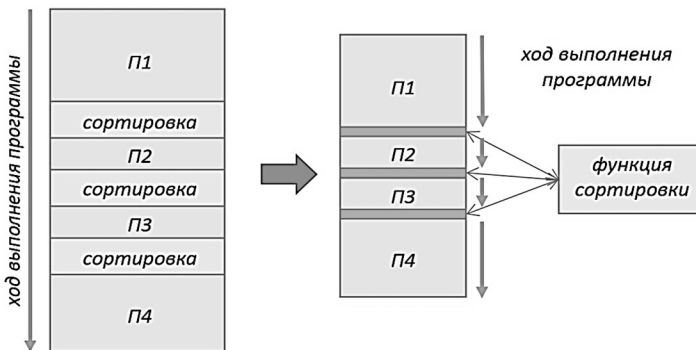


Рис. 8. Обособление повторяющегося фрагмента кода в виде функции

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

- в объёме используемой памяти: во время выполнения программы код функции находится в памяти всего лишь в единственном экземпляре;
- поддержке кода: если в функции нашли ошибку, то её нужно исправить всего в одном месте;
- многократном использовании: если код не является сугубо специфическим для данной программы и решает распространённую задачу, то его можно использовать и в других своих программах или выложить в сеть, чтобы им могли воспользоваться другие программисты;
- возможности декомпозиции сложной задачи.

С процессом декомпозиции связаны два основных метода разработки программного обеспечения: сверху вниз и снизу вверх (рис. 9). При проектировании сверху вниз сначала пишется главная функция программы, где сложные фрагменты кода заменены на вызовы пустых функций-заглушек. После этого расписывают каждую из этих функций, прибегая к тому же методу: если в коде функции встречается сравнительно сложный фрагмент логики, который легко обособляется в виде функции, вместо него вставляют вызов ещё одной заглушки, определение которой оставляют на потом, и т. д.

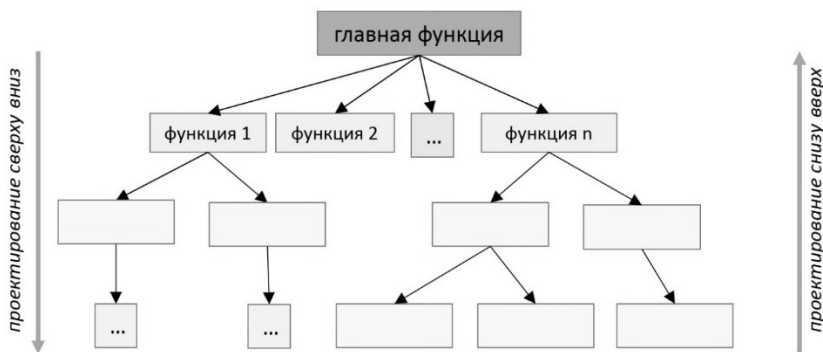


Рис. 9. Проектирование кода

При написании кода снизу вверх поступают наоборот: сначала пишут наиболее простые «примитивы», которые понадобятся в дальнейшем. Затем на их основе собирают уже более сложные программные модули, на основе которых – ещё более сложные, и так далее, пока не получится так, что главная функция программы собирается из уже имеющихся модулей как конструктор.

6.2. ОБЪЯВЛЕНИЕ, ОПРЕДЕЛЕНИЕ И ВЫЗОВ ФУНКЦИИ

Для вызова функции нужно написать её имя, а затем круглые скобки, внутри которых должны быть перечислены через запятую передаваемые в неё 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 нужно передавать функции, зависит от неё самой. Будучи выполненной, функция возвращает какое-то значение (единственное). Она 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 это следующим образом: сначала поочерёдно вызываются все функции, затем возвращённые ими значения подставляются в выражение на место соответствующих вызовов, и после этого всё 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 определённом 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`; и 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
}
```

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

Пример 1

Если, например, требуется найти максимум из двух целых чисел, то решение может быть таким:

```
// 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, в то время как объявление используется тогда, когда это необходимо. В частности, в примере 1 можно было обойтись и без отдельного объявления. Тогда зачем нужно разбивать функцию на объявление и определение? Дело в том, что, как и в случае с переменными, прежде чем использовать функцию, мы должны её объявить, чтобы компилятор «знал», как её вызывать, т. е. 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: всё, что мы используем сейчас, уже должно было быть объявлено ранее. Ничто не мешает нам определять

функции именно в том порядке, в котором они используются, однако иногда это невозможно.

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. Независимо, какую функцию мы определим первой, все равно оставшаяся из двух будет не определена в момент её вызова. Для решения этой проблемы 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 её объявление (собственно говоря, объявление – это «усечённое» определение), поэтому функцию A объявлять не требуется.

Пример 2. 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 её 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 компилятору этого будет достаточно. Действительно, функция **max** уже скомпилирована и её код будет связан с нашим кодом на следующем шаге сборки приложения – линковке, а сейчас компилятору важно лишь проверить соблюдение синтаксической корректности вызова функции.

Как уже говорилось, 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.

6.3. АРГУМЕНТЫ, ПАРАМЕТРЫ И ВОЗВРАТ ЗНАЧЕНИЯ

Функцию можно рассматривать как чёрный ящик с несколькими входами и одним выходом. Получив на вход какие-то данные, она выполняет свою работу и возвращает определённое значение.

Входных параметров может быть сколько угодно (или может даже и не быть), а возвращаемое значение только одно (его тоже может не быть, хотя такие функции – редкость).

Рассмотрим передачу параметров на примере функции **add**.

Необходимо написать функцию для сложения двух чисел. Решение может быть таким:

```
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 её в строке 3 и передаём 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` передаём 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? Точно 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`.

Ещё 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 данной функции, и этим переменным присваиваются значения аргументов, переданных в функцию в момент вызова (см. рис. 9). 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. Если говорить более точно, то с переменными связано такое понятие, как область видимости. Каждая функция задаёт свою область видимости, и все переменные, объявленные внутри некоторой функции, существуют только внутри неё.

Рассмотрим возврат результата из функции. Функция заканчивает свою работу в двух случаях:

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

Синтаксис оператора `return` следующий:

`return выражение;`

Выполнение этого оператора завершает работу функции и возвращает вызывавшему её коду значение выражения. `Void`-функцию можно завершить с помощью `return` без операнда:

`return;`

Оператор `return` можно образно сравнить с кнопкой катапультирования: нажав её, мы тут же «вылетаем» из функции. Одна из распространённых ошибок 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 в строке 3, которая никогда не будет выполнена. Это свойство `return` можно использовать также для того, чтобы вместо кода

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

писать код

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

Вызов любой не `void`-функции можно использовать внутри выражения и передавать её 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;
}
```

6.4. 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, т. е. после вызова `sortArgs(a,b)` значения исходных переменных `a` и `b` должны поменяться местами, если первое больше второго. Код, решающий эту задачу, 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. }
```

Код работает, так как мы объявили параметры функции `sortArgs` как указатели на целочисленные переменные (в строках 4 и 18), а в строке 12 передаём в качестве аргументов адреса переменных `a` и `b`, объявленных в строке 8. В строках 20–21 используется оператор разыменования – обращения к ячейкам памяти по указателю.

7. МАССИВЫ

7.1. МАССИВ КАК АГРЕГАТНЫЙ ТИП ДАННЫХ

Довольно часто требуется обрабатывать большие объёмы 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 ещё и так называемые агрегатные (сложные, составные) типы данных, одним из которых 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.

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

`<тип> <имя>[<размер>];`

Здесь квадратные скобки являются частью синтаксиса.

С оператором объявления мы уже знакомы – он использовался для объявления переменных. Разница в синтаксисе заключается в том, что

после имени в квадратных скобках записывается размер создаваемого массива. Например:

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

Можно смешивать в одном операторе объявление обычных переменных и массивов:

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

Для того 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, потому что это и есть обычные переменные, просто они не имеют своих собственных имён, а имеют лишь общее имя массива и свой порядковый номер.

Так же как и в случае указателей, не следует 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, например, от языка Pascal) нумерация элементов в массиве начинается с нуля, поэтому в примере выше первый элемент массива `mas` будет `mas[0]`, последний – `mas[13]`, а `mas[2]` обращается к третьему по порядку элементу. Как правило, номера элементов массива называются индексами, а процесс доступа к ним – индексированием.

7.2. РАЗМЕЩЕНИЕ В ПАМЯТИ И ИНИЦИАЛИЗАЦИЯ

Массивы, которые были рассмотрены выше, называются статическими. Это означает, что их размер задаётся 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 байтов, а `int mas[3]` – это три подряд идущие целочисленные переменные, каждая из которых занимает по 4 байта, т. е. всего такой массив занимает 12 байтов.

7.3. ПЕРЕДАЧА МАССИВОВ В ФУНКЦИЮ

Массивы передаются в функцию через указатель на его первый элемент. В языке Си массив как раз и является указателем на его первый элемент.

Рассмотрим пример. Пусть необходимо написать функцию, которая принимает на вход массив из 10 случайных чисел и печатает его на экран. Решение может быть таким:

```
void printMas(int* mas, int num); // прототип функции printMas  
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");  
}
```

Здесь функция `rand()` генерирует массив случайных чисел.

Обратим внимание на объявление функции `printMas`. Её первый параметр имеет тип указателя на `int`, через него функция получит доступ к массиву. Поскольку никакой другой информации, кроме адреса первого элемента, указатель в себе не несёт, то размер массива мы должны передать отдельно – через второй параметр. Сразу возникает вопрос: а как функция различает, что ей 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 передаём в качестве первого аргумента сам массив – `mas` (без каких-либо квадратных скобок).

7.4. МНОГОМЕРНЫЕ МАССИВЫ

До сих пор речь шла лишь об одномерных массивах, т. е. таких, которые образно можно представить в виде линейной последовательности ячеек, пронумерованных от нуля до N .

Предположим, необходимо в программе создать таблицу умножения. Для этих целей, конечно, можно 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. Однако это неудобно, потому что каждый раз придётся вычислять номер нужной ячейки по двум индексам. В языке Си есть возможность явным образом создавать двумерные массивы (их также 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, нужно только вместо одной пары квадратных скобок писать две и соответственно указывать значения двух координат. Нумерация в обоих измерениях начинается с нуля, таким образом, `mas[2][5]` – это значение, находящееся на пересечении третьей строки и шестого столбца (по порядку).

Точно так же можно объявить и использовать трёхмерный, четырёхмерный и массив любой другой размерности:

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

Всего в массиве, объявленном выше, будет 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)? Матрица записывается в память строка за строкой: сначала первая строка, потом вторая, третья и т. д. Например, `char M[3][4]`; размещается в памяти, как показано на рис. 10.

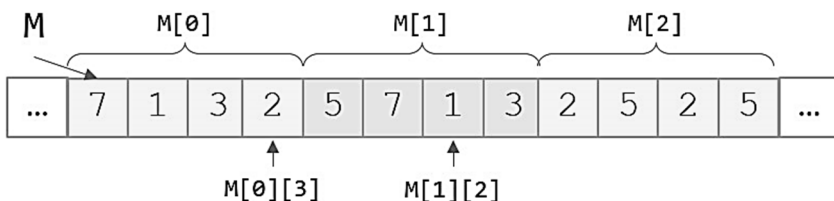


Рис. 10. Матрица `char M[3][4]` в памяти

Трёхмерный массив `char M[3][3][2]`; выглядит в памяти как показано на рис. 11.

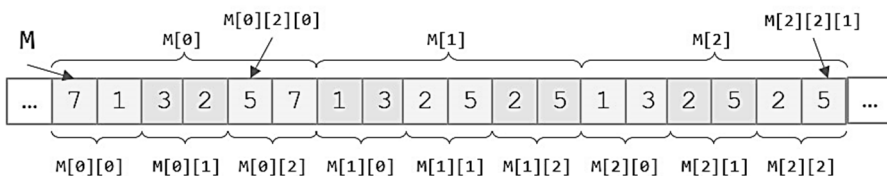


Рис. 11. Трёхмерный массив `char M[3][3][2]` в памяти

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]);
```

Аналогично можно передавать и одномерные массивы:

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

Хотя до этого 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 её второй размерности. Если исходный массив был размером 10×10 , то `mas[3][3]` обращается к 34-й ячейке, а если 10×5 , то к 19-й ячейке.

Рассмотрим пример с таблицей умножения. Необходимо создать матрицу 10×10 , заполнить её таблицей умножения, а затем вывести на экран. Решение может быть таким:

```
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("%d", mas[i][j]);
        printf("\n");
    }
    return 0;
}
```

В процессе выполнения программы мы получаем таблицу умножения от 1 до 10 с выводом результатов её работы на экран.

8. СЛОЖНЫЕ ТИПЫ ДАННЫХ В ЯЗЫКЕ СИ. СТРУКТУРЫ

8.1. ПОНЯТИЕ СТРУКТУРЫ

Структура – это одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем (в некоторых языках, в частности в Pascal, структуры называются записями). Структуры помогают с организацией сложных данных, особенно в больших программах, поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое.

Традиционный пример структуры – строка платёжной ведомости. Она содержит такие сведения о служащем, как его полное имя, адрес, номер карточки социального страхования, зарплата и др. Некоторые из этих характеристик сами могут быть структурами: например, полное имя состоит из нескольких компонентов (фамилии, имени и отчества); аналогично адрес и даже зарплата. Другой пример (более типичный для языка Си) – из области графики: точка есть пара координат, прямоугольник есть пара точек и т. д.

Главные изменения, внесённые стандартом ANSI в отношении структур, – это введение для них операции присваивания. Структуры могут копироваться, над ними могут выполняться операции присваивания, их можно передавать функциям в качестве аргументов, а функции могут возвращать их в качестве результатов. В большинстве компиляторов уже давно были реализованы эти возможности, но затем они стали точно оговорены стандартом. Для автоматических структур и массивов теперь также допускается инициализация.

8.2. ОСНОВНЫЕ СВЕДЕНИЯ О СТРУКТУРАХ

Сконструируем графическую структуру. В качестве основного объекта выступает точка с координатами $x = 4$ и $y = 3$ целочисленного типа.

Указанные две компоненты можно поместить в структуру, объявленную, например, следующим образом:

```
struct point {int x; int y};
```

Объявление структуры начинается с ключевого слова **struct** и содержит список объявлений, заключённый в фигурные скобки. За словом **struct** может следовать имя, называемое тегом. Тег даёт название структуре данного вида и далее может служить кратким обозначением той части объявления, которая заключена в фигурные скобки. Перечисленные в структуре переменные называются элементами. Имена элементов и тегов без каких-либо коллизий могут совпадать с именами обычных переменных (т. е. не элементов), так как они всегда различимы по контексту. Более того, одни и те же имена элементов могут встречаться в разных структурах, хотя если следовать хорошему стилю программирования, то лучше одинаковые имена давать только близким по смыслу объектам.

Объявление структуры определяет тип. За правой фигурной скобкой, закрывающей список элементов, могут следовать переменные — точно так же, как они могут быть указаны после названия любого базового типа. Таким образом, выражение

```
struct {...} x, y, z;
```

с точки зрения синтаксиса аналогично выражению

```
int x, y, z;
```

в том смысле, что и то и другое объявляет x , y и z переменными указанного типа. Поэтому и то и другое приведёт к выделению памяти соответствующего размера.

Объявление структуры, не содержащей списка переменных, не резервирует память, а просто описывает шаблон (или образец) структуры. Однако если структура имеет тег, то этим тегом можно пользоваться далее при определении структурных объектов. Например, с помощью заданного выше описания структуры **point** строка

```
struct point pt;
```

определяет структурную переменную `pt` типа `struct point`. Структурную переменную при определении можно инициализировать, формируя список инициализаторов её элементов в виде константных выражений:

```
struct point maxpt = {320, 200};
```

Инициализировать автоматические структуры можно также присваиванием или обращением к функции, возвращающей структуру соответствующего типа.

Доступ к отдельному элементу структуры осуществляется посредством конструкции вида

`имя_структуры.элемент`

Оператор доступа к элементу структуры соединяет имя структуры и имя элемента. Чтобы напечатать, например, координаты точки `pt`, годится следующее обращение к `printf`:

```
printf("%d, %d", pt.x, pt.y);
```

8.3. СТРУКТУРЫ И ФУНКЦИИ

Возможные операции над структурами – это их копирование, присваивание, взятие адреса с помощью `&` и осуществление доступа к её элементам. Копирование и присваивание также включают в себя передачу функциям аргументов и возврат ими значений. Структуры нельзя сравнивать. Инициализировать структуру можно списком константных значений её элементов; автоматическую структуру также можно инициализировать присваиванием. Чтобы лучше познакомиться со структурами, напишем несколько функций, манипулирующих точками и прямоугольниками. Возникает вопрос: а как передавать функциям названные объекты? Существует, по крайней мере, три подхода: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый подход имеет свои плюсы и минусы.

Первая функция, `makepoint`, получает два целых значения и возвращает структуру `point`:

```
/* makepoint: формирует точку по компонентам x и y */  
struct point makepoint(int x, int y) {  
    struct point temp;
```

```
temp.x = x;
temp.y = y;
return temp;
}
```

Заметим: никакого конфликта между именем аргумента и именем элемента структуры не возникает; более того, сходство подчёркивает родство обозначаемых им объектов. Теперь с помощью функции **makepoint** можно выполнять динамическую инициализацию любой структуры или формировать структурные аргументы для той или иной функции:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2, (screen.pt1.y
+ screen.pt2.y)/2);
```

Следующий шаг состоит в определении ряда функций, реализующих различные операции над точками. В качестве примера рассмотрим следующую функцию:

```
/* addpoint: сложение двух точек */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Здесь оба аргумента и возвращаемое значение — структуры. Мы увеличиваем компоненты прямо в **p1** и не используем для этого временную переменную, чтобы подчеркнуть, что структурные параметры передаются по значению, так же как и любые другие.

В качестве другого примера приведём функцию **ptinrect**, которая проверяет, находится ли точка внутри прямоугольника, относительно которого мы принимаем соглашение, что в него входят его левая и нижняя стороны, но не входят верхняя и правая.

```

/* ptinrect: возвращает 1, если p в r, и 0 в противном случае */
int ptinrect(struct point p, struct rect r) {
    return p.x >= r.pt1.x && p.x < r.pt2.x && p.y >= r.pt1.y && p.y <
r.pt2.y;
}

```

Здесь предполагается, что прямоугольник представлен в стандартном виде, т. е. координаты точки **pt1** меньше соответствующих координат точки **pt2**.

Следующая функция гарантирует получение прямоугольника в каноническом виде:

```

#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
/* canonrect: канонизация координат прямоугольника */
struct rect canonrect(struct rect r)
{
    struct rect temp;
    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}

```

Если функции передаётся большая структура, то вместо того чтобы копировать её целиком, эффективнее передать указатель на неё. Указатели на структуры ничем не отличаются от указателей на обычные переменные. Так, объявление

```
struct point *pp;
```

сообщает, что **pp** – это указатель на структуру типа **struct point**. Если **pp** указывает на структуру **point**, то ***pp** – это сама структура, а **(*pp).x** и **(*pp).y** – её элементы. Используя указатель **pp**, мы могли бы написать следующее:

```

struct point origin, *pp;
pp = &origin;
printf("origin: (%d,%d)\n", (*pp).x, (*pp).y);

```

Скобки в `(*pp).x` необходимы, поскольку приоритет оператора «.» выше, чем приоритет оператора «*». Выражение `*pp.x` будет интерпретировано как `*(pp.x)`, что неверно, поскольку `pp.x` не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к их элементам была придумана ещё одна форма записи, более короткая. Если `p` – указатель на структуру, то

`p->элемент_структуры;`

есть её отдельный элемент. Оператор «`->`» состоит из знака «`->`», за которым сразу следует знак «`>`».) Поэтому `printf` можно переписать в виде

`printf("origin: (%d,%d)\n", pp->x, pp->y);`

Операторы «.» и «`->`» выполняются слева направо. Таким образом, при наличии объявления

`struct rect r, *rp = &r;`

следующие четыре выражения будут эквивалентны:

`r.pt1.x; rp->pt1.x; (r.pt1).x; (rp->pt1).x;`

Операторы доступа к элементам структуры «.» и «`->`» вместе с операторами вызова функции `()` и индексации массива `[]` занимают самое высокое положение в иерархии приоритетов и выполняются раньше любых других операторов. Например, если задано объявление

`struct {int len; char *str;} *p;`

то оператор

`++p->len;`

увеличит на единицу значение элемента структуры `len`, а не указатель `p`, поскольку в этом выражении неявно присутствуют скобки:

`++(p->len);`

Чтобы изменить порядок выполнения операций, нужны явные скобки. Так, в `(++p)->len`, прежде чем взять значение `len`, программа прирастит указатель `p`. В `(p++)->len` указатель `p` увеличится после того, как будет взято значение `len` (в последнем случае скобки необязательны).

По тем же правилам оператор `*p->str`; обозначает содержимое объекта, на который указывает `str`, а оператор `*p->str++`; прирастит указатель `str` после получения значения объекта, на который он указывал (как и в выражении `*s++`).

Оператор `(*p->str)++`; увеличит значение объекта, на который указывает `str`, а оператор `*p++->str`; увеличит `p` после получения того, на что указывает `str`.

8.4. МАССИВЫ СТРУКТУР

Рассмотрим программу, определяющую число вхождений каждого ключевого слова в текст Си-программы. Нам нужно уметь хранить ключевые слова в виде массива строк и счётчики ключевых слов в виде массива целых. Один из возможных вариантов – это иметь два параллельных массива:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Однако именно тот факт, что они параллельны, подсказывает нам другую организацию хранения – через массив структур. Каждое ключевое слово можно описать парой характеристик:

```
char *word;
int count;
```

Такие пары составляют массив. Объявление

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

объявляет структуру типа `key` и определяет массив `keytab`, каждый элемент которого является структурой этого типа и которому где-то будет выделена память. Это же можно записать и по-другому:

```
struct key {
    char *word;
    int count;
};
struct key keytab[NKEYS];
```


Так как массив **keytab** содержит постоянный набор имён, его легче всего сделать внешним массивом и инициализировать один раз в момент определения. Инициализация структур аналогична ранее демонстрировавшимся инициализациям, т. е. за определением следует список инициализаторов, заключённый в фигурные скобки:

```
struct key {
char *word;
int count;
} keytab[] = {
"auto", 0,
"break", 0,
"case", 0,
"char", 0,
"const", 0,
"continue", 0,
"default", 0,
/* ... */
"unsigned", 0,
"void", 0,
"volatile", 0,
"while", 0
};
```

Инициализаторы задаются парами, чтобы соответствовать конфигурации структуры. Строго говоря, пару инициализаторов для каждой отдельной структуры следовало бы заключить в фигурные скобки, как, например, во фрагменте кода

```
{"auto", 0},
{"break", 0},
{"case", 0},
...
```

Однако если инициализаторы – простые константы или строки символов, и все они имеются в наличии, то во внутренних скобках нет необходимости. Число элементов массива **keytab** будет вычислено по количеству инициализаторов, поскольку они представлены полностью, а внутри квадратных скобок [] ничего не задано.

Программа подсчёта ключевых слов начинается с определения `keytab`. Программа `main` читает ввод, многократно обращаясь к функции `getword` и получая на каждом её вызове очередное слово. Каждое слово ищется в `keytab`. Для этого используется функция бинарного поиска, которую мы написали в разделе 3. Список ключевых слов должен быть упорядочен по алфавиту:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
int binsearch(char *, struct key *, int);
/* подсчет ключевых слов Си */
main()
{
    int n;
    char word[MAXWORD];
    while(getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n", keytab[n].count, keytab[n].word);
    return 0;
}
/* binsearch: найти слово в tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;
    low = 0;
    high = n-1;
    while (low <= high) {
        mid = (low + high)/2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
```

```

high = mid - 1;
else if (cond > 0)
low = mid + 1;
else
return mid;
}
return -1;
}

```

Чуть позже мы рассмотрим функцию `getword`, а сейчас нам достаточно знать, что при каждом её вызове получается очередное слово, которое запоминается в массиве, заданном первым аргументом.

NKEYS – это количество ключевых слов в массиве `keytab`. Хотя мы могли бы подсчитать число таких слов вручную, гораздо легче и безопаснее сделать это с помощью машины, особенно если список ключевых слов может быть изменён. Одно из возможных решений – поместить в конец списка инициализаторов пустой указатель (`NULL`) и затем перебирать в цикле элементы `keytab`, пока не встретится концевой элемент.

Возможно и более простое решение. Поскольку размер массива полностью определён во время компиляции и равен произведению количества элементов массива на размер его отдельного элемента, то число элементов массива можно вычислить как `размер keytab / размер struct key`.

В языке Си имеется унарный оператор `sizeof`, который работает во время компиляции. Его можно применять для вычисления размера любого объекта.

Выражения `sizeof объект` и `sizeof (имя типа)` выдают целые значения, равные размеру указанного объекта или типа в байтах. Строго говоря, `sizeof` выдаёт беззнаковое целое, тип которого `size_t` определён в заголовочном файле `stddef.h`. Что касается объекта, то это может быть переменная, массив или структура. В качестве имени типа может выступать имя базового типа (`int`, `double` ...) или имя производного типа, например структуры или указателя.

В нашем случае, чтобы вычислить количество ключевых слов, размер массива надо поделить на размер одного элемента. Указанное вычисление используется в инструкции `#define` для установки значения **NKEYS**:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Этот же результат можно получить другим способом – поделить размер массива на размер какого-то его конкретного элемента:

```
#define NKEYS (sizeof keytab / sizeof keytab[0])
```

Преимущество такого рода записей состоит в том, что их не надо корректировать при изменении типа.

Поскольку препроцессор не обращает внимания на имена типов, оператор `sizeof` нельзя применять в `#if`. Но в `#define` выражение препроцессором не вычисляется, так что предложенная нами запись допустима.

Вернёмся к функции `getword`. Мы написали `getword` в несколько более общем виде, чем требуется для нашей программы, но она от этого не стала заметно сложнее. Функция `getword` берёт из входного потока следующее «слово». Под словом понимается цепочка букв и цифр, начинающаяся с буквы, или отдельный символ, отличный от символа-разделителя. В случае конца файла функция возвращает `EOF`, в остальных случаях её значением является код первого символа слова или сам символ, если это не буква.

```
/* getword: принимает следующее слово или символ из ввода */
int getword (char *word, int lim) {
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for (; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
```

```

break;
}
*w = '\0';
return word[0];
}

```

Функция `getword` обращается к `getch` и `ungetch`. По завершении набора букв-цифр оказывается, что `getword` взяла лишний символ. Обращение к `ungetch` позволяет вернуть его назад во входной поток. В `getword` используются также процедуры: `isspace` – для пропуска символов-разделителей, `isalpha` – для идентификации букв и `isalnum` – для распознавания букв-цифр. Все они описаны в стандартном заголовочном файле `ctype.h`.

8.5. УКАЗАТЕЛИ НА СТРУКТУРЫ

Для иллюстрации некоторых моментов, касающихся указателей на структуры и массивов структур, перепишем программу подсчёта ключевых слов, пользуясь для получения элементов массива указателями вместо индексов.

Внешнее объявление массива `keytab` остаётся без изменения, а функции `main` и `binsearch` нужно модифицировать:

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
struct key *binsearch(char *, struct key *, int);
/* подсчёт ключевых слов Си: версия с указателями */
main()
{
    char word[MAXWORD];
    struct key *p;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))

```

```

if ((p = binsearch(word, keytab, NKEYS)) != NULL)
p->count++;
for (p = keytab; p < keytab + NKEYS; p++)
if (p->count > 0)
printf("%4d %s\n", p->count, p->word);
return 0;
}
/* binsearch: найти слово word в tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
int cond;
struct key *low = &tab[0];
struct key *high = &tab[n];
struct key *mid;
while (low < high) {
mid = low + (high - low) / 2;
if ((cond = strcmp(word, mid->word)) < 0)
high = mid;
else if (cond > 0)
low = mid + 1;
else
return mid;
}
return NULL;
}

```

Некоторые детали этой программы требуют пояснений. Во-первых, описание функции `binsearch` должно отражать тот факт, что она возвращает указатель на `struct key`, а не целое, — это объявлено как в прототипе функции, так и в функции `binsearch`. Если `binsearch` находит слово, то она выдаёт указатель на него, в противном случае она возвращает `NULL`. Во-вторых, к элементам `keytab` доступ в нашей программе осуществляется через указатели. Это потребовало значительных изменений в `binsearch`. Инициализаторами для `low` и `high` теперь служат

указатели на начало и на место сразу после конца массива. Вычисление положения среднего элемента с помощью формулы

$$\text{mid} = (\text{low} + \text{high}) / 2 \text{ /* НЕВЕРНО */}$$

не годится, поскольку указатели нельзя складывать. Однако к ним можно применить операцию вычитания. Поскольку `high-low` есть число элементов, то присваивание

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

превратит `mid` в указатель на элемент, лежащий посередине между `low` и `high`. Самое важное при переходе на новый вариант программы – сделать так, чтобы не генерировались неправильные указатели и не было попыток обратиться за пределы массива. Проблема в том, что и `&tab[-1]`, и `&tab[n]` находятся вне границ массива. Первый адрес определённо неверен, нельзя также осуществить доступ и по второму адресу. По правилам языка, однако, гарантируется, что адрес ячейки памяти, следующей сразу за концом массива (т. е. `&tab[n]`), в арифметике с указателями воспринимается правильно.

В главной программе `main` мы написали

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Если `p` – это указатель на структуру, то при выполнении операций с `p` учитывается размер структуры. Поэтому `p++` увеличит `p` на такую величину, чтобы выйти на следующий структурный элемент массива, а проверка условия вовремя остановит цикл.

Не следует, однако, полагать, что размер структуры равен сумме размеров её элементов. Вследствие выравнивания объектов разной длины в структуре могут появляться безымянные «дыры». Например, если переменная типа `char` занимает один байт, а `int` – четыре байта, то для структуры

```
struct {  
    char c;  
    int i;  
};
```

может потребоваться восемь байт, а не пять. Оператор `sizeof` возвращает правильное значение.

Следует сказать несколько слов относительно формата программы. Функция может возвращать значение сложного типа – например, в нашем случае она возвращает указатель на структуру

```
struct key *binsearch(char *word, struct key *tab, int n)
```

«Высмотреть» здесь имя функции оказывается совсем не просто. В подобных случаях иногда пишут так:

```
struct key *  
binsearch(char *word, struct key *tab, int n)
```

Какой форме отдать предпочтение – дело вкуса. Выберите ту, которая вам нравится больше всего, и придерживайтесь её.

8.6. ВЫДЕЛЕНИЕ ПАМЯТИ ПОД СТРУКТУРЫ

Для выделения динамической памяти под структуры обычно используется функция `malloc()`.

Вопрос об объявлении типа таких функций, как `malloc`, является камнем преткновения в любом языке с жёсткой проверкой типов. В языке Си этот вопрос решается естественным образом: `malloc` объявляется как функция, которая возвращает указатель на тип `void`. Полученный указатель затем явно приводится к желаемому типу. Замечание о приведении типа величины, возвращаемой функцией `malloc`, нужно переписать. Пример корректен и работает, но совет является спорным в контексте стандартов ANSI/ISO 1988–1989 гг. На самом деле это не обязательно (при условии, что приведение `void*` к `ALMOSTANYTYPE*` выполняется автоматически) и, возможно, даже опасно, если `malloc` или её заместитель не может быть объявлен как функция, возвращающая `void*`. Явное приведение типа может скрыть случайную ошибку. В прежние времена (до появления стандарта ANSI) приведение считалось обязательным, что также справедливо и для C++. Описания `malloc` и связанных с ней функций находятся в стандартном заголовочном файле `stdlib.h`. Таким образом, функцию `malloc` можно записать следующим образом:


```
#include <stdlib.h>
/* talloc: создаёт tnode */
struct tnode *talloc(void) {
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

Функция **strdup** просто копирует строку, указанную в аргументе, в место, полученное с помощью **malloc**:

```
char *strdup(char *s) /* делает дубликат s */
{
    char *p;
    p = (char *) malloc(strlen(s)+1); /* +1 для '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

Функция **malloc** возвращает **NULL**, если свободного пространства нет; **strdup** возвращает это же значение, оставляя заботу о выходе из ошибочной ситуации вызывающей программе.

Память, полученную с помощью **malloc**, можно освободить для повторного использования, обратившись к функции **free**.

9. СИМВОЛЫ И СТРОКИ

9.1. СИМВОЛЫ И КОДИРОВКА ASCII

До сих пор мы работали преимущественно с числовыми данными, но язык Си позволяет также оперировать символами и строками.

Для начала рассмотрим понятие символа. Символ – это просто печатный знак, который можно «нарисовать» или «напечатать». Буква, цифра, знак пунктуации или математической операции и тому подобное – всё это примеры символов, которые постоянно используются при работе за компьютером. Однако мы знаем, что память компьютера состоит из битов, каждый из которых хранит либо ноль, либо единицу. Таблица, устанавливающая соответствие между символами и целыми числами, называется кодировкой. Самой распространённой и самой старой стандартной кодировкой является ASCII20, с помощью которой кодируются 256 различных символов: буквы латиницы, цифры, 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. Именно поэтому для такого важного объекта, как символ, был создан отдельный тип данных `char` (от англ. «символ»), который как раз и занимает в памяти ровно один байт.

В настоящее время программы оперируют гораздо бóльшим количеством символов: это буквы всевозможных национальных алфавитов, иероглифы, математические знаки, символы мёртвых языков и множество других. Очевидно, что перенумеровать их все числами от 0 до 255 невозможно, поэтому сейчас применяют более объёмные и «хитрые» кодировки, в которых под один символ требуется не один

байт, а несколько. Однако во всех этих кодировках первые 256 номеров всё равно отводятся под кодировку ASCII, а в языке Си тип данных `char` по-прежнему занимает ровно один байт.

9.2. РАБОТА С СИМВОЛЬНЫМИ ДАННЫМИ

Присвоить переменной типа `char` какой-либо символ можно одним из трёх способов.

Первый способ – 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 по причинам, которые будут описаны ниже. Рассмотрим ещё 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 представляет собой ASCII-код символа '0', а второе – это просто число ноль.

Второй способ: если мы знаем ASCII-код какого-либо символа, мы можем его записать прямо 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 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, это '\n', которая соответствует символу с кодом 10. Наиболее часто используемые последовательности представлены в табл. 4.

Т а б л и ц а 4

| Escape-последовательность | Код | Описание |
|---------------------------|-----|--|
| \b | 8 | Удаление предыдущего символа (backspace) |
| \n | 10 | Новая строка |
| \r | 13 | Возврат каретки |
| \t | 9 | Горизонтальная табуляция |
| \' | 39 | Одинарная кавычка |
| \" | 34 | Двойная кавычка |
| \\ | 92 | Обратная косая черта |

В частности, если мы захотим в символьной переменной сохранить символ одинарной кавычки и 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:

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

О символах полезно знать следующее.

Во-первых, язык Си интерпретирует символьные константы как целые числа (как коды, которые соответствуют этим константам). То есть мы можем, например, написать

```
// в с 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 даёт возможность определять, какой именно символ у нас в переменной – буква, цифра или что-то другое, а также даёт возможность менять регистр букв и легко конвертировать символы цифр '0', '1', '2', ... в числа 0, 1, 2 и обратно.

9.3. СТРОКИ

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: длина строки и размер массива символов – это разные вещи. Массив может состоять из 250 ячеек (байтов), 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. 12 изображено, как это будет выглядеть в памяти.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|-----|----|-----|-----|-----|-----|----|----|-----|-----|-----|-----|-----|----|----|----|
| str | 72 | 101 | 108 | 108 | 111 | 44 | 32 | 119 | 111 | 114 | 108 | 100 | 33 | 0 | |

Рис. 12. Строка «Hello, world!» в 15-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 их коды.

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.

Первый способ: 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 следует забывать про нулевой маркер в конце строки. Без него мы просто не сможем понять, где в массиве заканчивается строка. Количество ячеек массива, занимаемое строкой из пяти 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 разобьёт 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 обязательно учитывать.

Есть ещё 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.

Второй способ записать в массив строку – считать её с клавиатуры или из файла:

```
1. char str[250];  
2. scanf_s("%s", str, 250);  
3. gets_s(str, sizeof(str));
```

В строке использована уже знакомая нам функция `scanf_s`, однако с двумя нововведениями: во-первых, мы указали спецификатор `%s`, благодаря которому функция понимает, что её просят считать строку; во-вторых, использован числовой аргумент – максимальное количество символов, которое функция может считать.

Функция `scanf_s` считывает символы до первого пробела или конца строки (смотря что встретится раньше). В третьей строке мы воспользовались ещё 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 ей передали не числовую константу, а значение встроенного в язык Си оператора `sizeof`, который возвращает размер переменной (массива, типа) данных в байтах. Если количество считанных символов больше указанного максимального размера, то эти 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 нас не устраивает, мы можем в спецификаторе функции `scanf_s` указать ширину поля, тогда она будет считывать указанное количество символов (или меньше).

Наконец, третий способ работы со строковыми данными – использование библиотечных функций для работы со строками. К сожалению, это единственный способ, чтобы, например, скопировать значение одной строки в другую.

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 массив `myString` объявлен вне какой-либо функции. Это способ создания так называемых глобальных переменных, которые видны из любой функции, поэтому к ней нельзя обратиться из `main`. Мы могли бы точно так же обратиться к строке и из функции `startsWithDigit()`, но 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` приведёт к ошибке компиляции. В функцию, объявленную таким образом (с параметром `const char* str`), можно передавать строковые константы, потому что они имеют именно такой тип:

```
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 `str28`;

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`, то функция возвращает код ошибки 29. Если всё прошло успешно, то 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 берётся 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, то возвращает отрицательное значение, если больше – положительное. Если строки полностью совпадают, то функция возвращает 0;

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 `str` или `NULL`, если вхождений не найдено;

9) `char* strrchr(char *str, int ch);` – возвращает указатель на последнее вхождение символа `ch` в строку `str` 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.

10. ВВОД-ВЫВОД

10.1. БУФЕР ВВОДА

Операции ввода и вывода данных – неотъемлемая составляющая абсолютно любой программы. Программа либо получает информацию извне, либо выдаёт 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 ещё 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, откуда они уже считываются функцией `scanf_s`. Данные всегда дописываются в конец буфера. Рассмотрим этот процесс на примере. Пусть имеется 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. 13.

Здесь `readStart` – это стартовая позиция, с которой функция `scanf_s` начинает считывать из буфера данные, а `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 строки 3 программы вызывается функция `scanf_s`. Она анализирует спецификатор `%d`, понимает, что её просят считать целое число, и обращается в системный буфер. Поскольку он пуст (состояние 1 на рис. 13), 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>`.

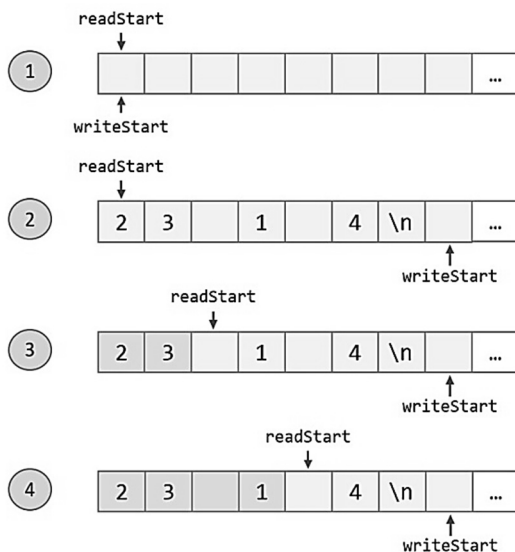


Рис. 13. Состояние буфера ввода

Предположим, что мы ввели строку 23 1 4. Очевидно, что никто не запрещает 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. 13.

Так как данные появились, программа возобновляет свою работу. 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 свою работу, 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 ещё остаются данные. Что с ними произойдёт? Ничего – они будут ждать своей очереди, когда в очередной раз вызванная функция `scanf_s` их считает. Что произойдёт, если в буфере ничего не будет? В этом случае `scanf_s` опять остановится и будет ждать, пока там что-нибудь появится. Внешне это выглядит как приостановка выполнения программы и появление мигающего курсора, предлагающего ввести данные. Текст, который мы вводим в консольном окне, не сразу попадает в буфер по той причине, что его ещё можно редактировать с помощью клавиши `<Backspace>`. Данные попадают в буфер в тот момент, когда мы нажимаем клавишу `<Enter>`, т. е. завершаем тем самым ввод одной строки. При этом важно помнить, что нажатие на `<Enter>` не только заносит набранные данные в буфер, но ещё и добавляет символ переноса строки в этот буфер (символ `\n` на рис. 13).

Отметим, что у каждой выполняющейся программы есть свой системный буфер чтения. Поэтому когда наша программа завершит своё выполнение на строке 6, то вместе с ней своё существование и прекратит буфер чтения.

10.2. ФОРМАТИРОВАННЫЙ ВВОД-ВЫВОД

Рассмотрим теперь подробнее синтаксис самих функций.

Функции `printf` и `scanf_s` называются функциями форматированного ввода-вывода, этим объясняется буква «f» (formatted) в их названии. Эти функции не просто позволяют вводить и выводить информацию, но и осуществлять при этом её форматирование.

Рассмотрим синтаксис форматированного вывода (рис. 14).

```
int a = 1, b = 2;
printf("Sum of numbers %d and %d is: %d + %d = %d\n", a, b, a, b, a+b);
```

Рис. 14. Работа функции printf

Вывод на экран результатов работы кода, изображённого на рис. 14, будет выглядеть следующим образом:

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

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

Помимо указания на место, куда надо вставить значение, спецификатор выполняет ещё 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 здесь означают, что поле является опциональным.

<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> – это 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).

Рассмотрим функцию форматированного ввода. Её синтаксис выглядит так же, как и для `printf`:

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

Первое поле – это форматная строка, указывающая, что и в каком формате надо считывать. Далее идут адреса переменных, в которые нужно записать считанные значения. Функция возвращает 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-приёмники 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. Если мы ввели число с большим количеством цифр, то оставшиеся знаки останутся лежать в буфере. Например:

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

Если мы введём на клавиатуре число 573, то в `n` попадёт только 5, а 7 и 3 останутся в буфере, решение становится исключительно простым:

```
int a, b, c, d, e, f;  
scanf_s("%1d%1d%1d%1d%1d%1d", &a, &b, &c, &d, &e, &f);  
if (a + b + c == d + e + f){  
    // билет счастливый  
}
```

Отсутствие пробелов в форматной строке совершенно непринципиально. Как уже говорилось выше, `scanf_s` их просто игнорирует, поэтому можно было 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 (кроме пробелов). Это в целом правильно, так как функция `scanf_s` ожидает команды-спецификаторы, описывающие, что ей нужно считывать. Однако если мы уверены в формате вводимой информации, то можем воспользоваться

ещё одной особенностью форматированного ввода – возможностью писать в форматной строке `scanf_s` произвольные символы. В этом случае, если вводимая строка выглядит соответствующим образом, `scanf_s` будет игнорировать эти символы во входной строке. Например, с помощью следующего кода мы можем ввести дату, разбив её сразу же на три поля (год, месяц, день):

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

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

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

10.3. РАБОТА С ФАЙЛАМИ

Рассмотрим теперь, как работать с файлами, а не с консолью. Это делается очень просто. Сначала создаём указатель:

```
FILE *pfile;
```

`FILE` – это специальный тип данных, определённый в файле `stdio.h`, который можно использоваться точно так же, 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`:

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

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

"r" (от слова read) – файл открывается на чтение. Если файла не существует, то `fopen_s` возвратит ненулевое значение;

"w" (от слова write) – файл открывается 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 при этом используются функции `fscanf_s` и `fprintf`, которые отличаются от `scanf_s` и `printf` наличием ещё одного параметра типа `FILE*`. В остальном работа происходит точно так же: функция `fscanf_s` вместо системного буфера чтения использует файл, а `fprintf` то, что простая функция `printf` вывела бы на экран, печатает в файл. Например, если мы хотим считать целое число из файла `file.txt` в переменную `n`, то нужно выполнить следующий код:

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

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

Осталось выяснить ещё один вопрос. Если мы считываем данные с клавиатуры и буфер чтения пуст, то программа приостанавливает свою работу и ожидает данные от пользователя. Что будет, если попытаться считать данные из файла, который закончился?

Таблица 5

| 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 |

В этом случае выполнение программы останавливаться не будет, а функция `fscanf_s` вернёт значение специальной макроконстанты `EOF`, определённой в заголовочном файле как

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

Подытожим смысл значений, возвращаемых функциями `scanf_s` и `fscanf_s`, в табл. 5.

10.4. ПЕРЕДАЧА ПАРАМЕТРОВ В ПРОГРАММУ ПРИ ЗАПУСКЕ

При написании программ на языке Си часто используют передачу параметров в программу непосредственно при её запуске. Например:

```
name input.txt output.txt
```

Здесь `name` – имя исполняемого файла задачи (`name.exe`). При этом функция `main()` должна быть оформлена в виде

```
main(int argc, char* argv[])
```

```
{  
...
```

При запуске программы с командной строкой параметр `argc` получает значение, равное количеству параметров в строке (включая имя задачи), а элементы массива `argv` – адреса строковых параметров, передаваемых через данную командную строку.

Для вышеприведённого примера `argc` равно трём, а `argv` будут выглядеть следующим образом:

```
argc = 3;  
argv[0] = "name";  
argv[1] = "input.txt";  
argv[2] = "output.txt";
```

Передача параметров через командную строку позволяет программе работать с файлами различных типов, осуществлять операции открытия, обработки, закрытия, а также чтения и записи информации.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Практическое задание № 1 Технология работы с программами на языке Си в системе программирования Visual Studio. Элементарные операции

Цель задания

Изучить процесс прохождения программы на языке Си в системе программирования Visual Studio. Ознакомиться с технологией отладки программ. Изучить основные конструкции операторов присваивания и директивы препроцессора.

Указания для выполнения практического задания

- При выполнении задания необходимо стремиться максимально упростить выражения в операциях присваивания. Для этого следует использовать операции инкремента и декремента, а также операторы «+=», «*=» и т. п.
- Все константы, используемые в программе, должны быть описаны директивами препроцессора `#define`.
- Для использования функций ввода-вывода `printf()` и `scanf()` необходимо включить в программу файл `stdio.h` директивой `#include <stdio.h>`.
- Все программы должны быть отлажены в пошаговом режиме с выводом информации о значениях переменных в окно отладки.

Порядок выполнения задания

- Написать программу, выполняющую арифметические и логические операции над целыми переменными A, B, C, D по вариантам (табл. 6). Использовать глобальные статические переменные и инициализацию при компиляции. Вывести значения переменных с помощью функции `printf()` в заданном формате (см. пример на с. 111). Переменные C и D печатать в восьмеричном виде.
- Модифицировать программу, описав переменные как локальные и задав их значения с помощью прямого присваивания. Сравнить результаты с предыдущим пунктом.
- Изменить программу, применив для ввода значений переменных функцию `scanf()`. Сравнить результаты с предыдущими пунктами.
- Описать переменные как автоматические. Вывести их значения до присваивания. Объяснить полученные результаты.
- Повторить первый пункт, разместив все описания в отдельном файле.
- Повторить первый пункт, используя статический массив вместо переменных A, B, C, D.

Т а б л и ц а 6

| Вариант | Задание |
|---------|--|
| 1 | A – сложить с B, C и D, увеличенными на 1. Результат умножить на 5. B – разделить по модулю 5. C – сложить с A, увеличенным на 1, и с B. Число B уменьшить на 1. D – выполнить поразрядное И 0–3-го разрядов числа B и 2-го разряда числа C, сдвинутого вправо на два разряда |
| 2 | A – сложить с B, C и D, уменьшенными на 1. Результат разделить на 2. B – умножить на 3. C – уменьшить на величину A, делённую по модулю B. D – выполнить поразрядное И 2–8-го разрядов числа B и 1–3-го разрядов числа C, сдвинутых вправо на два разряда |
| 3 | A – сложить с B и C, увеличенными на 1, и разделить по модулю D. B – разделить на 6. C – сложить с B, уменьшенным на 1, и прибавить A. Число A увеличить на 1. D – выполнить поразрядное И 1–6-го разрядов числа B и 0–2-го разрядов числа C, сдвинутых вправо на один разряд |

| Вариант | Задание |
|---------|--|
| 4 | <p>А – умножить на С, сложить с В и разделить по модулю D.</p> <p>В – уменьшить на 4.</p> <p>С – сложить с В, уменьшенным на 1, и прибавить А. Число А увеличить на 1.</p> <p>D – выполнить поразрядное ИЛИ 6–7-го разрядов числа В и 0–3-го разрядов числа С, сдвинутых влево на три разряда</p> |
| 5 | <p>А – сложить с произведением В и С, делённым по модулю D.</p> <p>В – увеличить на 3.</p> <p>С – из числа С вычесть А и В, уменьшенные на 1. Число В не изменять.</p> <p>D – выполнить поразрядное И 0–3-го разрядов числа В и 1–5-го разрядов числа С, сдвинутых вправо на один разряд</p> |
| 6 | <p>А – умножить на В, разделить по модулю С и вычесть D. Числа С и D уменьшить на 1.</p> <p>В – разделить по модулю 7.</p> <p>С – сложить с А и В и разделить на 3. Число А увеличить на 1, число В уменьшить на 1.</p> <p>D – выполнить поразрядное И числа В (1-го ИЛИ 5-го разряда) и числа С (3–7-го разрядов), сдвинутых влево на один разряд</p> |
| 7 | <p>А – умножить на сумму чисел В и С, делённую по модулю 6, затем прибавить D. Число D увеличить на 1.</p> <p>В – разделить на 7.</p> <p>С – из числа С вычесть А и В. Число А увеличить на 1, число В уменьшить на 1. Результат разделить на 3.</p> <p>D – выполнить поразрядное ИЛИ 0–5-го разрядов числа В и 0–3-го разрядов числа С, сдвинутых вправо на два разряда</p> |
| 8 | <p>А – разделить по модулю В, прибавить сумму С, увеличенного на 1, и D. Число D увеличить на 1.</p> <p>В – увеличить на 5.</p> <p>С – сложить с числом А, увеличенным на 1, и с В. Число В уменьшить на 1. Результат разделить на 5.</p> <p>D – выполнить поразрядное ИЛИ числа В (2-го ИЛИ 6-го разряда) и числа С (4–8-го разрядов), сдвинутых влево на два разряда</p> |
| 9 | <p>А – сложить с произведением чисел В и С, делённым по модулю 5. Числа В и С уменьшить на 1</p> |

| Вариант | Задание |
|---------|--|
| | <p>В – уменьшить на 8.</p> <p>С – умножить на число В, разделить на сумму А и С, число С увеличить на 2.</p> <p>Д – выполнить поразрядное И 4–7-го разрядов числа В, сдвинутых вправо на два разряда, и 0–1-го разрядов числа С, сдвинутых влево на один разряд</p> |
| 10 | <p>А – разделить по модулю В, прибавить произведение чисел С и D, число С увеличить на 1, число D уменьшить на 1.</p> <p>В – разделить по модулю 3.</p> <p>С – сложить с числами А и В, разделить по модулю 2, результат умножить на 3.</p> <p>Д – выполнить поразрядное ИЛИ 0–2-го разрядов числа В и 0–3-го разрядов числа С, сдвинутых влево на три разряда</p> |
| 11 | <p>А – сложить с числами В, С и D, уменьшенными на 1. Результат разделить по модулю 4.</p> <p>В – увеличить на 4.</p> <p>С – сложить с числом А, уменьшенным на 1, и с В. Результат разделить на 2.</p> <p>Д – выполнить поразрядное ИЛИ 1–2-го разрядов числа В и 4-го разряда числа С, сдвинутого вправо на один разряд</p> |
| 12 | <p>А – умножить на С, уменьшенное на 1, и сложить с D, увеличенным на 1. Результат разделить по модулю 5.</p> <p>В – разделить по модулю 2.</p> <p>С – сложить с числом А, уменьшенным на 1, и прибавить В. Результат разделить на 2.</p> <p>Д – выполнить поразрядное И числа В (2-го ИЛИ 6-го разряда) и 5-го разряда числа С, сдвинутого влево на два разряда</p> |
| 13 | <p>А – разделить по модулю С и сложить с числом В, уменьшенным на 1. Результат разделить по модулю D.</p> <p>В – уменьшить на 5.</p> <p>С – сложить с числом А, уменьшенным на 1, и прибавить В. Число В увеличить на 1.</p> <p>Д – выполнить поразрядное ИЛИ числа В (2–5-го разрядов) и числа С (5-го ИЛИ 6-го разряда), сдвинутого влево на четыре разряда</p> |

| Вариант | Задание |
|---------|--|
| 14 | <p>А – умножить на В, прибавить сумму чисел С и D, уменьшенных на 1. Результат разделить на 3.</p> <p>В – разделить на 7.</p> <p>С – из числа С вычесть А и В, увеличенные на 1. А не изменять.</p> <p>D – выполнить поразрядное И 5-го разряда числа В и 6–7-го разрядов числа С, сдвинутых вправо на один разряд</p> |
| 15 | <p>А – умножить на С, разделить на D и прибавить число В, уменьшенное на 1. Результат разделить на 4.</p> <p>В – уменьшить на 6.</p> <p>С – сложить с суммой чисел А и В, увеличенных на 1. Результат умножить на 2.</p> <p>D – выполнить поразрядное ИЛИ 7-го разряда числа В и 3-го разряда числа С, сдвинутого вправо на один разряд</p> |
| 16 | <p>А – умножить на сумму чисел В и С и разделить по модулю D. Числа А и В уменьшить на 1.</p> <p>В – умножить на 2.</p> <p>С – сложить с числами А и В, уменьшенными на 1. Результат умножить на 4.</p> <p>D – выполнить поразрядное ИЛИ числа В (2-го ИЛИ 4-го разряда) и числа С (1–2-го разрядов), сдвинутых вправо на три разряда</p> |
| 17 | <p>А – сложить с числами В и С, уменьшенными на 1, и с D, увеличенным на 1. Результат разделить на 3.</p> <p>В – разделить на 4.</p> <p>С – из числа С вычесть А, уменьшенное на 1, и прибавить В. Число В увеличить на 1.</p> <p>D – выполнить поразрядное ИЛИ 1-го разряда числа В и 4–5-го разрядов числа С, сдвинутых влево на один разряд</p> |
| 18 | <p>А – сложить с числами В, С и D. Результат разделить на 2. Числа В и С увеличить на 1, число D уменьшить на 1.</p> <p>В – разделить на 9.</p> <p>С – увеличить на величину А, делённую по модулю 5.</p> <p>D – выполнить поразрядное ИЛИ 3–6-го разрядов числа В и 8-го разряда числа С, сдвинутого вправо на два разряда</p> |
| 19 | <p>А – из числа А вычесть числа В, С и D. Результат умножить на 3. Числа В и С уменьшить на 1</p> |

| Вариант | Задание |
|---------|--|
| | <p>В – умножить на 8.</p> <p>С – число С сложить с суммой А и В, делённой на 3. Результат уменьшить на 4.</p> <p>Д – выполнить поразрядное И числа В (7-го ИЛИ 8-го разряда) и числа С (0–1-го разрядов), сдвинутых вправо на три разряда</p> |
| 20 | <p>А – сложить с В, умножить на С, разделить на Д. Число В увеличить на 1, число С уменьшить на 1.</p> <p>В – разделить по модулю 4.</p> <p>С – из числа С вычесть число А и прибавить произведение чисел С и Д, уменьшенных на 1.</p> <p>Д – выполнить поразрядное ИЛИ 4-го разряда числа В и 7-го разряда числа С, сдвинутого влево на один разряд</p> |

Пример

Исходные данные:

$A = 12 \quad B = 8 \quad C = 34 \quad D = 123$

Результат:

$A = 23 \quad B = 334 \quad C = 45 \quad D = 177$

Содержание отчёта

1. Цель задания.
2. Задание по варианту.
3. Листинги программ.
4. Полученные результаты.
5. Выводы.

Контрольные вопросы

1. Какие классы памяти существуют в языке Си?
2. Можно ли инициализировать автоматическую переменную?
3. Какая операция имеет больший приоритет: «+» или «++»?
4. Какая операция имеет больший приоритет: «&&» или «&»?

5. В каком случае размерность массива при его описании можно не указывать?
6. Какая логическая операция используется для обнуления группы двоичных разрядов?
7. Особенности выполнения операции арифметического сдвига вправо.
8. Чем определяется область видимости и время жизни переменной? Когда при описании переменной надо явно указывать, что она `static`, `auto`?

Практическое задание № 2

Условные операторы и циклические структуры в языке Си. Конструкции `if`, `if-else`, `switch`, `for`, `while`, `do-while`

Цель задания

Изучить правила оформления и использования условных операторов и операторов цикла в языке Си.

Указания для выполнения практического задания

- При выполнении работы нужно использовать все приведенные конструкции циклов. В табл. 7 указано, какие из них вы должны использовать при выполнении тех или иных пунктов задания.
- При написании программы с циклическими алгоритмами рекомендуется использовать фигурные скобки для явного указания тела цикла. Это уменьшает вероятность ошибок и делает понятной структуру программы.
- При вводе строк используйте функцию `gets()` / `gets_s()`, так как функция `scanf()` воспринимает пробелы в строке как разделители.
- Вывод строки на экран удобно выполнять с помощью функции `puts()`.

Порядок выполнения задания

1. Написать программу, которая позволит выполнить следующие действия.
 - 1.1. Ввести две строки текста с подсказками при вводе.

1.2. Определить и вывести на экран длины строк (без конечного нулевого байта).

1.3. Объединить две строки в одну, удалив все пробелы. Для модификации использовать одну из условных операций.

1.4. Выполнить обработку строки (варианты приведены в табл. 7). Обработку прекратить, если количество изменений будет больше четырёх. Использовать для реализации задания конструкцию **if-else**.

1.5. Вывести результирующую строку на экран.

2. Повторить программу, применив для выполнения задания оператор **switch** вместо **if-else**. Сравнить результаты.

Т а б л и ц а 7

| Номер варианта | Варианты циклов | | | Задание п. 1.4 |
|-------------------|--------------------|--------|--------|--|
| | п. 1.2 | п. 1.3 | п. 1.4 | |
| 1 | 1 | 2 | 3 | 1. Во всех парах одинаковых символов второй заменить на пробел. 2. Заменить буквы А, М, Р, R на цифры 0, 1, 2, 3 соответственно |
| 2 | 1 | 3 | 2 | 1. Во всех парах одинаковых символов первый заменить на «+». 2. Заменить буквы a, b, x, z на буквы e, k, m, p соответственно |
| 3 | 2 | 1 | 3 | 1. Заменить все пары букв «ХУ» на цифры «12». 2. Заменить все круглые и фигурные скобки на квадратные |
| 4 | 2 | 3 | 1 | 1. Заменить в строке все цифры пробелами. 2. Заменить знаки арифметических операций (+, -, *, /) на буквы p, m, u, r |
| 5 | 3 | 1 | 2 | 1. Заменить на пробел все символы, совпадающие с последним символом в строке. 2. Заменить цифры 2, 4, 6, 3 на 9, 7, 5, 8 соответственно |
| 6 | 3 | 2 | 1 | 1. Заменить все символы с кодами меньше 48 на пробелы. 2. Заменить все квадратные и фигурные скобки на круглые |

| Номер варианта | Варианты циклов | | | Задание п. 1.4 |
|-------------------|--------------------|--------|--------|--|
| | п. 1.2 | п. 1.3 | п. 1.4 | |
| 7 | 1 | 2 | 3 | 1. Заменить все символы с кодами больше 48 на пробелы. 2. Заменить знаки \$, %, &, # на U, R, L, X соответственно |
| 8 | 1 | 3 | 2 | 1. Заменить каждый третий символ на пробел. 2. Заменить буквы a, r, h, w на соответствующие буквы верхнего регистра A, R, H, W |
| 9 | 2 | 1 | 3 | 1. Заменить все пары цифр «89» на буквы «AB». 2. Заменить цифры 1, 3, 6, 7 на буквы A, R, N, E соответственно |
| 10 | 2 | 3 | 1 | 1. Заменить все пары одинаковых символов на пробелы. 2. Заменить буквы a, б, л, н на f, g, u, k соответственно |
| 11 | 3 | 1 | 2 | 1. Заменить все четные символы в строке на пробелы. 2. Заменить цифры 3, 6, 9, 7 на знаки @, #, &, + соответственно |
| 12 | 3 | 2 | 1 | 1. Заменить все нечетные символы в строке на пробелы. 2. Заменить знаки <, >, !, ? на буквы f, t, x, z соответственно |
| 13 | 1 | 2 | 3 | 1. Заменить на пробелы все символы, совпадающие с первым символом в строке. 2. Заменить символы \, __, ^, ~ на буквы q, w, e, r |
| 14 | 1 | 3 | 2 | 1. Заменить каждый третий символ в строке первым символом. 2. Заменить буквы q, s, x, v на буквы A, F, K, N соответственно |
| 15 | 2 | 1 | 3 | 1. Удалить из текстовой строки все чётные символы. 2. Заменить буквы D, L, G, S на буквы k, f, t, r соответственно |

| Номер варианта | Варианты циклов | | | Задание п. 1.4 |
|-------------------|--------------------|--------|--------|---|
| | п. 1.2 | п. 1.3 | п. 1.4 | |
| 16 | 2 | 3 | 1 | 1. Удалить из текстовой строки все нечётные символы. 2. Заменить буквы s, y, j, k на цифры 4, 8, 1, 6 соответственно |
| 17 | 3 | 1 | 2 | 1. Удалить из текстовой строки все символы «,». 2. Заменить символы :, ;, -, + на буквы S, F, V, N соответственно |
| 18 | 3 | 2 | 1 | 1. Удалить из текстовой строки все пары одинаковых символов. 2. Заменить цифры 5, 2, 0, 4 на буквы X, P, Y, D соответственно |
| 19 | 1 | 2 | 3 | 1. Все символы, совпадающие с последним символом строки, заменить цифрой 0. 2. Заменить все открывающиеся круглые и фигурные скобки на символ «+», а закрывающиеся круглые и фигурные скобки – на символ «-» |
| 20 | 1 | 3 | 2 | 1. Все символы, совпадающие с первым символом строки, заменить цифрой 1. 2. Заменить буквы Ш, Ы, Д, Ц на буквы й, у, т, г соответственно |

Примечание. В таблице обозначено:

- 1 – оператор for;
- 2 – оператор while;
- 3 – оператор do-while.

Содержание отчёта

- 1. Цель задания.
- 2. Задание по варианту.
- 3. Листинги программ.
- 4. Полученные результаты.
- 5. Выводы.

Контрольные вопросы

1. Пояснить логику работы оператора `if-else`. Возможны ли вложенные операторы `if`?
2. Какой оператор используется для выхода из оператора `switch`? Для чего используется метка `default` в операторе `switch`?
3. Реализовать с помощью условной операции функцию $x = \max(a, b)$.
4. Можно ли использовать выражения для задания констант в операторе `switch`? Можно ли в теле оператора `switch` использовать вложенные операторы `switch`?
5. В каких случаях целесообразно использовать оператор `goto`?
6. Написать цикл `while`, эквивалентный циклу `for`.
7. Изобразить алгоритм работы цикла `do-while`.
8. В какой из трёх конструкций циклов гарантируется хотя бы однократное выполнение тела цикла?
9. Какой оператор прерывает выполнение цикла? Назначение оператора `continue`.
10. Как записать цикл с бесконечным числом повторений (бесконечный цикл) для каждого из операторов цикла?
11. Нарисовать блок-схему цикла `while`.
12. Привести пример вложенного цикла.
13. Чему равно значение переменной цикла при его нормальном завершении и при прерывании?
14. Можно ли в теле цикла менять переменную цикла?

Практическое задание № 3

Указатели, функции

Цель задания

Освоить правила написания и использования функций в языке Си. Научиться использовать указатели при обработке массивов данных.

Указания для выполнения практического задания

- Необходимо выполнить сортировку массива строк. Для ускорения этой операции обычно используется дополнительный массив указателей. В этом случае вместо перестановки двух строк с помощью функции `strcpy()` используется перестановка указателей обычным присваиванием.
- Формирование массива указателей можно совместить с вводом строк.

Порядок выполнения задания

1. Написать программу сортировки массива строк по вариантам (табл. 8). Ввод данных, сортировку и вывод результатов оформить в виде функций. Входные и выходные параметры функции сортировки также указаны в табл. 8.

Входные и выходные параметры функций для ввода-вывода следующие.

- Прототип функции для ввода строк:

```
length = inp_str(char* string, int maxlen);  
// length – длина строки  
// string – введенная строка  
// maxlen – максимально возможная длина строки (размерность массива string)
```

- Прототип функции для вывода строк:

```
void out_str(char* string, int length, int number);  
// string – выводимая строка  
// length – длина строки  
// number – номер строки
```

2. Модифицировать программу п. 1, применив в функциях передачу параметров и возврат результатов по ссылке (с использованием указателей). Сравнить результаты.

Т а б л и ц а 8

| Номер варианта | Задание | Входные параметры | Выходные параметры |
|----------------|---|-------------------------------------|---|
| 1 | Расположить строки по возрастанию длины | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Длина меньшей строки |
| 2 | Расположить строки по убыванию длины | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Длина большей строки |
| 3 | Расположить строки в алфавитном порядке | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Первая буква первой строки |

Продолжение табл. 8

| Номер варианта | Задание | Входные параметры | Выходные параметры |
|-------------------|---|-------------------------------------|---|
| 4 | Расположить строки в обратном алфавитном порядке | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Длина первой строки |
| 5 | Расположить строки по возрастанию количества слов | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Первый символ последней строки |
| 6 | Расположить строки по убыванию количества слов | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Максимальное количество слов |
| 7 | Расположить строки по возрастанию количества цифр | 1. Массив 2. Размерность массива | 1. Количество цифр 2. Вторая цифра строки |
| 8 | Расположить строки по убыванию количества цифр | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Количество цифр |
| 9 | Расположить строки по возрастанию длины первого слова | 1. Массив 2. Размерность массива | 1. Максимальная длина слова 2. Количество перестановок |
| 10 | Расположить строки по убыванию длины первого слова | 1. Массив 2. Размерность массива | 1. Минимальная длина слова 2. Последняя буква первого слова |
| 11 | Расположить строки в алфавитном порядке по последней букве строки | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Последняя буква первой строки |

Продолжение табл. 8

| Номер варианта | Задание | Входные параметры | Выходные параметры |
|-------------------|--|-------------------------------------|---|
| 12 | Расположить строки в обратном алфавитном порядке по последней букве строки | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Последняя буква последней строки |
| 13 | Расположить строки по возрастанию количества пробелов в строке | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Максимальное количество пробелов в строке |
| 14 | Расположить строки по убыванию количества пробелов в строке | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Максимальное количество пробелов подряд в строке |
| 15 | Расположить строки по возрастанию длины последнего слова | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Максимальная длина последнего слова |
| 16 | Расположить строки по убыванию длины последнего слова | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Минимальная длина последнего слова |
| 17 | Расположить строки по возрастанию длины самого длинного слова | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Максимальная длина самого длинного слова |
| 18 | Расположить строки по убыванию длины самого длинного слова | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Минимальная длина самого длинного слова |

| Номер варианта | Задание | Входные параметры | Выходные параметры |
|----------------|--|-------------------------------------|--|
| 19 | Расположить строки по возрастанию длины самого короткого слова | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Максимальная длина самого короткого слова |
| 20 | Расположить строки по убыванию длины самого короткого слова | 1. Массив 2. Размерность массива | 1. Количество перестановок 2. Минимальная длина самого короткого слова |

Содержание отчёта

1. Цель задания.
2. Задание по варианту.
3. Листинги программ.
4. Полученные результаты.
5. Выводы.

Контрольные вопросы

1. Правила описания указателей.
2. Как связаны указатели и массивы?
3. Назначение прототипа функции.
4. Структура описания функции.
5. Какие операции допустимы с адресами?
6. Массивы указателей, их описание и использование.
7. Привести пример описания двумерного массива.
8. Привести пример доступа к элементу двумерного массива через указатель на него.
9. Какие классы памяти существуют в языке Си?

Практическое задание № 4

Структуры в языке Си

Цель задания

Ознакомиться с понятием структур. Научиться использовать структуру для организации простейших баз данных.

Указания для выполнения практического задания

- При написании программы следует использовать статические массивы структур или указателей на структуры. Размерность массивов: 3 или 4 структуры. Для динамического выделения памяти использовать функцию `malloc()`. Для определения размера структуры в байтах удобно использовать операцию `sizeof()`, возвращающую целую константу:

```
struct ELEM *sp;  
sp = malloc(sizeof(struct ELEM));
```

- При выполнении п. 2 задания потребуется выполнять операцию перестановки элементов массива. Для этого необходимо описать дополнительный указатель на структуру. Можно также использовать вспомогательный массив указателей.

- Ввод данных выполнить с помощью функций `scanf()`.

Порядок выполнения задания

1. Написать программу, работающую с базой данных в виде массива структур и выполняющую последовательный ввод данных в массив, а также последующую распечатку его содержимого. Состав структуры приведён в табл. 9. Типы данных выбрать самостоятельно.

2. Переписать программу п. 1, используя массив указателей на структуры и динамическое выделение памяти. Выполнить сортировку массива. Способ сортировки массива приведён в табл. 9.

Таблица 9

| Номер варианта | Структура данных | Задание для п. 2 |
|----------------|--|---|
| 1 | Фамилия Год рождения Номер отдела Оклад | Расположить записи в массиве в порядке возрастания по году рождения |

Продолжение табл. 9

| Номер варианта | Структура данных | Задание для п. 2 |
|-------------------|---|---|
| 2 | Название детали Год выпуска Стоимость Количество | Расположить записи в массиве в порядке возрастания стоимости |
| 3 | Название книги Год издания Количество страниц Стоимость | Расположить записи в массиве в алфавитном порядке по названию |
| 4 | Фамилия Дата рождения Телефон Сумма долга | Все записи с суммой долга, не равной нулю, разместить в начале массива |
| 5 | Название покупки Дата приобретения Стоимость Количество | Сгруппировать все записи по месяцам приобретения |
| 6 | Название команды Игры Очки Сумма призового фонда | Расположить записи в порядке воз- растания по сумме призового фонда |
| 7 | Фамилия Группа Номер в списке Стипендия | Расположить записи в порядке воз- растания номера в списке |
| 8 | Название маршрута Протяжённость Количество остановок Стоимость путевки | Расположить записи в порядке убыва- ния стоимости |
| 9 | Фамилия спортсмена Вид спорта Количество медалей Призовой фонд | Расположить в алфавитном порядке записи с ненулевым количеством ме- далей |

Продолжение табл. 9

| Номер варианта | Структура данных | Задание для п. 2 |
|-------------------|---|--|
| 10 | Название детали Количество Вес Адрес поставщика | Расположить записи в порядке возрастания количества деталей с весом меньше заданного |
| 11 | Фамилия Группа Место прохождения практики Оценка | Расположить записи в массиве в порядке убывания оценки |
| 12 | Гос. номер Марка / модель автомобиля Фамилия владельца Дата угона автомобиля | Расположить записи в массиве в порядке возрастания даты угона |
| 13 | Наименование товара Наименование заказчика Количество заказанного товара Фамилия менеджера | Расположить записи в массиве в алфавитном порядке по наименованию заказчика |
| 14 | Номер рейса Дата вылета Фамилия пассажира Номер кассы | Расположить записи в массиве в порядке возрастания номера кассы |
| 15 | Район города Количество комнат в квартире Площадь квартиры Цена квартиры | Расположить записи в массиве в порядке возрастания количества комнат |
| 16 | Фамилия студента Группа Наименование дисциплины Оценка | Расположить записи в массиве в порядке убывания оценки |

| Номер варианта | Структура данных | Задание п. 2 |
|----------------|--|--|
| 17 | Фамилия преподавателя Наименование дисциплины Номер курса Аудитория | Расположить записи в массиве в алфавитном порядке по фамилии преподавателя |
| 18 | Учетная запись пользователя Буква диска Дисковая квота пользователя Объем занятого дискового пространства | Расположить записи в массиве в алфавитном порядке по учётной записи пользователя |
| 19 | Название книги Издательство Цена книги Количество книг | Расположить записи в массиве в алфавитном порядке по названию издательства |
| 20 | Наименование микросхемы Тип корпуса Наименование изготовителя Цена | Расположить записи в массиве в порядке возрастания цены |

Содержание отчёта

1. Цель задания.
2. Задание по варианту.
3. Листинги программ.
4. Полученные результаты.
5. Выводы.

Контрольные вопросы

1. Правила описания структур.
2. Операции, используемые для выделения элементов структур.
3. Выделение элементов структур, адресуемых указателем.
4. Назначение функции `malloc()`.
5. Отличие функции `calloc()` от `malloc()`.

6. Правила инициализации структур при описании.
7. Особенности выделения памяти под структуры.
8. В каком случае размерность массива при его описании можно не указывать?
9. Назначение функций `realloc()` и `free()`.
10. Как передать структуру в функцию?

Практическое задание № 5

Файловые операции

Цель задания

Ознакомиться со стандартными функциями языка Си, используемыми для организации доступа к файлам.

Указания для выполнения практического задания

- Требуется написать две программы для обработки текстовых файлов. Одна из них выполняет построчную обработку, другая – посимвольную. Ввод параметров должен быть организован в командной строке запуска программы.
- Исходный файл должен быть создан с помощью любого текстового редактора. При обработке текста рекомендуется использовать функции из стандартной библиотеки Си для работы со строками, для преобразования и анализа символов.

Порядок выполнения задания

1. Написать программу, обрабатывающую текстовый файл и записывающую обработанные данные в файл с таким же именем, но с другим типом (варианты задания приведены в табл. 10).
 2. Написать программу, выполняющую посимвольную обработку текстового файла (варианты задания приведены в табл. 11).
- Ввод параметров организовать в командной строке запуска программы.

Т а б л и ц а 10

| Номер варианта | Задание | Параметры командной строки |
|----------------|---|---|
| 1 | Исключить строки с длиной больше заданной | 1. Имя входного файла 2. Заданная длина строки |

Продолжение табл. 10

| Номер варианта | Задание | Параметры командной строки |
|-------------------|--|---|
| 2 | Оставить только строки, начинающиеся с латинских букв | 1. Имя входного файла 2. Количество обрабатываемых строк |
| 3 | Исключить строки, начинающиеся с заданного слова | 1. Имя входного файла 2. Заданное слово |
| 4 | Оставить строки, начинающиеся с заданной буквы | 1. Имя входного файла 2. Заданная буква |
| 5 | Исключить строки с количеством пробелов больше заданного числа | 1. Имя входного файла 2. Заданное количество пробелов |
| 6 | Оставить строки, не содержащие цифр | 1. Имя входного файла 2. Количество обрабатываемых строк |
| 7 | Исключить строки, начинающиеся заданной парой символов | 1. Имя входного файла 2. Заданная пара символов |
| 8 | Оставить строки, заканчивающиеся цифрами | 1. Имя входного файла 2. Максимальная длина строки |
| 9 | Исключить строки, содержащие хотя бы один заданный символ | 1. Имя входного файла 2. Заданный символ |
| 10 | Оставить строки, содержащие количество цифр меньше заданного | 1. Имя входного файла 2. Заданное количество цифр |
| 11 | Исключить строки, содержащие заданное слово | 1. Имя входного файла 2. Заданное слово |
| 12 | Оставить строки, где все слова имеют длину больше указанной | 1. Имя входного файла 2. Длина слова |
| 13 | Исключить строки, начинающиеся и заканчивающиеся заданным символом | 1. Имя входного файла 2. Заданный символ |
| 14 | Оставить строки, заканчивающиеся заданным словом | 1. Имя входного файла 2. Заданное слово |
| 15 | Исключить строки, не содержащие ни одного заданного символа | 1. Имя входного файла 2. Заданный символ |

Окончание табл. 10

| Номер варианта | Задание | Параметры командной строки |
|----------------|--|---|
| 16 | Оставить строки, где все слова имеют длину меньше указанной | 1. Имя входного файла 2. Длина слова |
| 17 | Исключить строки, в которых есть слова короче указанной длины | 1. Имя входного файла 2. Длина слова |
| 18 | Оставить строки, в которых указанное слово встречается более одного раза | 1. Имя входного файла 2. Заданное слово |
| 19 | Исключить строки, в которых есть хотя бы один не алфавитно-цифровой символ | 1. Имя входного файла 2. Количество обрабатываемых строк |
| 20 | Оставить строки с количеством слов меньше указанного | 1. Имя входного файла 2. Количество слов |

Таблица 11

| Номер варианта | Задание | Параметры командной строки |
|----------------|--|--|
| 1 | Удалить из текста заданный символ | 1. Имя входного файла 2. Заданный символ |
| 2 | В конце каждой строки вставить заданный символ | 1. Имя входного файла 2. Заданный символ |
| 3 | Заменить цифры пробелами | 1. Имя входного файла 2. Количество замен |
| 4 | Заменить знаки заданными символами | 1. Имя входного файла 2. Заданный символ |
| 5 | Заменить каждый пробел двумя пробелами | 1. Имя входного файла 2. Количество замен |
| 6 | После каждой точки вставить символ 'n' | 1. Имя входного файла 2. Количество замен |
| 7 | Удалить из текста все пробелы | 1. Имя входного файла 2. Количество замен |
| 8 | Заменить заданные символы пробелами | 1. Имя входного файла 2. Заданный символ |

| Номер варианта | Задание | Параметры командной строки |
|----------------|---|---|
| 9 | После каждого пробела вставить точку | 1. Имя входного файла 2. Количество вставок |
| 10 | Заменить все пробелы последним символом текста | 1. Имя входного файла 2. Максимальное количество замен |
| 11 | Во всех парах одинаковых символов второй символ заменить пробелом | 1. Имя входного файла 2. Количество замен |
| 12 | Заменить пробелами все символы, совпадающие с первым символом в строке | 1. Имя входного файла 2. Количество замен |
| 13 | Заменить заданную пару букв символами #@ | 1. Имя входного файла 2. Заданная пара букв |
| 14 | Заменить все цифры заданным символом | 1. Имя входного файла 2. Заданный символ |
| 15 | Заменить пробелами все символы, совпадающие с последним символом в строке | 1. Имя входного файла 2. Количество замен |
| 16 | Заменить пробелами все символы с кодами меньше 48 | 1. Имя входного файла 2. Количество замен |
| 17 | Заменить пробелами все символы с кодами больше 48 | 1. Имя входного файла 2. Количество замен |
| 18 | Заменить каждый третий символ пробелом | 1. Имя входного файла 2. Количество замен |
| 19 | Заменить все пробелы заданными символами | 1. Имя входного файла 2. Заданный символ |
| 20 | Заменить все пары одинаковых символов пробелами | 1. Имя входного файла 2. Количество замен |

Содержание отчёта

1. Цель задания.
2. Задание по варианту.
3. Листинги программ.
4. Полученные результаты.
5. Выводы.

Контрольные вопросы

1. Назначение указателя FILE*.
2. Стандартные функции открытия / закрытия файлов.
3. Стандартные функции для построчной обработки файлов.
4. Стандартные функции для посимвольной обработки файлов.
5. Стандартные функции работы со строками.
6. Буферизованный и небуферизованный ввод-вывод.
7. Передача параметров в программу при её запуске.
8. Проверка корректности выполнения операций файлового ввода-вывода.
9. Режимы открытия файлов.
10. Перенаправление стандартного ввода-вывода.
11. Чем определяется область видимости и время жизни переменной?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Керниган Б. Язык программирования Си: пер. с англ. / Б. Керниган, Д. Ритчи. – 3-е изд., испр. – Санкт-Петербург: Невский диалект, 2001. – 352 с.
2. Керниган Б. Язык программирования С / Б. Керниган, Д. Ритчи. – 2-е изд., перераб. и доп. – Москва: Вильямс, 2019. – 288 с.
3. Фомин С. Курс программирования на языке Си: учебник / С. Фомин, В. Подбельский. – Москва: Пресс, 2013. – 384 с.
4. Голодных Г. П. Программирование: методические указания к лабораторным работам для I курса АВТФ, направление 27.03.04 «Управление в технических системах» / Г. П. Голодных, А. В. Гунько, Г. В. Саблина. – Новосибирск: Изд-во НГТУ, 2017. – 33 с.

ОГЛАВЛЕНИЕ

| | |
|--|-----------|
| Введение | 3 |
| 1. Современные методы и средства разработки программного обеспечения (ПО) | 5 |
| 1.1. Технологии разработки ПО..... | 5 |
| 1.2. Основные этапы развития технологии разработки ПО | 6 |
| 2. Введение в программирование на языке Си | 9 |
| 2.1. Языки программирования | 9 |
| 2.2. Работа в IDE | 11 |
| 2.3. Отладка | 17 |
| 3. Сборка решений | 19 |
| 3.1. Препроцессор | 19 |
| 3.2. Компиляция | 21 |
| 3.3. Компоновка | 21 |
| 4. Работа с памятью | 23 |
| 4.1. Константы и переменные | 23 |
| 4.2. Типы данных | 25 |
| 4.3. Комментарии в программе на языке Си..... | 26 |
| 4.4. Приведение типов | 27 |
| 4.5. Адреса переменных и указатели..... | 28 |
| 5. Основы структурированного программирования | 30 |
| 5.1. Операнды, операции, выражения | 30 |
| 5.2. Основные операторы языка Си..... | 45 |

| | |
|---|-----|
| 6. Функции | 59 |
| 6.1. Назначение функций | 59 |
| 6.2. Объявление, определение и вызов функции | 61 |
| 6.3. Аргументы, параметры и возврат значения | 64 |
| 6.4. Передача параметров по указателю | 67 |
| 7. Массивы | 68 |
| 7.1. Массив как агрегатный тип данных | 68 |
| 7.2. Размещение в памяти и инициализация | 69 |
| 7.3. Передача массивов в функцию | 70 |
| 7.4. Многомерные массивы | 71 |
| 8. Сложные типы данных в языке Си. Структуры | 74 |
| 8.1. Понятие структуры | 74 |
| 8.2. Основные сведения о структурах | 75 |
| 8.3. Структуры и функции | 76 |
| 8.4. Массивы структур | 80 |
| 8.5. Указатели на структуры | 85 |
| 8.6. Выделение памяти под структуры | 88 |
| 9. Символы и строки | 90 |
| 9.1. Символы и кодировка ASCII | 90 |
| 9.2. Работа с символьными данными | 91 |
| 9.3. Строки | 93 |
| 10. Ввод-вывод | 98 |
| 10.1. Буфер ввода | 98 |
| 10.2. Форматированный ввод-вывод | 100 |
| 10.3. Работа с файлами | 103 |
| 10.4. Передача параметров в программу при запуске | 105 |
| Практическая часть | 106 |
| Практическое задание № 1. Технология работы с программами на языке Си в системе программирования Visual Studio. | |
| Элементарные операции | 106 |

| | |
|--|-----|
| Практическое задание № 2. Условные операторы и циклические структуры в языке Си. Конструкции if, if-else, switch, for, while, do-while | 112 |
| Практическое задание № 3. Указатели, функции | 116 |
| Практическое задание № 4. Структуры в языке Си..... | 121 |
| Практическое задание № 5. Файловые операции | 125 |
| Библиографический список | 130 |

**Саблина Галина Владимировна
Ядрышников Олег Дмитриевич**

ПРОГРАММИРОВАНИЕ

ЯЗЫК СИ

Учебное пособие

Редактор *Е.Е. Татарникова*
Выпускающий редактор *И.П. Брованова*
Корректор *Л.Н. Киншт*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 23.05.2023. Формат 60 × 84 1/16. Бумага офсетная. Тираж 80 экз.
Уч.-изд. л. 7,9. Печ. л. 8,5. Изд. № 12. Заказ № 166. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20