

1.1. История и этапы развития языка C

C – универсальный язык программирования. Он тесно связан с системой UNIX, так как был разработан в этой системе. В свою очередь система, как и большинство программ, работающих в ней, написаны на C. Однако язык не привязан жестко к какой-то одной операционной системе или машине. Хотя он и назван «языком системного программирования», поскольку удобен для написания компиляторов и операционных систем, оказалось, что на нем столь же хорошо писать большие программы другого профиля.

Многие важные идеи C взяты Деннисом Ритчи из языка BCPL, автором которого является Мартин Ричардс. Влияние BCPL на C было косвенным – через язык B, разработанный Кеном Томпсоном в 1970 г. для первой системы UNIX, реализованной на PDP-7.

В течение многих лет единственным определением языка C было первое издание книги «Язык программирования C» Брайана Кернигана и Денниса Ритчи (1978). В 1983 г. Институтом американских национальных стандартов (ANSI) учреждается комитет для выработки современного исчерпывающего определения языка C. Результатом его работы явился стандарт для C («ANSI-C»), который был одобрен в декабре 1989 г.

После 1989 года в центре внимания программистов оказался язык C++, к которому были добавлены новые свойства и возможности, разработанные для поддержки объектно-ориентированного программирования. Развитие этого языка на протяжении 1990-х годов завершилось одобрением стандарта в конце 1998 года. Между тем работа над языком C продолжалась. В итоге в 1999 году появился новый стандарт языка C, который обычно называют C99. В целом стандарт C99 сохранил практически все свойства стандарта C89, не изменив основных аспектов языка.

С появлением языка C++ некоторые программисты считали, что язык C потеряет самостоятельность и сойдет со сцены. Однако этого не произошло. Во-первых, не все программы должны быть объектно-ориентированными. Во-вторых, существует огромное множество программ на языке C, которые активно эксплуатируются и нуждаются в модификации. Поскольку язык C является основой языка C++, он продолжает широко использоваться, имея блестящие перспективы.

1.2. Сравнительный анализ языка C с другими языками программирования

Как ни странно, не все языки программирования предназначены для программистов. Рассмотрим классические примеры язы-

ков, ориентированных не на программистов, – COBOL и BASIC. Язык COBOL был разработан для того, чтобы люди, не являющиеся программистами, могли читать и (по возможности) понимать (хотя это вряд ли) написанные на нем программы. В свою очередь, язык BASIC был разработан для пользователей, которые решают на компьютере простые задачи.

В противоположность этому, язык С был создан для программистов, учитывал их интересы и многократно проверялся на практике, прежде чем был окончательно реализован. В итоге этот язык дает программистам именно то, чего они желали: сравнительно небольшое количество ограничений, минимум претензий, блочные структуры, изолированные функции и компактный набор ключевых слов.

Язык С часто называют языком среднего уровня. Это не означает, что он менее эффективен, более неудобен в использовании или менее продуман, чем языки высокого уровня, такие как Basic или Pascal. Отсюда также не следует, что он запутан, как язык ассемблера (и порождает связанные с этим проблемы). Это выражение означает лишь, что язык С объединяет лучшие свойства языков высокого уровня, возможности управления и гибкость языка ассемблера (рис. 1.1).

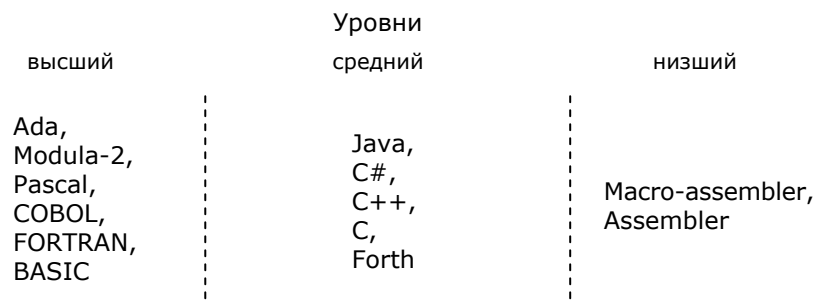


Рис. 1.1

Все языки высокого уровня используют концепцию типов данных. Тип данных определяет множество значений, которые может принимать переменная, а также множество операций, которые над ней можно выполнять. К основным типам данных относятся целое число, символ и действительное число. Несмотря на то что в языке С существует пять встроенных типов данных, он не является строго типизированным языком, как языки Pascal и Ada. В языке С разрешены практически все преобразования типов. Например, в одном и том же выражении можно свободно использовать переменные символьного и целочисленного типов.

В отличие от языков высокого уровня, язык С практически не проверяет ошибки, возникающие на этапе выполнения программ. Например, не осуществляется проверка возможного выхода индекса массива за пределы допустимого диапазона. Предотвращение ошибок такого рода возлагается на программиста.

Кроме того, язык С не требует строгой совместимости типов параметров и аргументов функций. Как известно, языки высокого уровня обычно требуют, чтобы тип аргумента точно совпадал с типом соответствующего параметра. Однако в языке С такого условия нет. Он позволяет использовать аргумент любого типа, если его можно разумным образом преобразовать в тип параметра. Кроме того, язык С предусматривает средства для автоматического преобразования типов.

Особенность языка С заключается в том, что он позволяет непосредственно манипулировать битами, байтами, машинными словами и указателями. Это делает его очень удобным для системного программирования, в котором эти операции широко распространены.

Язык С является структурированным языком. Отличительной особенностью структурированных языков является обособление кода и данных. Оно позволяет выделять и скрывать от остальной части программы данные и инструкции, необходимые для решения конкретной задачи. Этого можно достичь с помощью подпрограмм, в которых используются локальные (временные) переменные. Используя локальные переменные, можно создавать подпрограммы, не порождающие побочных эффектов в других модулях. Это облегчает координацию модулей между собой. Если программа разделена на обособленные функции, нужно лишь знать, что делает та или иная функция, не интересуясь, как именно она выполняет свою задачу. Чрезмерное использование глобальных переменных (которые доступны в любом месте программы) повышает вероятность ошибок и нежелательных побочных эффектов.

В структурированных языках использование оператора goto либо запрещено, либо нежелательно и не включается в набор основных средств управления потоком выполнения программы (как это принято в стандарте языка BASIC и в традиционном языке FORTRAN). Структурированные языки позволяют размещать несколько инструкций программы в одной строке и не ограничивают программиста жесткими полями для ввода команд (как это делалось в старых версиях языка FORTRAN).

Язык С не считается блочно-структурированным, поскольку не позволяет объявлять одну функции внутри других.

Структурированные языки считаются более современными. В настоящее время неструктурированность является признаком ус-

таревших языков программирования, и лишь немногие программисты выбирают их для создания серьезных приложений.

Новые версии старых языков программирования (например Visual Basic) включают элементы структурированности. И все же врожденные недостатки этих языков вряд ли будут до конца исправлены, поскольку структурированность не закладывалась в их основу с самого начала.

1.3. Microsoft Visual C++ .NET

Поскольку изучение языка С является начальным этапом для изучения C++, то писать программы на С мы будем сразу в интегрированной среде разработки (IDE – Integrated Development Environment) приложений Microsoft Visual Studio .NET, одним из компонентов которой является компилятор Microsoft Visual C++ .NET. Детальнее работа с IDE будет рассмотрена при изучении языка C++.

При инсталляции Microsoft Visual Studio .NET выполняется четыре шага (рис. 1.2): установка дополнительных пакетов; уста-

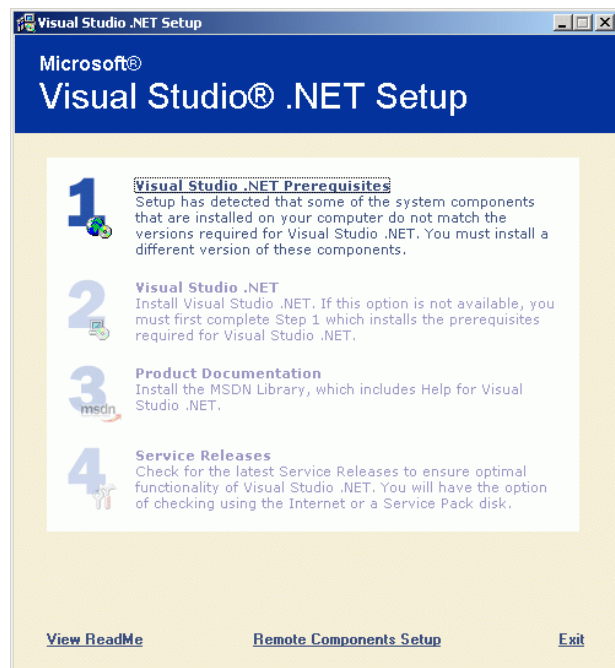


Рис. 1.2

новка Visual Studio; установка документации (MSDN Library); установка обновлений.

Сам по себе процесс установки настолько прост, что его описание не имеет смысла.

Запуск Visual Studio осуществляется через кнопку Пуск (Start) ⇒ Программы ⇒ Microsoft Visual Studio .NET 2003 ⇒ Microsoft Visual Studio .NET 2003 (рис. 1.3).



Рис. 1.3

После запуска появится окно показанное на рис. 1.4. Сначала необходимо настроить Visual Studio. Для этого необходимо вы-

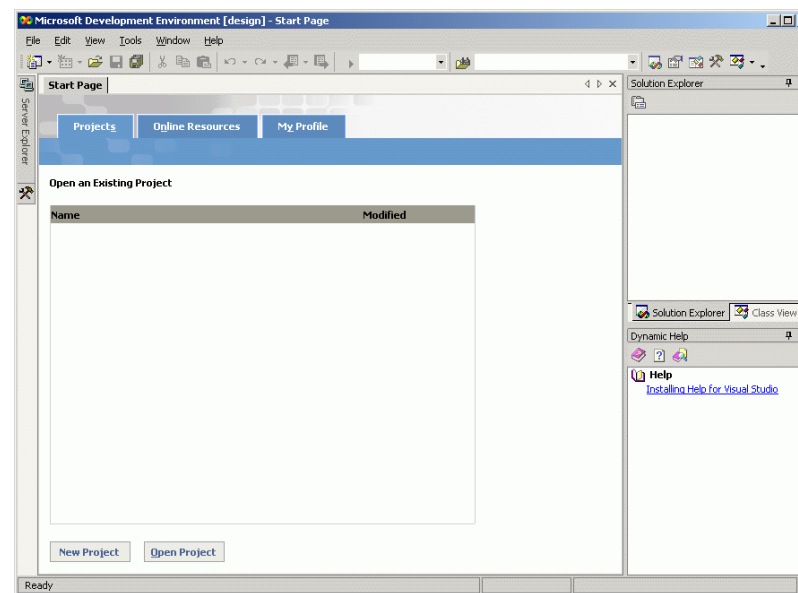


Рис. 1.4

брать закладку My Profile (рис. 1.5), а затем профиль поменять на Visual C++ Developer (рис. 1.6). Окна поменяют свое расположение и после этого необходимо снова выбрать закладку Projects.

Для создания нового проекта нажимаем кнопку New Project. На экране появится окно (рис. 1.7), в котором необходимо ввести тип проекта, его имя и расположение.

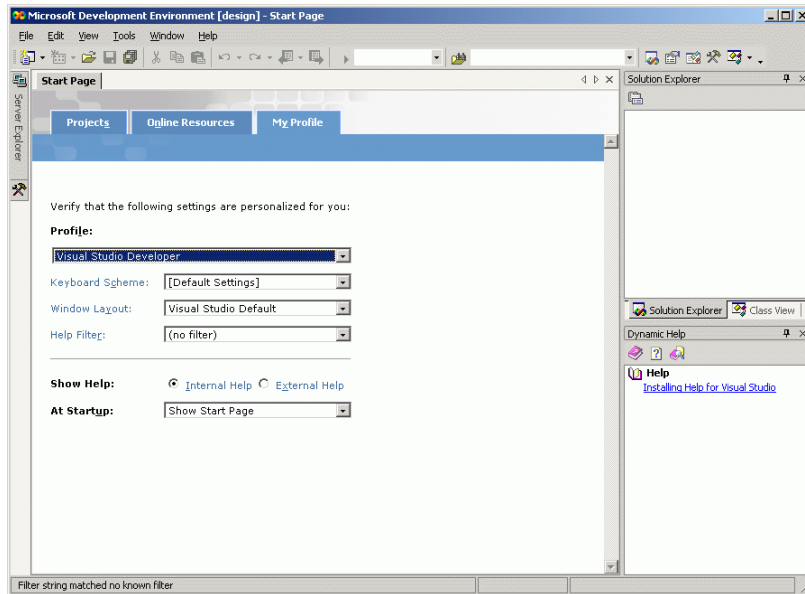


Рис. 1.5

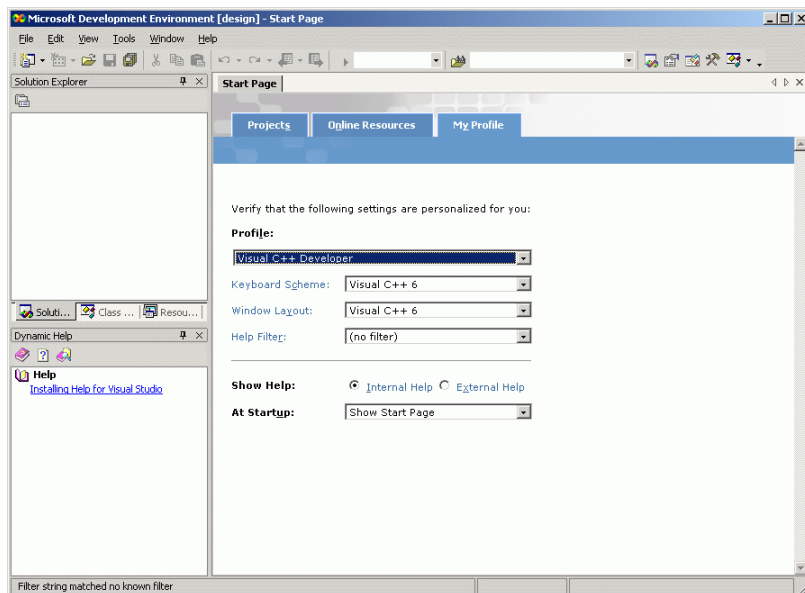


Рис. 1.6

Тип проекта должен быть Win32, а шаблон Win32 Console Project. После нажатия кнопки ОК появится окно Win32 Application Wizard (рис. 1.9), который позволяет генерировать «скелет» при-

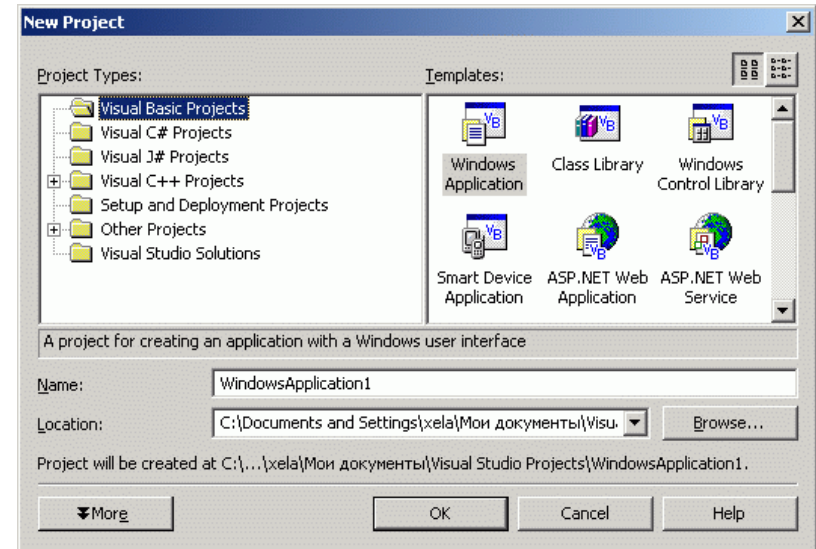


Рис. 1.7

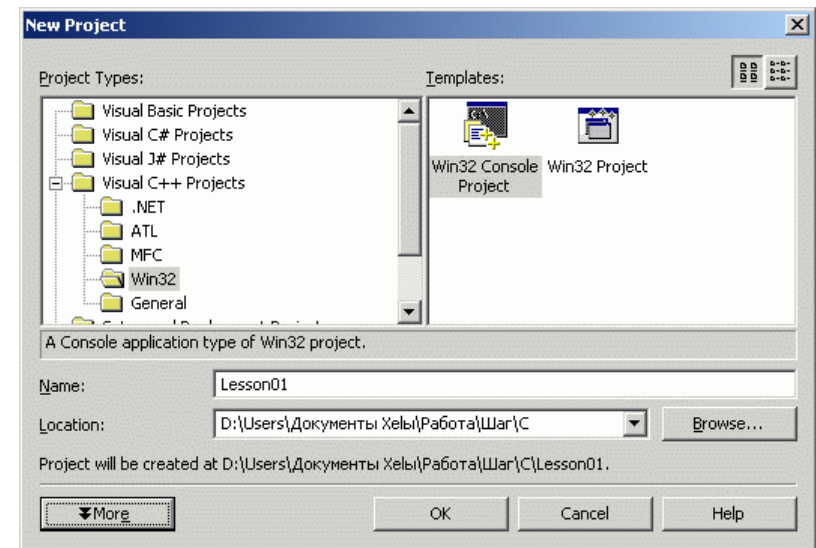


Рис. 1.8

ложения без написания кода вручную. Выбираем слева закладку Application Settings (рис. 1.10), выбираем Empty Project и нажимаем кнопку Finish.

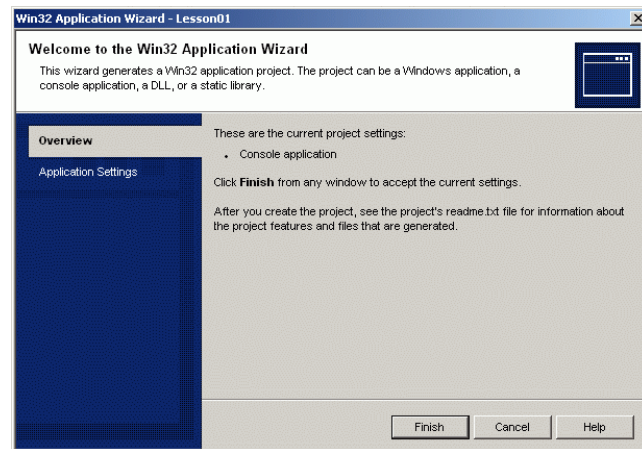


Рис. 1.9

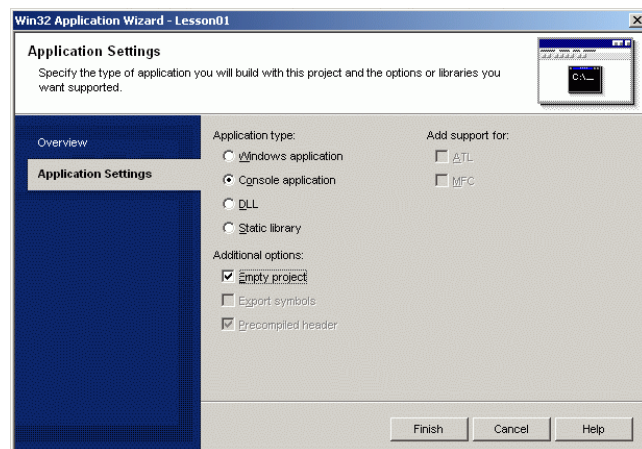


Рис. 1.10

Слева в окне Solution Explorer появится пустой проект (рис. 1.11). Добавим в этот проект нашу первую программу. Для этого выберем в меню пункт Project ⇒ Add New Item.... В появившемся окне в поле Name введите название файла и обязательно укажите расширение C, нажмите на кнопку Open.

У Вас появится окно для ввода текста программы. Прежде чем вводить текст, необходимо провести кое-какие подготовительные действия. Они необходимы потому, что программы, которые мы будем писать, будут консольными приложениями, то есть выполняться в среде DOS, а тексты программ мы будем набирать в среде Windows. Если мы будем выводить на экран кириллицу, то получим не тот результат который необходим. В связи с этим необходимо настроить редактор программы таким образом, что бы он работал с кодировкой DOS.

Закроем файл, который только что был создан (File ⇒ Close), и снова откроем его, только для этого на его названии (окно Solution Explorer) нажмем правую кнопку мыши и выберем пункт меню Open With.... В открывшемся окне (рис. 1.12) из списка выберем строку Source Code (Text) Editor With Encoding и нажмем

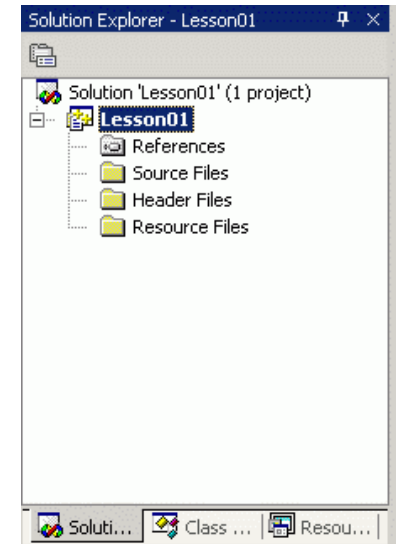


Рис. 1.11

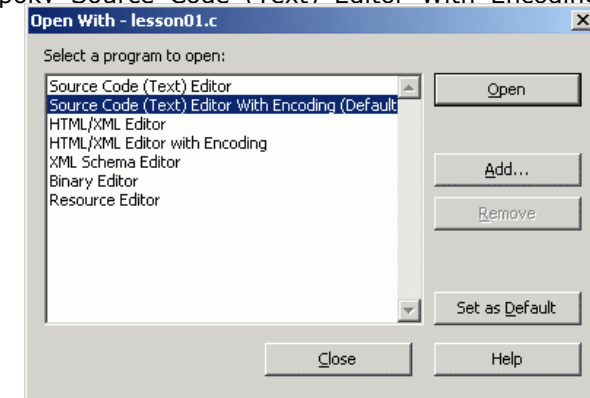


Рис. 1.12

рем строку Source Code (Text) Editor With Encoding и нажмем кнопку Set as Default, а потом кнопку Open. В появившемся окне в поле Encoding выберем пункт Cyrillic (DOS) – Codepage 866. Теперь можно приступать к написанию первой программы.

Занятие 2

2.1. Первая программа

Создадим проект Lesson02, а в нем файл lesson02.c. Наберем следующий текст программы:

```
/* lesson02 */
#include <stdio.h>
void main()
{
    printf("Здравствуй, мир!\n");
}
```

Запуск программы на выполнение осуществим с помощью команды меню Debug ⇒ Start Without Debugging или с помощью комбинации клавиш Ctrl+F5. После запуска на экране появится текстовое окно, в котором будут выведены следующие строки:

```
Здравствуй, мир!
Press any key to continue
```

Программа на C, каких бы размеров она ни была, состоит из функций и переменных. Функции содержат инструкции, описывающие вычисления, которые необходимо выполнить, а переменные хранят значения, используемые в процессе этих вычислений.

Приведенная программа – это функция с именем main. Обычно функциям можно давать любые имена, но «main» – особое имя: любая программа начинает свои вычисления с первой инструкции функции main.

Обычно main для выполнения своей работы пользуется услугами других функций; одни из них пишутся самим программистом, а другие берутся им готовыми из имеющихся в его распоряжении библиотек. Первая строка программы:

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

сообщает компилятору, что он должен включить информацию о стандартной библиотеке ввода-вывода. Эта строка встречается в начале многих исходных файлов C-программ.

Один из способов передачи данных между функциями состоит в том, что функция при обращении к другой функции передает ей список значений, называемых аргументами. Этот список обрамляется скобками и помещается после имени функции. В нашем примере main определена как функция, которая не ждет никаких аргументов, что отмечено пустым списком ().

Инструкции функции заключаются в фигурные скобки {}. Функция main содержит только одну инструкцию

```
printf("Здравствуй, мир!\n");
```

Функция вызывается по имени, после которого, в скобках, указывается список аргументов. Таким образом, приведенная строка – это вызов функции printf с аргументом "Здравствуй,

мир\n", которая в данном случае напечатает последовательность литер, заключенную в двойные кавычки.

Цепочка литер в двойных кавычках называется *стрингом* литер или *стринговой константой*.

2.2. Компиляция программ

Программа состоит из одной или нескольких *компонент трансляции*, хранящихся в виде файлов. Каждая такая компонента проходит ряд фаз трансляции. Начальные фазы осуществляют лексические преобразования нижнего уровня, выполняют директивы, задаваемые в программе строками, начинающимися со знака #, обрабатываются макроопределения и получают макrorасширения (см. занятия 75-76). По завершению препроцессирования программа представляется в виде последовательности лексем.

При отсутствии ошибок трансляции начинается процесс компиляции каждой из компонент трансляции. Такой процесс называется *раздельной компиляцией*.

Все компиляторы языка C сопровождаются стандартной библиотекой функций, выполняющих наиболее распространенные задачи. Существуют и другие библиотеки, например, содержащие графические функции. Функции представляют собой крупные строительные блоки, из которых можно сконструировать свою программу. В частности, если какую-то функцию вы используете в своих программах очень часто, ее следует поместить в библиотеку.

При вызове библиотечной функции компилятор "запоминает" ее имя. Позднее редактор связей объединит ваш откомпилированный код с объектным кодом этой библиотечной функции. Этот процесс называется редактированием связей (linking).

Скомпилировав все файлы, отредактировав связи между ними и библиотечными функциями, вы получите заверченный объектный код. Преимущество раздельной компиляции заключается в том, что при изменении кода, записанного в одном из файлов, нет необходимости компилировать заново всю программу. Это существенно экономит время на этапе компиляции.

2.3. Лексемы

Существует шесть классов лексем: идентификаторы, ключевые слова, константы, стринговые константы, операторы и прочие разделители. Пробелы, горизонтальные и вертикальные табуляции, новые строки, переводы-страницы и комментарии (имеющие общее название «пробельные литеры») рассматриваются компилятором только как разделители лексем и в остальном на результат трансляции влияния не оказывают. Любая из пробельных

литер годится, чтобы отделить друг от друга соседние идентификаторы, ключевые слова и константы.

В табл. 2.1 перечислены 32 ключевых слова, которые используются при формировании синтаксиса языка С.

Таблица 2.1

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

Обратите внимание на то, что все ключевые слова набраны строчными буквами. Ключевые слова нельзя использовать в программе для иных целей, например, в качестве имени переменной или функции.

Остальные лексемы будут рассмотрены далее.

2.4. Организация вывода данных на консоль

В языке С существует несколько библиотек, содержащих функции вывода данных. Рассмотрим функцию printf библиотеки stdio.h. Эта функция возвращает количество записанных ею символов (побочный эффект), а в случае ошибки – отрицательное число. Она содержит один обязательный параметр – *управляющая строка* и произвольное число необязательных параметров.

Управляющая строка состоит из элементов двух видов: символы, которые выводятся на консоль, и спецификаторы формата (см. занятие 4). Количество необязательных параметров должно соответствовать количеству спецификаторов формата в управляющей строке.

Как было сказано выше, управляющая строка представляет собой стринговую константу, некоторые литеры которой могут записываться с помощью эскейп-последовательностей (Escape). *Эскейп-последовательностями* называются комбинации литер, начинающиеся с обратной наклонной черты (например \n) для обозначения трудно представимых или невидимых литер.

Полный набор эскейп-последовательностей следующий:

- \n – новая строка
- \t – горизонтальная табуляция
- \v – вертикальная табуляция
- \b – возврат на шаг
- \r – возврат каретки

- \f – перевод страницы
- \a – сигнал-звонок
- \\ – обратная наклонная черта
- \? – знак вопроса
- \' – одиночная кавычка
- \" – двойная кавычка
- \ooo – восьмеричный код
- \xhh – шестнадцатеричный код

Литерная константа '\0' – это литера с нулевым значением – так называемая литера null.

2.5. Комментарии

Любые литеры, помещенные между /* и */, игнорируются компилятором, и ими можно свободно пользоваться, чтобы сделать программу более понятной. Комментарий можно располагать в любом месте, где могут стоять литеры пробела, табуляции или литера новой строки. Между звездочкой и косой чертой не должно стоять никаких символов.

В нашей первой программе комментарием является первая строка:

```
/* lesson02 */
```

Такой стиль комментариев называется *многострочным*, поскольку такие комментарии могут состоять из нескольких строк. Они не могут быть вложенными. Стандартом C89 предусмотрены только такие комментарии.

Язык C++ и стандарт C99 предусматривает два вида комментариев, то есть еще *однострочные комментарии*:

```
// lesson02
```

Однострочные комментарии особенно полезны для создания кратких пошаговых описаний программы. Однострочные комментарии можно вкладывать в многострочные.

Несмотря на то, что они не поддерживаются стандартом C89, многие компиляторы позволяют их применять.



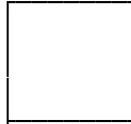
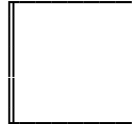
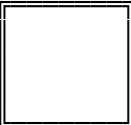
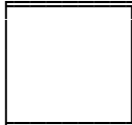
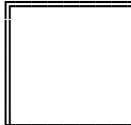

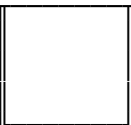
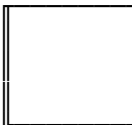
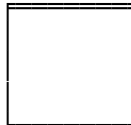
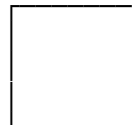




Комментарии следует размещать там, где необходимы пояснения. Например, все функции, за исключением очень коротких и самоочевидных, должны содержать комментарии, в которых описано их предназначение, способ вызова и возвращаемое значение.


2.6. Задания

1. Запустите программу, печатающую «Здравствуй, мир!». Поэкспериментируйте, удаляя некоторые части программы, и запишите, какие сообщения об ошибках Вы получите.

2. Выясните, что произойдет, если в строковую константу аргумента функции printf вставить \с, где с – литера не входящая в перечисленный в подразделе 2.4 список.
3. Измените программу так, чтобы приветствие на консоль выводилось в двойных кавычках.
4. Исправьте программу lesson02 так, чтобы она выводила приветствие в рамке (тип рамки выбрать в соответствии с вариантом из табл. 2.2). Символы для вывода рамки необходимо взять из таблицы символов ASCII – American Standard Code for Information Interchange (детальнее см. занятие 4)¹.

Таблица 2.2

№	Рамка	№	Рамка	№	Рамка	№	Рамка
1		2		3		4	
5		6		7		8	
9		10		11		12	
13		14		15		16	

¹ ASCII-коды символов, в шестнадцатеричной системе, представлены в табл. 4.1. Например, символ  имеет код 0xB1 (символ находится на пересечении строки В и столбца 1) и записывается при выводе на консоль с помощью эскейп-последовательности как \xB1. Десятичный код символа будет 177 = В(11)*16 + 1. Этот же код в восьмеричной системе будет равен 261 (\261 – в виде эскейп-последовательности).

Занятие 3 3.1. Переменные

На предыдущем занятии было уже сказано, что такое переменная – хранит значения используемые программой в процессе вычислений. *Переменная* представляет собой имя ячейки памяти, которую можно использовать для хранения модифицируемого значения.

В языке С имена переменных, функций и других объектов, определенных пользователем, называются идентификаторами.

Идентификатор – последовательность букв и цифр. Первой литерой должна быть буква; знак подчеркивания _ считается буквой. Буквы нижнего и верхнего регистров различаются.

Идентификаторы могут иметь любую длину; для внутренних идентификаторов значимыми являются первые 31 литеры. К внутренним идентификаторам относятся имена макросов и все другие имена не имеющие внешних связей. На идентификаторы с внешними связями могут накладываться большие ограничения: иногда воспринимаются не более шести первых литер и/или не различаются буквы верхнего и нижнего регистров.

В С любая переменная должна быть описана раньше, чем она будет использована; обычно все переменные описываются в начале функции перед первой исполняемой инструкцией. В декларации (описании) объявляются свойства переменных. Она состоит из названия типа и списка переменных:

тип список_переменных;

Здесь слово *тип* означает один из допустимых типов данных, включая модификаторы, а *список_переменных* может состоять из одного или нескольких идентификаторов, разделенных запятыми.

При объявлении переменной ей можно присвоить начальное значение. Общий вид инициализации выглядит следующим образом:

тип имя_переменной = значение;

Решим такую задачу: найти шестой член геометрической прогрессии 5, -10, ...

Если b_0 первый член геометрической прогрессии, а $q \neq 1$ – постоянное отношение следующего члена к предыдущему, называемое знаменателем прогрессии, то

$$b_i = b_0 q^i,$$

где i – номер члена геометрической прогрессии.

Таким образом, для нахождения шестого члена геометрической прогрессии необходимо найти знаменатель прогрессии, который равен: $q = b_1/b_0$.

Рассмотрим пример программы находящей шестой член геометрической прогрессии 5, -10,

```
#include <stdio.h>
#include <math.h>
void main()
{
    float b0 = 5, b1 = -10, q, b6;
    q = b1/b0;
    b6 = b0*pow(q, 6);
    printf("6-й член г/прогрессии %4.2f, %4.2f, ...
        равен %4.2f\n", b0, b1, b6);
}
```

В этой программе, в отличие от приведенной в занятии 1, подключается библиотека математических функций math.h (для возведения q в 6-ю степень – функция pow).

В начале программы описываются четыре переменных типа float (числа с плавающей точкой, то есть числа, которые могут иметь дробную часть). Переменные b0 и b1 при этом инициализируются начальными значениями, соответственно 5 и -10.

Далее находим знаменатель прогрессии q и шестой член геометрической прогрессии b6.

После вычисления на экран выводятся результаты: управляющая строка, кроме рассмотренного на занятии 2 символа новой строки, содержит спецификатор формата %4.2f (детальнее см. занятие 4). Этот спецификатор означает, что будет выведено число с плавающей точкой, причем оно на экране займет минимум 4 символа и 2 из них займет дробная часть числа.

После выполнения программы на экране появится текстовое окно, в котором будут выведены следующие строки:

6-й член г/прогрессии 5.00, -10.00, ... равен 320.00

3.2. Задания

Составить программу вычисления величин, где N – вариант.

№	Условие
1–4	Сумму всех четных чисел от 2 до $5 \times N$
5–8	Сумму всех двухзначных чисел, кратных N
9	Сумму одиннадцати первых членов арифметической прогрессии ² , если $a_3 + a_8 = 8$
10	Сумму всех двухзначных чисел
11–15	Сумму всех нечетных чисел от 3 до $5 \times N$

² Если a_0 – первый член арифметической прогрессии, а d – постоянная разность между следующим и предыдущим членами, называемая разностью прогрессии, то $a_i = a_0 + id$, а $S_i = (a_0 + a_i)(n + 1)/2$.

Занятие 4

4.1. Основные типы данных

В языке C существуют пять элементарных типов данных: символ, целое число, число с плавающей точкой, число с плавающей точкой удвоенной точности и переменная, не имеющая значений. Им соответствуют следующие ключевые слова: char, int, float, double и void. Все другие типы данных в языке C создаются на основе элементарных типов, указанных выше. Размер переменных и диапазон их значений зависит от типа процессора и компилятора.

Размер целого числа обычно равен длине машинного слова, принятой в операционной системе. Значения переменных типа char, как правило, используются для представления символов, предусмотренных в системе кодирования ASCII (табл. 4.1), однако во всех случаях размер символа равен 1 байт. Диапазон изменения переменных типа float и double зависит от способа представления чисел с плавающей точкой. В любом случае, этот диапазон достаточно широк. Стандарт языка C определяет минимальный диапазон изменения чисел с плавающей точкой: от $1E-37$ до $1E+37$. Минимальное количество цифр, определяющих точность чисел с плавающей точкой, указано в табл. 4.2.

Таблица 4.1

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☻	♥	♦	♣	♠	●	◼	○	◼	♂	♀	♪	♫	✽
1	▶	◀	↑	!!	¶	\$	—	‡	↑	↓	→	←	↶	↷	▲	▼
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	␣
8	A	B	V	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
9	P	C	T	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
A	a	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
B	▒	▒	▒													
C	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
D	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
E	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F	Ё	ё	Є	е	İ	ı	Ÿ	ÿ	°	•	•	√	№	¤	■	

Таблица 4.2

Тип	Обычный размер, бит	Минимальный диапазон
char	8	от -128 до 127
unsigned char	8	от 0 до 255
signed char	8	от -128 до 127
int	16 или 32	от -32768 до 32767
unsigned int	16 или 32	от 0 до 65535
signed int	16 или 32	такой же, как у int
short int	16	от -32768 до 32767
unsigned short int	16	от 0 до 65535
signed short int	16	такой же, как у short int
long int	32	от -2147483648 до 2147483647
signed long int	32	такой же, как у long int
unsigned long int	32	от 0 до 4294967295
float	32	шесть значащих цифр
double	64	десять значащих цифр
long double	80	десять значащих цифр

Тип void используется для определения функции, не возвращающей никаких значений (см. занятие 36), либо для создания обобщенного указателя (см. занятие 45).

4.2. Модификация основных типов

За исключением типа void, основные типы данных могут иметь различные модификаторы (modifiers), которые используются для более точной настройки: signed, unsigned, long, short.

Целочисленные типы можно модифицировать с помощью всех ключевых слов. Символьные типы можно уточнять с помощью модификаторов unsigned и signed. Кроме того, тип double можно модифицировать ключевым словом long. В табл. 4.2 указаны все возможные комбинации типов данных, а также их минимальные диапазоны и приблизительный размер в битах.

Применение модификатора signed допускается, но является излишним, поскольку по умолчанию все целые числа имеют знак. Наиболее важен этот модификатор при уточнении типа char в тех реализациях языка C, где тип char по умолчанию знака не имеет.

Бели модификатор типа используется отдельно (т.е. без указания элементарного типа), по умолчанию предполагается, что объявляемая переменная имеет тип int.

Хотя одного спецификатора int вполне достаточно, многие программисты предпочитают явно указывать его модификации.

Перепишем программу нахождения шестого члена геометрической прогрессии так, чтобы она работала с целыми числами:

```
#include <stdio.h>
#include <math.h>
void main()
{
    int b0 = 5, b1 = -10, q, b6;
    q = b1/b0;
    b6 = b0*(int)pow(q,6);
    printf("6-й член г/прогрессии %i, %i, ... равен\n",b0,b1,b6);
}
```

Отличие этой программы заключается в описании переменных, в вычислении b6 и в выводе результатов. При вычислении b6 используется приведение типов, которое мы рассмотрим на занятии 7.

4.3. Спецификаторы формата

Функция printf() выполняет форматированный ввод-вывод на консоль. Функция может оперировать любыми встроенными типами данных, включая символы, строки и числа. Она допускает широкий выбор спецификаторов формата, показанных в табл. 4.3.

Таблица 4.3

Код	Формат
%c	Символ
%d	Десятичное целое число со знаком
%i	Десятичное целое число со знаком
%e	Научный формат (строчная буква e)
%E	Научный формат (прописная буква E)
%f	Десятичное число с плавающей точкой
%g	В зависимости от того, какой формат короче, применяется либо %e, либо %f
%G	В зависимости от того, какой формат короче, применяется либо %E, либо %f
%o	Восьмеричное число без знака
%s	Строка символов
%u	Десятичное целое число без знака
%x	Шестнадцатеричное число без знака (строчные буквы)
%X	Шестнадцатеричное число без знака (прописные буквы)
%p	Указатель
%n	Указатель на целочисленную переменную. Присваивает этой переменной количество символов, выведенных перед ним
%%	Знак %

Многие спецификаторы формата имеют свои модификаторы, которые немного изменяют их смысл. Например, с их помощью можно изменять минимальную ширину поля, количество цифр после десятичной точки, а также выполнять выравнивание по левому краю. Модификатор формата указывается между символом процента и кодом формата.

Целое число, размещенное между символом процента и кодом формата, задает минимальную ширину поля. Если строка вывода короче, чем нужно, она дополняется пробелами, если длиннее, строка все равно выводится полностью. Строку можно дополнять не только пробелами, но и нулями. Для этого достаточно поставить 0 перед модификатором ширины поля. Например, спецификатор %05d дополнит число, количество цифр которого меньше пяти, ведущими нулями.

Задайте подобный модификатор в примере подраздела 3.2.

Модификатор минимальной ширины поля чаще всего используется для форматирования таблиц.

Модификатор точности указывается после модификатора ширины поля (если он есть). Этот модификатор состоит из точки, за которой следует целое число. Точный смысл модификатора зависит от типа данных, к которым он применяется.

Если модификатор точности применяется к числам с плавающей точкой с форматами %f, %e или %E, он означает количество десятичных цифр после точки.

Если модификатор применяется к спецификаторам формата %g или %G, он задает количество значащих цифр.

Если модификатор используется для вывода строк, он задает максимальную длину поля.

Если модификатор точности применяется к целым типам, он задает минимальное количество цифр, из которых должно состоять число. Если число состоит из меньшего количества цифр, оно дополняется ведущими нулями.

По умолчанию вывод выравнивается по правому краю. Иначе говоря, если ширине поля больше, чем выводимые данные, результаты "прижимаются" к правому краю. Вывод можно выровнять по левому краю, поставив после символа % знак "минус".

Для вывода значений переменных типа short int и long int функция printf() использует соответственно два модификатора h и l, которые можно применять к спецификаторам d, i, o, u и x. Модификаторы h и l можно также применять к спецификатору n.

Модификатор L можно использовать как префикс перед спецификаторами e, f и g. Он означает, что на экран выводится значение типа long double.

Функция printf() имеет еще два модификатора: * и #.

Если перед спецификаторами g, G, f, F или e стоит модификатор #, это означает, что число будет содержать десятичную точку, даже если оно не имеет дробной части. Если этот модификатор стоит перед спецификаторами x или X, шестнадцатеричные числа выводятся с префиксом 0x. Если же символ # указан перед спецификатором o, число будет дополнено ведущими нулями. К другим спецификаторам модификатор # применять нельзя.

Ширину поля и точность представления числа можно задавать не только константами, но и переменными. Для этого вместо точных значений в спецификаторе следует указать символ *. При сканировании строки вывода функция printf() поочередно сопоставляет модификатор * с каждым аргументом:

```
printf("&*. *f", 10, 4, 123.3).
```

4.4. Перечисления

Перечисление (enumeration) представляет собой набор именованных целочисленных констант, задающих все допустимые значения переменной данного типа. Перечисления часто встречаются в повседневной жизни. Например, месяцы года образуют перечисление

jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec.

Перечисление определяется как структура и начинается с ключевого слова enum. Вот как выглядит перечисление:

```
enum тип_переч {список_констант} список_перем
```

Тип_переч и список_перем указывать не обязательно (однако хотя бы один из этих элементов объявлений должен присутствовать).

Следующий фрагмент определяет перечисление с именем month:

```
enum month {jan, feb, mar, apr, may, jun, jul, aug,
            sep, oct, nov, dec};
```

Тип перечисления можно использовать для объявления переменных. В языке C оператор, приведенный ниже, объявляет переменную m типа month с начальным значением apr.

```
enum month m = apr;
```

Главное в перечислении то, что каждая константа обозначает целое число. Следовательно, их можно использовать вместо целочисленных переменных. Значение каждой константы на единицу превышает значение предыдущей. Первая константа равна 0.

Значения констант можно задавать с помощью инициализатора. Для этого после имени константы следует поставить знак равенства и целое число. Константа, указанная после инициализатора, на единицу превышает значение, присвоенное предыдущей константе. Например:

```
enum month {jan = 1, feb, mar, apr, may, jun, jul,
            aug, sep, oct, nov, dec};
```

Константы перечисления нельзя вводить и выводить как строки. Константа `jan` и другие – это просто имя целого числа.

4.5. Задания

1. Составить программу вычисления следующих величин³ (табл. 4.4). Вывести результаты.
2. Описать перечисление (табл. 4.5). Создать переменную этого типа и инициализировать ее любым значением. Вывести целочисленное значение этой переменной.

Таблица 4.4

№	Условие
1	Модуль вектора $5a + 10b$, если $a=\{3; 2\}$ и $b=\{0; -1\}$
2	Угол между векторами $a=\{1; 2\}$ и $b=\{1; -0,5\}$
3	Площадь четырехугольника с вершинами $A(0; 0)$, $B(-1; 3)$, $C(2; 4)$, $D(3; 1)$
4	Периметр треугольника с вершинами $A(1; 1)$, $B(4; 1)$, $C(4; 5)$
5	Модуль вектора $-2a + 4b$, если $a=\{3; 2\}$ и $b=\{0; -1\}$
6	Углы треугольника с вершинами $A(0; 1,7)$, $B(2; 1,7)$, $C(1,5; 0,85)$
7	Модуль вектора $a-b$, если $ a =3$, $ b =5$ и a и b составляют угол в 120°
8	Модуль вектора $a+b$, если $ a =11$, $ b =23$, $ a-b =30$
9	Угол между векторами $a=\{2; -4; 4\}$ и $b=\{-3; 2; 6\}$

Таблица 4.5

№	Перечисление	№	Перечисление
1	Дни недели	2	Цвета
3	Времена года	4	Время суток
5	Члены семьи	6	Ноты
7	Носители информации	8	Виды транспорта
9	Стиль музыки	10	Жанры игр

³ Модуль вектора $|\vec{a}| = \sqrt{a_x^2 + a_y^2}$ (для двухмерного случая) и

$|\vec{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}$ (для трехмерного случая); угол между векторами

$\cos \varphi = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|}$; скалярное произведение векторов $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y$

(для двухмерного случая) и $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$ (для трехмерного случая).

Занятие 5

5.1. Организация ввода данных

Для того чтобы писать полноценные программы, недостаточно только выводить результаты расчетов, необходимо также уметь их получать от пользователя для организации взаимодействия программы с окружающим миром.

Функция `scanf()` представляет собой процедуру ввода. Она может считывать данные всех встроенных типов и автоматически преобразовывать числа в соответствующий внутренний формат. Данная функция выглядит полной противоположностью функции `printf()`. Она возвращает количество переменных, которым она успешно присвоила свои значения. Если при чтении произошла ошибка, функция возвращает константу EOF. Параметр управляющая строка определяет порядок считывания значений и присваивания их переменным, указанным в списке аргументов.

Управляющая строка, также как и в функции `printf()`, состоит из символов, разделенных на три категории:

- спецификаторы формата;
- разделители;
- символы, не являющиеся разделителями.

Перед спецификаторами формата стоит символ `%`. Они сообщают функции `scanf()` тип данных, подлежащих вводу. Эти коды похожи на спецификаторы функции `printf()`, но все-таки имеют некоторые отличия (табл. 5.1).

Таблица 5.1

Код	Формат
<code>%c</code>	Символ
<code>%d</code>	Десятичное целое число со знаком
<code>%i</code>	Десятичное целое число со знаком, восьмеричное или шестнадцатеричное число
<code>%e</code>	Десятичное число с плавающей точкой
<code>%f</code>	Десятичное число с плавающей точкой
<code>%g</code>	Десятичное число с плавающей точкой
<code>%o</code>	Восьмеричное число
<code>%s</code>	Строка символов
<code>%u</code>	Десятичное целое число без знака
<code>%x</code>	Шестнадцатеричное число без знака (строчные буквы)
<code>%p</code>	Указатель
<code>%n</code>	Указатель на целочисленную переменную. Присваивает этой переменной количество символов, выведенных перед ним
<code>%[]</code>	Набор сканируемых символов
<code>%%</code>	Знак <code>%</code>

Несмотря на то что пробелы, знаки табуляции и символы перехода на новую строку используются как разделители при чтении данных любых типов, при вводе отдельных символов они считаются наравне со всеми. Например, если поток ввода содержит строку "x y", то фрагмент кода

```
scanf("%c%c%c", &a, &b, &c);
```

присвоит символ x переменной a, пробел – переменной b и символ y – переменной c.

Ввод строк мы рассмотрим подробнее на занятии 58.

Функция `scanf()` поддерживает спецификатор универсального формата, называемый набором сканируемых символов. При обработке этого спецификатора функция будет вводить только те символы, которые входят в заданный набор. Символы записываются в массив, на который ссылается соответствующий аргумент. Для того чтобы определить набор сканируемых символов, достаточно перечислить их в квадратных скобках: `%[XYZ]`.

При использовании набора сканируемых символов функция `scanf()` продолжает считывать символы, помещая их в соответствующий массив, пока не обнаружится символ, не входящий в заданный набор.

Если первым символом набора является знак ^, набор сканируемых символов трактуется наоборот. Знак ^ сообщает функции `scanf()`, что она должна вводить только символы, не определенные в наборе.

В большинстве реализаций можно задавать диапазон символов, используя дефис: `%[A-Z]`.

Следует помнить, что набор сканируемых символов чувствителен к регистру. Для того чтобы ввести как прописные, так и строчные буквы, их следует указать отдельно.

Если в управляющей строке задан разделитель, функция пропускает один или несколько разделителей во входном потоке. Разделителем может быть пробел, символ табуляции, символ вертикальной табуляции, символ прогона бумаги, а также символ перехода на новую строку.

Символы, не являющиеся разделителями, вынуждают функцию считывать и пропускать заданные символы. Например, управляющая строка `"%d,%d"` заставляет функцию считать целое число, прочитать и отбросить запятую, а затем считать следующее целое число. Если указанный символ не найден, функция прекращает работу. Если нужно считать и отбросить знак процента, в управляющей строке используется обозначение `%%`.

При использовании функции `scanf()` в качестве аргументов необходимо передавать не сами переменные, а их адреса (детальнее см. занятие 34):

```
scanf("%u",&num);
```

Как и функция `printf()`, функция `scanf()` допускает модификацию спецификатора формата. К ним относятся модификатор максимальной ширины поля (для строк); модификаторы `l` и `h` (для чисел).

Можно заставить функцию `scanf()` считывать поле, но не присваивать его ни одной переменной – *подавление ввода*. Для этого перед кодом соответствующего формата следует поставить символ `*`. Например, вызов

```
scanf("%d%*c%d", &x, &y);
```

означает, что если в программу вводится пара чисел 10,10, то запятая между числами должна считываться, но не присваиваться ни одной переменной. Подавление присваивания особенно полезно, когда программа обрабатывает лишь часть входной информации.

5.2. Чтение и запись символов

Простейшими консольными функциями ввода и вывода являются функция `getchar()`, считывающая символ с клавиатуры, и функция `putchar()`, выводящая символ на экран. Функция `getchar()` ожидает нажатия клавиши и возвращает ее значение, которое автоматически выводится на экран. Функция `putchar()` выводит символ на экран в точку, определенную текущим положением курсора. Прототипы функций выглядят так:

```
int getchar(void);
```

```
int putchar(int c);
```

Как видим, функция `getchar()` возвращает целое число. В его младшем байте содержится код символа, соответствующего нажатой клавише (старший байт обычно содержит нулевое значение). Это позволяет присвоить полученное целочисленное значение какой-нибудь символьной переменной. Если при вводе произошла ошибка, функция `getchar()` возвращает константу `EOF`.

Несмотря на то что функция `putchar()` по определению должна получать целочисленный аргумент, ей можно передавать только символьные значения. На экран поводится лишь младший байт аргумента. Функция `putchar()` возвращает либо символ, выведенный ею на экран, либо константу `EOF`, если произошла ошибка. Константа `EOF` определена в заголовочном файле `stdio.h` и обычно равна -1.

Функция `getchar()` может породить несколько проблем. Обычно эта функция помещает входные данные в буфер, пока не будет нажата клавиша `<ENTER>`. Такой способ называется буферизованным вводом. Для того чтобы, данные, которые вы ввели, действительно были переданы программе, следует нажать клавишу

<ENTER>. Кроме того, при каждом вызове функция `getchar()` вводит символы по одному, последовательно размещая их в очереди. Если программа использует интерактивный диалог, такое торможение становится раздражающим фактором. Несмотря на то, что стандарт языка C позволяет сделать функцию `getchar()` интерактивной, эта возможность используется редко.

В этом случае можно использовать другие функции, позволяющие считывать символы с клавиатуры. Наиболее известными альтернативами являются функции `getch()` и `getche()`. Их прототипы выглядят так:

```
int getch(void);
int getche(void);
```

Большинство компиляторов размещают прототипы этих функций в заголовочном файле `conio.h`.

После нажатия клавиши функция `getch()` немедленно возвращает результат, введенный символ на экране не отображается. Функция `getche()` аналогична функции `getch()`, за одним исключением: введенный символ отображается на экране.

Следующая программа ждет нажатия любой клавиши и выводит ее на экран (тремя способами):

```
/* lesson 05.c */
#include <stdio.h>
#include <conio.h>
void main()
{
    char ch;
    printf("Нажмите любую клавишу и нажмите Enter: ");
    ch = getch();
    printf("\nВы нажали %c\n", ch);
    printf("Нажмите любую клавишу: ");
    ch = getch();
    printf("\nВы нажали %c\n", ch);
    printf("Нажмите любую клавишу: ");
    ch = getche();
    printf("\nВы нажали %c\n", ch);
}
```

5.3. Задание

1. Описать чем отличается ввод символа с помощью функций `getchar()`, `getch()` и `getche()` в предыдущей программе.
2. В программе из задания 4 предусмотреть ввод произвольных значений координат векторов или точек (в зависимости от варианта), а также переменной перечисления.

Занятие 6

6.1. Константы

Константами называются фиксированные значения, которые программа не может изменить. Способ представления константы зависит от ее типа. Иногда константы также называют литералами.

Символьные константы заключаются в одинарные кавычки. Например, символы `'a'` и `'%'` являются константами. В языке C предусмотрены расширенные символы, которые позволяют использовать другие языки, помимо английского. Их длина равна 16 бит. Для того чтобы определить расширенную символьную константу, перед символом следует поставить букву L:

```
wchar_t wc;
wc = L'A';
```

Здесь переменной `wc` присваивается расширенная символьная константа, эквивалентная букве A. Расширенные символы имеют тип `wchar_t`. В языке C этот тип определяется в заголовочном файле и не является встроенным типом.

Целочисленные константы считаются числами, не имеющими дробной части. Например, числа 10 и -100 являются целочисленными константами. Константы с плавающей точкой содержат дробную часть, которая отделяется десятичной точкой. Например, число 11.123 представляет собой константу с плавающей точкой.

По умолчанию компилятор приводит числовые константы к наименьшему подходящему типу. Таким образом, если предположить, что целое число занимает 16 бит, то число 10 по умолчанию имеет тип `int`, а число 103000 – тип `long`. Несмотря на то, что число 10 можно привести к символьному типу, компилятор не нарушит границы типов. Единственное исключение из этого правила представляет собой константа с плавающей точкой, которая по умолчанию имеет тип `double`.

Используя суффиксы, типы констант можно задавать явно. Если после константы с плавающей точкой поставить суффикс `F`, она будет иметь тип `float`. Если вместо буквы `F` поставить букву `L`, константа получит тип `long double`. Для целочисленных типов суффикс `U` означает `unsigned`, а суффикс `L` – `long`.

Иногда позиционные системы счисления по основанию 8 или 16 оказываются удобнее, чем обычная десятичная система. Позиционная система счисления по основанию 8 называется восьмеричной. В ней используются цифры от 0 до 7. Числа в восьмеричной системе раскладываются по степеням числа 8. Система счисления с основанием 16 называется шестнадцатеричной. В ней используются цифры от 0 до 9 и буквы от A до F, которые соответствуют числам 10, 11, 12, 13, 14 и 15. Например, шестнадцатерич-

ное число 10 в десятичной системе равно 16. Поскольку эти системы счисления применяются очень часто, в языке C предусмотрены средства для представления шестнадцатеричных и восьмеричных констант. Для этого перед шестнадцатеричным числом указывается префикс 0x. Восьмеричные константы начинаются с нуля:

```
int hex = 0x80;      /* 128 в десятичной системе */
int oct = 012;        /* 10 в десятичной системе */
```

В языке C есть еще один вид констант – строковые. Строка – это последовательность символов, заключенная в двойные кавычки. Например, "пример строки" – это строка. Мы уже видели примеры использования строк, когда применяли функцию printf(). Несмотря на то что в языке C можно определять строковые константы, строго говоря, в нем нет отдельного типа данных для строк.

Не следует путать символы и строки. Отдельная символьная константа заключается в одинарные кавычки, например 'a'. Если заключить букву a в двойные кавычки, получим строку "a", состоящую из одной буквы.

Практически все символы можно вывести на печать, заключив их в одиночные кавычки. Однако некоторые символы, например, символ перехода на новую строку, невозможно ввести в строку с клавиатуры. Для этого в языке C предусмотрены специальные управляющие символьные константы, которые называются эскейп-последовательностями (см. занятие 2).

6.2. Операторы

В языке C предусмотрено большое количество операторов. Операторы разделяются на четыре основные группы: арифметические, сравнения, логические (см. занятие 8) и побитовые (см. занятие 80). Кроме того, для конкретных целей предусмотрено еще несколько специальных операторов.

Оператор присваивания можно использовать в любом корректном выражении. В языке C (в отличие от многих языков программирования, включая Pascal, BASIC и FORTRAN) оператор присваивания не считается особым. Общий вид оператора присваивания выглядит следующим образом:

```
имя_переменной = выражение;
```

Здесь выражение может состоять как из отдельной константы, так и комбинации сложных операторов. В качестве оператора присваивания в языке C используется знак равенства (отличие от языков Pascal и Modula-2, в которых оператор присваивания обозначается символами :=). Цель или левая часть оператора присваивания, должна быть либо переменной, либо указателем, но не функцией или константой.

В языке C разрешается присваивать одно и то же значение нескольким переменным одновременно. Например, во фрагменте программы, приведенном ниже, переменным x, y и z одновременно присваивается число 0:

```
x = y = z = 0;
```

Такой способ присваивания часто применяется в профессиональных программах.

В табл. 6.1 приведен список арифметических операторов языка C. Операторы +, -, * и / выполняются точно так же, как и в большинстве других языков программирования. Их можно применять практически к любым встроенным типам данных. Если оператор / применяется к целому числу или символу, дробная часть отбрасывается. Например, 5/2 равно 2.

Таблица 6.1

Оператор	Действие
-	Вычитание, а также унарный минус
+	Сложение
*	Умножение
/	Деление
%	Деление по модулю
--	Декрементация
++	Инкрементация

Оператор деления по модулю %, как и в других языках программирования, возвращает остаток целочисленного деления. Однако этот оператор нельзя применять к числам с плавающей точкой.

Унарный минус умножает свой операнд на -1. Иными словами, унарный минус меняет знак операнда на противоположный.

В языке C есть два полезных оператора, которыми не обладают некоторые другие языки. Это операторы инкрементации и декрементации ++ и --. Оператор ++ добавляет 1 к своему операнду, а оператор -- вычитает ее. Таким образом, оператор два следующих операторы эквивалентны:

```
x = x + 1;
++x;
```

Операторы инкрементации и декрементации имеют две формы: префиксную (++x) и постфиксную (x++). Однако между ними существует важное отличие, когда они используются внутри выражений. Если используется префиксная форма, операторы инкрементации и декрементации возвращают значению операнда после изменения, а если постфиксная – до:

```
x = 10; y = ++x;      /* y присвоится 11 */
```



```
x = 10; y = x++;          /* y присвоится 10 */
```

Большинство компиляторов языка С создают для операторов инкрементации и декрементации очень быстрый и эффективный объектный код. Он выполняется намного быстрее, чем код, соответствующий оператору присваивания. По этой причине операторы инкрементации и декрементации следует применять всегда, когда это возможно.

Ниже приведены приоритеты арифметических операторов:

```
высший      ++ --
              - (унарный минус)
              * / %

низший      + -
```

Операторы, имеющие одинаковый приоритет, выполняются слева направо. Разумеется, для изменения порядка вычисления операторов можно применять скобки. В языке С скобки интерпретируются точно так же, как и во всех других языках программирования. Они позволяют присвоить некоторым операторам или группе операторов более высокий приоритет.

Круглые скобки – это оператор, повышающий приоритет операций, заключенных внутри. Лишние скобки не приводят к ошибкам и не замедляют выполнение программ. Однако они позволяют прояснить точный порядок вычислений:

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

Пробелы и символы табуляции облегчают чтение программ.

В языке С существуют составные формы оператора присваивания, представляющие собой комбинации оператора присваивания и арифметических операторов. Например, оператор $x = x + 10$ можно переписать в виде $x += 10$.

Оператор "+=" сообщает компилятору, что к старому значению переменной x следует прибавить число 10.

Оператор присваивания образует составные формы практически со всеми бинарными операторами.

Составные формы присваивания широко используются в профессиональных программах на языке С. Часто их называют сокращенными операторами присваивания.

Оператор последовательного вычисления связывает в одно целое несколько выражений. Символом этого оператора является запятая. Подразумевается, что левая часть оператора последовательного вычисления всегда имеет тип void. Это означает, что значение выражения, стоящего в его правой части, становится значением всего выражения, разделенного запятыми. Например, оператор

```
x = (y=3, y+1);
```

сначала присваивает переменной y значение 3, а затем присваивает переменной x значение 4. Скобки здесь необходимы, поскольку приоритет оператора последовательного вычисления меньше, чем приоритет оператора присваивания.

По существу, выполнение оператора последовательного вычисления сводится к выполнению нескольких операторов подряд. Если этот оператор стоит в правой части оператора присваивания, то его результатом всегда будет результат выражения, стоящего последним в списке.

Оператор последовательного вычисления можно сравнить с союзом "и" в выражении "сделай это и это и это".

6.3. Задание

Написать программу вычисления математического выражения (табл. 6.2)⁴ для произвольных входных данных.

Таблица 6.2

№	Математическое выражение	№	Математическое выражение
1	$\frac{\ln 3z + \operatorname{arctg} 2z^3}{3(z+1)^2 + 2,1 \cdot 10^6}$	2	$\frac{10^{-7} \ln 3z^3 + \sin 2z^2}{(z+1)^{0,5} + 10^6}$
3	$\frac{\ln 7z + \operatorname{arctg} 2z^2}{7(z+1)^{0,5} + 2,7 \cdot 10^6}$	4	$\frac{10^{-7} \ln 3z + b^{0,4}}{\ln(z+1)^2 + 4,2 \cdot 10^4}$
5	$\frac{10^{-5} e^{-5f} + \sin^2 z^3 }{5(z+1)^5 + 10^6}$	6	$\frac{10^{-7} \sin 3z + b^{1,2}}{(z+1)^2 + 1,2 \cdot 10^6}$
7	$\frac{10^{-7} \ln 9z^3 + \cos 2z^2}{ z+1 ^2 + 2 \cdot 10^6}$	8	$\frac{10^{-4} e^{-2f} + \ln z^3 }{2(z+2)^{1,5}}$
9	$\frac{10^{-7} \ln 2z + \sin 2z^3}{3(z+3)^2 + 2,1 \cdot 10^7}$	10	$\frac{10^{-6} \ln 3z^3 + \ln^2 z^3}{6(z+1)^6 + 10^6}$

⁴ В математической библиотеке math.h имеются следующие функции для вычисления математического выражения:

$ x $ – fabs(x);	x^a – pow(x,a);
\sqrt{x} – sqrt(x);	$\cos x$ – cos(x);
$\sin x$ – sin(x);	$\operatorname{arctg} x$ – atan(x);
e^x – exp(x);	$\ln x$ – log(x).

Занятие 7

7.1. Преобразования типов в операторе присваивания

Когда в выражении смешиваются переменные разных типов, необходимо выполнить *преобразование типов*. Для оператора присваивания правило преобразования типов формулируется просто: значение правой части преобразовывается к типу левой части. Рассмотрим конкретный пример:

```
void main(void)
{
    int x = 550;
    char ch = '0';
    float f = 3.15;
    ch = x;    /* Строка 1 */
    x = f;     /* Строка 2 */
    f = ch;    /* Строка 3 */
    f = x;     /* Строка 4 */
}
```

В строке 1 левый старший разряд целой переменной *x* отбрасывается, и переменной *ch* присваиваются 8 младших битов, а поскольку число 550 в шестнадцатеричной системе равно 0x226, то переменной *ch* будет присвоен символ '0' (с кодом 0x26 или 38). Если значение переменной *x* изменяется от 0 до 255, то переменные *ch* и *x* будут эквивалентными. В строке 2 переменной *x* будет присвоена целая часть переменной *f*, то есть 3. В строке 3 восьмьбитовое целое число, хранящееся в переменной *ch*, будет преобразовано в число с плавающей точкой – 38,0. То же самое произойдет и в строке 4, только теперь в число с плавающей точкой будет преобразовано целочисленное значение переменной *x*, то есть 3,0.

В процессе преобразования типа *int* в тип *char* и типа *long int* в тип *int* старшие разряды отбрасываются. Во многих 16-битовых операционных системах это означает, что 8 бит будет потеряно. В 32-разрядных операционных системах при преобразовании типа *int* в тип *char* будет потеряно 24 бит, а при преобразовании типа *int* в тип *short int* – 16 бит.

Правила преобразования типов показаны в табл. 7.1. Помните, что преобразование типа *int* в тип *float*, типа *float* в тип *double* и так далее не увеличивает точности. Этот вид преобразований лишь изменяет способ представления числа.

Если вам необходимо выполнить преобразование типа, не указанное в табл. 7.1, преобразуйте его сначала в промежуточный тип, который есть в таблице, а затем – в результирующий тип. Например, чтобы преобразовать тип *double* в тип *int*, сначала следует преобразовать тип *double* в тип *float*, а затем тип *float* – в тип *int*.

Таблица 7.1

Результирующий тип	Тип выражения	Возможные потери
signed char	char	Если значение > 127, результатом будет отрицательное число
char	short int	Старшие 8 бит
char	int (16 бит)	Старшие 8 бит
char	int (32 бит)	Старшие 24 бит
char	long int	Старшие 24 бит
short int	int (16 бит)	Нет
short int	int (32 бит)	Старшие 16 бит
int (16 бит)	long int	Старшие 16 бит
int (32 бит)	long int	Нет
int	float	Дробная часть и, возможно, что-то еще
float	double	Точность, результат округляется
double	long double	Точность, результат округляется

7.2. Преобразование типов в выражениях

Если в выражение входят константы и переменные разных типов, они преобразуются к одному типу. Компилятор преобразует все операнды в тип "наибольшего" операнда. Этот процесс называется расширением типов. Во-первых, все переменные типов *char* и *short int* автоматически преобразуются в тип *int*. Этот процесс называется целочисленным расширением. Во-вторых, все преобразования выполняются одно за другим, следуя приведенному ниже алгоритму.

```
ЕСЛИ операнд имеет тип long double,
    ТО второй операнд преобразуется в тип long double
ИНАЧЕ, ЕСЛИ операнд имеет тип double,
    ТО второй операнд преобразуется в тип double
ИНАЧЕ, ЕСЛИ операнд имеет тип float,
    ТО второй операнд преобразуется в тип float
ИНАЧЕ, ЕСЛИ операнд имеет тип unsigned long,
    ТО второй операнд преобразуется в тип unsigned long
ИНАЧЕ, ЕСЛИ операнд имеет тип long,
    ТО второй операнд преобразуется в тип long
ИНАЧЕ, ЕСЛИ операнд имеет тип unsigned int,
    ТО второй операнд преобразуется в тип unsigned int
```

Кроме того, есть одно правило: если один из операторов имеет тип *long*, а второй – *unsigned int*, и значение переменной типа *unsigned int* невозможно представить с помощью типа *long*, то оба операнда преобразуются в тип *unsigned long*.

После выполнения преобразований тип обоих операндов

одинаков и совпадает с типом результата выражения.

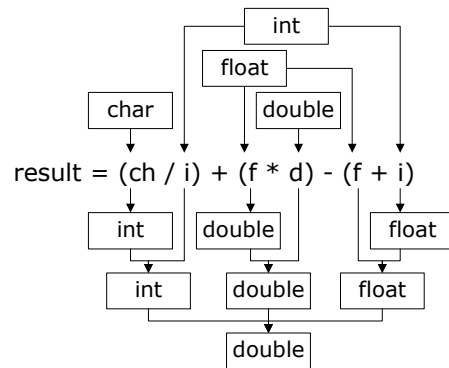


Рис. 7.1

Рассмотрим пример приведенный на рис. 7.1. Здесь символ `ch` сначала преобразуется в целое число. Затем результат операции `ch/i` преобразуется в тип `double`, поскольку именно этот тип имеет результат выражения `f*d`. После этого результат операции `f+i` преобразуется в тип переменной `f`, то есть в тип `float`. Окончательный результат имеет тип `double`.

7.3. Приведение типов

Используя приведение типов, можно принудительно задать тип выражения. Приведение типов имеет следующий вид:

(тип) выражение

Здесь тип – это любой допустимый тип данных. Например, в приведенном ниже фрагменте программы результат выражения `x/2` приводится к типу `float`:

(float) `x/2`

С формальной точки зрения приведение типов является унарным оператором, и его приоритет совпадает с приоритетами всех других унарных операторов.

Несмотря на то что приведение типов не слишком широко используется в программировании, иногда этот оператор оказывается очень полезным.

Предположим, что нам необходимо найти знаменатель геометрической прогрессии, заданной целочисленными значениями: 3, -7, Без оператора приведения типа (float) деление было бы целочисленным и дробная часть была бы потеряна, то есть:

```

q = -7/3;           /* -2.000000 */
q = (float) -7/3;   /* -2.333333 */
  
```

Занятие 8

8.1. Операторы сравнения и логические операторы

В термине *оператор сравнения* слово "сравнение" относится к значениям операндов. В термине *логический оператор* слово "логический" относится к способу, которым устанавливаются эти отношения. Поскольку операторы сравнения и логические операторы тесно связаны друг с другом, мы рассмотрим их вместе.

В основе и операторов сравнения, и логических операторов лежат понятия "истина" и "ложь". В языке C истинным считается любое значение, не равное нулю. Ложное значение всегда равно 0. Выражения, использующие операторы сравнения и логические операторы, возвращают 0, если результат ложен, и 1, если результат истинен.

Операторы сравнения и логические операторы приведены в табл. 8.1.

Таблица 8.1

Оператор	Обозначение в математике	Действие
Операторы сравнения		
<code>></code>	$>$	Больше
<code>>=</code>	\geq	Больше или равно
<code><</code>	$<$	Меньше
<code><=</code>	\leq	Меньше или равно
<code>==</code>	$=$	Равно
<code>!=</code>	\neq	Не равно
Логические операторы		
<code>&&</code>	\wedge	И
<code> </code>	\vee	ИЛИ
<code>!</code>	\neg	НЕ

Рассмотрим таблицу истинности для логических операторов, используя значения 1 и 0 (табл. 8.2).

Таблица 8.2

a	b	<code>a && b</code>	<code>a b</code>	<code>!a</code>
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Операторы сравнения и логические операторы имеют более низкий приоритет, чем арифметические операторы. Таким образом, выражение `10 > 1 + 12` будет вычислено так, будто оно запи-

сано следующим образом: $10 > (1 + 12)$. Разумеется, это выражение будет ложным.

В одном и том же выражении можно использовать несколько операторов:

$10 > 5 \&\& !(10 < 9) || 3 <= 4$

В данном случае результатом будет истинное значение.

Ниже приведены приоритеты операторов сравнения и логических операторов.

высший	!
	> >= < <=
	== !=
	&&
низший	

Как и для арифметических операторов, естественный порядок вычислений можно изменять с помощью скобок. Например, выражение

$!0 \&\& 0 || 0$

является ложным. Однако, если перегруппировать его операнды с помощью скобок, результат станет противоположным:

$!(0 \&\& 0) || 0$.

Существует еще несколько логических операций, но их описание отсутствует в языке C. Самой широко используемой в программировании является "исключающее ИЛИ" (XOR). Ее можно легко реализовать в виде функции, используя основные логические операторы. Операция XOR в математике обозначается как \oplus . Результатом операции XOR является истинное значение, если оба операнда имеют разные логические значения, то есть:

$a \oplus b = \neg(a = b)$

Поскольку в языке C истинным считается любое значение не равное 0, то эта формула не совсем подходит, так как если $a = 3$, $a = b = 5$, то результатом будет "истинна", а должна быть "ложь". Поэтому воспользуемся другой формулой:

$a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$

Запишем это выражение на языке C:

$a \&\& !b || !a \&\& b$

Скобки здесь не нужны, поскольку приоритет И больше чем ИЛИ.

Помните: все выражения, использующие операторы сравнения и логические операторы, имеют либо истинное, либо ложное значение. Следовательно, фрагмент программы, приведенный ниже, верен. В результате его выполнения на экран будет выведено значение 1:

```
int x = 100;
printf("%d", x>10);
```

8.2. Приоритеты операторов

Приоритеты всех операторов приведены ниже.

высший	() [] -> .
	! ~ ++ -- + - (тип) * & sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	&&
	?:
низший	= += -= *= /= и так далее

Обратите внимание на то, что все операторы, за исключением унарных (вторая строка), тернарного, присваивания и сокращенных операторов присваивания, выполняются слева направо. Операторы инкрементации и декрементации могут выполняться и слева направо, и справа на лево (см. занятие 6).

8.3. Задания

Написать программу вычисления математических выражений (табл. 8.3) для произвольных входных данных.

Таблица 8.3

№	Математическое выражение	№	Математическое выражение
1	$\ln x+z > 0 \wedge 0 < b < 1;$ $(A \wedge B) \vee (C = D)$	2	$x + z < 0 \vee 0 < f < 0,2;$ $(A \vee B) = (C \vee D)$
3	$ x+z > 1 \wedge 1 < b < 2;$ $(A \wedge B) = (C \wedge D)$	4	$ x > 2 \vee 0 < b < 3;$ $(A = B) \wedge (C \vee D)$
5	$0 < b < 1 \vee 0 < f < 0,5;$ $(A \wedge B) = (C \vee D)$	6	$\cos x+z > 0 \vee 0 < b < 6;$ $(A = B) \wedge (C = D)$
7	$ x+z > 0 \wedge 0 < b < 7;$ $(A \vee B) \wedge (C = D)$	8	$\ln x+z > 0 \vee 0 < b < 1;$ $(A = B) \vee (C \wedge D)$
9	$ x+z > 0 \wedge b > 9;$ $(A \vee B) = (C \wedge D)$	10	$ x+z > 0 \vee 0 < b < 1;$ $(A = B) \vee (C = D)$

Занятие 9

9.1. Условные операторы

В языке С предусмотрены два условных оператора: if и switch. Кроме того, в некоторых ситуациях в качестве альтернативы условному оператору if можно применять тернарный оператор "?".

Оператор if имеет следующий вид:

if (выражение) оператор;

else оператор;

Здесь оператор может состоять из одного или нескольких операторов (блока) или отсутствовать вовсе (пустой оператор). Раздел else является необязательным.

Если выражение истинно (т.е. не равно нулю), выполняется оператор или блок, указанный в разделе if, в противном случае выполняется оператор или блок, предусмотренный в разделе else. Операторы, указанные в разделах if или else, являются взаимоисключающими.

В языке С результатом условного выражения является скаляр, т.е. целое число, символ, указатель или число с плавающей точкой. Число с плавающей точкой редко применяется в качестве результата условного выражения, входящего в условный оператор, поскольку значительно замедляет выполнение программы (это объясняется тем, что операции над числами с плавающей точкой выполняются медленнее, чем над целыми числами или символами).

Проиллюстрируем применение оператора if с помощью программы, которая представляет собой упрощенную версию игры "угадай волшебное число". Если игрок выбрал задуманное число, программа выводит на экран сообщение ** Верно **. "Волшебное число" генерируется с помощью стандартного датчика псевдослучайных чисел rand(), возвращающего произвольное число, лежащее в диапазоне от 0 до RAND_MAX. Как правило, это число не превышает 32767. Функция rand() объявлена в заголовочном файле stdlib.h.

```
/* Волшебное число. Вариант #1 */
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int magic;          /* Волшебное число */
    int guess;          /* Предположение игрока */
    magic = rand();     /* Генерируем волшебное число */
    printf("Угадайте волшебное число: ");
    scanf("%d", &guess);
```

```
    if(guess == magic) printf("*** Верно ***");
}
```

Следующая версия программы иллюстрирует применение оператора else в случае, если игрок ошибся.

```
/* Волшебное число. Вариант #2 */
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int magic;          /* Волшебное число */
    int guess;          /* Предположение игрока */
    magic = rand();     /* Генерируем волшебное число */
    printf("Угадайте волшебное число: ");
    scanf("%d", &guess);
    if(guess == magic) printf("*** Верно ***");
    else printf("*** Неверно ***");
}
```

9.2. Блок

Блок – это группа связанных между собой операторов, рассматриваемых как единое целое. Операторы, образующие блок, логически связаны друг с другом. Блок иногда называется также *составным оператором*. Блок начинается открывающей фигурной скобкой { и завершается закрывающей фигурной скобкой }. Блок чаще всего используется как составная часть другого оператора, например, условного оператора if. Однако блок может являться и самостоятельной единицей программы.

9.3. Задание

Составить программу вычисления значений функции $y = f(x)$. Функция задана в табл. 9.1. Предусмотреть ошибки.

Таблица 9.1

№	$f(x)$	№	$f(x)$	№	$f(x)$	№	$f(x)$
1	$\begin{cases} \frac{1}{x-1}, & x < 1 \\ \cos x, & x \geq 1 \end{cases}$	2	$\frac{\ln(x-3)}{x}$	3	$\operatorname{ctg} \sqrt[3]{x}$	4	$\frac{x}{1+\operatorname{tg} x}$
5	$\begin{cases} \sin x, & x < -1 \\ \sqrt{x+1}, & x \geq -1 \end{cases}$	6	$\frac{\operatorname{tg}(x+2)}{3x}$	7	$\frac{\operatorname{tg} x}{x-1}$	8	$\frac{x}{1-x^2}$
9	$\begin{cases} \operatorname{tg} x, & x \neq \frac{\pi}{2} + \pi n \\ 1, & x = \frac{\pi}{2} + \pi n \end{cases}$	10	$\begin{cases} \sqrt{1- x }, & x \leq 1 \\ e^x, & x > 1 \end{cases}$				

Занятие 10

10.1. Вложенные операторы if

Вложенным называется оператор if, который находится внутри другого оператора if или else. Вложенные операторы if встречаются довольно часто. Во вложенном условном операторе раздел else всегда связан с ближайшим оператором if, находящимся с ним в одном блоке и не связанным с другим оператором else:

```
if(i)
{
    if(j) оператор1;
    if(k) оператор2; /* данный if */
    else оператор3; /* связан с данным оператором */
}
else оператор4; /* связан с оператором if(i) */
```

Последний раздел else связан не с оператором if(j), который находится в другом блоке, а с оператором if(i). Внутренний раздел else связан с оператором if(k), потому что этот оператор if является ближайшим.

Язык C допускает до 15 уровней вложенности условных операторов. На практике многие компиляторы предусматривают намного большую глубину. Однако на практике глубоко вложенные условные операторы используются крайне редко, поскольку это значительно усложняет логику программы.

Модифицируем программу "Волшебное число" из занятия 9:

```
/* Волшебное число. Вариант #3 */
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int magic; /* Волшебное число */
    int guess; /* Предположение игрока */
    magic = rand(); /* Генерируем волшебное число */
    printf("Угадайте волшебное число: ");
    scanf("%d", &guess);
    if(guess == magic) {
        printf("*** Верно **\n");
        printf(" %d - волшебное число\n", magic);
    }
    else {
        printf("*** Неверно **\n");
        if (guess > magic) printf("Слишком много\n");
        else printf("Слишком мало\n");
    }
}
```

10.2. Цепочка операторов if-then-else

В программах на языке C часто используется конструкция, которая называется цепочка if-then-else, или лестница if-then-else. Общий вид этой конструкции выглядит так:

```
if(выражение) оператор;
else
    if (выражение) оператор;
    else
        if(выражение) оператор;
    ...
    else оператор;
```

Эти условия вычисляются сверху вниз. Как только значение условного выражения становится истинным, выполняется связанный с ним оператор, и оставшаяся часть конструкции игнорируется. Если все условные выражения оказались ложными, выполняется оператор, указанный в последнем разделе else. Если этого раздела нет, то не выполняется ни один оператор.

Поскольку по мере увеличения глубины вложенности количество отступов в строке возрастает, лестницу if-then-else часто записывают иначе:

```
if(выражение)
    оператор;
else if (выражение)
    оператор;
else if(выражение)
    оператор;
...
else
    оператор;
```

Используя цепочку операторов if-then-else, программу для угадывания "волшебного числа" можно переписать следующим образом:

```
/* Волшебное число. Вариант #4 */
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int magic; /* Волшебное число */
    int guess; /* Предположение игрока */
    magic = rand(); /* Генерируем волшебное число */
    printf("Угадайте волшебное число: ");
    scanf("%d", &guess);
    if(guess == magic) {
        printf("*** Верно **\n");
    }
```

```

        printf(" %d - волшебное число\n", magic);
    }
    else if (guess > magic)
        printf("*** Неверно **\nСлишком много\n");
    else printf("*** Неверно **\nСлишком мало\n");
}

```

Напишем программу, которая по введенному номеру месяца выводит его название, используя цепочку операторов if-then-else:

```

#include <stdio.h>
enum month {jan, feb, mar, apr, may, jun, jul, aug,
            sep, oct, nov, dec};

void main()
{
    int num;
    printf("Введите номер месяца: ");
    scanf("%d", &num);
    if (num > 0 && num <= 12) {
        enum month m = num - 1;
        printf("Вы ввели ");
        if (m == jan) printf("январь\n");
        else if (m == feb) printf("февраль\n");
        else if (m == mar) printf("март\n");
        else if (m == apr) printf("апрель\n");
        else if (m == may) printf("май\n");
        else if (m == jun) printf("июнь\n");
        else if (m == jul) printf("июль\n");
        else if (m == aug) printf("август\n");
        else if (m == sep) printf("сентябрь\n");
        else if (m == oct) printf("октябрь\n");
        else if (m == nov) printf("ноябрь\n");
        else printf("декабрь\n");
    }
    else printf("Вы ввели не тот номер месяца\n");
}

```

В условном операторе if, где в качестве оператора используется блок, можно объявлять переменные и использовать их (см. пример программы). При выходе из блока, объявленные в нем переменные автоматически уничтожаются компилятором. Поэтому при их использовании, вне блока, возникнет ошибка.

10.3. Задания

Измените программу из заданий 4 и 5 (о перечислениях) таким образом, чтобы при вводе значения не входящего в перечисление программа об этом сообщала, а при вводе значения входящего в перечисление программа выводила на консоль текстовое значение введенной переменной.

Занятие 11

11.1. Тернарная альтернатива

Вместо операторов if-else можно использовать тернарный оператор "?". Общий вид заменяемых операторов if-else выглядит следующим образом:

if (условие) выражение; else выражение;

Однако в данном случае с операторами if и else связаны отдельные выражения, а не операторы.

Оператор "?" называется тернарным, поскольку имеет три операнда. Его общий вид таков:

Выражение1 ? Выражение2 : Выражение3

Обратите внимание на использование и местоположение двоеточия.

Оператор "?" выполняется следующим образом. Сначала вычисляется *Выражение1*. Если оно является истинным, вычисляется *Выражение2*, и его значение становится значением всего тернарного оператора. Если *Выражение1* является ложным, вычисляется *Выражение3*, и результатом выполнения тернарного оператора считается именно его значение. Рассмотрим пример:

x = 10;

y = x > 9 ? 100 : 200;

В данном случае переменной y присваивается значение 100. Если бы переменная x была меньше 9, то переменная y получила бы значение 200. Аналогично:

x = 10;

if(x > 9) y = 100;

else y = 200;

В следующем примере оператор "?" используется для "возведения в квадрат с сохранением знака" числа, введенного пользователем (например, 10 в квадрате равно 100, а -10 в квадрате равно -100):

```

#include <stdio.h>
void main()
{
    int isqrd, i;
    printf("Введите число: ");
    scanf("%d", &i);
    isqrd = i > 0 ? i*i : -(i*i);
    printf("%d в квадрате равно %d", i, isqrd);
}

```

Тернарный оператор можно использовать вместо конструкции if-else не только для присвоения значений. Как известно, все функции возвращают какое-либо значение (кроме функций, возвращающих значение типа void). Следовательно, вместо выраже-

ний в операторе "?" можно использовать вызовы функций. Если в операторе "?" встречается имя функции, она вызывается, а ее результат используется вместо значения соответствующего выражения. Это означает, что, используя вызовы функций в качестве операндов тернарного оператора, можно выполнить одну или несколько функций сразу.

Используя тернарный оператор, перепишите программу для угадывания "волшебного числа". В этой программе оператор "?" в зависимости от результата проверки должен выводить на экран соответствующие сообщения.

11.2. Условное выражение

Иногда начинающих программистов на языке C приводит в смятение тот факт, что в качестве условных выражений в операторах if или "?" можно использовать любые допустимые выражения. Иначе говоря, условное выражение не обязано состоять из операторов сравнения или логических операторов, как в языках BASIC и Pascal. Значением выражения может быть любое число, которое в зависимости от своего значения интерпретируется как "истина" или "ложь". Приведенный ниже фрагмент программы считывает с клавиатуры два целых числа и выводит на экран их частное.

```
/* Деление первого числа на второе */
#include <stdio.h>
void main()
{
    int a, b;
    printf("Введите два числа: ");
    scanf("%d%d", &a, &b);
    if(b) printf("%d\n", a/b);
    else printf("Делись на нуль нельзя.\n");
}
```

Если число b равно нулю, то условие оператора if является ложным, и выполняется оператор else. В противном случае условие является истинным, и выполняется деление.

Оператор if можно было бы записать иначе:

```
if(b != 0) printf("%d\n", a/b);
```

Однако следует заметить, что последняя запись избыточна, может снизить эффективность вычислений и поэтому считается признаком плохого стиля программирования. Для проверки условия оператора if достаточно значения самой переменной b и нет никакой необходимости сравнивать ее с нулем.

11.3. Задание

Напишите программу из задания 10 с тернарным оператором.

Занятие 12

12.1. Оператор switch

В языке C предусмотрен оператор многовариантного ветвления switch, который последовательно сравнивает значение выражения со списком целых чисел или символьных констант. Если обнаруживается совпадение, выполняется оператор, связанный с соответствующей константой. Оператор switch имеет следующий вид:

```
switch (выражение)
{
    case константа1:
        последовательность операторов
        break;
    case константа2:
        последовательность операторов
        break;
    case константа3:
        последовательность операторов
        break;
    ...
    default:
        последовательность операторов
}
```

Значением *выражения* должен быть символ или целое число. Например, выражения, результатом которых является число с плавающей точкой, не допускаются. Значение выражения последовательно сравнивается с константами, указанными в операторах case. Если обнаруживается совпадение, выполняется последовательность операторов, связанных с данным оператором case, пока не встретится оператор break или не будет достигнут конец оператора switch. Если значение выражения не совпадает ни с одной из констант, выполняется оператор default. Этот раздел оператора switch является необязательным. Если он не предусмотрен, в отсутствие совпадений не будет выполнен ни один оператор.

В языке C оператор switch допускает до 257 операторов case! На практике количество разделов case в операторе switch следует ограничивать, поскольку оно влияет на эффективность программы. Оператор case используется только внутри switch.

Оператор break относится к группе операторов перехода. Его можно использовать как в операторе switch, так и в циклах (см. занятие 20). Когда поток управления достигает оператора break, программа выполняет переход к оператору, следующему за оператором switch.

Следует знать три важных свойства оператора switch.

- Оператор switch отличается от оператора if тем, что значение его выражения сравнивается исключительно с константами, в то время как в операторе if можно выполнять какие угодно сравнения или вычислять любые логические выражения.
- Две константы в разных разделах case не могут иметь одинаковых значений, за исключением случая, когда один оператор switch вложен в другой.
- Если в операторе switch используются символьные константы, они автоматически преобразовываются в целочисленные.

Оператор switch часто используется для обработки команд, введенных с клавиатуры, например, при выборе пунктов меню.

С формальной точки зрения наличие оператора break внутри оператора switch не обязательно. Этот оператор прерывает выполнение последовательности операторов, связанных с соответствующей константой. Если его пропустить, будут выполнены все последующие операторы case, пока не встретится следующий оператор break, либо не будет достигнут конец оператора switch.

В приведенном ниже примере при вводе номера месяца выводится, к какой поре года он относится. Приведенная программа использует эффект "сквозного" выполнения ветвей case (пропуска оператора break):

```
#include <stdio.h>
enum month {jan, feb, mar, apr, may, jun, jul, aug,
            sep, oct, nov, dec};

void main()
{
    int num;
    printf("Введите номер месяца: ");
    scanf("%d", &num);
    if (num > 0 && num <= 12) {
        enum month m = num - 1;
        printf("Вы ввели ");
        switch (m)
        {
            case dec: case jan: case feb:
                printf("зимний"); break;
            case mar: case apr: case may:
                printf("весенний"); break;
            case jun: case jul: case aug:
                printf("летний"); break;
            default:
                printf("осенний");
        }
    }
}
```

```
    }
    printf(" месяц\n");
}
else printf("Вы ввели не тот номер месяца\n");
}
```

"Сквозное" выполнение ветвей case позволяет несколько ветвей case объединить в одну.

Рекомендуется в конце последней ветви помещать оператор break, хотя с точки зрения логики в ней нет необходимости. Но эта маленькая предосторожность спасет, когда однажды потребуется добавить в конец еще одну ветвь case.

В операторе case также как и в операторе if могут быть описаны переменные, но их инициализация не будет выполняться:

```
switch (x)
{
    int i = 1;           /* i не присвоится значение 1 */
    case 1:
        i = 2;           /* если x==1, то i = 2 */
    case 2:
        printf( "%d\n", i); /* и выводится на экран */
}
```

12.2. Вложенные операторы switch

Операторы switch могут быть вложены друг в друга. Даже если константы разделов case внешнего и внутреннего операторов switch совпадают, проблемы не возникают. Например, приведенный ниже фрагмент программы является вполне приемлемым:

```
switch (x)
{
    case 1:
        switch(y)
        {
            case 0:
                printf( "Деление на нуль.\n");
                break;
            case 1:
                process(x,y);
        }
        break;
    case 2:
        ...
}
```

12.3. Задание

Напишите программу из задания 10 с оператором switch.

Занятие 13

13.1. Операторы цикла

В языке C, как и во всех других современных языках программирования, операторы цикла предназначены для выполнения повторяющихся инструкций, пока действует определенное правило. Это условие может быть как задано заранее (в цикле `for`), так и меняться во время выполнения цикла (в операторах `while` и `do-while`).

13.2. Цикл `for`

В том или ином виде цикл `for` есть во всех процедурных языках программирования. Однако в языке C он обеспечивает особенно высокую гибкость и эффективность.

Общий вид оператора `for` таков:

`for` (инициализация; условие; приращение)

Цикл `for` имеет много вариантов. Однако наиболее общая форма этого оператора работает следующим образом. Сначала выполняется *инициализация* – оператор присваивания, который задает начальное значение счетчика цикла. Затем проверяется *условие*, представляющее собой условное выражение. Цикл выполняется до тех пор, пока значение этого выражения остается истинным. *Приращение* изменяет значение счетчика цикла при очередном его выполнении. Эти разделы оператора отделяются друг от друга точкой с запятой. Как только условие цикла станет ложным, программа прекратит его выполнение и перейдет к следующему оператору.

В следующем примере цикл `for` выводит на экран числа от 1 до 100:

```
#include <stdio.h>
void main()
{
    int x;
    for (x = 1; x <= 100; x++)
        printf("%d ", x);
}
```

Сначала переменной `x` присваивается число 1, а затем она сравнивается с числом 100. Поскольку ее значение меньше 100, вызывается функция `printf()`. Затем переменная `x` увеличивается на единицу, и условие цикла проверяется вновь. Как только ее значение превысит число 100, выполнение цикла прекратится. В данном случае переменная `x` является счетчиком цикла, который изменяется и проверяется на каждой итерации.

Рассмотрим пример цикла `for`, тело которого состоит из не-

скольких операторов:

```
for (x = 100; x != 65; x -= 5) {
    z = x*x;
    printf("Квадрат числа %d равен %f", x, z);
}
```

Возведение числа `x` в квадрат и вызов функции `printf()` выполняются до тех пор, пока значение переменной `x` не станет равным 65. Обратите внимание на то, что в этом цикле счетчик уменьшается: сначала ему присваивается число 100, а затем на каждой итерации из него вычитается число 5.

В цикле `for` проверка условия выполняется перед каждой итерацией. Иными словами, если условие цикла с самого начала является ложным, его тело не будет выполнено ни разу. Рассмотрим пример:

```
x = 10;
for(y = 10; y != x; ++y)
    printf("%d", y);
printf("%d", y); /* Это единственный вызов функции
                  printf(), который выполняется в
                  данном фрагменте */
```

Этот цикл никогда не будет выполнен, поскольку значения переменных `x` и `y` при входе в цикл равны. Следовательно, условие цикла является ложным, и ни тело цикла, ни приращение счетчика выполняться не будут. Таким образом, значение переменной `y` останется равным 10, и именно оно будет выведено на экран.

Пример табуляции функции `sin x` на интервале `[0; 1]`:

```
#include <stdio.h>
#include <math.h>
const int N = 10;
void main()
{
    double a = 0, b = 1, h = (b - a)/N, x;
    int i;
    printf("    X        Y\n");
    for (i = 0; i <= N; i++) {
        x = a + i*h;
        printf("%4.2f%9.5f\n", x, sin(x));
    }
}
```

13.3. Задание

Протабулировать функцию из задания 9 на интервале `[a; b]`. Интервал разбить на 10 частей. Начало и конец интервала ввести с клавиатуры.

Занятие 14

Варианты цикла for

Оператор for имеет несколько вариантов, повышающих его гибкость и эффективность.

Наиболее распространенным является вариант, в котором используется оператор последовательного выполнения ("запятая"), что позволяет применять несколько счетчиков цикла одновременно. Напомним, что оператор последовательного выполнения связывает между собой несколько операторов, вынуждая их выполняться друг за другом (занятие 6). Например, переменные x и y являются счетчиками приведенного ниже цикла. Их инициализация выполняется в одном и том же разделе цикла:

```
for(x = 0, y = 0; x + y < 10; ++x) {
    y = getchar();
    y = y - '0';          /* Вычесть из переменной y
                           ASCII-код нуля */
}
```

Как видим, два оператора инициализации разделены запятой. При каждой итерации значение переменной x увеличивается на единицу, а переменная y вводится с клавиатуры. Несмотря на это, переменная y должна иметь какое-то начальное значение, иначе перед первой итерацией цикла условие может оказаться ложным.

Условное выражение не обязательно связано с проверкой счетчика цикла. В качестве условия цикла может использоваться любой допустимый оператор сравнения или логический оператор. Это позволяет задавать несколько условий цикла одновременно.

Рассмотрим программу, которая ждет от пользователя нажатия клавиши <Esc>. Пользователю дается пять попыток:

```
#include <stdio.h>
void main()
{
    char c;
    int num;
    printf("Нажмите клавишу Esc\n");
    for (num = 1, c = 0; num <= 5 && c != 27; num++) {
        printf("\tпопытка %d: ", num);
        c = getch();
        printf("\n");
    }
    if (c == 27)
        printf("Вы нажали Esc с %d попытки!\n", --num);
    else printf("Вы так и не нажали Esc!\n");
}
```

Каждый из трех разделов цикла for может состоять из любых допустимых выражений. Эти выражения могут быть никак не связаны с предназначением разделов. Учитывая вышесказанное, рассмотрим следующий пример:

```
#include <stdio.h>
void main()
{
    int t, S = 0;
    for (printf("Введите число: "); scanf("%d",&t), t;

        S +=t;
        printf("Сумма чисел равна %d\n",S);
    }
```

Обратите внимание на цикл for в функции. В разделе инициализации выводится сообщение предлагающее пользователю ввести число. В разделе условия программа ждет ввода числа и возвращает это число. Таким образом, условием завершения цикла будет ввод числа 0, которое интерпретируется компилятором как ложь. В разделе приращения счетчика выводится приглашение к вводу следующего числа. В теле цикла вычисляется сумма введенных чисел.

Другая интересная особенность цикла for заключается в том, что его разделы можно пропускать. Каждый из его разделов является необязательным. Например, цикл, приведенный ниже, выполняется до тех пор, пока пользователь не введет число 123:

```
for (x = 0; x != 123; )
    scanf("%d", &x);
```

Обратите внимание на то, что раздел приращения счетчика в данном цикле for отсутствует. Это значит, что при каждой итерации значение переменной x сравнивается с числом 123, и никакие действия с ней больше не выполняются. Однако, если пользователь введет с клавиатуры число 123, условие цикла станет ложным, и программа прекратит его выполнение.

Счетчик можно инициализировать вне цикла for. Этим способом пользуются, когда начальное значение счетчика является результатом сложных вычислений, как в следующем примере:

```
ch = getch(); /* Считать символ */
if (ch >= '0' && ch <= '9') x = ch - '0';
else x = 10;
for ( ; x < 10; ) {
    printf("%d", x);
    ++x;
}
```

Здесь раздел инициализации оставлен пустым, а переменная x инициализируется до входа в цикл.

Занятие 15

Бесконечный цикл

Хотя в качестве бесконечного можно использовать любой цикл, традиционно для этой цели применяется оператор `for`. Поскольку все разделы оператора `for` являются необязательными, его легко сделать бесконечным, не задав никакого условного выражения.

```
for( ; ; )
    printf("Этот цикл выполняется бесконечно.\n");
```

Если условное выражение не указано, оно считается истинным. Разумеется, в этом случае можно по-прежнему выполнять инициализацию и приращение счетчика, однако программисты на языке C в качестве бесконечного цикла чаще всего используют конструкцию `for (; ;)`.

На самом деле конструкция `for(; ;)` не гарантирует бесконечное выполнение цикла, поскольку его тело может содержать оператор `break`, приводящий к немедленному выходу (детальнее этот оператор будет рассмотрен на занятии 20). В этом случае программа передаст управление следующему оператору, находящемуся за пределами тела цикла `for`, как показано ниже:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    double a = 6, b = 3;
    char ch;
    int sign;
    for ( ; ; ) {
        printf("Введите операцию (+ - * /): ");
        ch = getch();
        sign = ch == '+' || ch == '-' || ch == '*' ||
              ch == '/';
        if (ch == 27) break;
        if (sign) {
            printf("%f %c %f = ", a, ch, b);
            switch (ch) {
                case '+': printf("%f", a+b); break;
                case '-': printf("%f", a-b); break;
                case '*': printf("%f", a*b); break;
                case '/': printf("%f", a/b); break;
            }
        }
        printf("\n");
    }
}
```

Занятие 16

16.1. Пустой цикл `for`

Оператор может быть пустым. Это значит, что тело цикла `for` (как и любого другого цикла) может не содержать ни одного оператора. Этот факт можно использовать для повышения эффективности некоторых алгоритмов и задержки выполнения программы.

Преобразуем программу, суммирующую числа введенные пользователем, таким образом, чтобы она содержала пустой цикл. В разделе инициализации выводится сообщение, предлагающее пользователю ввести число, и сумме присваивается 0. В разделе условия программа ждет ввода числа и возвращает это число. В разделе приращения счетчика суммируются числа и выводится приглашение к вводу следующего числа:

```
#include <stdio.h>
void main()
{
    int t, S = 0;
    for (printf("Введите число: "), S = 0;
        scanf("%d",&t), t;
        S += t, printf("\tеще: "));
    printf("Сумма чисел равна %d\n", S);
}
```

В качестве пустого тела цикла используется пустой оператор `(";")`. Иногда он также называется *фиктивным*.

Циклы часто используются для задержки выполнения программы. Ниже показано, как этого можно достичь, используя оператор `for`.

```
for (t = 0; t < SOME_VALUE; t++);
```

16.2. Задание

Написать программу нахождения суммы или произведения (табл. 16.1) введенных пользователем с клавиатуры чисел. Признаком окончания ввода служит введение в качестве числа нуля.

Таблица 16.1

		Сумма	Произведение
Вариант	1	6	положительных чисел
	2	7	отрицательных чисел
	3	8	четных чисел
	4	9	нечетных чисел
	5	10	чисел кратных 5

Занятие 17

17.1. Цикл while

Второй по значимости цикл в языке C – оператор while. Он имеет следующий вид:

while (условие) оператор;

Здесь оператор может быть пустым, отдельным оператором или блоком операторов. Условие может задаваться любым выражением. Условие цикла считается истинным, если значение этого выражения не равно нулю. Как только условие цикла становится ложным, программа передает управление оператору, стоящему сразу после оператора while. Рассмотрим пример программы, обрабатывающей ввод с клавиатуры, которая терпеливо ждет, пока пользователь не введет букву A:

```
char ch;
ch = '\0'; /* начальное значение переменной ch */
while(ch != 'A') ch = getchar();
```

Сначала переменной ch присваивается ноль. Затем цикл while проверяет, не равно ли значение переменной ch символу A. Поскольку начальное значение переменной ch равно нулю, условие цикла является истинным, и его выполнение продолжается. Условие цикла проверяется каждый раз, когда пользователь вводит символ с клавиатуры. Как только он введет букву A, условие цикла станет ложным, и программа прекратит его выполнение.

Как и цикл for, цикл while проверяет свое условие перед началом выполнения тела. Следовательно, если условие цикла с самого начала является ложным, его тело не будет выполнено ни разу. Благодаря этому свойству нет необходимости отдельно проверять условие цикла перед входом в него.

Какой оператор цикла for или while выбрать – дело вкуса, поскольку их конструкции эквивалентны:

```
– оператор for
for (выражение1; выражение2; выражение3)
    оператор;
– оператор while
выражение1;
while (выражение2) {
    оператор;
    выражение3;
}
```

Там, где есть простая инициализация и пошаговое увеличение значения некоторой переменной, больше подходит цикл for, так как в этом цикле организующая его часть сосредоточена в начале записи. Но тем не менее для этого можно использовать и

цикл while. Переделаем программу табуляции функции $\sin x$ на интервале $[0; 1]$ из занятия 13:

```
#include <stdio.h>
#include <math.h>
const int N = 10;
void main()
{
    double a = 0, b = 1, h = (b - a)/N, x;
    int i = 0;
    printf("      X          Y\n");
    while (i <= N) {
        x = a + i*h;
        printf("%4.2f%9.5f\n", x, sin(x));
        i++;
    }
}
```

Если выход из цикла while зависит от нескольких условий, обычно в качестве условия цикла используют отдельную переменную, значение которой изменяется несколькими операторами в разных точках цикла. Рассмотрим пример.

```
int working;
working = 1; /* т.е., истина */
while(working) {
    working = process1();
    if(working) working = process2();
    if(working) working = process3();
}
```

Каждая функция, вызванная в этом фрагменте программы, может вернуть ложное значение и привести к выходу из цикла.

Тело цикла while может быть пустым. Например, приведенный ниже пустой цикл выполняется до тех пор, пока пользователь не введет букву A:

```
while ((ch = getchar()) != 'A') ;
```

Оператор присваивания внутри условного выражения цикла while выглядит непривычно, но все встанет на свои места, если вспомнить, что его значением является значение операнда, расположенного в правой части.

17.2. Задание

1. В программе из задания 13 заменить оператор цикла for на оператор цикла while. Объяснить чем отличаются эти операторы цикла.
2. В программе из задания 16 заменить оператор цикла for на оператор цикла while.

Занятие 18

18.1. Цикл do-while

В отличие от операторов for и while, которые проверяют условие в начале цикла, оператор do-while делает это в конце. Иными словами, цикл do-while выполняется по крайней мере один раз. Общий вид оператора do-while таков:

```
do {
    оператор;
} while (условие);
```

Если тело цикла состоит лишь из одного оператора, фигурные скобки не обязательны, хотя они делают этот цикл понятнее (программисту, но не компилятору). Цикл do-while повторяется до тех пор, пока условие не станет ложным.

Рассмотрим пример, в котором цикл do-while считывает числа с клавиатуры, пока они не превысят 100:

```
do {
    scanf("%d", &num);
} while(num > 100);
```

Чаще всего оператор do-while применяется для выбора пунктов меню:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    double a = 6, b = 3;
    char ch;
    printf("Введите операцию (+ - * /):\n");
    do {
        ch = getch();
        switch (ch) {
            case '+' : printf("%f %c %f = %f\n",
                               a, ch, b, a+b); break;
            case '-' : printf("%f %c %f = %f\n",
                               a, ch, b, a-b); break;
            case '*' : printf("%f %c %f = %f\n",
                               a, ch, b, a*b); break;
            case '/' : printf("%f %c %f = %f\n",
                               a, ch, b, a/b); break;
        }
    } while (ch != 27);
}
```

18.2. Задание

Переделать задания 17 с оператором do-while.

Занятие 19

Объявление переменных в операторах

В языке C (но не в стандарте C89) можно объявлять переменные внутри условных выражений, входящих в операторы if или switch, внутри условия цикла while, а также в разделе инициализации цикла for. Область видимости переменной, объявленной в одном из этих трех мест, ограничена блоком, который относится к данному оператору. Например, переменная, объявленная внутри цикла for, является локальной по отношению к нему.

Рассмотрим пример, в котором переменная объявлена в разделе инициализации цикла for:

```
/* Переменная i является локальной по отношению к
   циклу for, а переменная j существует как внутри,
   так и вне цикла */
int j;
for(int i = 0; i<10; i++)
    j = i*i;
/* i = 10; // Ошибка - переменная i здесь невидима! */
```

В этом примере переменная i объявлена внутри раздела инициализации цикла for и используется в качестве счетчика. Вне цикла for эта переменная не существует.

Поскольку счетчик нужен лишь внутри цикла, его объявление в разделе инициализации оператора for стало общепринятой практикой. Однако учтите, что стандартом C89 это не допускается (в стандарте C99 это ограничение снято).

Утверждение, что переменная, объявленная в разделе инициализации цикла for, является локальной, в некоторых компиляторах не выполняется. В старых версиях такие переменные были доступны и вне оператора for. Однако стандарт C99 ограничил область видимости таких переменных телом оператора for.

Если ваш компилятор полностью поддерживает стандарт C99 языка C, то переменные можно объявлять не только внутри циклов, но и внутри обычных условных выражений. Рассмотрим пример.

```
if (int x = 20) {
    x -= y;
    if (x>10) y = 0;
}
```

Здесь объявляется переменная x, которой присваивается число 20. Поскольку это значение считается истинным, выполняется блок, связанный с оператором if. Область видимости переменной ограничивается этим блоком. Таким образом, вне оператора if переменной x не существует. Следует сказать, что такой способ объявления переменных не все считают приемлемым.

Занятие 20

20.1. Функция exit

Функция `exit()` не относится к управляющим операторам. Вызов стандартной библиотечной функции `exit()` приводит к прекращению работы программы и передаче управления операционной системе. Ее эффект можно сравнить с катапультированием из программы.

Функция `exit()` выглядит следующим образом:

```
void exit(int код_возврата);
```

Значение переменной `код_возврата` передается вызывающему процессу, в роли которого чаще всего выступает операционная система. Нулевое значение кода возврата соответствует нормальному завершению работы. Другие значения аргумента указывают на вид ошибки. В качестве кода возврата можно применять макросы `EXIT_SUCCESS` и `EXIT_FAILURE`. Для вызова функции `exit()` необходим заголовочный файл `stdlib.h`.

Функция `exit()` часто используется, когда обязательное условие, гарантирующее правильную работу программы, не выполняется. Рассмотрим, например, виртуальную компьютерную игру, для которой требуется специальный графический адаптер. Функция `main()` в этой программе может выглядеть следующим образом.

```
#include <stdlib.h>
void main()
{
    if(!virtual_graphics()) exit(1);
    play();
    /* ... */
}
/* ... */
```

Здесь функция `virtual_graphics()` определяется пользователем и возвращает истинное значение, когда в компьютере есть необходимый графический адаптер. Если же его нет, она возвращает ложное значение, и функция `exit()` прекращает работу программы.

20.2. Оператор break

Оператор `break` применяется в двух ситуациях. Во-первых, он используется для прекращения выполнения раздела `case` внутри оператора `switch`. Во-вторых, с помощью оператора `break` можно немедленно выйти из цикла независимо от истинности или ложности его условия.

Если в цикле встречается оператор `break`, итерации прекращаются и выполнение программы возобновляется с оператора, следующего за оператором цикла. Рассмотрим пример:

```
#include <stdio.h>
void main()
{
    int t;
    for(t = 0; t < 100; t++) {
        printf("%d ", t);
        if (t == 10) break;
    }
}
```

Эта программа выводит на экран числа от 0 до 10, а затем цикл прекращается, поскольку выполняется оператор `break`. Условие `t < 100` при этом игнорируется.

Оператор `break` часто применяется в циклах, выполнение которых следует немедленно прекратить при наступлении определенного события.

Если циклы вложены друг в друга, оператор `break` выполняет выход из внутреннего цикла во внешний. Например, приведенный ниже фрагмент программы 100 раз выводит на экран числа от 1 до 10, причем каждый раз, когда счетчик достигает значения 10, оператор `break` передает управление внешнему циклу `for`:

```
for (t = 0; t < 100; ++t) {
    count = 1;
    for(;;) {
        printf("%d ", count);
        count++;
        if (count == 10) break;
    }
}
```

Если оператор `break` содержится внутри оператора `switch`, который вложен в некий цикл, то выход будет осуществлен только из оператора `switch`, а управление останется во внешнем цикле.

20.3. Оператор continue

Оператор `continue` напоминает оператор `break`. Они различаются тем, что оператор `break` прекращает выполнение всего цикла, а оператор `continue` – лишь его текущей итерации, вызывая переход к следующей итерации и пропуская все оставшиеся операторы в теле цикла. В цикле `for` оператор `continue` вызывает проверку условия и приращение счетчика цикла. В циклах `while` и `do-while` оператор `continue` передает управление операторам, входящим в условие цикла.

В следующем примере оператор `continue` выполняет выход из цикла `while`, передавая управление оператору, входящему в условие цикла:

```

char done, ch;
done = 0;
while(!done) {
    ch = getchar();
    if (ch == '$') {
        done = 1;
        continue;
    }
    putchar(ch + 1); /* Перейти к следующей букве ал-
                     * фавита */
}

```

Фрагмент программы кодирует сообщение, прибавляя единицу к коду каждого символа. Например, в сообщении на английском языке буква А заменяется буквой В и т.д. Фрагмент программы прекращает свою работу, если пользователь ввел символ \$. После этого вывод сообщений на экран прекращается, поскольку переменная done принимает истинное значение, и, соответственно, условие цикла становится ложным.

20.4. Разбор числа на цифры

Пусть дано произвольное целое число. Необходимо все его цифры вывести на экран в обратном порядке.

Для решения данной задачи необходимо взять из числа последнюю цифру (% 10) и вывести ее на экран. Затем необходимо взять число без последней цифры (/ 10) и повторить действия. Действия необходимо повторять пока от числа ничего не останется. Таким образом:

```

scanf("%d", &num);
while (num) {
    printf("%d", num % 10);
    num /= 10;
}

```

20.5. Задание

Дано произвольное целое число. Выполнить над его цифрами такие действия (табл. 20.1) и вывести новое число на экран.

Таблица 20.1

№	Действие	№	Действие
1	Оставить цифры кратные 3	6	Удалить цифры кратные 3
2	Большие 5 поделить на 2	7	К четным прибавить 1
3	Удалить нечетные цифры	8	Оставить четные цифры
4	Из нечетных вычесть 1	9	Меньшие 5 умножить на 2
5	Оставить нечетные цифры	10	Удалить четные цифры

Занятие 21

21.1. Отладка приложения

Отладка является существенно важной частью процесса программирования. Если программа выполняется до конца, но при этом делает не то, что для нее планировалось, нужно выяснить, что же происходит на самом деле. Для этого придется обратиться к отладчику.

В сущности, самым важным понятием при отладке является точка прерывания (breakpoint). Точка прерывания – это место в программе, в котором вы хотели бы остановиться. Возможно, вы не знаете, сколько раз должен выполняться цикл, пока управление не передается конкретному оператору if, или пока не будет вызвана некоторая функция. Установка точки прерывания на определенной строке кода заставляет программу приостановить свое выполнение как раз перед обработкой этой строки. В этом месте вы можете прервать программу и запустить ее снова или же проследовать по тексту программы построчно. Вы можете также узнать значения некоторых переменных или проверить стек вызовов для того, чтобы увидеть, каким образом управление ходом выполнения программы было передано на этот оператор. Часто вы обнаруживаете ошибку в точке прерывания и тут же ее исправляете.

Чтобы продолжить выполнение программы, вы можете использовать следующие команды:

- Continue – продолжить выполнение программы до следующей точки прерывания, или, если таковых не встретится, до ее конца;
- Restart – запустить выполнение программы с самого начала;
- Step Over – выполнить только следующий оператор и затем остановиться. Если оператор представляет собой вызов функции, то выполнить всю функцию и остановиться после ее завершения;
- Step Into – выполнить только следующий оператор, но если он окажется вызовом функции, войти в нее и остановиться на первом же операторе внутри функции;
- StepOut – выполнить всю оставшуюся часть функции и остановиться на первом же операторе вызывающей функции;
- Run To Cursor – начать выполнение программы и остановиться на строке, на которую указывает курсор.

Большая часть необходимой вам информации, имеющейся в распоряжении отладчика, оформлена в виде новых окон. Visual Studio имеет мощный отладчик с богатым пользовательским интерфейсом. Он содержит команды меню, пиктограммы панели инструментов и окна, которые используются только при отладке.

21.2. Команды меню

Когда вы приступите к отладке, в меню Debug появятся новые пункты, список которых представлен ниже:

- Windows ⇒ Breakpoints (Окна ⇒ Точки прерывания);
- Windows ⇒ Running Documents (Окна ⇒ Открытые документы);
- Windows ⇒ Watch ⇒ Watch 1-4 (Окна ⇒ Просмотр значения выражения или переменной ⇒ 1-4);
- Windows ⇒ Autos (Окна ⇒ Просмотр переменных, использующихся в текущем и предыдущем операторах);
- Windows ⇒ Locals (Окна ⇒ Просмотр переменных, локальных в текущем контексте);
- Windows ⇒ This (Окна ⇒ Просмотр переменных объекта, ассоциированного с текущим методом);
- Windows ⇒ Immediate (Окна ⇒ Окно немедленного выполнения команд);
- Windows ⇒ Call Stack (Окна ⇒ Стек вызовов);
- Windows ⇒ Threads (Окна ⇒ Потоки);
- Windows ⇒ Modules (Окна ⇒ Модули);
- Windows ⇒ Memory ⇒ Memory 1-4 (Окна ⇒ Память ⇒ 1-4);
- Windows ⇒ Disassembly (Окна ⇒ Дизассемблирование);
- Windows ⇒ Registers (Окна ⇒ Регистры);
- Continue (Продолжить);
- Break All (Прервать все);
- Stop Debugging (Остановить отладку);
- Detach All (Отсоединить все);
- Restart (Перезапустить);
- Apply Code Changes (Применить изменения кода);
- Processes... (Процессы...);
- Exceptions... (Исключения...);
- Step Into (Начать пошаговое выполнение функции);
- Step Over (Пропустить выполнение функции в пошаговом режиме);
- Step Out (Закончить выполнение функции в пошаговом режиме);
- QuickWatch... (Средство быстрого просмотра);
- New Breakpoint (Новая точка прерывания);
- Clear All Breakpoints (Удалить все точки прерывания);
- Disable All Breakpoints (Заблокировать точки прерывания);
- Save Dump As (Сохранить "снимок" памяти как).

Чтобы понять эти пункты меню, нужно познакомиться с окнами отладки и уяснить, что такое точка прерывания и окно просмотра.

Занятие 22

22.1. Установка точек прерывания

Вероятно, самым простым способом установки точки прерывания является помещение курсора на оператор программы, перед выполнением которого вы желаете остановиться. Собственно установка точки прерывания выполняется с помощью клавиши <F9> или щелчка на серой границе окна редактора кода. Точка прерывания будет отмечена красным маркером, расположенным слева от оператора на серой границе окна редактора кода.

Выполнив команду Debug ⇒ Windows ⇒ Breakpoints, вы увидите диалоговое окно установки и управления простыми или условными точками прерывания. Например, вам может понадобиться остановиться в момент, когда изменится значение определенной переменной. Поиск в программе строк, которые изменяют значение этой переменной и установка на них точек прерывания – это очень утомительное занятие. Вместо этого выполните команду Debug ⇒ New Breakpoint или щелкните на кнопке New, расположенной на мини-панели инструментов окна Breakpoints, и переключитесь на вкладку Data (Данные) диалогового окна New Breakpoint.

В поле Variable (Переменная) введите имя переменной, а в поле Context (Контекст) – ограниченное фигурными скобками имя функции, в которой значение этой переменной может измениться. Когда значение переменной изменится, вам будет сообщено о причине остановки выполнения программы, после чего вы сможете проанализировать исходный код и переменные.

Вы можете также установить условные точки прерывания. Например, выполнение программы может быть остановлено на конкретной строке в случае, если значение переменной *i* превысит 100, что при стократном выполнении тела цикла избавит вас от необходимости щелкать на кнопке Continue 100 раз.

22.2. Анализ значений переменных

Когда вы установите точку прерывания и запустите отладку программы, выполнение будет происходить в обычном режиме до тех пор, пока не будет достигнута точка прерывания. После этого на сцене появляется Visual Studio. Она выводит поверх окна приложения несколько дополнительных окон и отображает желтую стрелку возле красного маркера, отмечающего текущую точку прерывания.

Переместите курсор мыши на имя переменной. Около курсора появится подсказка, содержащая значение этой переменной. Вы можете проверить значение скольких угодно переменных, за-

тем продолжить выполнение программы и проверить их снова. Следует отметить, что существуют и другие способы проверки значений переменных. Наиболее известными из них являются использование окна быстрого просмотра QuickWatch, окна просмотра Watch и окна просмотра Local.

Окно QuickWatch показывает значения переменных или выражений, предоставляя более детальную информацию, чем подсказка. Для вызова окна QuickWatch щелкните на переменной (или воспользуйтесь каким-либо другим способом перемещения курсора) и выполните команду Debug ⇒ QuickWatch.

Окно просмотра Watch подобно окну быстрого просмотра QuickWatch за исключением того, что оно показывает значения только тех переменных и выражений, которые вы в него добавите собственноручно, к тому же, это окно остается доступным и во время выполнения приложения.

Окно Locals расположено в левом нижнем углу. В него не нужно добавлять переменные, поскольку это окно показывает все переменные, локальные для функции, которая выполняется в текущий момент. В окне Locals отображается больше информации, чем в окне Watch, что иногда затрудняет его использование.

22.3. Пошаговое выполнение программы

После просмотра всех желаемых переменных пора двигаться дальше. Несмотря на то, что в меню Debug имеются команды Step Over, Step Into и т.д., большинство разработчиков используют кнопки панели инструментов или горячие клавиши. Задержите указатель мыши на каждой кнопке для того, чтобы увидеть, какие команды с ними связаны. Например, кнопка с изображением стрелки, направленной внутрь абзаца с отступом, означает команду Step Into, которой соответствует горячая клавиша <F11>.

С помощью базовых средств отладки можно быстро обнаружить ошибки и понять как работают другие аналогичные программы.

Отладчик Visual Studio является самым полезным средством, интегрированным в эту среду разработки. Никакое добавление операторов вывода не может соревноваться с простотой остановки выполнения программы и анализа состояния переменных.

Для овладения техникой отладки рекомендуется начать с установок точек прерывания и пошагового наблюдения за работой программы. Далее можно переходить к установке условных точек прерывания или изменению значений переменных во время выполнения программы. Таким образом, Вы наработаете навыки поиска и устранения ошибок в программе в считанные секунды.

Занятие 23

23.1. Массивы

Массив (array) – это совокупность переменных, имеющих одинаковый тип и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса. Согласно правилам языка C все массивы состоят из смежных ячеек памяти. Младший адрес соответствует первому элементу массива, а старший – последнему.

Массивы могут быть одномерными и многомерными. Наиболее распространенным массивом является строка, завершающаяся нулевым байтом. Она представляет собой обычный массив символов, последним элементом которого является нулевой байт.

Массивы и указатели тесно связаны между собой. Трудно описывать массивы, не упоминая указатели, и наоборот. На этом занятии будут рассмотрены массивы, а указатели рассмотрим на занятии 45.

23.2. Декларация одномерных массивов

Объявление одномерного массива выглядит следующим образом:

```
тип имя_переменной[размер]
```

Как и другие переменные, массив должен объявляться явно, чтобы компилятор мог выделить память для него. Здесь тип объявляет базовый тип массива, т.е. тип его элементов, а размер определяет, сколько элементов содержится в массиве. Вот как выглядит объявление массива с именем balance, имеющего тип double и состоящего из 100 элементов:

```
double balance[100];
```

Доступ к элементу массива осуществляется с помощью имени массива и индекса. Для этого индекс элемента указывается в квадратных скобках после имени массива. Например, оператор, приведенный ниже, присваивает третьему элементу массива balance значение 12.23:

```
balance[3] = 12.23;
```

Индекс первого элемента любого массива в языке C равен нулю. Следовательно, оператор

```
char p[10];
```

объявляет массив символов, состоящий из 10 элементов – от p[0] до p[9]. Следующая программа заполняет целочисленный массив числами от 0 до 99:

```
#include <stdio.h>
void main()
{
```

```

int x[100];      /* Объявление целочисленного массива, состоящего из 100 элементов */

int t;
/* Заполнение массива числами от 0 до 99 */
for(t = 0; t < 100; ++t)
    x[t] = t;
/* Вывод на экран элементов массива x */
for(t = 0; t < 100; ++t)
    printf("%d ", x[t]);
}

```

Объем памяти, необходимый для хранения массива, зависит от его типа и размера. Размер одномерного массива в байтах вычисляется по формуле:

количество_байт = sizeof(тип) * количество_элементов

В языке С не предусмотрена проверка выхода индекса массива за пределы допустимого диапазона. Иными словами, во время выполнения программы можно по ошибке выйти за пределы памяти, отведенной для массива, и записать данные в соседние ячейки, в которых могут храниться другие переменные и даже программный код. Ответственность за предотвращение подобных ошибок лежит на программисте.

Например, фрагмент программы, приведенный ниже, будет скомпилирован без ошибок, однако во время выполнения программы индекс массива выйдет за пределы допустимого диапазона:

```

int count[10], i;
/* Выход индекса массива за пределы диапазона */
for(i=0; i<100; i++)
    count[i] = i;

```

По существу, одномерный массив представляет собой список переменных, имеющих одинаковый тип и хранящихся в смежных ячейках памяти в порядке возрастания их индексов.

Ниже показано (табл. 23.1), как хранится в памяти массив *a*, начинающийся с адреса 1000 и объявленный с помощью оператора `char a[7];`

Таблица 23.1

Элемент	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Адрес	1000	1001	1002	1003	1004	1005	1006

23.3. Задание

Создайте целочисленные массивы, содержащие количество дней в каждом месяце (не високосного года). По введенному номеру вывести количество дней в месяце.

Занятие 24

24.1. Инициализация массива

В языке С допускается инициализация массивов при их объявлении. Общий вид инициализации массива не отличается от инициализации обычных переменных:

тип имя_массива[размер1]...[размерN] = {список_значений}

Список_значений представляет собой список констант, разделенных запятыми. Тип констант должен быть совместимым с типом массива. Первая константа присваивается первому элементу массива, вторая – второму и т.д. Обратите внимание на то, что после закрывающей фигурной скобки `}` обязательно должна стоять точка с запятой.

Рассмотрим пример, в котором целочисленный массив, состоящий из 10 элементов, инициализируется числами от 1 до 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Здесь элементу `i[0]` присваивается значение 1, а элементу `i[9]` – число 10. Символьные массивы можно инициализировать строковыми константами:

```
char имя_массива[размер] = "строка";
```

Например, следующий фрагмент инициализирует массив `str` строкой "Я люблю С":

```
char str[10] = "Я люблю С";
```

То же самое можно переписать иначе:

```
char str[10] = {'Я', ' ', 'л', 'ю', 'б', 'л', 'ю', ' ', 'С', '\0'};
```

Поскольку строки завершаются нулевым байтом (детальнее см. занятие 58), размер массива должен быть достаточным, поэтому размер массива `str` равен 10, хотя строка "Я люблю С" состоит из 9 символов. Если массив инициализируется строковой константой, компилятор автоматически добавляет нулевой байт.

24.2. Инициализация безразмерного массива

При инициализации сообщений об ошибках, каждое из которых хранится в одномерном массиве, необходимо подсчитать точное количество символов в каждом сообщении:

```
char e1[18] = "Ошибка при чтении";
```

```
char e2[24] = "Невозможно открыть файл";
```

Компилятор может сам определить размер массива. Для этого не нужно указывать размер массива:

```
char e1[] = "Ошибка при чтении";
```

```
char e2[] = "Невозможно открыть файл";
```

Такой массив называется безразмерным. Инициализация безразмерных массивов, в данном случае, позволяет изменять сообщения, не заботясь о размере массивов.

Занятие 25

25.1. Нахождение суммы элементов массива

Для обработки массивов существует несколько стандартных алгоритмов. Одним из них является процесс нахождения суммы элементов массива. Он состоит из двух этапов: присваивания сумме нулевого значения; увеличение в цикле суммы на элемент массива, номер которого совпадает с номером итерации цикла:

```
int a[9] = {34, -3, 44, 16, 53, -55, 1, -21, 2};
int i, Sum = 0;
for (i = 0; i < 9; i++)
    Sum += a[i];
```

На рис. 25.1, а показана блок-схема нахождения суммы элементов массива.

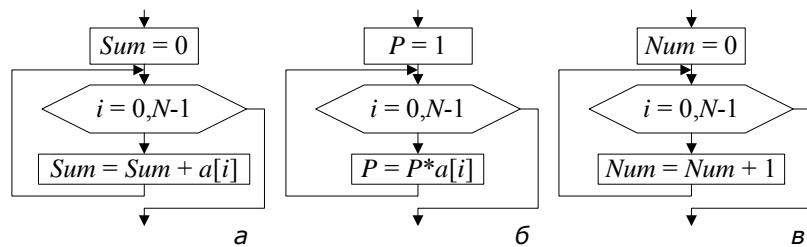


Рис. 25.1

Алгоритм нахождения суммы можно модифицировать, если, например, находить суммы только положительных элементов:

```
int i, Sum = 0;
for (i = 0; i < 9; i++)
    if (a[i] > 0) Sum += a[i];
```

25.2. Нахождение произведения элементов массива

Процесс нахождения произведения элементов массива также состоит из двух этапов: присваивания произведению единичного значения; умножение в цикле произведения на элемент массива, номер которого совпадает с номером итерации цикла:

```
int i, P = 1;
for (i = 0; i < 9; i++)
    P *= a[i];
```

На рис. 25.1, б показана блок-схема нахождения произведения элементов массива.

Алгоритм нахождения произведения также можно модифицировать, если, например, находить произведение только положительных элементов:

```
int i, P = 1;
```

```
for (i = 0; i < 9; i++)
    if (a[i] > 0) P *= a[i];
```

25.3. Нахождение количества элементов массива

Сам по себе процесс нахождения количества элементов массива не имеет смысла поскольку для его реализации необходимо знать количество итераций в цикле. Но он имеет смысл, если необходимо найти, например, количество положительных элементов в массиве.

Алгоритм нахождения количества элементов полностью повторяет алгоритм нахождения суммы элементов массива за тем исключением, что в цикле количество увеличивается на 1 (рис. 25.1, в). Рассмотрим пример нахождения количества положительных элементов массива:

```
int i, Num = 0;
for (i = 0; i < 9; i++)
    if (a[i] > 0) Num++;
```

Суммируя все сказанное выше, напомним программу нахождения среднего арифметического положительных элементов массива:

```
#include <stdio.h>
void main()
{
    int a[] = {34, -3, 44, 16, 53, -55, 1, -21, 2};
    int i, Sum = 0, Num = 0;
    for (i = 0; i < 9; i++)
        if (a[i] > 0) {
            Sum += a[i];
            Num++;
        }
    printf("%5.2f\n", (double) Sum/Num);
}
```

25.4. Задание

Найти сумму, произведение и количество определенных элементов массива (табл. 25.1). Размер массива 15 элементов. Элементы массива вводить каждый раз с клавиатуры.

Таблица 25.1

№	Обработать	№	Обработать
1	Нечетные, большие 20	6	Меньшие 10 по модулю
2	Положительные, кратные 3	7	Нечетные отрицательные
3	Отрицательные, кратные 3	8	Кратные 2 и 3
4	Нечетные положительные	9	Четные отрицательные
5	Четные, большие 20	10	Четные положительные

Занятие 26

26.1. Нахождение минимального элемента массива

К стандартным относятся также алгоритмы нахождения минимального и максимального элементов массива.

Процесс нахождения минимального элемента массива состоит из двух этапов (рис. 26.1):

- присваивание минимальному значению нулевого элемента массива;
- сравнение всех элементов массива (кроме нулевого) с минимальным в цикле (если текущий элемент окажется меньше минимального, то его следует сделать минимальным).

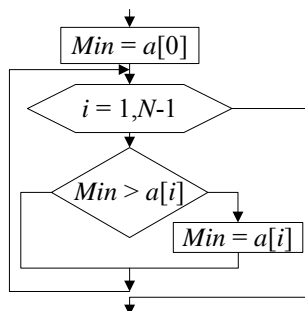


Рис. 26.1

Ниже приведен фрагмент программы нахождения минимального элемента массива:

```

int a[9] = {34, -3, 44, 16, 53, -55, 1, -21, 2};
int i, Min = a[0];
for (i = 1; i < 9; i++)
    if (Min > a[i]) Min = a[i];
  
```

26.2. Нахождение максимального элемента массива

Процесс нахождения максимального элемента массива сходен с алгоритмом нахождения минимального элемента (рис. 26.2).

Отличие заключается в сравнении максимального элемента с текущим: если текущий элемент окажется больше максимального, то его следует сделать максимальным.

Ниже приведен фрагмент программы нахождения максимального элемента массива:

```

int a[9] = {34, -3, 44, 16, 53, -55, 1, -21, 2};
int i, Max = a[0];
for (i = 1; i < 9; i++)
    if (Max < a[i]) Max = a[i];
  
```

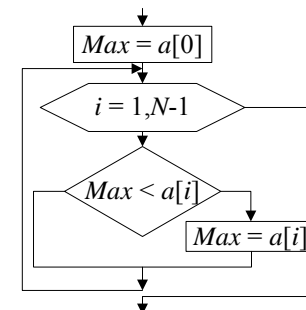


Рис. 26.2

26.3. Нахождение экстремальных элементов по условию

При нахождении минимального или максимального элемента массива по условию, например среди положительных элементов, приведенная выше схема не подходит, поскольку нулевой элемент может не удовлетворять заданному условию.

В этом случае, прежде чем приступить к нахождению экстремального элемента, необходимо найти первый подходящий под условие элемент массива, а затем среди оставшихся найти экстремальный.

Ниже приведена программа нахождения максимального отрицательного числа в массиве:

```

#include <stdio.h>
void main()
{
    int a[] = {34, -3, 44, 16, 53, -55, 1, -21, 2};
    int i, Max, N = 9;
    for (i = 0; i < N; i++)
        if (a[i] < 0) {
            Max = a[i];
            break;
        }
    for (i++; i < N; i++)
        if (a[i] < 0 && Max < a[i]) Max = a[i];
    printf("%d\n", Max);
}
  
```

26.4. Задание

Найти максимальный и минимальный элемент массива среди отвечающих условию (см. табл. 25.1).

Занятие 27

27.1. Сортировка массивов

При решении задачи сортировки обычно выдвигается требование минимального использования дополнительной памяти, из которого вытекает недопустимость применения дополнительных массивов.

Для оценки быстродействия алгоритмов различных методов сортировки, как правило, используют два показателя:

- количество присваиваний;
- количество сравнений.

Все методы сортировки можно разделить на две большие группы:

- прямые методы сортировки;
- улучшенные методы сортировки.

Прямые методы сортировки по принципу, лежащему в основе метода, в свою очередь разделяются на четыре подгруппы:

- 1) сортировка выбором;
- 2) сортировка вставкой;
- 3) сортировка заменой;
- 4) сортировка обменом ("пузырьковая" сортировка).

Улучшенные методы сортировки основываются на тех же принципах, что и прямые, но используют некоторые оригинальные идеи для ускорения процесса сортировки.

Прямые методы на практике используются довольно редко, так как имеют относительно низкое быстродействие. Однако они хорошо показывают суть основанных на них улучшенных методов. Кроме того, в некоторых случаях (как правило, при небольшой длине массива и/или особом исходном расположении элементов массива) некоторые из прямых методов могут даже превзойти улучшенные методы.

27.2. Сортировка выбором

Принцип метода:

Находим (выбираем) в массиве элемент с минимальным значением на интервале от 1-го (0 индекс) элемента до n-го (последнего) элемента и меняем его местами с первым элементом.

На втором шаге находим элемент с минимальным значением на интервале от 2-го до n-го элемента и меняем его местами со вторым элементом.

И так далее для всех элементов до n-1-го.

На рис. 27.1 приведен пример упорядочивания массива целых чисел длиной 7 элементов.

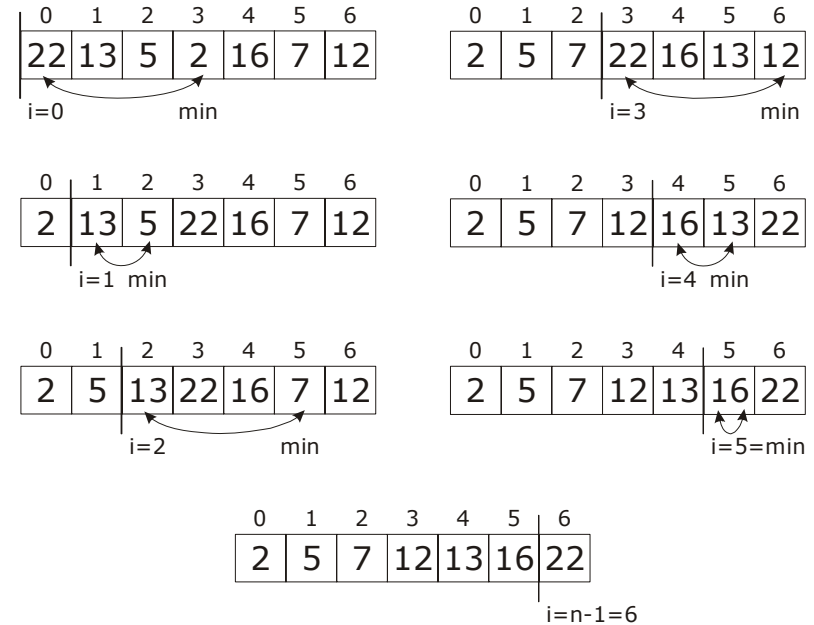


Рис. 27.1

Программа, реализующая метод выбора, будет иметь следующий вид:

```
#include <stdio.h>
const int N = 7;
void main()
{
    int vector[] = {22, 13, 5, 2, 16, 7, 12},
        i, j, Imin, Min;
    for (i = 0; i < N - 1; i++) {
        Imin = i;
        for (j = i + 1; j < N; j++)
            if (vector[j] < vector[Imin])
                Imin = j;
        Min = vector[Imin];
        vector[Imin] = vector[i];
        vector[i] = Min;
    }
    for (i = 0; i < N; i++)
        printf("%4d", vector[i]);
    printf("\n");
}
```

Занятие 28

Сортировка вставкой

Принцип метода:

Массив разделяется на две части: отсортированную и неотсортированную. Элементы из неотсортированной части поочередно выбираются и вставляются в отсортированную часть так, чтобы не нарушить в ней упорядоченность элементов. В начале работы алгоритма в качестве отсортированной части массива принимают только один первый элемент, а в качестве неотсортированной части – все остальные элементы.

Таким образом, алгоритм будет состоять из $n-1$ -го прохода (n – размерность массива), каждый из которых будет включать четыре действия:

- взятие очередного i -го неотсортированного элемента и сохранение его в дополнительной переменной;
- поиск позиции j в отсортированной части массива, в которой присутствие взятого элемента не нарушит упорядоченности элементов;
- сдвиг элементов массива от $i-1$ -го до $j-1$ -го вправо, чтобы освободить найденную позицию вставки;
- вставка взятого элемента в найденную j -ю позицию.

Для реализации данного метода можно предложить несколько алгоритмов, которые будут отличаться способом поиска позиции вставки.

Рассмотрим схему реализации одного из возможных алгоритмов. Схематично описанные действия можно представить следующим образом (рис. 28.1). На рисунке цифрами в кружочках показаны действия из которых состоит алгоритм сортировки вставкой.

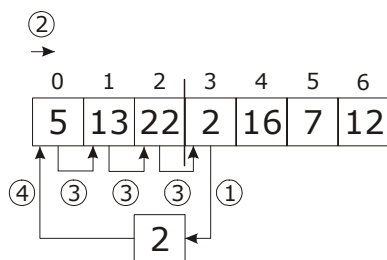


Рис. 28.1

На рис. 28.2 приведен пример упорядочивания массива целых чисел длиной 7 элементов.

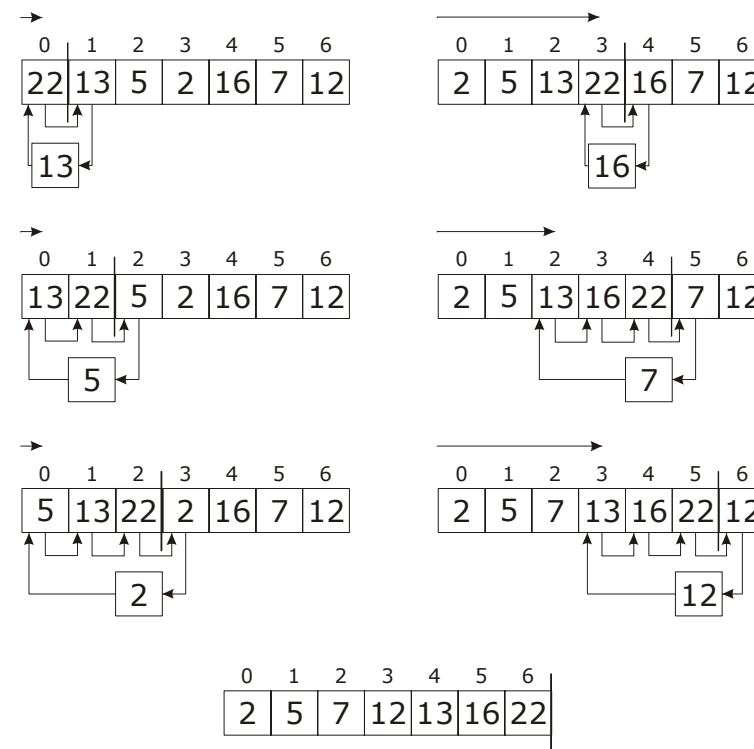


Рис. 28.2

Программа будет иметь следующий вид:

```
#include <stdio.h>
const int N = 7;
void main()
{
    int vector[] = {22, 13, 5, 2, 16, 7, 12},
        i, B, j, k;
    for (i = 1; i < N; i++) {
        B = vector[i], j = 0;
        while (B > vector[j]) j++;
        for (k = i - 1; k >= j; k--)
            vector[k + 1] = vector[k];
        vector[j] = B;
    }
    for (i = 0; i < N; i++)
        printf("%4d", vector[i]);
    printf("\n");
}
```

Занятие 29

Сортировка обменом ("пузырьковая" сортировка)

Принцип метода:

Слева направо поочередно сравниваются два соседних элемента, и если их взаимное расположение не соответствует заданному условию упорядоченности, то они меняются местами. Далее берутся два следующих соседних элемента и так далее до конца массива.

После одного такого прохода на последней n -ой позиции массива будет стоять максимальный элемент ("всплыл" первый "пузырек"). Поскольку максимальный элемент уже стоит на своей последней позиции, то второй проход обменов выполняется до $n-1$ -го элемента. И так далее. Всего требуется $n-1$ проход.

Рассмотрим схему алгоритма сортировки методом прямого обмена по неубыванию (рис. 29.1).

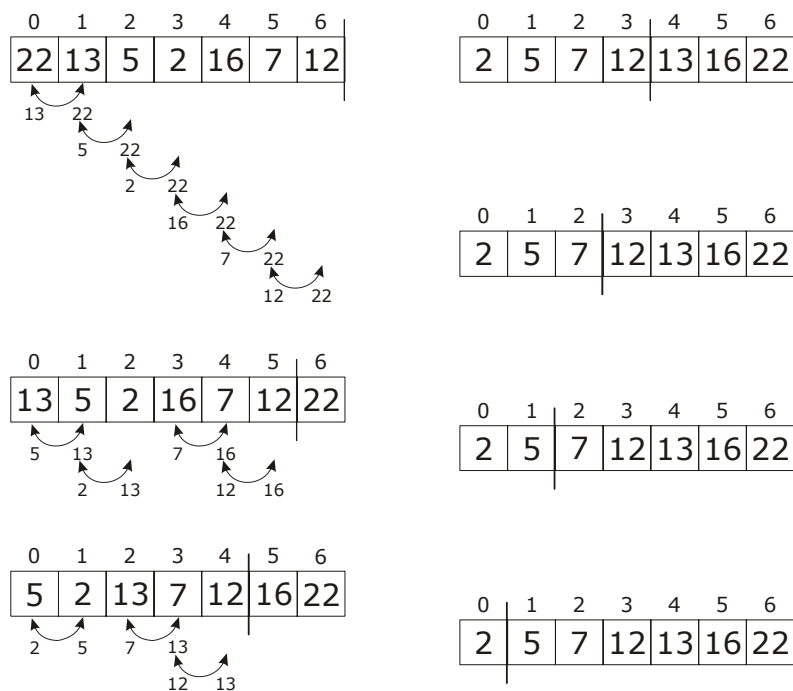


Рис. 29.1

Рассмотрим фрагмент программы, реализующей приведенный алгоритм:

```
for (i = n - 2; i > 0; i--) {
    int j;
    for (j = 0; j <= i; j++)
        if (x[j] > x[j + 1]) {
            int y = x[j];
            x[j] = x[j + 1];
            x[j + 1] = y;
        }
}
```

В частном случае, сортировку можно прекратить, если при прохождении массива, не один элемент не был переставлен, то есть массив был отсортирован.

Программно это реализуется с помощью дополнительной переменной, которая имеет смысл логического переключателя: в начале обмена она сигнализирует о том, что массив отсортирован, а если хотя бы один обмен был произведен – ее значение меняется на противоположное. Проходы заканчиваются, если переменная не изменила своего значения.

Программа, реализующая метод обмена ("пузырька"), будет иметь следующий вид:

```
#include <stdio.h>
const int N = 7;
void main()
{
    int x[] = {22, 13, 5, 2, 16, 7, 12};
    int i, sort;
    for (i = N - 2; i > 0; i--) {
        int j;
        sort = 1;
        for (j = 0; j <= i; j++)
            if (x[j] > x[j + 1]) {
                int y = x[j];
                x[j] = x[j + 1];
                x[j + 1] = y;
                sort = 0;
            }
        if (sort) break;
    }
    for (i = 0; i < N; i++)
        printf("%4d", x[i]);
    printf("\n");
}
```

В этой программе реализуется сортировка в порядке возрастания. Для реализации сортировки в порядке убывания достаточно поменять знак сравнения в условном операторе

```
if (x[j] < x[j + 1])
```

Занятие 30 Бинарный поиск

Поиск – процесс нахождения заданного элемента в массиве. Алгоритм бинарного поиска допустимо использовать для нахождения заданного элемента только в упорядоченных массивах.

Принцип бинарного поиска:

Исходный массив делится пополам и для сравнения выбирается средний элемент. Если он совпадает с искомым, то поиск заканчивается. Если же средний элемент меньше искомого, то все элементы левее его также будут меньше искомого. Следовательно, их можно исключить из зоны дальнейшего поиска, оставив только правую часть массива. Аналогично, если средний элемент больше искомого, то отбрасывается правая часть, а остается левая.

На втором этапе выполняются аналогичные действия над оставшейся половиной массива. В результате после второго этапа остается 1/4 часть массива.

И так далее, пока или элемент будет найден, или длина зоны поиска станет равной нулю. В последнем случае искомый элемент найден не будет.

Рассмотрим схему алгоритма бинарного поиска (рис. 30.1).

Программа, реализующая один из возможных вариантов бинарного поиска, имеет следующий вид:

```
#include <stdio.h>
const int N = 15;
void main()
{
    int a[] = { 2, 3, 5, 12, 16,
                17, 20, 22, 23, 25,
                29, 36, 37, 42, 52};
    int x, L = 0, R = N - 1, i;
    printf("Введите искомый элемент: ");
    scanf("%d", &x);
    while (L <= R) {
        i = (L + R)/2;
        if (a[i] == x) break;
        if (a[i] < x) L = i + 1;
        else R = i - 1;
    }
    printf("Искомый элемент ");
    if (a[i] == x) printf("найден на позиции %d\n", i);
    else printf("не найден\n");
}
```

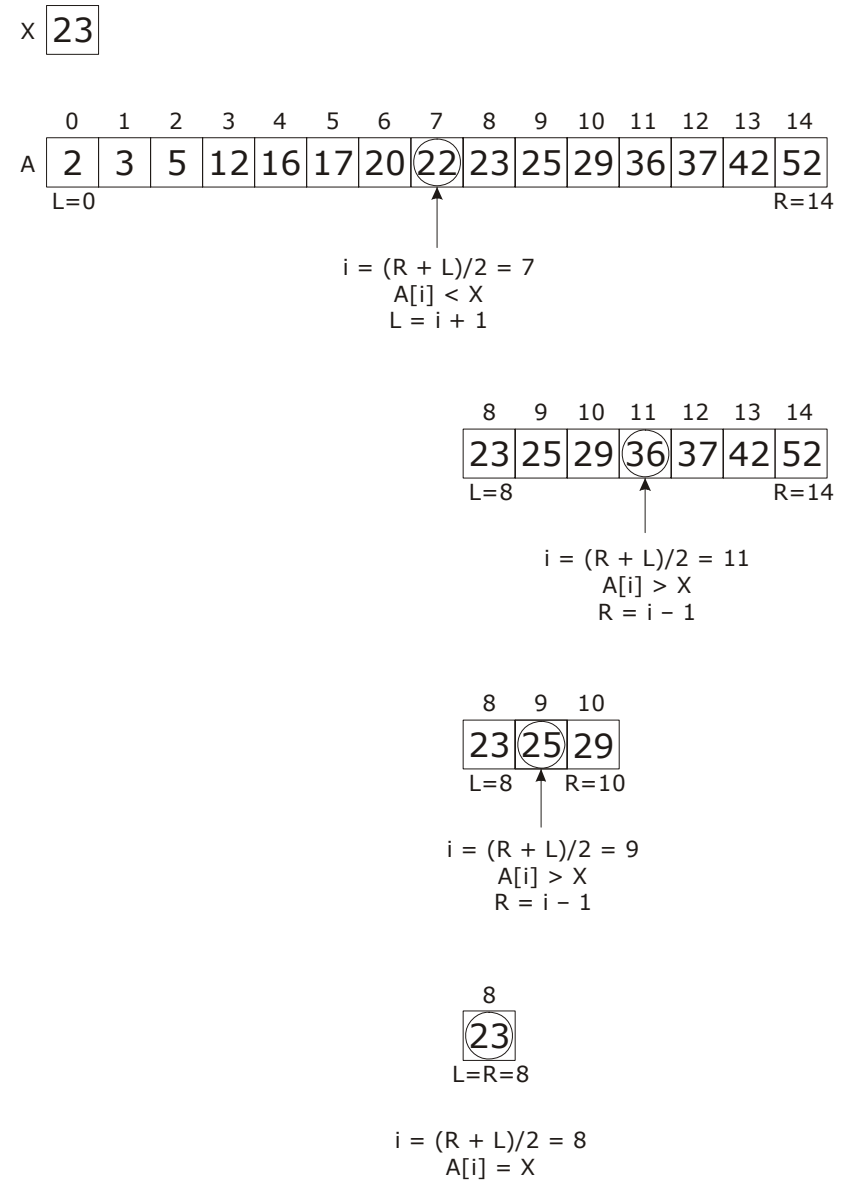


Рис. 30.1

Занятие 31

31.1. Двухмерные массивы

В языке C предусмотрены многомерные массивы. Простейшим из них является двухмерный. По существу, двухмерный массив – это массив одномерных массивов. Объявление двухмерного массива `d`, состоящего из 10 строк и 20 столбцов, выглядит следующим образом:

```
int d[10][20];
```

Объявляя двухмерные массивы, следует проявлять осторожность. В некоторых языках программирования размерности массивов отделяются запятыми. В отличие от них, в языке C размерности массива заключаются в квадратные скобки. Обращение к элементу двухмерного массива выглядит так:

```
d[1][2]
```

Следующая программа заполняет двухмерный массив числами от 1 до 19 и выводит их на экран построчно:

```
#include <stdio.h>
void main()
{
    int t, i, num[3][4];
    for (t = 0; t < 3; ++t)
        for (i = 0; i < 4; ++i)
            num[t][i] = (t*4) + i + 1;
    for (t = 0; t < 3; ++t) {
        for (i = 0; i < 4; ++i)
            printf("%3d ", num[t][i]);
        printf("\n");
    }
}
```

В данном примере элемент `num[0][0]` равен 1, `num[0][1]` равен 2, `num[0][2]` равен 3 и так далее. Значение элемента `num[2][3]` равно 12. Массив `num` можно изобразить следующим образом:

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Двухмерный массив хранится в виде матрицы, в которой первый индекс задает номер строки, а второй – номер столбца. Таким образом, при обходе элементов в порядке их размещения в памяти правый индекс изменяется быстрее, чем левый.

Объем памяти, занимаемый двухмерным массивом, выраженный в байтах, задается следующей формулой:

$$\text{колбайтов} = \text{колстрок} * \text{колстолбцов} * \text{sizeof(базовый_тип)}$$

Считая, что целое число занимает 4 байта, можно вычислить, что для хранения массива, состоящего из 10 строк и 5 столбцов, необходимо $10 * 5 * 4 = 200$ байт.

31.2. Многомерные массивы

В языке C массивы могут иметь больше двух размерностей. Максимально допустимое количество размерностей задается компилятором. Общий вид объявления многомерного массива таков:

тип имя[Размер1][Размер2][Размер3]...[РазмерN]

Массивы, имеющие больше трех размерностей, используются редко, поскольку они занимают слишком большой объем памяти.

При обращении к многомерным массивам компилятор вычисляет каждый индекс. Следовательно, доступ к элементам многомерного массива происходит значительно медленнее, чем к элементам одномерного массива.

31.3. Инициализация многомерных массивов

Многомерные массивы инициализируются аналогично одномерным. Рассмотрим пример инициализации массива `sgrs` числами от 1 до 10 и их квадратами:

```
int sgrs[10][2] = { 1, 1, 2, 4, 3, 9, 4, 16, 5, 25,
                   6, 36, 8, 64, 9, 81, 10, 100 };
```

Инициализируя многомерный массив, можно заключать элементы списка в фигурные скобки. Этот способ называется субагрегатной группировкой. Например, предыдущую инициализацию можно переписать следующим образом:

```
int sgrs[10][2] = {{1, 1}, {2, 4}, {3, 9}, {4, 16},
                  {5, 25}, {6, 36}, {7, 49},
                  {8, 64}, {9, 81}, {10, 100}};
```

Если при группировке данных указаны не все начальные значения, оставшиеся элементы группы автоматически заполняются нулями.

Инициализировать можно и безразмерные многомерные массивы. Для этого требуется указать все размеры, кроме первого (остальные размеры нужны для того, чтобы компилятор правильно выполнял индексацию массива). Это позволяет создавать таблицы переменной длины, поскольку компилятор автоматически размещает их в памяти. Например, объявление массива `sgrs` как безразмерного выглядит так:

```
int sgrs[][2] = {{1, 1}, {2, 4}, {3, 9}, {4, 16},
                {5, 25}, {6, 36}, {7, 49}, {8, 64},
                {9, 81}, {10, 100}}
```

Занятие 32

32.1. Вложенные циклы и обработка матриц

Тело цикла одного оператора цикла может содержать другие операторы цикла. Такие циклы называются вложенными. При составлении программ с вложенными циклами необходимо помнить о следующем:

- передача управления разрешена из внутреннего цикла во внешний, но не наоборот;
- область действия внутреннего цикла не должна выходить за пределы внешнего цикла.

Кроме того, для операторов цикла `for` действует еще одно правило: имена параметров внешнего и внутреннего циклов должны быть разными.

В случае вложенных циклов `for` при каждом фиксированном значении параметра внешнего цикла параметр внутреннего цикла *пробегаёт* все свои значения.

При работе с двумерными массивами (матрицами) вложенные циклы рекомендуется организовывать так, чтобы во внешнем цикле изменялся индекс строки (первый индекс), а во внутреннем – индекс столбца (второй индекс), что увеличивает эффективность программы благодаря последовательному считыванию элементов массива из памяти.

Рассмотрим пример заполнения двумерной матрицы 6×6 произвольными числами от 1 до 36:

```
#include <stdio.h>
#include <stdlib.h>
const int N = 6;
void main()
{
    int x[6][6];
    int i, j;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            x[i][j] = (rand() % 36) + 1;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++)
            printf("%7d", x[i][j]);
        printf("\n");
    }
}
```

32.2. Задание

Вычислить сумму элементов матрицы, показанных на рис. 32.1. Размер матрицы соответствует рисунку (7×7).

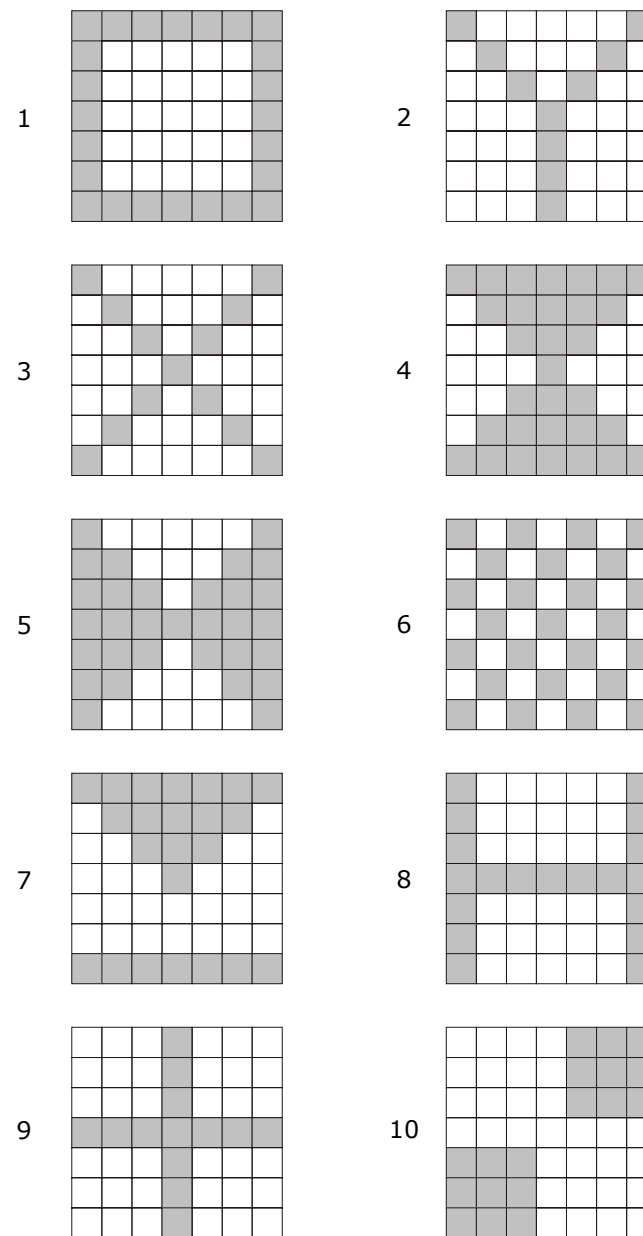


Рис. 32.1

Занятие 33

33.1. Общий вид функции

Функции – это строительные блоки для всех программ на языке C. Общий вид функции выглядит следующим образом.

```
тип_возвращаемого_знач имя_функции(список_параметров)
{
    тело функции
}
```

Тип_возвращаемого_значения определяет тип переменной, которую возвращает функция. Функция может возвращать переменные любого типа, кроме массива. В списке параметров перечисляются типы аргументов и их имена, разделенные запятыми. Если функция не имеет аргументов, ее список_параметров пуст. В этом случае скобки все равно необходимы.

В языке C с помощью одного оператора можно объявлять несколько переменных одинакового типа, разделяя их запятыми. В противоположность этому, каждый параметр функции должен быть объявлен отдельно. Таким образом, общий вид объявления параметров в заголовке функции выглядит так:

```
f(тип имя_перемен1, тип имя_перемен2, ..., тип имя_переменN)
```

Таким образом:

```
f(int i, int k, int j)          /* Правильно */
f(int i, k, float j)           /* Неправильно */
```

Определим функцию, выводящую на консоль число в обратном порядке (см. занятие 20).

Опишем эту функцию в программе и вызовем несколько раз с различными параметрами (функцию необходимо описать до основной программы – функции main):

```
#include <stdio.h>
void reverse(unsigned long int number)
{
    while (number) {
        printf("%d", number % 10);
        number /= 10;
    }
    printf("\n");
}
void main()
{
    int x = 1245;
    reverse(x);
    reverse(4329);
    reverse(17328);
}
```

Приведем пример функции возвращающей минимальное из двух значений:

```
#include <stdio.h>
double minimum(double a, double b)
{
    return (a < b)? a: b;
}
void main()
{
    double x = 3.5, y = -4.6, min;
    min = minimum(x,y);
    printf("%5.2f\n", min);
    min = minimum(x,1.8);
    printf("%5.2f\n", min);
    printf("%5.2f\n", minimum(-5.4,x));
    printf("%5.2f\n", minimum(6.6,9.2));
}
```

33.2. Область видимости функции

Правила, определяющие область видимости, устанавливают, видит ли одна часть программы другую часть кода или данных и может ли к ним обращаться.

Каждая функция представляет собой отдельный блок операторов. Код функции является закрытым и недоступным для любых операторов, расположенных в других функциях, кроме операторов вызова. Код, образующий тело функции, скрыт от остальной части программы, если он не использует глобальные переменные. Этот код не может влиять на другие функции, а они, в свою очередь, не могут влиять на него. Иначе говоря, код и данные, определенные внутри некой функции, никак не взаимодействуют с кодом и данными, определенными в другой функции, поскольку их области видимости не перекрываются.

Переменные, определенные внутри функции, называются локальными. Они создаются при входе в функцию и уничтожаются при выходе из нее. Иными словами, локальные переменные не сохраняют свои значения между вызовами функции. Единственное исключение из этого правила составляют статические локальные переменные. Они хранятся в памяти вместе с глобальными переменными, но их область видимости ограничена функцией, в которой они определены. Глобальные и локальные переменные подробно будут рассмотрены на занятии 37.

В языке C невозможно определить одну функцию внутри другой. По этой причине с формальной точки зрения язык C нельзя считать блочно-структурированными.

Занятие 34

34.1. Аргументы функции

Если функция имеет аргументы, в ее заголовке должны быть объявлены переменные, принимающие их значения. Эти переменные называются формальными параметрами функции. Как и локальные переменные, они создаются при входе в функцию и уничтожаются при выходе из нее.

Изменим функцию приведенную на предыдущем занятии так, чтобы она выводила цифры числа в правильном порядке, но через пробел:

```
void reverse(unsigned long int number)
{
    int numeral[10], i = 0;
    while (number) {
        numeral[i++] = number % 10;
        number /= 10;
    }
    for (--i; i >= 0; i--)
        printf("%d ", numeral[i]);
    printf("\n");
}
```

Несмотря на то что формальные параметры должны, в основном, получать значения аргументов, передаваемых функции, их можно использовать наравне с другими локальными переменными, в частности, присваивать им значения или использовать в любых выражениях, что и видно в данном примере. В нем также использованы две локальные переменные: массив целых значений `numeral` и целая переменная `i`.

34.2. Передача параметров по значению и по ссылке

В языках программирования существуют два способа передачи аргументов в подпрограмму. Первый из них известен как *передача параметров по значению*. В этом случае формальному параметру подпрограммы присваивается копия значения аргумента, и все изменения параметра никак не отражаются на аргументе.

Второй способ называется *передачей параметров по ссылке*. При использовании этого метода параметру присваивается адрес аргумента. Внутри подпрограммы этот адрес открывает доступ к фактическому аргументу. Это значит, что все изменения, которым подвергается параметр, отражаются на аргументе.

По умолчанию в языке C применяется передача по значению. Как правило, это означает, что код, образующий тело функции, не может изменять аргументы, указанные при ее вызове. Но

на практике такие изменения очень часты (приведенный пример тому подтверждение).

34.3. Передача параметров по ссылке

Хотя в языке C по умолчанию применяется передача параметров по значению, их можно передавать и по ссылке. Для этого в функцию вместо аргумента нужно передать указатель на него (детальнее указатели будут рассмотрены на занятиях 45–56). Поскольку функция получает адрес аргумента, ее код может изменять значение фактического аргумента вне функции.

Указатели передаются функции как обычные переменные. Естественно, они должны быть объявлены как указатели. Рассмотрим в качестве примера функцию `swap()`:

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Функция `swap()` может менять местами значения двух переменных, на которые ссылаются указатели `x` и `y`, поскольку она получает их адреса, а не копии.

Помните, что функции `swap()` передаются адреса аргументов. Рассмотрим фрагмент, который демонстрирует правильный вызов функции `swap()`:

```
int i, j;
i = 10; j = 20;
swap(&i, &j); // Передаются адреса переменных i и j
printf("%d %d", i, j);
```

В этом примере переменной `i` присваивается значение 10, а переменной `j` – число 20. Затем функции `swap()` передаются адреса переменных `i` и `j`, а не их значения (для этого используется унарный оператор получения адреса `&`). После возврата управления из функции `swap()` переменные `x` и `y` оказываются переставленными.

34.4. Задание

Выполнить задание 20 с использованием функции. Функция должна *вернуть* число, в котором цифры расположены в правильном порядке, а не в обратном. В основной программе полученное число вывести на консоль.

Занятие 35

35.1. Возврат управления из функции

Существует два способа прекратить выполнение функции и вернуть управление вызывающему модулю. В первом случае функция полностью выполняет свои операторы, и управление достигает закрывающей фигурной скобки `}` (сама фигурная скобка не порождает никакого объектного кода, однако ее можно так интерпретировать). Примером может послужить функция `reverse` приведенная на предыдущем занятии.

Как только будет выведен на консоль символ перевода строки, функции `reverse()` ничего не остается делать, и она возвратит управление в точку, откуда была вызвана.

На самом деле этот способ используется довольно редко, поскольку, во-первых, это снижает наглядность и эффективность функции, а во-вторых, в большинстве случаев она должна вернуть некий результат.

Функция может содержать несколько операторов `return`. Например, функция `find_number()` в следующей программе возвращает позицию числа в неотсортированном массиве, а если число не найдено, возвращает число `-1`:

```
#include <stdio.h>
int find_number(int number)
{
    int array[] = {21, 2, -4, 6, 1, 11, -3}, N = 7, i;
    for (i = 0; i < N; i++)
        if (number == array[i]) return i;
    return -1;
}

void main()
{
    int x, num;
    printf("Введите искомое число: ");
    scanf("%d", &x);
    if ((num = find_number(x)) == -1)
        printf("Такое число не найдено\n");
    else printf("Позиция числа в массиве %d\n", num);
}
```

35.2. Возвращаемые значения

Все функции, за исключением функций, объявленных со спецификатором `void`, с помощью оператора `return` возвращают некий результат. В соответствии со стандартом C89, если функция, возвращающая значение, не использует для этого оператор `return`, в вызывающий модуль возвращается "мусор", то есть случайное

значение. В стандарте C99 все функции, возвращающие значение, должны делать это с помощью оператора `return`. Иными словами, если в объявлении указано, что функция возвращает некое значение, любой оператор `return`, расположенный в ее теле, должен быть связан с определенной переменной. Однако, если поток управления достигнет конца функции, возвращающей значение, не встретив на своем пути оператора `return`, в вызывающий модуль также вернется "мусор".

Несмотря на то что эта ошибка не относится к разряду синтаксических, она является принципиальным изъяном программы и должна быть исключена.

Если функция не объявлена со спецификатором `void`, ее можно использовать в качестве операнда в любом выражении. Таким образом, фрагмент программы, приведенный ниже, является совершенно правильным:

```
x = power(y);
if (max(x,y) > 100) printf("больше");
for(ch = getchar(); isdigit(ch); ) ... ;
```

В языке программирования C, функция не должна стоять в левой части оператора присваивания. Следовательно, оператор

```
swap(x,y) = 100; // Неправильный оператор
```

является ошибочным.

Функции можно разделить на три категории. К первой категории относятся вычислительные функции. Они выполняют некие операции над своими аргументами и возвращают результат вычислений. Их можно назвать "настоящими" функциями. К примеру, "настоящими" являются стандартные библиотечные функции `sqrt()` и `sin()`, вычисляющие квадратный корень и синус соответственно.

Функции второго типа выполняют обработку информации. Их возвращаемое значение просто означает, успешно ли завершены операции. В качестве примера можно указать приведенную выше функцию `find_number()`, осуществляющую поиск числа в массиве. Если число найдено, функция возвращает его индекс, в противном случае она возвращает `-1`.

И, наконец, функции третьего вида не имеют явно возвращаемого значения. По существу, эти функции являются процедурами и не вычисляют никакого результата. К этой категории относится функция `exit()`, прекращающая выполнение программы.

Все функции, не возвращающие никаких значений, следует объявлять с помощью спецификатора `void` (детальнее спецификатор будет рассмотрен на следующем занятии). Такие функции нельзя использовать в выражениях.

Иногда функции, не вычисляющие ничего интересного, все равно что-то возвращают. Например, функция `printf()` возвращает

количество символов, выведенных на экран. Было бы интересно увидеть программу, в которой это значение имело какой-то смысл.

Хотя все функции, за исключением функций, объявленных с помощью спецификатора `void`, возвращают какие-то значения, использовать их в дальнейших вычислениях совершенно не обязательно. При обсуждении функций часто возникает вопрос: "Обязательно ли присваивать значение, возвращаемое функцией, какой-то переменной, объявленной в вызывающем модуле?". Нет, не обязательно. Если оператор присваивания не указан, возвращаемое значение просто игнорируется. Рассмотрим программу, в которой используется функция `mul()`.

```
#include <stdio.h>
int mul(int a, int b)
{
    return a*b;
}
void main()
{
    int x, y, z;
    x = 10, y = 20;
    z = mul(x, y);          /* 1 */
    printf("%d", mul(x,y)); /* 2 */
    mul(x, y);              /* 3 */
}
```

В первой строке значение, возвращаемое функцией `mul()`, присваивается переменной `z`.

Во второй строке возвращаемое значение ничему не присваивается, но используется внутри функции `printf()`.

В третьей строке значение, возвращаемое с помощью оператора `return`, отбрасывается, поскольку оно ничему не присваивается и не является частью какого-либо выражения.

35.3. Оператор `return` в функции `main()`

Во всех примерах, приведенных на занятиях, функция `main()` ничего возвращает. На самом деле функция `main()` возвращает вызвавшему ее процессу операционной системы целое число. Другими словами ее можно описать как

```
int main()
```

В этом случае, она должна завершать свою работу оператором `return` с аргументом. Вызвав функцию `exit()` с этим же аргументом, Вы получите тот же самый эффект.

На практике многие компиляторы языка C по умолчанию возвращают число 0, но полагаться на это не стоит, поскольку это снизит машинезависимость вашей программы.

Занятие 36

36.1. Функции типа `void`

Функции, объявленные с помощью спецификатора `void`, не возвращают никаких значений. Это не позволяет использовать такие функции в выражениях и предотвращает их неправильное применение. Например, функция `print_chessdesk()` выводит на экран шахматную доску. Поскольку эта функция ничего не возвращает, на месте типа возвращаемого значения указано ключевое слово `void`. Она использует аналогичную функцию:

```
#include <stdio.h>
void chessline(int j,
               char cb, char c, char cl, char ce)
{
    int i;
    printf("%c",cb);
    for (i = 0; i < 8; i++) {
        char c0 = (j + i)%2?0x20:0xB0;
        if (j >= 0) printf("%c%c%c",c0,c0,cl);
        else printf("%c%c%c",c,c,cl);
    }
    printf("\b%c\n",ce);
}
void print_chessdesk()
{
    //          0   1   2   3   4   5   6
    char box[] = " \xC9\xD1\xBB\xB6\xBC\xCF\xC8
                  \xC7\xCD\xBA\xC4\xB3\xC5";

    int i;
    chessline(-1,box[0],box[8],box[1],box[2]);
    for (i = 0; i < 7; i++) {
        chessline(i,box[9],box[12],box[11],box[9]);
        chessline(-1,box[7],box[10],box[12],box[3]);
    }
    chessline(7,box[9],box[12],box[11],box[9]);
    chessline(-1,box[6],box[8],box[5],box[4]);
}
void main()
{
    print_chessdesk();
}
```

В ранних версиях языка C ключевое слово `void` не определялось. Таким образом, в старых программах функции, не возвращающие никаких значений, вместо спецификатора `void` по умолчанию использовали тип `int`. Не удивляйтесь, встретив такие примеры где-нибудь в архиве.

Занятие 37

37.1. Передача массивов в качестве параметров

Если аргументом функции является массив, ей передается его адрес. Эта ситуация представляет собой исключение из общепринятого правила передачи параметров по значению. Функция, получившая массив, получает доступ ко всем его элементам и может его модифицировать. Рассмотрим в качестве примера функцию `print_upper()`, предназначенную для преобразования строчных букв в прописные (с кириллицей программа не работает):

```
#include <stdio.h>
#include <ctype.h>
/* Выводит строку, состоящую из прописных букв */
void print_upper(char *string)
{
    int t;
    for(t = 0; string[t]; ++t) {
        string[t] = toupper(string[t]);
        putchar(string[t]);
    }
}

void main()
{
    char s[80];
    gets(s);
    print_upper(s);
    printf("\nСтрока из прописных букв: %s", s);
}
```

После вызова функции `print_upper()` буквы, хранящиеся в массиве `s`, заменяются прописными. Если нет необходимости изменять содержимое массива `s`, то можно переписать функцию следующим образом:

```
void print_upper(char *string)
{
    int t;
    for(t = 0; string[t]; ++t)
        putchar(toupper(string[t]));
}
```

В этой версии содержимое массива в остается постоянным, поскольку его значение в функции `print_upper()` не изменяется.

37.2. Передача одномерного массива

Если аргументом функции является одномерный массив, ее формальный параметр можно объявить тремя способами: как указатель, как массив фиксированного размера и как массив неопре-

деленного размера. Например, чтобы предоставить функции `func1()` доступ к массиву `i`, можно использовать три варианта (первый вариант приведен в предыдущем подразделе):

```
void func1(int x[10]) // Массив фиксированного размера
void func1(int x[])  // Массив неопределенного размера
```

Все объявления эквивалентны друг другу, поскольку их смысл совершенно одинаков: в функцию передается указатель на целочисленную переменную. В первом случае действительно используется указатель. Во втором применяется стандартное объявление массива. В последнем варианте объявляется, что в функцию будет передан целочисленный массив неопределенной длины. Размер массива, передаваемого в функцию, не имеет никакого значения, поскольку проверка выхода индекса за пределы допустимого диапазона в языке C не предусмотрена. При указании размера массива можно написать любое число – это ничего не изменит, поскольку в функцию будет передан не массив, состоящий из `n` элементов, а лишь указатель на его первый элемент.

37.3. Передача многомерного массива

Если двухмерный массив используется в качестве аргумента функции, то в нее передается только указатель на его первый элемент. Однако при этом необходимо указать, по крайней мере, количество столбцов (можно, разумеется, задать и количество строк, но это не обязательно). Количество столбцов необходимо компилятору для того, чтобы правильно вычислить адрес элемента массива внутри функции, а для этого должна быть известна длина отроки. Например, функция, получающая в качестве аргумента двухмерный массив, состоящий из 10 строк и 10 столбцов:

```
void func1(int x[][10])
```

Компилятор должен знать количество столбцов, иначе он не сможет правильно вычислять выражения, подобные следующему:

```
x[2][4]
```

Если бы длина строки была неизвестна, компилятор не нашел бы начало третьей строки.

Передавая многомерный массив в функции, следует указывать все размеры, кроме первого. Например, если массив `m` объявлен оператором

```
int m[4][3][6][5];,
```

то функция `func1()`, получающая его в качестве аргумента, может выглядеть так:

```
void func1(int d[][3][6][5])
```

Разумеется, при желании можно указать и первый размер, но это не обязательно.

Занятие 38

38.1. Формальные параметры

Если функция имеет аргументы, следует объявить переменные, которые будут принимать их значения. Эти переменные называются формальными параметрами. Внутри функции они ничем не отличаются от других локальных переменных.

Тип формальных параметров задается при их объявлении. После этого их можно использовать как обычные локальные переменные. Учтите, что, как и локальные переменные, формальные параметры являются динамическими и разрушаются при выходе из функции. Их можно использовать в любых конструкциях и выражениях. Свои значения они получают от аргументов, передаваемых извне функции.

Корректность передачи параметров основывается на их порядке перечисления в заголовке функции и совместимости по присваиванию между соответствующими формальными и фактическими (см. далее) параметрами.

Многие функции имеют несколько параметров. Задача программиста – убедиться, что параметры, которые он указывает при вызове (фактические параметры), соответствуют по смыслу формальным параметрам. Компилятор может проверить только очевидные случаи – неправильное число параметров или несовместимость типов.

38.2. Фактические параметры

Параметры, указываемые при вызове функции, называются фактическими параметрами.

Фактическими параметрами могут быть не только переменные, но и значения. Например, если описана функция

```
double max(double a, double b)
{
    return (a > b)?a:b;
}
```

то вызвать ее можно как с переменными соответствующего типа

```
y = max(x1, x2);
```

так и со значениями

```
y = max(3.1, -4.2);
```

Не является исключением и смешанное использование фактических параметров:

```
y = max(x, -4.2);
```

Все это справедливо только если параметры передаются по значению. Если же параметры передаются по ссылке, то фактические параметры должны быть только параметрами.

Занятие 39

39.1. Локальные переменные

Как правило, переменные объявляют в трех местах: внутри функций, в определении параметров функции и за пределами всех функций. Соответственно такие переменные называются локальными, формальными параметрами и глобальными.

Переменные, объявленные внутри функции, называются локальными. Локальные переменные можно использовать только в операторах, расположенных внутри блока, где они объявлены. Иначе говоря, локальные переменные невидимы снаружи их блока (ограничен открывающей и закрывающей фигурными скобками).

Чаще всего локальные переменные объявляются внутри функций. Рассмотрим два примера:

```
void func1()
{
    int x;
    x = 10;
}
void func2()
{
    int x;
    x = -199;
}
```

Переменная *x* объявлена дважды: сначала – в функции *func1()*, а затем – в функции *func2()*. Переменная *x* из функции *func1()* не имеет никакого отношения к переменной *x*, объявленной внутри функции *func2()*. Каждая из этих переменных существует только внутри блока, где она была объявлена.

Язык C содержит ключевое слово *auto*, которое можно использовать для объявления локальных переменных. Однако, поскольку все локальные переменные по умолчанию считаются автоматическими, это ключевое слово на практике никогда не используется.

Поскольку локальная переменная создается при входе в блок, где она объявлена, и разрушается при выходе из него, ее значение теряется. Это особенно важно учитывать при вызове функции. Локальные переменные функции создаются при ее вызове и уничтожаются при возврате управления в вызывающий модуль. Это значит, что между двумя вызовами функции значения ее локальных переменных не сохраняются (исключение составляют переменные объявленные с помощью модификатора *static*).

Локальную переменную можно проинициализировать неким значением. Оно будет присваиваться переменной каждый раз при входе в блок. Неинициализированные локальные переменные

имеют неопределенное значение, пока к ним не будет применен оператор присваивания.

39.2. Объявление локальных переменных внутри блоков

Из соображений удобства и по традиции большинство программистов объявляют все переменные, используемые в функции, сразу после открывающей фигурной скобки и перед всеми остальными операторами. Однако следует иметь в виду, что локальные переменные можно объявлять в начале любого блока до выполнения каких-либо "активных" операторов. Например, попытка объявить локальные переменные так, как показано в приведенном ниже фрагменте, вызовет ошибку:

```
void f()
{
    int i;
    i = 10;
    int j;          // Этот оператор порождает ошибку
    j = 20;
}
```

Объявление переменных внутри блоков позволяет избежать побочных эффектов. Поскольку локальная переменная вне блока не существует, ее значение невозможно изменить непреднамеренно.

Примером объявления локальной переменной внутри блока может служить объявление переменных внутри условных или циклических операторов (рассматривалось ранее).

39.3. Локальные статические переменные

Статические переменные, в отличие от глобальных, неизвестны вне своей функции или файла, и сохраняют свои значения между вызовами. Это очень полезно при создании обобщенных функций и библиотек функций, которые могут использоваться другими программистами. Статические переменные отличаются как от локальных, так и от глобальных переменных.

Если локальная переменная объявлена с помощью спецификатора `static`, компилятор выделит для нее постоянное место хранения, как и для глобальной переменной. Принципиальное отличие локальной статической переменной от глобальной состоит в том, что первая остается доступной лишь внутри своего блока. Проще говоря, локальная статическая переменная – это локальная переменная, сохраняющая свои значения между вызовами функции.

Такие переменные очень полезны при создании изолированных функций, поскольку между вызовами их можно использовать в других частях программы. Если бы статических переменных не было, пришлось бы применять глобальные переменные, порождая неизбежные побочные эффекты. Примером удачного применения статических переменных является генератор чисел, который порождает новое число, используя предыдущее значение. Для хранения этого числа можно было бы использовать глобальную переменную, однако при каждом новом вызове ее пришлось бы объявлять заново, постоянно проверяя, не конфликтует ли она с другими глобальными переменными. Применение статической переменной легко решает эту проблему:

```
int series()
{
    static int series_num;
    series_num += 23;
    return series_num;
}
```

В этом примере переменная `series_num` продолжает существовать между вызовами функции, а локальная переменная каждый раз создавалась бы при входе и уничтожалась при выходе из функции. Таким образом, каждый вызов функции `series()` порождает новый элемент ряда, используя предыдущее значение и не прибегая к глобальной переменной.

Локальную статическую переменную можно инициализировать. Начальное значение присваивается лишь один раз, а не при каждом входе в блок, как это происходит с локальными переменными. Например, в приведенной ниже версии функции `series()` переменная `series_num` инициализируется числом 100:

```
int series()
{
    static int series_num = 100;
    series_num += 23;
    return series_num;
}
```

Теперь ряд будет всегда начинаться с числа 123. Хотя во многих приложениях это вполне приемлемо, обычно генераторы чисел предоставляют пользователю право выбора начального значения. Для этого можно сделать переменную `series_num` глобальной. Однако именно для того, чтобы избежать этого, и были созданы статические переменные. Это приводит ко второму способу использования статических переменных.

Практическим примером использования статических переменных является создание счетчика вызовов функции.

Занятие 40

40.1. Глобальные переменные

В отличие от локальных, глобальные переменные доступны из любой части программы и могут быть использованы где угодно. Кроме того, они сохраняют свои значения на всем протяжении выполнения программы. Объявления глобальных переменных должны размещаться вне всех функций. Эти переменные можно использовать в любом выражении, в каком бы блоке оно не находилось.

В приведенной ниже программе переменная `count` объявлена вне всех функций. Несмотря на то что ее объявление расположено до функции `main()`, эту переменную можно было бы с таким же успехом объявить в другом месте, но вне функций и до ее первого использования. И все же лучше всего размещать объявления глобальных переменных в самом начале программы:

```
#include <stdio.h>
int count;
void func2()
{
    int count;
    for(count = 1; count < 10; count++)
        putchar('.');
}
void func1()
{
    int temp;
    temp = count;
    func2();
    printf("count = %d", count);
}
void main()
{
    count = 100;
    func1();
}
```

Глобальные переменные хранятся в специально отведенном для них месте. Они оказываются весьма полезными, если разные функции в программе используют одни и те же данные. Однако, если в них нет особой необходимости, глобальных переменных следует избегать. Дело в том, что они занимают память во время всего выполнения программы, даже когда уже не нужны. Кроме того, использование глобальных переменных там, где можно было бы обойтись локальными, ослабляет независимость функций, поскольку они вынуждены зависеть от сущности, определенной в

другом месте. Итак, применение большого количества глобальных переменных повышает вероятность ошибок вследствие непредсказуемых побочных эффектов. Большинство проблем, возникающих при разработке больших программ, является следствием непредвиденного изменения значений переменных, которые используются в любом месте программы. Это может случиться, если в программе объявлено слишком много глобальных переменных.

Глобальные переменные инициализируются только при запуске программы. Неинициализированные глобальные переменные автоматически устанавливаются равными нулю.

40.2. Глобальные статические переменные

Применение спецификатора `static` к глобальной переменной заставляет компилятор создать глобальную переменную, видимую только в пределах текущего файла. Несмотря на то что эта переменная остается глобальной, в других файлах она не существует. Следовательно, изменить ее значение путем вмешательства извне невозможно. Это предотвращает побочные эффекты. В некоторых ситуациях, в которых локальные статические переменные оказываются недостаточными, можно создать небольшой файл, содержащий лишь функции, которые используют конкретную глобальную статическую переменную, отдельно скомпилировать его и применять без риска возникновения побочных эффектов.

Чтобы проиллюстрировать применение глобальной статической переменной, перепишем генератор чисел из предыдущего занятия таким образом, чтобы начальное значение задавалось при вызове функции `series_start()`:

```
static int series_num;
void series_start(int seed)
{
    series_num = seed;
}
int series(void)
{
    series_num += 23;
    return series_num;
}
```

Вызов функции `series_start()` инициализирует генератор чисел. После этого следующий элемент ряда порождается новым вызовом функции `series()`.

Локальные статические переменные видимы лишь в пределах блока, где они объявлены, а глобальные статические переменные – в пределах файла. Модификатор `static` позволяет скрывать часть программы от других модулей.

Занятие 41

41.1. Прототипы функций

Для объявления функций до своего первого вызова используются прототипы функций. В исходном варианте языка C прототипов не было. Они были добавлены при разработке стандарта, в котором настоятельно рекомендуется использовать прототипы. Следует заметить, что это лишь пожелание, а не категорическое требование. Если в программе используются прототипы, компилятор может обнаружить несоответствия между типами аргументов и параметрами функций, а также несовпадение их количества.

Общий вид прототипа таков:

тип имя_функции(тип парам1, тип парам2, ..., тип парамN);

Указывать имена параметров не обязательно. Однако они дают компилятору возможность обнаружить имя параметра, тип которого не совпадает с именем аргумента, поэтому рекомендуется использовать имена аргументов в прототипах.

Единственная функция, для которой не требуется прототип, – функция `main()`, поскольку она всегда вызывается первой.

В языке C прототип

тип `f()`;

означает отсутствие информации о параметрах. В зависимости от компилятора это может означать, что функция либо не имеет параметров, либо имеет несколько параметров. Поэтому прототип функции, не имеющей параметров, должен содержать ключевое слово `void`. Вот как должен выглядеть прототип функции `f()` в программе на языке C:

тип `f(void)`;

Это объявление сообщает компилятору, что функция не имеет параметров, и любой вызов этой функции с какими-либо параметрами является ошибкой.

41.2. Прототипы стандартных библиотечных функций

Программа должна содержать прототипы всех вызываемых стандартных библиотечных функций. Для этого в программу следует включить соответствующие заголовочные файлы, поддерживаемые компиляторами языка C и которые обычно имеют расширения `.h`.

Заголовочный файл состоит из двух элементов: определений переменных и функций, используемых в библиотечных функциях, а также прототипов самих библиотечных функций. Например, заголовочный файл `stdio.h` включается почти во все программы, поскольку он содержит прототип функции `printf()`.

Занятие 42

42.1. Рекурсия

В языке C функция может вызывать саму себя. Функции, вызывающие сами себя, называются рекурсивными. Рекурсия – это процесс определения какого-либо понятия через него же. Иногда его называют круговым определением.

Простым примером рекурсивной функции является функция `factr()`, вычисляющая факториал целого числа. Факториалом называется число `n`, представляющее собой произведение всех целых чисел от 1 до `n`. Например, факториал числа 3 равен $1 \times 2 \times 3 = 6$. Рассмотрим рекурсивный и итеративный варианты функции `factr()`:

```
/* Рекурсивный вариант */
unsigned long int factr(unsigned int n)
{
    if (n == 0) return 1;
    else return factr(n - 1)*n; // рекурсивный вызов
}
/* Итеративный вариант */
unsigned long int fact(unsigned int n)
{
    unsigned int t;
    unsigned long int answer;
    for(t = answer = 1; t <= n; answer *= t, t++);
    return(answer);
}
```

Итеративный вариант функции `factr()` выглядит проще. В нем используется цикл, счетчик которого пробегает значения от 1 до `n` и последовательно умножается на предыдущий результат.

Рекурсивная версия функции `factr()` несколько сложнее. Если функция `factr()` вызывается с аргументом 0, она возвращает число 1. В противном случае она возвращает значение `factr(n-1)*n`. Чтобы вычислить это выражение, функция `factr()` вызывается `n-1` раз до тех пор, пока значение переменной `n` не станет равным 0.

В этот момент начнется последовательное выполнение операторов `return`, относящихся к разным вызовам функции `factr()`.

При вычислении факториала числа 2 первый вызов функции `factr()` порождает второй вызов этой функции – теперь уже с аргументом, равным 1. Второй вызов порождает третий – с аргументом, равным 0. Он вернет число 1, которое будет умножено сначала на 1, а потом на 2 (исходное значение аргумента `n`). Попробуйте сами проследить за вызовами функции `factr()` при вычислении факториала числа 3. Чтобы увидеть уровень каждого вызова функции

factr()), можно вставить в нее вызов функции printf() и вывести промежуточные результаты.

Когда функция вызывает саму себя, в стеке размещается новый набор ее локальных переменных и параметров, и функция выполняется с начала. Рекурсивный вызов не создает новую копию функции. Копируются лишь переменные, с которыми она работает. При каждом рекурсивном возврате управления копии переменных и параметров удаляются из стека, а выполнение программы возобновляется с места вызова функции. Рекурсивные функции напоминают телескоп, который то раскладывается, то складывается.

Как правило, рекурсивные функции незначительно уменьшают размер кода и ненамного повышают эффективность использования памяти по сравнению с ее итеративными аналогами. Кроме того, рекурсивные версии большинства функций выполняются несколько медленнее, чем их итеративные эквиваленты. Большое количество рекурсивных вызовов может вызвать переполнение стека, поскольку при каждом вызове функции в стеке размещается новый набор ее локальных переменных и параметров. При этом программист может пребывать в полном неведении, пока не произойдет аварийное прекращение работы программы.

Основное преимущество рекурсивных функций заключается в том, что они упрощают и делают нагляднее некоторые алгоритмы. Например, алгоритм быстрой сортировки трудно реализовать итеративным способом. Кроме того, некоторые проблемы, связанные с искусственным интеллектом, легче решить, применяя рекурсию. В заключение отметим, что некоторым людям легче думать рекурсивно, чем итеративно.

Создавая рекурсивную функцию, необходимо предусмотреть условный оператор, например, оператор if, с помощью которого можно прекратить выполнение рекурсивных вызовов. Если этого не сделать, функция может бесконечно вызывать саму себя. Отсутствие условия остановки рекурсии является весьма распространенной ошибкой. Для отслеживания промежуточных результатов следует широко применять функцию printf(). Это позволит сохранять контроль за вычислениями и прервать выполнение программы, если возникла ошибка.

42.2. Задание

Выполнить задание 20.5 рекурсивно. Создать две рекурсивные функции: первая возвращает результирующее число в правильном порядке, вторая – в обратном.

Занятие 43

43.1. Определение списка параметров переменной длины

В программе можно объявить функцию с переменным количеством параметров. В большинстве случаев это относится к функции printf(). Чтобы сообщить компилятору, что функция может быть вызвана с разным количеством параметров, объявление ее параметров следует завершить многоточием. Например, прототип, приведенный ниже, означает, что функция func() может иметь по меньшей мере два параметра и сколько угодно дополнительных параметров (или вовсе не иметь их):

```
int func(int a, int b, ...);
```

Этот способ объявления параметров используется и в определении функции.

Любая функция с переменным количеством параметров должна иметь по меньшей мере один фактический параметр. Например, следующее объявление является неправильным:

```
int func(...); /* Неправильно */
```

43.2. Объявление параметров функции в классическом и современном стиле

Объявление параметров функции в ранних версиях языка C отличается от стандартного (современного). Иногда этот способ называют классическим. Стандарт языка C допускает оба стиля, но настоятельно рекомендует использовать лишь современный.

Классическое объявление параметров функции состоит из двух частей: списка параметров, заключенного в скобки, и указанного вслед за именем функции, и объявлений фактических параметров, расположенных между заголовком и телом функции:

```
тип имя_функции(парам1, ..., парамN)
```

```
тип парам1;
```

```
...
```

```
тип парамN;
```

```
{ тело функции }
```

Например, современное объявление параметров

```
float f(int a, int b, char ch)
```

```
{ /* ... */ }
```

в классическом варианте выглядит так:

```
float f(a, b, ch)
```

```
int a, b;
```

```
char ch;
```

```
{ /* ... */ }
```

Обратите внимание на то, что классическое определение позволяет объявлять несколько параметров одновременно.

Занятие 44

44.1. Аргументы функции main()

Иногда бывает удобно вводить данные, указывая их в командной строке при запуске программы. Для этого обычно используются аргументы функции main(). Аргументами командной строки называются данные, указанные в командной строке операционной системы вслед за именем программы. Например, часто используют командную строку

```
ping 192.168.100.3
```

Здесь параметр 192.168.100.3 является аргументом командной строки, который задает IP-адрес компьютера в сети.

Для получения аргументов командной строки предназначены встроенные аргументы функции main(): argv и argc. Целочисленный параметр argc содержит количество аргументов командной строки. Его значение не может быть меньше 1, поскольку имя программы считается первым аргументом. Параметр argv представляет собой указатель на массив символьных указателей. Каждый элемент этого массива ссылается на аргумент командной строки. Все аргументы командной строки являются строками – введенные числа должны конвертироваться в соответствующее внутреннее представление. Рассмотрим простую программу, которая выводит на экран строку "Привет," и ваше имя, введенное в командной строке:

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
    if (argc != 2) {
        printf("Вы забыли указать свое имя\n");
        exit(1);
    }
    printf("Привет, %s!", argv[1]);
}
```

Если в командной строке набрать имя программы и указать свой ник, например, lesson44 xela, то программа выведет на экран строку "Привет, xela!".

Во многих операционных системах каждый аргумент командной строки должен отделяться пробелом или знаком табуляции. Запятые, точки с запятой и другие знаки пунктуации разделителями не считаются. Рассмотрим пример:

```
run Spot, run
```

Эта командная строка состоит из трех строк. А командная строка

```
run,Spot,run
```

содержит только одну строку, поскольку запятая разделителем не является.

В некоторых операционных системах строку с пробелами можно заключать в двойные кавычки. В этом случае она считается отдельным аргументом. Прежде чем передавать параметры через командную строку, прочитайте документацию, сопровождающую вашу операционную систему.

Аргумент argv необходимо объявлять самому. Чаще всего применяется следующий способ:

```
char * argv[];
```

Пустые квадратные скобки означают, что массив имеет неопределенную длину. Доступ к отдельным элементам массива argv осуществляется с помощью индексации. Например, элемент argv[0] ссылается на первую строку, которая всегда является именем программы, элемент argv[1] ссылается на первый аргумент и т.д.

Рассмотрим программу countdown, в которой используются аргументы командной строки. Счетчик цикла в этой программе уменьшается с начального значения, заданного аргументом командной строки, до 0. Обратите внимание на то, что первый аргумент преобразуется в целое число с помощью стандартной функции atoi(). Если вторым аргументом является строка "display", значение счетчика выводится на экран:

```
/* Программа countdown */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
void main(int argc, char *argv[])
{
    int disp, count;
    if (argc < 2)
    {
        printf("Введите начальное значение счетчика в командной строке. Попробуйте еще\n");
        exit(1);
    }
    if (argc == 3 && !strcmp(argv[2], "display"))
        disp = 1;
    else disp = 0;
    for (count = atoi(argv[1]); count; --count)
        if(disp) printf("%d\n", count);
    putchar('\a'); /* Сигнал */
    printf("Готово");
}
```

Если командная строка не содержит аргументов, на экран выдается сообщение об ошибке. Программа, ожидающая ввода аргументов из командной строки, должна анализировать их и давать пользователю соответствующие инструкции.

Чтобы выделить отдельный символ из аргумента командной строки, необходимо использовать индекс массива `argv`. Следующая программа выводит на экран все свои аргументы посимвольно:

```
#include <stdio.h>
void main(int argc, char *argv[])
{
    int t, i;
    for (t = 0; t < argc; ++t) {
        i = 0;
        while (argv[t][i]) {
            putchar(argv[t][i]);
            ++i;
        }
        printf("\n");
    }
}
```

Первый индекс предоставляет доступ к строке, а второй – к отдельному символу.

44.2. Использование аргументов функции `main()`

Обычно аргументы `argc` и `argv` используются для передачи начальных параметров программы. Теоретически можно указывать до 32767 аргументов, однако в большинстве операционных систем это количество намного меньше.

Как правило, аргументы командной строки задают имя файла или какую-либо опцию. Применение командных аргументов придает программе профессиональный вид и позволяет применять пакетные файлы.

Если программа не использует аргументов, указываемых в командной строке, как правило, функция `main()` явно объявляется без параметров. В программах на языке C для этого достаточно просто указать в списке параметров ключевое слово `void`. Список параметров пустым оставлять не рекомендуется. Начиная с этого занятия, при отсутствии аргументов у функции `main()`, будем писать `void main(void)`.

Несмотря на то что имена `argc` и `argv` являются традиционными, аргументы функции `main()` можно называть как угодно. Кроме того, компиляторы могут поддерживать расширенный список аргументов функции `main()`. Информацию об этом следует искать в документации.

Занятие 45

45.1. Что такое указатель

Существуют три причины, по которым невозможно написать хорошую программу на языке C без использования указателей. Во-первых, указатели позволяют функциям изменять свои аргументы. Во-вторых, с помощью указателей осуществляется динамическое распределение памяти. И, в-третьих, указатели повышают эффективность многих процедур.

Указатели – одно из самых мощных и, в то же время, самых опасных средств любого языка программирования, а особенно C. Например, неинициализированные указатели (или указатели, содержащие неверные адреса) могут уничтожить операционную систему компьютера. И, что еще хуже, неправильное использование указателей порождает ошибки, которые крайне трудно обнаружить.

Указатель – это переменная, в которой хранится адрес другого объекта (как правило, другой переменной). Например, если одна переменная содержит адрес другой переменной, говорят, что первая переменная ссылается на вторую (рис. 45.1).

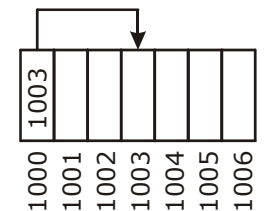


Рис. 45.1

45.2. Указатели

Переменная, хранящая адрес ячейки памяти, должна быть объявлена как указатель. Объявление указателя состоит из имени базового типа, символа `*` и имени переменной. Общая форма этого объявления такова:

тип_указателя *имя_указателя;

Здесь тип_указателя означает базовый тип указателя. Им может быть любой допустимый тип.

С формальной точки зрения указатель любого типа может ссылаться на любое место в памяти. Однако операции адресной арифметики тесно связаны с базовым типом указателей, поэтому очень важно правильно их объявить (адресная арифметика обсуждается на занятии 50).

Занятие 46

Адрес переменной

Существуют два специальных оператора для работы с указателями: оператор разыменования указателя * и оператор получения адреса &.

Оператор & является унарным и возвращает адрес своего операнда. Не забывайте: унарные операторы имеют только один операнд. Например, оператор присваивания

```
m = &count;
```

записывает в указатель m адрес переменной count. Этот адрес относится к ячейке памяти, которую занимает переменная count. Адрес и значение переменной никак не связаны друг с другом. Оператор & означает "адрес". Следовательно, предыдущий оператор присваивания можно прочитать так: присвоить указателю m адрес переменной count.

Если переменная занимает несколько ячеек памяти, ее адресом считается адрес первой ячейки.

Необходимо иметь гарантии, что указатель всегда ссылается на переменную правильного типа (исключением является указатель на void). Например, если в программе объявлен указатель на целочисленную переменную, компилятор полагает, что адрес, который в нем содержится, относится к переменной типа int, независимо от того, так ли это на самом деле. Поскольку указателю можно присвоить любой адрес, следующая программа будет скомпилирована без ошибок, но желаемого результата не даст:

```
void main(void)
{
    double x = 100.1;
    int *p;
    /* Следующий оператор заставит указатель p (имеющий целочисленный тип) ссылаться на переменную типа double */
    p = &x;
}
```

Рассмотрим пример, в котором показано, что при объявлении переменных они располагаются в памяти последовательно:

```
#include <stdio.h>
void main(void)
{
    double x;
    char a, b;
    void *p1 = &x, *p2 = &a, *p3 = &b;
    printf("%p %p %p\n", p1, p2, p3);
}
```

Занятие 47

47.1. Оператор разыменования

Оператор разыменования указателя * является антиподом оператора &. Этот унарный оператор возвращает значение, хранящееся по указанному адресу. Например, если указатель m содержит адрес переменной count, то оператор присваивания

```
q = *m;
```

поместит значение count в переменную q. Следовательно, переменная q станет равной 100, поскольку именно это число записано в ячейке, адрес которой хранится в указателе m. Символ * можно интерпретировать как "значение, хранящееся по адресу". В данном случае предыдущий оператор означает: "присвоить переменной q значение, хранящееся по адресу m".

Приоритет операторов & и * выше, чем приоритет всех арифметических операторов, за исключением унарного минуса.

Как было сказано на предыдущем занятии, указатель должен ссылаться на переменную правильного типа. Модифицируем программу приведенную ранее:

```
#include <stdio.h>
void main(void)
{
    double x = 100.1, y;
    int *p;
    p = &x;
    y = *p;
    printf("%f\n", y);
}
```

Эта программа никогда не присвоит переменной y значение переменной x. Поскольку указатель p является целочисленным, в переменную y будут скопированы лишь 4 байта (поскольку целые числа занимают 4 байта), а не 8.

Для того, чтобы присвоение прошло правильно, необходимо либо указатель p описать типа double, либо ввести переменную целого типа и привести явно переменную x к этому типу, а затем указателю присвоить адрес этой переменной.

47.2. Связанные переменные

Две переменные, значения которых находятся по одному адресу, называются связными. Если меняется значение одной переменной, то меняется соответственно значение и второй переменной.

Механизм создания связанных переменных детально будет рассмотрен на занятии 68.

Занятие 48

48.1. Понятие стека и динамической памяти

Указатели позволяют динамически распределять память в программах на языке С. Термин динамическое распределение памяти означает, что программа может получать необходимую ей память уже в ходе своего выполнения.

Как известно, память для глобальных переменных выделяется на этапе компиляции. Локальные переменные хранятся в стеке. Следовательно, в ходе выполнения программы невозможно объявить новые глобальные или локальные переменные. И все же иногда в ходе выполнения программы возникает необходимость выделить дополнительную память, размер которой заранее не известен.

Например, программа может использовать динамические структуры – связанные списки или деревья. Такие структуры данных являются динамическими по своей природе. Они увеличиваются или уменьшаются по мере необходимости. Для того чтобы реализовать такие структуры данных, программа должна иметь возможность распределять и освобождать память.

48.2. Система динамического распределения памяти

Память, выделяемая функциями динамического распределения, находится в куче (heap), которая представляет собой область свободной памяти, расположенную после кода программы, сегмента данных и стека (рис. 48.1). Хотя размер кучи заранее не известен, ее объем обычно достаточно велик.



Рис. 48.1

Пример из занятия 46 как раз и демонстрирует то, что показано на рисунке – у самой первой локальной переменной самый большой адрес, поскольку она находится в стеке.

Занятие 49

49.1. Функции динамического распределения памяти

В основе системы динамического распределения памяти в языке С лежат функции `malloc()` и `free()`. Во многих компиляторах предусмотрены дополнительные функции, позволяющие динамически выделять память, однако эти две – самые важные. Эти функции создают и поддерживают список свободной памяти. Функция `malloc()` выделяет память для объектов, а функция `free()` освобождает ее. Иными словами, при каждом вызове функция `malloc()` выделяет дополнительный участок памяти, а функция `free()` возвращает его операционной системе. Любая программа, использующая эти функции, должна включать в себя заголовочный файл `stdlib.h`.

49.2. Функция `malloc()`

Прототип функции `malloc()` имеет следующий вид:

```
void *malloc(size_t количество_байтов)
```

Параметр `количество_байтов` задает размер памяти, которую необходимо выделить. Тип `size_t` определен в заголовочном файле `stdlib.h` как целое число без знака. Функция `malloc()` возвращает указатель типа `void *`. Это означает, что его можно присваивать указателю любого типа. В случае успеха функция `malloc()` возвращает указатель на первый байт памяти, выделенной в куче. Если размера кучи недостаточно для успешного выделения памяти, функция `malloc()` возвращает нулевой указатель. В приведенном ниже фрагменте программы выделяется 1000 байт непрерывной памяти:

```
char *p;  
p = malloc(1000);           /* выделить 1000 байт */
```

После выполнения оператора присваивания указатель `p` ссылается на первый из 1000 байт выделенной памяти.

Обратите внимание на то, что в примере указатель, возвращаемый функцией `malloc()`, присваивается указателю `p` без приведения типа. В языке С это допускается, поскольку указатель типа `void *` автоматически преобразовывается в тип указателя, стоящего в левой части оператора присваивания, но все-таки рекомендуется использовать приведение типа:

```
p = (char *) malloc(1000);
```

В следующем примере выделяется память для 50 целых чисел. Обратите внимание на то, что применение оператора `sizeof` гарантирует машинезависимость этого фрагмента программы.

```
int *p;  
p = (int *) malloc(50*sizeof(int));
```

Поскольку размер кучи ограничен, при выделении памяти необходимо проверять указатель, возвращаемый функцией `malloc()`. Если он равен нулю, продолжать выполнение программы опасно. Проиллюстрируем эту мысль следующим примером:

```
p = (int *) malloc(100);
if (!p)
{
    printf("Память исчерпана\n");
    exit(1);
}
```

Разумеется, если указатель `p` является нулевым, не обязательно выпрыгивать на ходу – вызов функции `exit()` можно заменить какой-нибудь обработкой возникшей ошибки и продолжить выполнение программы. Достаточно убедиться, что указатель `p` больше не равен нулю.

49.3. Статический оператор `sizeof`

Статический унарный оператор `sizeof` вычисляет длину операнда в байтах. Операндом может быть как отдельная переменная, так и имя типа, заключенное в скобки. Поскольку размер типа `int` равен 4 байта, а типа `double` – 8 байт (в Microsoft Visual Studio .NET), то в этом случае фрагмент программы выведет числа 8 и 4:

```
double x;
printf("%d %d\n", sizeof x, sizeof(int));
```

Следует помнить, что для вычисления размера типа его имя должно быть заключено в скобки. Имя переменной в скобки заключать не обязательно, хотя и вреда от этого не будет.

С формальной точки зрения результат оператора `sizeof` имеет тип `size_t`, но на практике его можно считать целым без знака.

Оператор `sizeof` позволяет создавать машинонезависимые программы, в которых автоматически учитываются размеры типов.

49.4. Функция `free()`

Функция `free()` является антиподом функции `malloc()` – она освобождает ранее занятую область динамической памяти. Освобожденную память можно снова использовать с помощью повторного вызова функции `malloc()`. Прототип функции `free()` выглядит следующим образом:

```
void free(void *p)
```

Здесь параметр `p` является указателем на участок памяти, ранее выделенный функцией `malloc()`. Принципиально важно то, что функцию `free()` ни в коем случае нельзя вызывать с неправильным аргументом – это разрушит список свободной памяти.

Занятие 50

50.1. Присваивание указателей

Указатель можно присваивать другому указателю. Рассмотрим пример:

```
#include <stdio.h>
void main(void)
{
    int x;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;
    printf("%p", p2);    /* Выводит адрес переменной
                           x, а не ее значение */
}
```

Теперь на переменную `x` ссылаются оба указателя `p1` и `p2`. Адрес переменной `x` выводится на экран с помощью форматного спецификатора `%p` и функции `printf()`.

50.2. Адресная арифметика

К указателям можно применять только две арифметические операции: сложение и вычитание. Предположим, что указатель `p1` ссылается на целочисленную переменную, размещенную по адресу 2000. Поскольку целые числа занимают 4 байта в памяти компьютера, то после вычисления выражения

```
p1++;
```

переменная `p1` будет равна не 2001, а 2004, поскольку при увеличении указателя `p1` на единицу он ссылается на следующее целое число. Это же относится и к оператору декрементации. Например, если указатель `p1` равен 2000, выражение

```
p1--;
```

присвоит указателю `p1` значение 1996.

Таким образом, существуют следующие правила адресной арифметики. При увеличении указатель ссылается на ячейку, в которой хранится следующий элемент базового типа. При уменьшении он ссылается на предыдущий элемент. Для указателей на символы сохраняются правила "обычной" арифметики, поскольку размер символов равен 1 байт. Все остальные указатели увеличиваются или уменьшаются на длину соответствующих переменных, на которые они ссылаются. Это гарантирует, что указатели всегда будут ссылаться на элемент базового типа.

Действия над указателями не ограничиваются операторами инкрементации и декрементации. Например, к ним можно добавлять и целые числа. Выражение

```
p1 = p1 + 12;
```

смещает указатель p1 на 12-й элемент базового типа, расположенный после текущего адреса.

Кроме того, указатели можно вычитать. Это позволяет определить количество объектов базового типа, расположенных между двумя указателями. Все другие арифметические операции запрещены. В частности, указатели нельзя складывать, умножать и делить. К ним нельзя применять побитовые операции, суммировать их со значениями переменных, имеющих тип float, double и т.п., а также вычитать такие значения из них.

50.3. Сравнение указателей

Указатели можно сравнивать между собой. Например, следующий оператор, в котором сравниваются указатели p и q, является совершенно правильным:

```
if (p < q)
    printf("Указатель p содержит меньший адрес,
           чем указатель q\n");
```

Как правило, указатели сравниваются между собой, когда они ссылаются на один и тот же объект, например массив. Рассмотрим в качестве примера пару функций для работы со стеками, которые записывают и извлекают целые числа. Стек – это список, в котором доступ к элементам осуществляется по принципу "первым вошел, последним вышел". Его часто сравнивают со стопкой тарелок на столе – нижняя тарелка будет снята последней. Стеки используются в компиляторах, интерпретаторах, программах обработки электронных таблиц и других системных утилитах. Чтобы создать стек, необходимы две функции: push() и pop(). Функция push() заносит числа в стек, а функция pop() извлекает их оттуда. В приведенной ниже программе они управляются функцией main(). При вводе числа с клавиатуры программа заталкивает его в стек. Если пользователь ввел число 0, значение выталкивается из стека. Программа прекратит свою работу, когда пользователь введет число -1:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 50
void push(int i);
int pop(void);
int *tos, *p1, stack[SIZE];
void main(void)
{
    int value;
    tos = stack;      // tos ссылается на вершину стека
```

```
p1 = stack;          // инициализация указателя p1
do {
    printf("Введите число: ");
    scanf("%d", &value);
    if (value != 0) push(value);
    else printf("Число на вершине стека равно
               %d\n", pop());
} while (value != -1);
}
void push(int i)
{
    p1++;
    if (p1 == (tos + SIZE))
    {
        printf("Стек переполнен\n");
        exit(1);
    }
    *p1 = i;
}
int pop(void)
{
    if (p1 == tos)
    {
        printf("Стек исчерпан\n");
        exit(1);
    }
    p1--;
    return *(p1 + 1);
}
```

Как видим, стек реализован в виде массива stack. Сначала указатели p1 и tos ссылаются на первый элемент стека. Затем указатель p1 начинает перемещаться по стеку, а переменная tos по-прежнему хранит адрес его вершины. Это позволяет избежать переполнения стека и обращения к пустому стеку. Функции push() и pop() можно применять сразу после инициализации стека. В каждой из них выполняется проверка, не вышел ли указатель за пределы допустимого диапазона значений. В функции push() это позволяет предотвратить переполнение (значение указателя p1 не должно превышать адрес tos + SIZE, где переменная SIZE определяет размер стека). В функции pop() эта проверка позволяет избежать обращения к пустому стеку (значение указателя p1 не должно быть меньше указателя tos, ссылающегося на вершину).

Обратите внимание на то, что в функции pop() значение, возвращаемое оператором return, необходимо заключить в скобки. Без них возвращается значение, записанное по адресу p1, увеличенное на единицу.

Занятие 51

Косвенная адресация

Иногда указатель может ссылаться на другой указатель, который, в свою очередь, содержит адрес обычной переменной. Такая схема называется косвенной адресацией, или указателем на указатель. Применение косвенной адресации снижает наглядность программы.

Концепция косвенной адресации проиллюстрирована на рис. 51.1. Обычный указатель хранит адрес объекта, содержащего требуемое значение. В случае косвенной адресации один указатель ссылается на другой указатель, а тот, в свою очередь, содержит адрес объекта, в котором записано нужное значение.

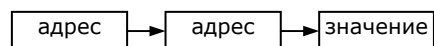


Рис. 51.1

Глубина косвенной адресации не ограничена, однако почти всегда можно обойтись "указателем на указатель". Более громоздкие схемы адресации труднее понять, следовательно, вероятность ошибок при их использовании резко возрастает.

Переменная, представляющая собой указатель на указатель, объявляется следующим образом: перед ее именем записывается дополнительная звездочка. Например, в следующем операторе переменная `newbalance` объявляется как указатель на указатель на число с плавающей точкой:

```
float **newbalance;
```

Следует помнить, что переменная `newbalance` не является указателем на число с плавающей точкой, она лишь ссылается на указатель этого типа.

Чтобы извлечь значение переменной с помощью косвенной адресации, необходимо дважды применить оператор разыменования:

```
#include <stdio.h>
void main(void)
{
    int x, *p, **q;
    x = 10;
    p = &x;
    q = &p;
    printf("%d", **q);    /* Вывод числа x */
}
```

Здесь переменная `p` объявлена как целочисленный указатель, а переменная `q` представляет собой указатель на целочисленный указатель. Функция `printf()` выводит на экран число 10.

Занятие 52

Инициализация указателей

Если нестатический локальный указатель объявлен, но не инициализирован, его значение остается неопределенным (глобальные и статические локальные указатели автоматически инициализируются нулем). При попытке применить указатель, содержащий неопределенный адрес, может разрушиться как программа, так и вся операционная система.

При работе с указателями большинство профессиональных программистов на языке C придерживаются следующего соглашения: указатель, не ссылающийся на конкретную ячейку памяти, должен быть равен нулю. По определению любой нулевой указатель ни на что не ссылается и не должен использоваться. Однако это еще не гарантирует безопасности. Использование нулевого указателя – всего лишь общепринятое соглашение. Это вовсе не правило, диктуемое языком C. Например, если нулевой указатель поместить в левую часть оператора присваивания, риск разрушения программы и операционной системы остается высоким.

Поскольку предполагается, что нулевой указатель в вычислениях не используется, его можно применять для повышения наглядности и эффективности программ. Например, его можно использовать в качестве признака конца массива (см. занятие 53).

Рассмотрим пример поиска символа в области памяти, на которую ссылается указатель `p`. Признаком окончания данных является нулевой символ:

```
#include <stdio.h>
#include <stdlib.h>
int search(char *p, char name)
{
    int t;
    for (t = 0; *(p + t); t++)
        if (*(p + t) == name)
            return t;
    return -1;
}

void main(void)
{
    char *p = (char *) malloc(27*sizeof (char));
    int t;
    for (t = 0; t < 26; t++)
        *(p + t) = t + 'a';
    *(p + t) = '\0';
    printf("%d\n", search(p, 'f'));
}
```

Занятие 53

53.1. Создание указателя на массив

Имя массива является указателем на первый его элемент. Допустим, массив объявлен с помощью оператора

```
int sample[10];
```

Указателем на его первый элемент при этом является имя `sample`. Таким образом, в следующем фрагменте указателю присваивается адрес первого элемента массива `sample`:

```
int *p;
int sample[10];
p = sample;
```

Адрес первого элемента массива можно также вычислить с помощью оператора `&`. Например, выражения `sample` и `&sample[0]` эквивалентны. Однако в профессионально написанных программах на языке C никогда не встречается выражение `&sample[0]`.

53.2. Указатели и массивы

Как было сказано выше, указатели и массивы тесно связаны между собой. Рассмотрим следующий фрагмент программы:

```
char str[80], *p1;
p1 = str;
```

Здесь указателю `p1` присвоен адрес первого элемента массива `str`. Чтобы получить доступ к пятому элементу этого массива, следует выполнить один из двух операторов:

```
str[4]
```

или

```
*(p1 + 4)
```

Оба оператора вернут значение пятого элемента массива `str`. Поскольку индексация массивов начинается с нуля, то индекс пятого элемента массива `str` равен 4. Кроме того, пятый элемент массива можно получить, прибавив 4 к указателю `p1`, который вначале ссылается на первый элемент. Имя массива без индекса представляет собой указатель на начало массива, то есть на его первый элемент.

Этот пример можно обобщить. В языке C существуют два способа обращения к элементам массива: индексация и адресная арифметика. Хотя индексация массива нагляднее, адресная арифметика иногда оказывается эффективнее. Поскольку быстрое действие программы относится к одним из ее важнейших свойств, программисты на языке C широко используют указатели для обращения к элементам массива.

В следующем фрагменте программы приведены два варианта функции `putstr()`, иллюстрирующей доступ к элементам массива. В

первом варианте используется индексация, а во втором – адресная арифметика. Функция `putstr()` предназначена для посимвольной записи строки в стандартный поток вывода:

```
/* Индексация указателя */
void putstr(char *s)
{
    int t;
    for (t = 0; s[t]; ++t) putchar(s[t]);
}
/* Доступ с помощью указателя */
void putstr(char *s)
{
    while(*s) putchar(*s++);
}
```

Большинство профессиональных программистов сочтут вторую версию более понятной и наглядной. На практике именно этот способ доступа к элементам массива распространен наиболее широко.

53.3. Индексация многомерных массивов

Если переменная `a` – это указатель на целочисленный массив, состоящий из 10 строк и 10 столбцов, то следующие два оператора будут эквивалентными:

```
a
&a[0][0]
```

Более того, доступ к элементу, стоящему на пересечении первой строки и пятого столбца, можно получить двумя способами: либо индексируя массив – `a[0][4]`, либо используя указатель – `*((int *)a + 4)`. Аналогично элемент, стоящий во второй строке и третьем столбце, является значением выражений `a[1][2]` и `*((int *)a + 12)`. Для двухмерного массива справедлива следующая формула:

```
a[j][k] = * ((базовый_тип *) a + (j * длина_строки) + k)
```

Правила адресной арифметики требуют приведения типа указателя на массив к его базовому типу. Обращение к элементам массива с помощью указателей используется довольно широко, поскольку операции адресной арифметики выполняются быстрее, чем индексация.

Двухмерный массив можно представить с помощью указателя на массив одномерных массивов. Следовательно, используя отдельный указатель, можно обращаться к элементам, стоящим в строках двухмерного массива. Этот способ иллюстрируется следующей функцией. Она выводит на печать содержимое заданной строки глобального целочисленного массива `num`:


```

int num[10][10];
...
void pr_row(int j)
{
    int *p, t;
    p = (int *) &num[j][0]; /* Получить адрес первого
                               элемента в строке с
                               индексом j */

    for (t = 0; t < 10; ++t)
        printf("%d ", *(p + t));
}

```

Эту функцию можно обобщить, включив в список аргументов индекс строки, ее длину и указатель на первый элемент массива:

```

void pr_row(int j, int row_dimension, int *p)
{
    int t;
    p = p + (j * row_dimension);
    for (t = 0; t < row_dimension; ++t)
        printf("%d ", *(p + t));
}
...
void f(void)
{
    int num[10][10];
    pr_row(0, 10, (int *) num); // Вывод первой строки
}

```

Этот способ можно применять и к массивам более высокой размерности. Например, трехмерный массив можно свести к указателю на двумерный массив, который, в свою очередь, можно представить с помощью указателя на одномерный массив. В общем случае n -мерный массив можно свести к указателю на $(n - 1)$ -мерный массив и так далее. Процесс завершается получением указателей на одномерный массив.

53.4. Массивы указателей

Как и все обычные переменные, указатели можно помещать в массив. Объявление массива, состоящего из 10 целочисленных указателей, выглядит следующим образом:

```
int *x[10];
```

Чтобы присвоить адрес целочисленной переменной `var` третьему элементу массива указателей, нужно выполнить оператор `x[2] = &var;`

Чтобы извлечь значение переменной `var`, используя указатель `x[2]`, необходимо его разыменовать:

```
*x[2]
```

Массив указателей передается в функцию как обычно – достаточно указать его имя в качестве параметра. В следующем фрагменте на вход функции поступает массив указателей.

```

void display_array(int *q[])
{
    int t;
    for (t = 0; t < 10; t++)
        printf("%d ", *q[t]);
}

```

Переменная `q` не является указателем на целочисленные переменные, ее следует интерпретировать как указатель на массив целочисленных указателей. Следовательно, необходимо объявить, что параметр `q` представляет собой массив целочисленных указателей. Это объявление сделано в заголовке функции.

Массивы часто содержат указатели на строки. Попробуем создать функцию, выводящую на экран сообщение об ошибке с заданным номером.

```

void syntax_error(int num)
{
    static char *err[] = {
        "Невозможно открыть файл\n",
        "Ошибка при чтении\n",
        "Ошибка при записи\n",
        "Отказ оборудования\n";
    }

    printf("%s", err[num]);
}

```

Указатели на строки содержатся в массиве `err`. Функция `printf()` вызывается внутри функции `syntax_error()` и выводит на экран строку с указанным номером. Например, если параметр `num` равен 2, на экране появится сообщение "Ошибка при записи".

Аргумент командной строки `argv` (см. занятие 44) также представляет собой массив указателей на символьные переменные.

53.5. Нулевой указатель

Рассмотрим пример использования нулевого указателя в качестве признака конца массива, содержащего указатели:

```

int search(char *p[], char *name)
{
    int t;
    for (t = 0; p[t]; ++t)
        if (!strcmp(p[t], name))
            return t;
    return -1; /* Имя не найдено */
}

```

Занятие 54

Возврат указателей

Функции, возвращающие указатели, ничем не отличаются от обычных, но с ними связано несколько важных понятий.

Указатели не являются целочисленными переменными. Они содержат адреса переменных, имеющих определенный тип. Используя адресную арифметику, следует иметь в виду, что результаты ее операций зависят от базового типа указателей. Как правило, при каждом увеличении (или уменьшении) указателя на единицу, он перемещается на следующую (или предыдущую) переменную этого типа. Поскольку размер переменных, имеющих разные типы, варьируется, компилятор должен знать, с каким типом данных связан тот или иной указатель. По этой причине функция, возвращающая указатель, должна явно объявлять его тип. Например, нельзя применять оператор `return` для возврата указателя, имеющего тип `int *`, если в объявлении функции указано, что она возвращает указатель типа `char *`.

Чтобы вернуть указатель, в объявлении функции следует указать соответствующий тип возвращаемого значения. Например, следующая функция возвращает указатель на первое вхождение символа `c` в строку `s`:

```
/* Возвращает указатель на первое вхождение символа c
   в строку s */
char *match(char c, char *s)
{
    while (c != *s && *s) s++;
    return s;
}
```

Если символ не найден, возвращается ноль. Рассмотрим короткую программу, в которой используется функция `match()`:

```
#include <stdio.h>
char *match(char c, char *s); /* Прототип */
void main(void)
{
    char s[80], *p, ch;
    gets(s);
    ch = getchar();
    p = match(ch, s);
    if (*p) printf("%s\n", p);
    else printf("Символ не обнаружен\n");
}
```

Эта программа сначала считывает строку, а затем – символ. Если символ входит в строку, программа выводит ее на экран, начиная с позиции, в которой обнаружен искомый символ.

Занятие 55

Указатели на функции

Особенно малопонятным, хотя и действенным механизмом языка `C` являются указатели на функции. Несмотря на то, что функция не является переменной, она располагается в памяти, и, следовательно, ее адрес можно присваивать указателю. Этот адрес считается точкой входа в функцию. Именно он используется при ее вызове. Поскольку указатель может ссылаться на функцию, ее можно вызывать с помощью этого указателя. Это позволяет также передавать функции другим функциям в качестве аргументов.

Адрес функции задается ее именем, указанным без скобок и аргументов. Чтобы разобраться в этом механизме, рассмотрим следующую программу:

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,
            int (*cmp) (const char *, const char *));
void main(void)
{
    char s1[80], s2[80];
    int (*p) (const char *, const char *);
    p = strcmp;
    gets(s1);
    gets(s2);
    check(s1, s2, p);
}
void check(char *a, char *b,
            int (*cmp) (const char *, const char *))
{
    printf("Проверка равенства - ");
    if (!(*cmp) (a, b)) printf("равны\n");
    else printf("не равны\n");
}
```

При вызове функция `check()` получает два указателя на символные переменные и указатель на функцию. Соответствующие аргументы объявлены в ее заголовке. Обратите внимание на то, как объявлен указатель на функцию. Эту форму объявления следует применять для любых указателей на функции, независимо от того, какой тип имеют их аргументы и возвращаемые значения. Объявление `*cmp` заключено в скобки для того, чтобы компилятор правильно его интерпретировал. Выражение

```
(*cmp) (a, b)
```

внутри функции `check()` означает вызов функции `strcmp()`, на которую ссылается указатель `cmp`, с аргументами `a` и `b`. Скобки по-

прежнему необходимы. Это – один из способов вызвать функцию с помощью указателя. Альтернативный вызов выглядит так:

```
cmp(a, b);
```

Первый способ применяется чаще, поскольку он позволяет явно продемонстрировать, что функция вызывается через указатель. Тем не менее эти два способа эквивалентны.

Обратите внимание на то, что функцию `check()` можно вызывать, непосредственно указав функцию `strcmp()` в качестве ее аргумента:

```
check(s1, s2, strcmp);
```

В этом случае отпадает необходимость в дополнительном указателе на функцию.

Вызов функции с помощью указателя, на первый взгляд, лишь усложняет программу, не предлагая взамен никакой компенсации. Тем не менее, иногда выгоднее вызывать функции через указатели и даже создавать массивы указателей на функции. Рассмотрим в качестве примера синтаксический анализатор – составную часть компилятора, вычисляющую выражения. Он часто вызывает различные математические функции (синус, косинус, тангенс и так далее), средства ввода-вывода или функции доступа к ресурсам системы. Вместо создания большого оператора `switch`, в котором пришлось бы перечислять все эти функции, можно создать массив указателей на них. В этом случае к функциям можно было бы обращаться по индексу. Чтобы оценить эффективность такого подхода, рассмотрим расширенную версию предыдущей программы. В этом примере функция `check()` проверяет на равенство строки, состоящие из букв или цифр. Для этого она просто вызывает разные функции, выполняющие сравнение:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void check ...           // Из предыдущего примера
int numcmp(const char *a, const char *b)
{
    return (atoi(a) != atoi(b));
}

void main(void)
{
    char s1[80], s2[80];
    gets(s1);
    gets(s2);
    if (isalpha(*s1)) check(s1, s2, strcmp);
    else check(s1, s2, numcmp);
}
```

Занятие 56

Проблемы, возникающие при работе с указателями

Указатели – головная боль программистов. С одной стороны, они представляют собой чрезвычайно мощный и необходимый во многих ситуациях механизм. С другой стороны, если указатель содержит непредвиденное значение, обнаружить эту ошибку крайне трудно.

Неверный указатель трудно найти, поскольку сам по себе он ошибкой не является. Проблемы возникают тогда, когда, пользуясь ошибочным указателем, вы начинаете считывать или записывать информацию в неизвестной области памяти. Если Вы считываете ее, в худшем случае получите мусор, но если записываете – можете повредить данные, код программы и даже операционную систему. Сначала ошибка может никак не проявиться, однако в ходе выполнения программы это может привести к непредсказуемым последствиям. Такая ошибка напоминает мину замедленного действия, причем совершенно непонятно, где ее следует искать.

Рассмотрим несколько рекомендаций, позволяющих избежать ненужных проблем. Классический пример ошибки при работе с указателями – неинициализированный указатель:

```
void main(void)
{
    int x, *p;
    x = 10;
    *p = x;
}
```

В этой программе число 10 записывается в некую неизвестную область памяти. Причина заключается в том, что в момент присваивания `*p = x` указатель `p` содержит неопределенный адрес. Следовательно, невозможно предсказать, куда будет записано значение переменной `x`. Если программа невелика, такие ошибки могут себя никак не проявлять, поскольку вероятность того, что указатель `p` содержит "безопасный" адрес, достаточно велика, и ваши данные, программа и операционная система могут избежать опасности. Однако, если программа имеет большой размер, вероятность повредить жизненно важные области памяти резко возрастает. В конце концов программа перестанет работать. Для того чтобы избежать таких ситуаций, указатели всегда следует "заземлять", присваивая им корректные адреса.

Второй вид наиболее распространенных ошибок порождается простым неумением работать с указателями:

```
#include <stdio.h>
void main(void)
{
```

```

int x, *p;
x = 10;
p = x;
printf("%d", *p);
}

```

При вызове функции printf() число 10 не будет выведено на экран. Вместо этого вы получите совершенно непредсказуемое значение, поскольку присваивание `p = x` является неверным. Этот оператор присваивает число 10 указателю `p`. Однако указатель `p` должен содержать адрес, а не значение. Исправить программу можно следующим образом `p = &x`.

Еще одна разновидность ошибок вызывается неверными предположениями о размещении переменных в памяти компьютера. Программист не может заранее знать, где именно будут размещены его данные, окажутся ли они в том же месте при следующем запуске программы, и будут ли разные компиляторы обрабатывать их одинаково. Именно поэтому любые сравнения указателей, которые ссылаются на разные объекты, не имеют смысла:

```

void main(void)
{
    char s[80], y[80];
    char *p1, *p2;
    p1 = s;
    p2 = y;
    if (p1 < p2) ...
    ...
}

```

Этот фрагмент иллюстрирует типичную ошибку, связанную со сравнением указателей – в очень редких ситуациях такие сравнения позволяют определить взаимное местоположение переменных (и все же это скорее исключение, чем правило).

Аналогичные ошибки возникают, когда программист считает, будто два смежных массива можно индексировать одним указателем, пересекая границу между ними:

```

void main(void)
{
    int first[10], second[10];
    int *p, t;
    p = first;
    for(t=0; t<20; ++t)
        *p++ = t;
}

```

Не стоит рисковать, инициализируя массивы подобным образом. Даже если в некоторых ситуациях этот способ сработает, нет никакой гарантии, что массивы `first` и `second` всегда будут распо-

лагаться в соседних ячейках памяти.

Следующая программа демонстрирует крайне опасную ошибку. Попробуйте сами найти ее:

```

#include <string.h>
#include <stdio.h>
void main(void)
{
    char *p1;
    char s[80];
    p1 = s;
    do {
        gets(s);
        while (*p1)
            printf(" %d", *p1++);
    } while(strcmp(s, "готово"));
}

```

В этой программе указатель `p1` используется для вывода на печать ASCII-кодов символов, содержащихся в строке `s`. Проблема заключается в том, что адрес строки `s` присваивается указателю `p1` только один раз, поэтому при втором проходе цикла он будет содержать адрес ячейки, на которой остановился в прошлый раз – ведь адрес начала строки ему не присваивается снова. В этой ячейке может быть записано все что угодно: символ из второй строки, другая переменная и даже инструкция программы. Исправить программу следует так:

```

#include <string.h>
#include <stdio.h>
void main(void)
{
    char *p1;
    char s[80];
    do {
        p1 = s;
        gets(s);
        while (*p1)
            printf(" %d", *p1++);
    } while(strcmp(s, "готово"));
}

```

Теперь в начале каждой итерации указатель `p1` содержит адрес первого символа строки `s`. Итак, при повторном использовании указателя его необходимо инициализировать вновь.

Неправильная работа с указателями может привести к тяжелым последствиям. Однако это не значит, что от указателей следует отказаться. Просто нужно быть внимательным и следить за тем, чтобы указатели содержали корректные адреса.

Занятие 57

57.1. Массивы символов

Наиболее распространенным массивом является обычный массив символов:

```
char name[20];
```

Массив символов занимает в памяти компьютера количество байтов, равное длине массива.

Поскольку символ в памяти занимает ровно один байт, то все арифметические операции, которые можно выполнить над байтами, можно выполнить и над символами:

```
int num = name[1] - '0';
```

57.2. Простейшее шифрование данных

Приведем пример использования символьных массивов для шифрования текста. В программе рассмотрен один из простейших методов симметричного шифрования когда каждый символ исходного текста складывается с символом ключа по модулю 2 (^):

```
#include <stdio.h>
void shifr(char *text, int lt, char *key, int lk)
{
    int i, j;
    for (i = 0, j = 0; i < lt; i++, j++)
    {
        if (j >= lk) j = 0;
        text[i] ^= key[j];
    }
}

void putCarr(char *arr, int la)
{
    int i;
    for (i = 0; i < la; i++)
        printf("%c", arr[i]);
    printf("\n");
}

void main(void)
{
    char text[22] = "Строка для шифрования", key[4];
    printf("Введите ключ (3): ");
    gets(key);
    shifr(text, 21, key, 3);    // зашифруем
    putCarr(text, 21);
    shifr(text, 21, key, 3);    // расшифруем
    putCarr(text, 21);
}
```

Занятие 58

58.1. Строки

Чаще всего одномерный массив используется для представления строк символов. В языке C предусмотрены строки, завершающиеся нулевым байтом (null-terminated string), представляющие собой массивы символов, последним элементом которых является нулевой байт. Это единственный вид строки, предусмотренный в языке C. Иногда строки, завершающиеся нулевым байтом, называются C-строками (C-string).

Объявляя массив символов, следует зарезервировать одну ячейку для нулевого байта, то есть указать размер, на единицу больше, чем длина наибольшей предполагаемой строки. Например, чтобы объявить массив str, предназначенный для хранения строки, состоящей из 10 символов, следует выполнить следующий оператор:

```
char str[11];
```

В этом объявлении предусмотрена ячейка, в которой будет записан нулевой байт.

Строка символов, заключенных в двойные кавычки, например, "Здравствуйте, я ваша тетя!", называется строковой константой (string constant). Компилятор автоматически добавляет к ней нулевой байт, поэтому программист может не беспокоиться об этом:

```
char str[] = "Здравствуйте, я ваша тетя!";
```

Не следует путать ноль (0) и нулевой байт (\0), или нулевой символ. Признаком конца строки является именно нулевой байт. Если не указать обратную косую черту, ноль будет считаться обычным символом.

Несмотря на то что в языке C++ для строк предусмотрен отдельный класс, строки, завершающиеся нулевым байтом, по-прежнему весьма популярны. Возможно, это происходит благодаря тому, что они очень эффективны и позволяют программистам полностью контролировать операции над строками.

Учтите, что отношения "больше" и "меньше" между строками понимаются в лексикографическом смысле, то есть сравниваются ASCII-коды первых несовпадающих между собой символов.

58.2. Задание

1. Напишите функцию strend(s,t), которая возвращает 1, если строка t расположена в конце строки s, и ноль в противном случае.
2. Напишите функцию strbalance(s), которая возвращает 1, если в строке есть баланс скобок, и ноль иначе.

Занятие 59

Строки и указатели

Работа со строками с помощью указателей осуществляется как с обычными массивами. В качестве примера переделаем программу простейшего шифрования из занятия 57:

```
#include <stdio.h>
#include <stdlib.h>
void shifr(char *text, int num, char *key)
{
    char *pt, *pk;
    for (pt=text, pk=key; num; pt++, pk++, num--) {
        if (!(*pk)) pk = key;
        *pt ^= *pk;
    }
}
void putCarr(char *arr, int la)
{
    int i;
    char *pa = arr;
    for (i = 0; i < la; i++, pa++)
        printf("%c", *pa);
    printf("\n");
}
int length(char *text)
{
    int i = 0;
    char *p;
    for (p = text; *p; p++, i++);
    return i;
}
void main(void)
{
    char *text = (char*) malloc(26*sizeof (char)),
        *key = (char*) malloc(4*sizeof (char));
    int num;
    printf("Введите строку для шифрования (до 25): ");
    scanf("%25s", text);
    num = length(text);
    printf("Введите ключ (до 3): ");
    scanf("%3s", key);
    shifr(text, num, key), putCarr(text, num);
    shifr(text, num, key), putCarr(text, num);
}
```

В функцию shifr передается длина строки поскольку при шифровании может получиться символ с кодом 0, который будет трактоваться как конец строки.

Занятие 60

60.1. Чтение и запись строк

Функция gets() считывает строку символов, введенных с клавиатуры, и размещает их по адресу, указанному в аргументе. Символы на клавиатуре набираются до тех пор пока не будет нажата клавиша **Enter**. В конец строки ставится не символ перехода на новую строку, а нулевой байт. После этого функция gets() завершает свою работу. Функцию gets() в отличие от функции getchar() нельзя использовать для перехода на новую строку. Ошибки, допущенные при наборе строки, можно исправить с помощью клавиши **BackSpace**. Прототип функции gets() имеет следующий вид:

```
char *gets(char *строка)
```

Здесь параметр строка представляет собой массив, в который записываются символы, введенные пользователем. Следующая программа считывает строку в массив str и выводит его длину:

```
#include <stdio.h>
#include <string.h>
void main(void)
{
    char str [80];
    gets(str);
    printf("Длина массива равна %d", strlen(str));
}
```

При работе с функцией gets () следует проявлять осторожность, поскольку она не проверяет выход индекса массива за пределы допустимого диапазона. Следовательно, количество символов, введенных пользователем, может превысить длину массива. Хотя функция gets() прекрасно работает в простых программах, ее не следует применять в коммерческих приложениях. Ее альтернативой является функция fgets(), которая предотвращает переполнение массива.

Функция puts () выводит на экран строку символов и переводит курсор на следующую строку экрана. Ее прототип такой

```
int puts (const char *строка)
```

Функция puts(), как и функция printf(), распознает эскейп-последовательности например '\n', предназначенную для перехода на новую строку. На вызов функции puts() тратится меньше ресурсов, чем на функцию printf(), поскольку она может выводить лишь строки, но не числа. Кроме того, она не форматирует вывод. Таким образом, функция puts() занимает меньше места и выполняется быстрее чем функция printf(). По этой причине функцию puts() часто применяют для оптимизации кода. Если при выводе возникла ошибка, функция puts() возвращает константу EOF. В против-

ном случае она возвращает неотрицательное значение. Однако при выводе данных на консоль программисты редко учитывают возможности ошибки, поэтому значение, возвращаемое функцией puts(), проверяется редко.

60.2. Библиотека string.h

В языке C предусмотрен богатый выбор функций для работы со строками. Эти функции объявлены в стандартном заголовочном файле string.h.

Строковые функции оперируют массивами, завершающимися нулевым символом. Для работы с ними необходимо включить в программу заголовки <string.h>.

Поскольку в языке C проверка выхода за пределы диапазона не предусмотрена, ответственность за предотвращение переполнения массива возлагается на программиста. Пренебрежение этой опасностью может закончиться крахом программы. В языке C печатаемым называется символ, который можно отобразить на экране. Обычно такие символы расположены между пробелом (0x20) и тильдой (0xFE). Управляющие символы имеют значения, лежащие в диапазоне от 0 до 0x1F, а также символ del (0x7F).

По историческим причинам параметрами символьных функций считаются целые числа, при этом используется только их младший байт. Символьные функции автоматически преобразовывают свои аргументы в тип unsigned char.

60.3. Функции работы с символами

Функция isalnum
`int isalnum(int ch);`

Если аргумент является буквой или цифрой, функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция isalpha
`int isalpha(int ch);`

Если аргумент является буквой, функция возвращает ненулевое значение, в противном случае возвращается нуль. Является ли символ буквой, зависит от языка.

Функция iscntrl
`int iscntrl(int ch);`

Если аргумент ch лежит в диапазоне от нуля до 0x1F или равен 0x7F (del), функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция isdigit
`int isdigit(int ch);`

Если аргумент является цифрой, функция возвращает ненулевое

значение, в противном случае возвращается нуль.

Функция isgraph
`int isgraph(int ch);`

Если аргумент является печатаемым символом, отличным от пробела, функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция islower
`int islower(int ch);`

Если аргумент является строчной буквой, функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция isprint
`int isprint(int ch);`

Если аргумент является печатаемым символом, включая пробел, функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция ispunct
`int ispunct(int ch);`

Если аргумент является знаком пунктуации, функция возвращает ненулевое значение, в противном случае возвращается нуль. К знакам пунктуации относятся все печатаемые символы, не являющиеся буквами, цифрами и пробелами.

Функция isspace
`int isspace(int ch);`

Если аргумент является пробелом, знаком горизонтальной или вертикальной табуляции, символом возврата каретки или перехода на новую строку, функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция isupper
`int isupper(int ch);`

Если аргумент является прописной буквой, функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция isxdigit
`int isxdigit(int ch);`

Если аргумент является шестнадцатеричной цифрой, функция возвращает ненулевое значение, в противном случае возвращается нуль.

Функция tolower
`int tolower(int ch);`

Если символ ch является буквой, функция возвращает его строчный эквивалент. В противном случае символ не изменяется.

Функция toupper
`int toupper(int ch);`

Если символ ch является буквой, функция возвращает его прописной эквивалент. В противном случае символ не изменяется.

Занятие 61

61.1. Конкатенация строк

Функция strcat

```
char *strcat(char *str1, const char *str2);
```

Функция конкатенирует копию строки str2 в строку str1 и записывает в конец строки str1 нулевой символ. Исходный нулевой символ, содержащийся в строке str1, накрывается первым символом строки str2. Строка str2 остается неизменной. Если массивы перекрываются, поведение функции становится неопределенным.

Функция strncat

```
char *strncat(char *str1, const char *str2, int count);
```

Функция конкатенирует первые count символов строки str2 в строку str1 и записывает в конец строки str1 нулевой символ. Исходный нулевой символ, содержащийся в строке str1, накрывается первым символом строки str2. Строка str2 остается неизменной. Если массивы перекрываются, поведение функции становится неопределенным.

Функции strcat() и strncat() возвращает указатель на строку str1. Следует помнить, что проверка выхода за пределы допустимого диапазона при копировании строк не выполняется, поэтому программист должен сам гарантировать, чтобы размер строки str1 был достаточен для хранения исходного содержимого строки str1 и содержимого строки str2.

61.2. Сравнения строк

Функция memcmp

```
int memcmp(const void *buf1, const void *buf2,  
           int count);
```

Функция сравнивает первые count элементов массивов, на которые ссылаются указатели buf1 и buf2.

Она возвращает целое число: если оно больше нуля, то массив buf1 меньше массива buf2; если равно нулю – массив buf1 равен массиву buf2; меньше нуля – массив buf1 больше buf2.

Функция strcmp

```
int strcmp(const char *str1, const char *str2);
```

Функция strcmp() выполняет лексикографическое сравнение двух строк, возвращая целое число, интерпретация которого аналогична функции memcmp().

Функция strncmp

```
int strncmp(const char *str1, const char *str2,  
            int count);
```

Функция выполняет лексикографическое сравнение первых count символов двух строк, завершающихся нулевым байтом.

Занятие 62

62.1. Функции работы с символом в строке

Функция memchr

```
void *memchr(const void *buffer, int ch, int count);
```

Функция ищет в массиве buffer первое вхождение символа ch среди первых count элементов. В случае успеха она возвращает указатель на первое вхождение символа ch в массив buffer, в противном случае возвращается нулевой указатель.

Функция strchr

```
char *strchr(const char *str, int ch);
```

Функция возвращает указатель на первое вхождение младшего байта числа ch в строку str. Если вхождение не обнаружено, возвращается нулевой указатель.

Функция strrchr

```
char *strrchr(const char *str, int ch);
```

Возвращает указатель на последнее вхождение младшего байта числа ch в строку str. Если вхождение не обнаружено, возвращается нулевой указатель.

Функция memset

```
void *memset(void *buf, int ch, int count);
```

Функция копирует младший байт символа ch в первые count символов массива buf. Она возвращает указатель на массив buf.

62.2. Функции работы со строками

Функция memcpy

```
void *memcpy(void *to, const void *from, int count);
```

Функция копирует первые count элементов массива from в массив to. Если массивы перекрываются, поведение функции становится неопределенным. Она возвращает указатель на массив to.

Функция strcpy

```
char *strcpy(char *str1, const char *str2);
```

Функция копирует содержимое строки str2 в строку str1. Указатель str1 должен ссылаться на строку, завершаемую нулевым символом.

Функция strncpy

```
char * strncpy(char *str1, const char *str2,  
               int count);
```

Копирует первые count символов строки str2 в строку str1. Указатель str1 должен ссылаться на строку, завершающуюся нулевым символом. Если строки перекрываются, поведение функции становится неопределенным.

Если длина строки str2 меньше числа count, строка str1 дополняется нулями. И, наоборот, если длина строки str2 больше числа count, результирующая строка не содержит нулевого симво-

ла. Функция возвращает указатель на строку str1.

Функция strspn

```
int strspn(const char *str1, const char *str2);
```

Возвращает длину начальной подстроки строки str1, содержащей только символы из строки str2. Иначе говоря, функция возвращает индекс первого символа в строке str1, не совпадающего с каким-либо символом из строки str2.

Функция strcspn

```
int strcspn (const char *str1, const char *str2);
```

Функция возвращает длину начальной подстроки строки str1, не содержащей символов из строки str2. Иначе говоря, функция возвращает индекс первого символа в строке str1, совпадающего с каким-либо символом из строки str2.

Функция strpbrk

```
char *strpbrk(const char *str1, const char *str2);
```

Возвращает указатель на первое вхождение символа из строки str1, совпадающего с каким-либо символом из строки str2. Нулевой символ не учитывается. Если вхождение не обнаружено, возвращается нулевой указатель.

Функция strstr

```
char *strstr(const char *str1, const char *str2);
```

Возвращает указатель на первое вхождение символа из строки str1, совпадающего с каким-либо символом из строки str2. Если вхождение не обнаружено, возвращается нулевой указатель.

Функция strerror

```
char *strerror(int errnum);
```

Возвращает указатель на строку, связанную со значением параметра errnum. Эта строка определяется операционной системой и не может изменяться.

Функция strlen

```
int strlen(const char *str);
```

Возвращает длину строки str, завершающейся нулем. Нулевой символ, служащий признаком конца строки, не учитывается.

Функция strtok

```
char *strtok(char *str1, const char *str2);
```

Возвращает указатель на следующую лексему в строке str1. Символы, образующие строку str2, являются разделителями, определяющими лексему. Если лексемы не обнаружены, возвращается нулевой указатель.

Для того чтобы разбить строку на лексемы, сначала необходимо вызвать функцию strtok() и получить указатель на разбиваемую строку str1. В дальнейшем при вызове функции strtok() вместо строки str1 следует задавать нулевой указатель и строка последовательно разбивается на лексемы.

Занятие 63

63.1. Алгоритмы обработки строк

Для выделения слов в строке предварительно выполняют следующие действия: все знаки пунктуации заменяют пробелами; удаляют все лишние пробелы (вместо группы идущих подряд – оставляют по одному); если в конце строки нет пробела, то добавляют.

Затем в цикле ищут первое вхождение пробела; выделяют в слово часть строки от ее начала до пробела; удаляют слово из строки; выполняют необходимые манипуляции со словом. Приведенные действия выполняют до тех пор, пока в строке не закончатся слова (станет пустой).

63.2. Задание

Создать функцию обработки строки (табл. 63.1).

Таблица 63.1

№	Способ обработки
1	имеются ли все буквы входящие в заданное слово
2	имеется ли заданная пара букв, например: ОН и НО
3	имеется ли заданная пара одинаковых букв
4	удалить все заданные группы букв, например: ТИП
5	удалить каждый заданный символ, а остальные продублировать
6	посчитать наибольшее количество идущих подряд пробелов
7	удалить лишние пробелы
8	подсчитать количество слов
9	преобразовать в список, где каждый атом – слово
10	подсчитать количество слов начинающихся с заданной буквы
11	найти длину самого короткого слова
12	удалить все символы не являющиеся буквами или цифрами
13	удалить слово, предшествующее лексикографически остальным словам
14	указать слово в котором количество гласных букв минимально
15	указать слово в котором количество согласных букв максимально
16	сколько раз встречается заданная буква
17	процентное отношение согласных букв
18	найти самое длинное симметричное слово
19	выделить те символы которые встречаются один раз
20	изменить регистр букв на противоположный

Занятие 64

Массивы строк

Массивы строк используются довольно часто. Например, сервер базы данных может сравнивать команды, введенные пользователем, с массивом допустимых команд. Для того чтобы создать массив строк, завершающихся нулевым байтом, необходим двумерный массив символов. Максимальное значение левого индекса задает количество строк, а правого индекса – максимальную длину каждой строки. В приведенном ниже фрагменте объявлен массив, состоящий из 30 строк, каждая из которых может содержать до 79 символов и нулевой байт:

```
char str_array[30][80];
```

Доступ к отдельной строке весьма прост: нужно лишь указать левый индекс. Например, следующий оператор применяет функцию `gets()` к третьей строке массива `str_array`:

```
gets(str_array[2]);
```

С функциональной точки зрения этот оператор эквивалентен такому вызову:

```
gets(&str_array[2][0]);
```

И все же первая из этих форм записи предпочтительнее.

Чтобы лучше понять, как устроен массив строк, рассмотрим короткую программу, в которой массив строк используется как основа для текстового редактора:

```
#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];
void main(void)
{
    int t, i, j;
    printf("Для выхода введите пустую строку:\n");
    for (t = 0; t < MAX; t++) {
        printf("%d: ", t);
        gets(text[t]);
        if (!*text[t]) break;
    }
    for (i = 0; i < t; i++) {
        for (j = 0; text[i][j]; j++)
            putchar(text[i][j]);
        putchar('\n');
    }
}
```

Ввод текста в эту программу продолжается до тех пор, пока пользователь не введет пустую строку. Затем все строки выводятся на экран по очереди.

Занятие 65

65.1. Двухмерные массивы символов

Приведенный ниже пример иллюстрирует способы манипулирования массивами в языке C. В нем описана простая программа для игры в крестики-нолики. В качестве игровой доски используется двумерный массив символов.

Компьютер играет очень просто. Когда наступает его очередь ходить, он просматривает матрицу с помощью функции `get_computer_move()` в поисках свободной ячейки. Найдя такую ячейку, заполняет ее символом 0. Если все ячейки заняты, фиксируется ничья, и программа прекращает свою работу. Функция `get_player_move()` предлагает игроку сделать очередной ход и занести символ X в указанную им ячейку. Координаты левого верхнего угла равны (1, 1), а правого нижнего – (3, 3).

Массив `matrix` инициализируется пробелами. Каждый ход, сделанный игроком или компьютером, заменяет пробел в соответствующей ячейке символами 0 или X. Это позволяет легко вывести матрицу на экран.

После каждого хода вызывается функция `check()`. Если победитель еще не определен, она возвращает пробел, если выиграл игрок – символ X, а если компьютер – символ 0. Эта функция просматривает матрицу в поисках строк, столбцов и диагоналей, заполненных одинаковыми символами.

Текущее состояние игры отображается функцией `disp_matrix()`. Обратите внимание, насколько инициализация матрицы пробелами упрощает эту функцию.

Все функции получают доступ к массиву `matrix` по-разному. Изучите эти способы внимательно, чтобы лучше понять их механизм.

65.2. Игра в крестики-нолики

```
#include <stdio.h>
#include <stdlib.h>
char matrix[3][3];           // Игровая доска
char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);

void main(void)
{
    char done = ' ';
```

```

printf("Это - игра в крестики нолики\n");
printf("Вы будете играть с компьютером\n") ;
init_matrix();
do {
    disp_matrix();
    get_player_move();
    done = check();          // Есть ли победитель?
    if (done != ' ') break;
    get_computer_move();
    done = check();          // Есть ли победитель?
} while (done == ' ');
if (done == 'X') printf("Вы победили!\n");
else printf("Я выиграл!!!!\n");
disp_matrix();              // Финальное положение
}

/* Инициализация матрицы */
void init_matrix(void)
{
    int i, j;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            matrix[i][j] = ' ';
}

/* Ход игрока */
void get_player_move(void)
{
    int x, y;
    printf("Введите координаты X,Y: ");
    scanf("%d%c%d", &x, &y);
    x--; y--;
    if (matrix[x][y] != ' ') {
        printf("Неверный ход, попробуйте еще\n");
        get_player_move();
    } else matrix[x][y] = 'X';
}

/* Ход компьютера */
void get_computer_move(void)
{
    int i, j;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++)
            if (matrix[i][j] == ' ') break;
        if (matrix[i][j] == ' ') break;
    }
}

```

```

if (i*j == 9) {
    printf("Ничья\n");
    exit(0);
} else matrix[i][j] = 'O';
}

/* Вывести матрицу на экран */
void disp_matrix(void)
{
    int t;
    for (t = 0; t < 3; t++) {
        printf(" %c | %c | %c ", matrix[t][0], matrix[t][1], matrix[t][2]);
        if (t != 2) printf("\n-----\n");
    }
    printf("\n");
}

/* Проверить, есть ли победитель */
char check(void)
{
    int i;
    for (i = 0; i < 3; i++) /* Проверка строк */
        if (matrix[i][0] == matrix[i][1] && matrix[i][0] == matrix[i][2])
            return matrix[i][0];
    for (i = 0; i < 3; i++) /* Проверка столбцов */
        if (matrix[0][i] == matrix[1][i] && matrix[0][i] == matrix[2][i])
            return matrix[0][i];
    /* Проверка диагоналей */
    if (matrix[0][0] == matrix[1][1] && matrix[1][1] == matrix[2][2])
        return matrix[0][0];
    if (matrix[0][2] == matrix[1][1] && matrix[1][1] == matrix[2][0])
        return matrix[0][2];
    return ' ';
}

```

65.3. Задание

Переделайте программу приведенную выше таким образом, чтобы компьютер мог выиграть или хотя бы сыграть вничью⁵.

⁵ Для этого необходимо заменить функцию get_computer_move().

Занятие 66

66.1. Динамические массивы

Ранее были рассмотрены многомерные массивы и их индексация с помощью указателей. Рассмотрим процесс создания многомерных динамических массивов.

Динамические массивы – это массивы которые создаются не на этапе компиляции программы, а на этапе работы программы. Другими словами размер таких массивов может быть вычислен во время работы программы, а затем создан:

```
int *a, n;
...
// вычисленное значение n = 20
a = (int *) malloc(n*sizeof (int));
...
// вывод элемента с индексом 15
printf("%d\n",*(a + 15));
...
free(a);
```

После окончания работы с динамическими массивами их необходимо обязательно удалять из памяти, потому что рано или поздно произойдет ее переполнение.

66.2. Многомерные динамические массивы

Аналогично создаются и многомерные динамические массивы. Например нам необходимо создать трехмерный динамический массив действительных чисел размером $n \times m \times l$:

```
a = (float *) malloc(n*m*l*sizeof (float));
```

тогда доступ к его элементу с индексами i, j, k можно получить с помощью такой записи

```
*(a + (i*m + j)*l + k)
```

Можно также написать функцию, которая упростит доступ к элементам массива:

```
float *geta(float *a, int m, int l,
           int i, int j, int k)
{
    return a + (i*m + j)*l + k;
}
```

Тогда если нам необходимо, например, получить значение элемента массива с индексами 1, 2, 3 мы можем записать

```
x = *geta(a,m,l,1,2,3);
```

Поскольку, определенная выше, функция возвращает указатель на действительное число, то ее можно применять в левой части присвоения, то есть:

```
*geta(a,m,l,1,2,3) = 5.467;
```

Занятие 67

67.1. Оператор typedef

С помощью ключевого слова typedef можно определить новое имя типа данных. Новый тип при этом не создается, просто уже существующий тип получит новое имя. Это позволяет повысить машинезависимость программ. Если в программе используется машинозависимый тип, достаточно его переименовать, и на новом компьютере для модификации программы придется изменить лишь одну строку с оператором typedef. Кроме того, с помощью оператора typedef можно давать типам осмысленные имена, что повышает наглядность программы. Общий вид оператора typedef таков:

typedef тип новое_имя

Здесь элемент тип обозначает любой допустимый тип, а элемент новое_имя – псевдоним этого типа. Новое имя является дополнительным. Оно не заменяет существующее имя типа.

Например, тип float можно переименовать следующим образом:

```
typedef float real;
```

Этот оператор сообщает компилятору, что real – это новое имя типа float. Теперь в программе вместо переменных типа float можно использовать переменные типа real:

```
real over_due;
```

Здесь переменная over_due является числом с плавающей точкой, но имеет тип real, а не float. В свою очередь, используя оператор typedef, тип real можно переименовать еще раз. Например, оператор

```
typedef real length;
```

сообщает компилятору, что length – это синоним слова real, которое является другим именем типа float.

Используя оператор typedef, можно повысить наглядность программы и ее машинезависимость, но нельзя создать новый физический тип данных.

67.2. Декларация структур

Структура – это набор переменных, объединенных общим именем. Она обеспечивает удобный способ организации взаимосвязанных данных. Объявление структуры создает ее шаблон, который можно использовать при создании объектов структуры (т.е. ее экземпляров). Переменные, входящие в структуру, называются ее членами (или элементами, или полями.)

Как правило, все члены структуры логически связаны друг с другом. Например, имя и адрес в списке рассылки естественно

представлять в виде структуры. Следующий фрагмент программы демонстрирует, как объявляется структура, состоящая из полей, в которых хранятся имена и адреса. Ключевое слово `struct` сообщает компилятору, что объявляется именно структура:

```
struct addr
{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};
```

Обратите внимание на то, что объявление завершается точкой с запятой. Это необходимо, поскольку объявление структуры является оператором. Данная структура имеет тип `addr`. Таким образом, тип `addr` идентифицирует конкретную структуру данных и является ее спецификатором.

В приведенном выше фрагменте еще не создана ни одна переменная. В нем лишь определен составной тип данных, а не сама переменная. Для того чтобы возникла реальная переменная данного типа, ее нужно объявить отдельно. В языке C переменная типа `addr` (т.е. физический объект в памяти компьютера) создается оператором

```
struct addr addr_info;
```

Здесь объявляется переменная типа `addr` с именем `addr_info`.

После объявления переменной, представляющей собой структуру, компилятор автоматически выделяет память для ее членов.

Одновременно с определением структуры можно объявить несколько ее экземпляров. Например, в операторе

```
struct addr
{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_info, binfo, cinfo;
```

определяется структура `addr` и объявляются переменные `addr_info`, `binfo` и `cinfo`, являющиеся ее экземплярами. Важно понимать, что каждый экземпляр структуры содержит свои собственные копии членов структуры. Например, поле `zip` в объекте `binfo` отличается от поля `zip`, принадлежащего переменной `cinfo`. Таким образом, изменения поля `zip`, относящегося к объекту `binfo`, никак не повлияют на поле `zip` в переменной `cinfo`.

Если в программе нужен лишь один экземпляр структуры, ее имя указывать не обязательно. Иначе говоря, оператор

```
struct {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_info;
```

объявляет одну переменную с именем `addr_info`, представляющую собой экземпляр структуры, определенной выше.

Объявление структуры имеет следующий общий вид:

```
struct имя_типа_структуры
{
    тип имя_члена;
    тип имя_члена;
    тип имя_члена;
    ...
} имена_экземпляров;
```

67.3. Доступ к членам структуры

Доступ к отдельным членам структуры обеспечивается оператором `"."` (обычно его называют оператором "точка" или оператором доступа к члену структуры). Например, в следующем фрагменте программы полю `zip` структуры `addr_info`, объявленной ранее, присваивается почтовый индекс 12345:

```
addr_info.zip = 12345;
```

Имя экземпляра структуры указывается перед точкой, а имя члена структуры – после нее. Оператор, предоставляющий доступ к члену структуры, имеет следующий вид:

имя_экземпляра.имя_члена

Таким образом, чтобы вывести на экран индекс, нужно выполнить следующий оператор:

```
printf("%d", addr_info.zip);
```

В результате на экране появится почтовый индекс, содержащийся в поле `zip` переменной `addr_info`.

Аналогично символьный массив `addr_info.name` можно использовать при вызове функции `gets()`:

```
gets(addr_info.name);
```

В этом операторе функции `gets()` передается указатель типа `char *` на начало массива `name`.

Поскольку переменная `name` представляет собой символьный массив, доступ к отдельным символам строки `addr_info.name` можно получить с помощью оператора индексирования. Например, в

следующем фрагменте программы содержимое строки `addr_info.name` посимвольно выводится на экран.

```
int t;
for (t = 0; addr_info.name[t]; ++t)
    putchar(addr_info.name[t]);
```

67.4. Присваивание структур и их инициализация

Информацию, содержащуюся в одной структуре, можно присваивать другой структуре того же типа, используя обычный оператор присваивания. Иначе говоря, нет никакой необходимости присваивать каждый член отдельно. Рассмотрим программу, иллюстрирующую применение оператора присваивания к структурам:

```
#include <stdio.h>
void main(void)
{
    struct {
        int a;
        int b;
    } x, y;
    x.a = 10;
    y = x; // Присваивание одной структуры другой
    printf("%d\n", y.a);
}
```

После выполнения оператора присваивания член `y.a` будет содержать число 10.

Инициализировать структуру можно таким же способом, как и массив:

```
struct ab {
    int a;
    float b;
};
...
struct ab x = {4, 5.6};
```

67.5. Передача членов структур функциям

Если в функцию передается член структуры, на самом деле передается лишь копия его значения. Следовательно, в этом отношении член структуры ничем не отличается от обычной переменной (разумеется, если он сам не является составным элементом, например массивом). Рассмотрим следующую структуру:

```
struct fred
{
    char x;
    int y;
```

```
float z;
char s[10];
} mike;
```

Вот как ее члены передаются функциям:

```
func(mike.x);      // символ
func2(mike.y);     // целое число
func3(mike.z);     // число с плавающей точкой
func4(mike.s);     // адрес строки
func(mike.s[2]);   // символ s[2]
```

Если необходимо передать адрес отдельного члена структуры, следует указать оператор `&` перед именем структуры. Например, чтобы передать адреса членов структуры `mike`, нужно выполнить следующие операторы:

```
func(&mike.x);     // адрес символа
func2(&mike.y);    // адрес целого числа
func3(&mike.z);    // адрес числа с плавающей точкой
func4(mike.s);     // адрес строки s
func(&mike.s[2]);  // адрес символа s[2]
```

Обратите внимание на то, что оператор `&` стоит перед именем структуры, а не перед именами ее отдельных членов. Кроме того, имя строки `s` и так является адресом, поэтому указывать перед ней символ `&` не следует.

67.6. Передача целых структур функции

Если структура является аргументом функции, она передается по значению. Естественно, это значит, что все изменения структуры, происходящие внутри функции, никак не отразятся на структуре, являющейся ее фактическим аргументом.

При передаче структур тип аргументов должен совпадать с типом параметров. Они должны быть не просто похожи, а идентичны. Например, следующий фрагмент программы является неправильным и не будет скомпилирован, поскольку тип аргумента, указанный при вызове функции `f1()`, отличается от типа ее параметра:

```
struct struct_type { ... }
struct struct_type2 { ... }
...
void f1(struct struct_type2 prm) { ... };
void main(void)
{
    struct struct_type arg;
    ...
    f1(arg);
    ...
}
```

Занятие 68 Объединения

Объединение – это область памяти, которая используется для хранения переменных разных типов. Объединение позволяет интерпретировать один и тот же набор битов по-разному. Объявление объединения напоминает объявление структуры. Вот его общий вид:

```
union имя_типа_объединения
{
    тип имя_члена;
    тип имя_члена;
    тип имя_члена;
    ...
} имена_экземпляров_объединения;
Например:
union u_type
{
    int i;
    char ch;
}
```

Это объявление не создает никаких переменных. Чтобы объявить экземпляр объединения, нужно либо указать его имя в конце объявления, либо применить отдельный оператор объявления. Рассмотрим пример программы на языке C, в котором объявляется экземпляр `cnvt` объединения `u_type`:

```
union u_type cnvt;
```

В переменной `cnvt` целочисленная переменная `i` и символьная переменная `ch` хранятся в одной и той же области памяти. Разумеется, переменная `i` занимает 4 байта, а переменная `ch` – только 1. Обе переменные используют одну и ту же область памяти. В любом месте программы на переменную `cnvt` можно ссылаться, считая, что в ней хранится либо целое число, либо символ.

При объявлении экземпляра объединения компилятор автоматически выделяет память, достаточную для хранения наибольшего члена объединения. Например, если считать, что целые числа занимают 4 байта, размер переменной `cnvt` равен 4 байта, потому что в ней должна храниться целочисленная переменная `i`, хотя переменная `ch` занимает только один байт.

Для доступа к членам объединения используются те же синтаксические конструкции, что и для доступа к членам структуры. Например, чтобы присвоить элементу `i` объединения `cnvt` число 10, следует выполнить следующий оператор:

```
cnvt.i = 10;
```

Занятие 69 Битовые поля

В отличие от многих языков программирования, язык C обладает встроенной поддержкой битовых полей, предоставляющих доступ к отдельным битам. Битовые поля могут оказаться полезными во многих ситуациях:

- Если память ограничена, в одном байте можно хранить несколько логических переменных, принимающих значение `true` и `false`.
- Когда в одном байте нужно хранить информацию о состоянии некоторых устройств, закодированную несколькими битами.
- Если шифровальным процедурам требуется доступ к отдельным битам.

Хотя для решения этих задач можно использовать побитовые операторы, битовые поля позволяют создавать более простые и эффективные программы.

Битовое поле представляет собой особую разновидность члена структуры, размер которого можно указывать в битах. Определение битового поля имеет следующий вид:

```
struct имя_типа_структуры
{
    тип имя1: длина;
    тип имя2: длина;
    ...
    тип имя: длина;
} список_переменных;
```

Здесь спецификатор `тип` означает тип битового поля, а параметр `длина` – размер этого поля, выраженный в битах. Битовые поля, длина которых равна 1, должны быть объявлены с помощью спецификатора `unsigned`, поскольку отдельный бит не может иметь знака. Битовые поля часто используют для анализа информации, поступающей от какого-либо устройства. Например, байт состояния последовательного порта организован следующим образом:

- 0 Изменение в линии сигнала разрешения на передачу (change in clear-to-send line)
- 1 Изменение состояния готовности устройства сопряжения (change in data-set-ready)
- 2 Обнаружен конец записи (trailing edge detected)
- 3 Изменение в приемной линии (change in receive line)
- 4 Разрешение на передачу (clear-to-send)
- 5 Готовность к приему (data-set-ready)
- 6 Телефонный звонок (telephone ringing)

7 Сигнал принят (received signal)

Информацию, записанную в байте состояния, можно представить в виде следующего битового поля:

```
struct status_type {
    unsigned delta_cts: 1;
    unsigned delta_dsr: 1;
    unsigned tr_edge: 1;
    unsigned delta_rec: 1;
    unsigned cts: 1;
    unsigned dsr: 1;
    unsigned ring: 1;
    unsigned rec_line: 1;
} status;
```

Для того чтобы позволить программе определять, можно ли посылать или получать данные, следует использовать такие операторы:

```
status = get_port_status();
if (status.cts) printf("Можно посылать");
if (status.dsr) printf("Можно принимать");
```

Чтобы присвоить битовому полю некое значение, используется тот же оператор, что и для обычного члена структуры. Например, в следующем фрагменте программы с помощью операторов доступа к члену структуры и оператора присваивания обнуляется поле ring:

```
status.ring = 0;
```

Имена битовым полям давать не обязательно. Это позволит обращаться только к нужным полям, игнорируя ненужные. Например, если вас интересуют только биты cts и dsr, структуру status_type можно объявить так:

```
struct status_type {
    unsigned : 4;
    unsigned cts: 1;
    unsigned dsr: 1;
} status;
```

Запомните: если биты, расположенные после бита dsr, не используются, указывать их не обязательно.

В одной и той же структуре обычные члены структуры можно использовать наряду с битовыми полями.

Битовые поля накладывают некоторые ограничения. Например, нельзя получить адрес битового поля. Кроме того, битовые поля нельзя организовывать в массивы. Битовые поля нельзя объявлять статическими. Невозможно предугадать, будут ли биты считываться слева направо или справа налево при переносе на компьютер другого типа. Все это делает программы, использующие битовые поля, машинозависимыми.

Занятие 70

70.1. Массивы структур

Чаще всего структуры используются как элементы массивов. Для того чтобы объявить массив структур, необходимо сначала определить структуру и объявить массив переменных этого типа. Например, чтобы объявить массив структур, состоящий из 100 элементов типа addr, необходимо выполнить оператор

```
struct addr addr_info[100];
```

Он создает набор, состоящий из 100 переменных, представляющих собой структуры типа addr.

Для того чтобы получить доступ к конкретной структуре, необходимо указать ее индекс. Например, выведем на экран почтовый индекс, хранящийся в третьей структуре:

```
printf("%d", addr_info[2].zip);
```

Как во всех массивах, нумерация элементов массива структур начинается с нуля.

70.2. Массивы и структуры внутри структур

Членами структуры могут быть как простые, так и составные переменные. Простыми членами структур называются переменные, имеющие встроенный тип, например, целые числа или символы. С одной из разновидностей агрегированных элементов мы уже встречались, используя массив символов в структуре addr. Другим агрегированным типом данных могут быть одномерные и многомерные массивы, а также структуры.

Член структуры, представляющий собой массив, обрабатывается, как обычно. Рассмотрим следующую структуру.

```
struct x {
    int a[10][10];
    float b;
} y;
```

Для того чтобы обратиться к элементу с индексами 3 и 7 соответственно, следует выполнить оператор y.a[3][7].

Если членом структуры является другая структура, она называется вложенной. Например:

```
struct emp {
    struct addr address;
    float wage;
} worker;
```

Следующий фрагмент программы присваивает элементу zip структуры address лотовый индекс 93456.

```
worker.address.zip = 93456;
```

В языке C допускается до 15 уровней вложения.

Занятие 71

71.1. Объявление указателей на структуры

В языке С на структуры можно ссылаться точно так же, как и на любой другой тип данных. Однако указатели на структуры имеют несколько особенностей.

Указатели на структуры объявляются с помощью символа *, стоящего перед именем экземпляра структуры. Например, указатель `addr_pointer` на структуру `addr` объявляется так:

```
struct addr *addr_pointer;
```

71.2. Использование указателей на структуры

Указатели на структуры используются в двух ситуациях: для передачи структуры в функцию по ссылке и для создания структур данных, основанных на динамическом распределении памяти (например, связанных списков).

Передача структур в качестве аргументов функции имеет один существенный недостаток: в стек функции приходится копировать целую структуру. Если структура невелика, дополнительные затраты памяти будут относительно небольшими. Однако, если структура состоит из большого количества членов или ее членами являются большие массивы, затраты ресурсов могут оказаться чрезмерными. Этого можно избежать, если передавать функции не сами структуры, а лишь указатели на них.

Если функции передается указатель на структуру, в стек закладывается только ее адрес. В результате вызовы функции выполняются намного быстрее. Второе преимущество, которое проявляется в некоторых случаях, заключается в том, что, получив адрес, функция может модифицировать структуру, являющуюся ее фактическим параметром.

Для того чтобы определить адрес структуры, достаточно поместить перед ее именем оператор `&`. Рассмотрим следующий фрагмент программы:

```
struct bal {  
    float balance;  
    char name[80];  
} person;  
struct bal *p;
```

Теперь оператор `p = &person;` присваивает указателю `p` адрес структуры `person`.

Для того чтобы обратиться к элементу структуры через указатель на нее, нужно применить оператор `"->"`. Например, вот как выглядит ссылка на поле `balance`:

```
p->balance
```

Занятие 72

72.1. Связанные динамические данные

Линейные списки – это данные динамической структуры, которые представляют собой совокупность линейно связанных однородных элементов (рис. 72.1), для которых разрешается добавлять элементы между любыми двумя другими, и удалять любой элемент.

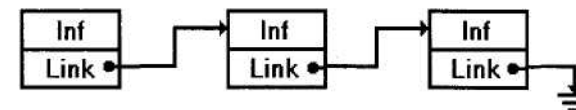


Рис. 72.1

Кольцевые списки – это такие же данные, как и линейные списки, но имеющие дополнительную связь между последним и первым элементами списка.

Очередь – частный случай линейного односвязного списка, для которого разрешены только два действия: добавление элемента в конец (хвост) очереди и удаление элемента из начала (головы) очереди.

Стек – частный случай линейного односвязного списка, для которого разрешено добавлять или удалять элементы только с одного конца списка, который называется вершиной (головой) стека.

Деревья – это динамические данные иерархической структуры произвольной конфигурации. Элементы дерева называются вершинами (узлами).

Пирамидой (упорядоченным деревом) называется дерево, в котором значения вершин (узлов) всегда возрастают или убывают при переходе на следующий уровень.

72.2. Организация взаимосвязей в связанных динамических данных

Связанные динамические данные характеризуются высокой гибкостью создания структур данных различной конфигурации. Это достигается благодаря возможности выделять и освобождать память под элементы в любой момент времени работы программы и возможности установить связь между любыми двумя элементами с помощью указателей.

Для организации связей между элементами динамической структуры данных требуется, чтобы каждый элемент содержал кроме информационных значений как минимум один указатель. Отсюда следует, что в качестве элементов таких структур необходимо использовать записи, которые могут объединять в единое целое разнородные элементы.

В простейшем случае элемент динамической структуры данных должен состоять из двух полей: информационного и указательного.

Схематично такую структуру данных можно представить как на рис. 72.1.

Соответствующие ей объявления будут иметь такой вид:

```
struct NODE {  
    INFO info;  
    struct NODE * next;  
};
```

где тип INFO может быть произвольного типа.

72.3. Работа со стеком

Для работы со стеком необходимо иметь один основной указатель на вершину стека top и один дополнительный временный указатель prom, который используется для выделения и освобождения памяти элементов стека.

Пустой стек характеризуется тем, что основной указатель указывает на пустой адрес, то есть NULL. Таким образом в начале программы работающей со стеком должна быть следующая строка:

```
struct NODE * top = NULL;
```

Для добавление элемента в стек, необходимо сначала выделить память под этот элемент

```
struct NODE * prom = (struct NODE *)  
    malloc(sizeof (struct NODE));
```

а затем внести значения в информационное поле нового элемента и установить связь между ним и "старой" вершиной стека Top:

```
INFO val;  
prom->info = val, prom->next = top;
```

И последнее что надо сделать при добавлении элемента в стек, переместить вершину стека top на новый элемент:

```
top = prom;
```

Удаление элемента стека происходит в таком порядке:

```
prom = top;  
val = prom->info, top = prom->next;  
free(prom);
```

72.4. Пример программы

```
#include <stdio.h>  
#include <stdlib.h>  
typedef double INFO;  
struct NODE {  
    INFO info;  
    struct NODE * next;
```

```
};  
struct NODE * top = NULL;  
void push(INFO val)  
{  
    struct NODE * prom = (struct NODE *)  
        malloc(sizeof (struct NODE));  
    prom->info = val;  
    prom->next = top;  
    top = prom;  
}  
INFO pop(void)  
{  
    struct NODE * prom = top;  
    INFO val = prom->info;  
    top = prom->next;  
    free(prom);  
    return val;  
}  
void show(void)  
{  
    struct NODE * prom = top;  
    printf("Содержимое стека\n-----\n");  
    if (!prom) {  
        printf("Стек пуст!\n");  
        return;  
    }  
    while (prom) {  
        printf("%g\n", prom->info);  
        prom = prom->next;  
    }  
}  
void main(void)  
{  
    INFO val;  
    show();  
    do {  
        printf("Введите элемент (0 - конец ввода): ");  
        scanf("%lg", &val);  
        if (!val) break;  
        printf("-> %g\n", val);  
        push(val);  
    } while (1);  
    show();  
    val = pop();  
    printf("<- %g\n", val);  
    show();  
}
```

Занятие 73

Работа с очередью

Для создания очереди и работы с ней необходимо иметь как минимум два указателя:

- на начало очереди (возьмем идентификатор head);
- на конец очереди (возьмем идентификатор tail).

Кроме того, также как и при работе со стеком, для добавления элементов и освобождения памяти удаляемых элементов требуется дополнительный временный указатель (возьмем идентификатор prom). Дополнительный указатель также часто используется в других ситуациях для удобства работы с очередью.

Пустая очередь характеризуется тем, что указатели на начало и конец очереди равны NULL. Таким образом, в начале программы работающей с очередью должна быть такая строка:

```
struct NODE *head = NULL, *tail = NULL;
```

Добавление элемента в очередь осуществляется по разному, в зависимости от того пуста очередь или нет. Но в любом случае необходимо сначала выделить память под этот элемент

```
struct NODE * prom = (struct NODE *)
    malloc(sizeof (struct NODE));
```

а затем внести значения в информационное поле нового элемента и установить указатель на следующий элемент в NULL:

```
INFO val;
prom->info = val, prom->next = NULL;
```

А теперь в зависимости от того пустая ли очередь, необходимо выполнить следующие действия:

- если очередь пуста, то необходимо установить указатели на начало и конец очереди на созданный элемент
- если очередь содержит элементы, то указатель на следующий элемент конца очереди и конец очереди установить на созданный элемент

```
tail->next = prom, tail = prom;
```

Удаление элемента очереди происходит в таком порядке:

```
val = head->info, prom = head, head = prom->next;
free(prom);
```

В качестве примера приведем программу создания и удаления очереди:

```
#include <stdio.h>
#include <stdlib.h>
typedef double INFO;
struct NODE {
    INFO info;
    struct NODE * next;
```

```
};
struct NODE * head = NULL, * tail = NULL;
void add(INFO val)
{
    struct NODE * prom = (struct NODE *)
        malloc(sizeof (struct NODE));
    prom->info = val, prom->next = NULL;
    if (!tail) head = prom;
    else tail->next = prom;
    tail = prom;
}
INFO del(void)
{
    INFO val = head->info;
    struct NODE * prom = head;
    head = prom->next, free(prom);
    return val;
}
void show(void)
{
    struct NODE * prom = head;
    printf("Содержимое очереди\n-----\n");
    if (!prom) {
        printf("Очередь пуста!\n");
        return;
    }
    while (prom) {
        printf("%g ", prom->info);
        prom = prom->next;
    }
    printf("\n");
}
void main(void)
{
    INFO val;
    show();
    do {
        printf("Введите элемент (0 - конец ввода): ");
        scanf("%lg", &val);
        if (!val) break;
        printf("-> %g\n", val);
        add(val);
    } while (1);
    show();
    printf("<- %g\n", del());
    show();
}
```

Занятие 74

74.1. Линейные списки

Линейные списки отличаются от рассмотренных ранее стеков и очередей тем, что элементы в список можно добавлять и удалять в любое место списка: начало, конец, после заданного элемента.

Мы будем рассматривать только линейные односвязные списки, поэтому для работы с ними необходим указатель на начало списка `first`, который в начале работы будет пустым:

```
struct NODE *first = NULL;
```

Также как и при работе со стеком и очередью, для добавления элементов и освобождения памяти удаляемых элементов требуется дополнительный указатель (`prom`).

Механизм добавления и удаления элемента в любую часть списка реализован в приведенной ниже программе. Следует отметить, что добавление и удаление последнего элемента реализовано с помощью добавления и удаления после заданного элемента.

74.2. Пример программы

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef double INFO;
struct NODE {
    INFO info;
    struct NODE * next;
};
typedef struct NODE sNODE;
sNODE * first = NULL;
int length = 0;
sNODE * getNode(int num)
{
    sNODE * prom = first;
    int i;
    for (i = 0; i < num; i++)
        prom = prom->next;
    return prom;
}
void addhead(INFO val)
{
    sNODE * prom = (sNODE *) malloc(sizeof (sNODE));
    prom->info = val, prom->next = first;
    first=prom, length++;
}
void addnum(INFO val, int num)
```

```
{
    sNODE * last = getNode(num);
    sNODE * prom = (sNODE *) malloc(sizeof (sNODE));
    prom->info = val, prom->next = last->next;
    last->next = prom, length++;
}
INFO delhead(void)
{
    INFO val = first->info;
    sNODE * prom = first;
    first = prom->next, length--, free(prom);
    return val;
}
INFO delnum(int num)
{
    sNODE * predlast = getNode(num - 1),
        * prom = predlast->next;
    INFO val = prom->info;
    predlast->next = prom->next, length--, free(prom);
    return val;
}
void show(void)
{
    sNODE * prom = first;
    printf("Содержимое списка: ");
    if (!prom) {
        printf("Список пуст!\n");
        return;
    }
    while (prom) {
        printf("%g ",prom->info);
        prom = prom->next;
    }
    printf("\n");
}
void main(void)
{
    char ch;
    do {
        printf("Длина списка %d\n",length);
        show();
        puts("\n1. Добавить элемент в начало списка");
        puts("2. Добавить элемент в конец списка");
        puts("3. Добавить после заданного элемента");
        puts("4. Удалить элемент из начала списка");
        puts("5. Удалить элемент из конца списка");
        puts("6. Удалить после заданного элемента");
```

```

puts("0. Выйти");
do {
    int num;
    ch = getch();
    if (!length && (ch == '2' || ch == '3'))
        ch = '1';
    if (!length && (ch >= '4' && ch <= '6'))
        ch = 'E';
    switch (ch) {
        case '0': break;
        case '1': addhead(rand() % 256);
                    break;
        case '2': addnum(rand()%256,length-1);
                    break;
        case '3':
            if (length == 1) {
                addnum(rand()%256,0);
                break;
            }
            printf("Введите номер элемента:");
            do { scanf("%d",&num);
                } while (num < 1 || num > length);
            addnum(rand()%256,num - 1);
            break;
        case '4': delhead();
                    break;
        case '5':
            if (length == 1) delhead();
            else delnum(length - 1);
            break;
        case '6':
            if (length <= 2) {
                if (length == 2) delnum(1);
                else delhead();
                break;
            }
            printf("Введите номер элемента:");
            do { scanf("%d",&num);
                } while (num<1 || num>length - 1);
            delnum(num);
            break;
        default: putchar('\a');
    }
} while (ch < '0' || ch > '6');
printf("-----\n");
} while (ch != '0');
}

```

Занятие 75

75.1. Препроцессорные директивы

В программы на языке C можно включать различные инструкции, регламентирующие работу компилятора. Эти инструкции называются директивами препроцессора. Хотя они не являются частью языка C, их применение позволяет расширить возможности программ.

Препроцессор содержит следующие директивы:

```
#define  #elif   #else   #endif  #error  #if
#ifdef   #ifndef  #include #line   #pragma #undef
```

Все директивы препроцессора начинаются со знака #. Кроме того, каждая директива должна располагаться в отдельной строке программы.

75.2. Директива #include

Директива #include вынуждает компилятор считать и подставить в исходный текст программы файл с заданным именем. Это имя заключается в двойные кавычки или угловые скобки.

Включаемые файлы сами могут содержать директивы #include. Такие директивы называются вложенными. Глубина вложения директив зависит от компилятора. Стандарт языка C допускает восемь уровней вложения.

Кавычки и угловые скобки, в которых указываются имена включаемых файлов, определяют способ их поиска на жестком диске. Если имя файла содержится в угловых скобках, он должен находиться в каталоге, указанном компилятором. Обычно это каталог include, предназначенный для хранения всех заголовочных файлов. Если имя файла заключено в кавычки, как правило, его поиск выполняется в рабочем каталоге. Если файл не найден, поиск повторяется так, будто имя файла содержалось в угловых скобках.

75.3. Другие директивы

Директива #error вынуждает компилятор прекратить компиляцию. Она используется в основном при отладке программ.

Препроцессор содержит несколько директив, позволяющих выборочно компилировать отдельные части программы (#elif, #else, #endif, #if, #ifdef, #ifndef). Этот процесс называется условной компиляцией, позволяющей настраивать программы.

Директива #pragma зависит от реализации. Она позволяет передавать компилятору различные команды. Детали и допустимые опции перечисляются в документации компилятора.

Занятие 76

76.1. Макроопределения

Директива `#define` определяет идентификатор и последовательность символов, которая заменяет его в тексте программы. Этот идентификатор называется именем макроса, а процесс замены – макроподстановкой. Общий вид этой директивы таков:

```
#define имя_макроса последовательность символов
```

Обратите внимание на то, что эта директива не содержит точки с запятой. Между идентификатором и последовательностью символов может располагаться сколько угодно пробелов, но сама последовательность должна заканчиваться символом перехода на новую строку.

Например, если вы хотите использовать вместо единицы слово `left`, а вместо нуля – слово `right`, необходимо объявить две директивы `#define`:

```
#define LEFT 1
#define RIGHT 0
```

Эти директивы вынудят компилятор заменять слова `left` и `right` в исходном файле значениями 1 и 0 соответственно. Например, следующий оператор выводит на экран числа 0 1 2:

```
printf("%d %d %d", RIGHT, LEFT, LEFT+1);
```

Если в программе определено имя макроса, его можно использовать для определения другого макроса. Например, в приведенном ниже фрагменте программы определяются значения идентификаторов `one`, `two` и `three`:

```
#define ONE 1
#define TWO ONE+ONE
#define THREE ONE+TWO
```

Макроподстановка означает простую замену идентификатора последовательностью символов, связанной с ним. Следовательно, при необходимости с помощью директивы `#define` можно определить стандартное сообщение об ошибке:

```
#define E_MS "стандартная ошибка при вводе\n"
...
printf(E_MS);
```

Если идентификатор является частью строки, заключенной в кавычки, макроподстановка не производится. Например, фрагмент программы

```
#define XYZ "это проверка"
...
printf("XYZ");
```

выведет на экран не строку "это проверка", а символы `хуз`.

Если последовательность символов занимает несколько строк, в конце каждой из них следует поставить обратную косую

черту:

```
#define LONG_STRING "в этом примере \
используется очень длинная строка"
```

Для названия идентификаторов программисты обычно используют прописные буквы. Это позволяет легко обнаруживать макроподстановки в тексте программ. Кроме того, лучше помещать все директивы `#define` в самом начале исходного файла или в отдельном заголовочном файле, а не разбрасывать их по всей программе.

Макросы часто применяются для определения имен констант, встречающихся в программе. Допустим, в программе определен массив, который используется в нескольких модулях. Крайне нежелательно "зашивать" его размер в текст программы. Вместо этого следует применить макроподстановку, используя директиву `#define`. Тогда для того, чтобы изменить размер массива, достаточно будет модифицировать одну строку программы, содержащую директиву `#define`, а не выискивать все ссылки на размер массива в тексте. После этого программу необходимо перекомпилировать.

76.2. Определение функций в виде макросов

Директива `#define` обладает еще одним мощным свойством: макрос может иметь формальные аргументы. Каждый раз, когда в тексте программы встречается имя макроса, его формальные аргументы заменяются фактическими. Этот вид макроса называется функциональным. Рассмотрим пример:

```
#include <stdio.h>
#define ABS(a) (a)<0 ? -(a) : (a)
void main(void)
{
    printf("abs of -1 and 1: %d %d", ABS(-1), ABS(1));
}
```

Скобки, в которые заключен макрос `a`, гарантируют правильную подстановку.

Использование функциональных макросов вместо настоящих функций увеличивает скорость выполнения программы, поскольку в ней отсутствуют вызовы функций. Однако, если размер функционального макроса достаточно велик, быстрое действие программы достигается в ущерб ее размеру, поскольку многие фрагменты программы просто дублируются.

Препроцессор имеет два оператора: `#` и `##`. Оператор `#`, который называется оператором превращения в строку, преобразует свой аргумент в строку, заключенную в кавычки. Оператор `##`, называемый оператором конкатенации, "склеивает" две лексемы.

Занятие 77

77.1. Стандартная библиотека ввода-вывода

Заголовочный файл, связанный с системой ввода-вывода ANSI C, называется `stdio.h`. В этом заголовочном файле определены несколько макросов и типов, используемых файловой системой. Наиболее важным является тип `FILE`, использующийся при объявлении указателя на файл. Двумя другими типами являются типы `size_t` и `fpos_t`. Тип `size_t` определяет объект, который может содержать наибольший файл, допускаемый операционной системой. Тип `fpos_t` определяет объект, предназначенный для хранения всей информации, необходимой для однозначной идентификации любой позиции внутри файла.

Многие функции ввода-вывода задают значение встроенной глобальной целочисленной переменной `errno`. При возникновении ошибки эта переменная позволяет получить доступ к более подробной информации о возникшей ситуации. Значения переменной `errno` являются машинозависимыми.

Все функции стандартной библиотеки ввода-вывода делятся на такие группы: операции над файлами; форматный вывод; форматный ввод; функции ввода-вывода литер; функции прямого ввода-вывода; функции позиционирования файла; функции обработки ошибок.

77.2. Пример

Для примера приведем программу, считывающую текстовый файл и выводящую его содержимое на консоль:

```
#include <stdio.h>
void main(void)
{
    FILE *f;
    char name[15], fs[80];
    puts("Введите имя файла");
    gets(name);
    if (!(f = fopen(name, "r"))) {
        fprintf(stderr,
            "Невозможно открыть файл %s...\n", name);
        return;
    }
    while (!feof(f)) {
        fgets(fs, 80, f);
        puts(fs);
    }
    fclose(f);
}
```

Занятие 78

78.1. Потоки и файлы

Прежде чем перейти к обсуждению файловой системы языка C, следует понять разницу между потоками и файлами. Система ввода-вывода языка C обеспечивает единообразный интерфейс, не зависящий от физических устройств. Иначе говоря, система ввода-вывода создает между программистом и устройством абстрактное средство связи. Эта абстракция называется потоком, а физическое устройство – файлом.

78.2. Потоки

Файловая система языка C предназначена для широкого спектра средств ввода-вывода, включая терминалы, дисководы и принтеры. Несмотря на разнообразие физических устройств, файловая система преобразовывает каждое из них в некое логическое устройство, называемое потоком. Все потоки функционируют одинаково. Поскольку они практически не зависят от конкретного устройства, одна и та же функция может выводить данные как на жесткий диск, так и на консоль. Существуют два вида потоков: текстовый и бинарный.

78.3. Текстовые потоки

Текстовый поток представляет собой последовательность символов. Стандарт языка C позволяет (но не требует) организовывать потоки в виде строк, заканчивающихся символом перехода. В последней строке символ перехода указывать не обязательно (на самом деле большинство компиляторов языка C не завершают текстовые потоки символом перехода на новую строку). В зависимости от окружения некоторые символы в текстовых потоках могут подвергаться преобразованиям. Например, символ перехода на новую строку может быть заменен парой символов, состоящей из символа возврата каретки и прогона бумаги. Следовательно, между символами, записанными в текстовом потоке, и символами, выведенными на внешние устройства, нет взаимно однозначного соответствия. По этой же причине количество символов в текстовом потоке и на внешнем устройстве может быть разным.

78.4. Бинарные потоки

Бинарный поток – это последовательность байтов, однозначно соответствующая последовательности байтов, записанной на внешнем устройстве. Кроме того, количество записанных (или счи-

танных) байтов совпадает с количеством байтов на внешнем устройстве. Однако бинарный поток может содержать дополнительные нулевые байты, количество которых зависит от конкретной реализации. Эти байты применяются для выравнивания записей, например для того, чтобы данные заполняли весь сектор на диске.

78.5. Файлы

В языке С файлом считается все – от файла на диске до дисплея или принтера. Выполнив операцию открытия, поток можно связать с конкретным файлом, который можно использовать для обмена данными с программой.

Не все файлы обладают одинаковыми возможностями. Например, файл на жестком диске предоставляет прямой доступ к своим записям, а некоторые принтеры – нет. Это приводит нас к следующему выводу: все потоки в файловой системе языка С одинаковы, а файлы могут различаться.

Если файл может поддерживать запрос позиции, при его открытии курсор файла устанавливается в начало. При чтении или записи очередного символа курсор перемещается на одну позицию вперед.

При закрытии файла его связь с потоком разрывается. Если файл был открыт для записи, его содержимое записывается на внешнее устройство. Этот процесс обычно называют очисткой потока. Он гарантирует, что после закрытия файла в потоке не останется никакой случайно забытой информации. При нормальном завершении программы все файлы закрываются автоматически. Если работа программы была завершена аварийно, например вследствие ошибки или выполнения функции `abort()`, файлы не закрываются.

Каждый поток, связанный с файлом, имеет управляющую структуру типа `FILE`, которую нельзя модифицировать. При работе с файлами следует объявить указатель на тип `FILE`, а не структуру типа `FILE`:

```
FILE *f;
```

Если вы новичок в программировании, различия между файлами и потоками могут показаться вам надуманными. Просто помните, что их единственное предназначение – обеспечить унифицированный интерфейс.

При выполнении операций ввода-вывода следует мыслить терминами потоков, используя при этом единственную файловую систему. Она автоматически преобразует исходные операции ввода или вывода, связанные с конкретным физическим устройством, в легко управляемый поток.

Занятие 79

79.1. Основы файловой системы

Файловая система языка С состоит из нескольких взаимосвязанных функций. В табл. 79.1 приведены наиболее распространенные из них. Для их использования необходим заголовочный файл `stdio.h`.

Таблица 79.1

Функция	Операция
<code>fopen()</code>	Открывает файл
<code>fclose()</code>	Закрывает файл
<code>putc()</code>	Записывает символ в файл
<code>fputc()</code>	То же, что и <code>putc()</code>
<code>getc()</code>	Считывает символ из файла
<code>fgetc()</code>	То же, что и <code>getc()</code>
<code>fgets()</code>	Считывает строку из файла
<code>fputs()</code>	Записывает строку в файл
<code>fseek()</code>	Устанавливает курсор на заданный байт файла
<code>ftell()</code>	Возвращает текущую позицию курсора
<code>fprintf()</code>	Файловый аналог функции <code>printf()</code>
<code>fscanf()</code>	Файловый аналог функции <code>scanf()</code>
<code>feof()</code>	Возвращает истинное значение, если достигнут конец файла
<code>ferror()</code>	Возвращает истинное значение, если произошла ошибка
<code>rewind()</code>	Устанавливает курсор в начало файла
<code>remove()</code>	Стирает файл
<code>fflush()</code>	Очищает поток

В файле `stdio.h` определены макросы `NULL`, `EOF`, `FOPEN_MAX`, `SEEK_SET`, `SEEK_CUR` и `SEEK_END`. Макрос `NULL` определяет нулевой указатель. Макрос `EOF` обычно определяет константу, равную `-1`. Он задает значение, возвращаемое функцией ввода при попытке прочесть несуществующую запись после конца файла. Макрос `FOPEN_MAX` определяет количество файлов, которые можно открыть одновременно. Остальные макросы используются функцией `fseek()`, осуществляющей прямой доступ к записям файла.

79.2. Указатель файла

Указатель файла – это звено, связывающее между собой все компоненты системы ввода-вывода. Он представляет собой указатель на структуру, имеющую тип `FILE`. В этой структуре хранится информация о файле, в частности, его имя, статус и текущее по-

ложение курсора. По существу, указатель файла описывает конкретный файл и используется соответствующим потоком при выполнении операций ввода-вывода. Выполнить эти операции без указателя файла невозможно.

79.3. Открытие файла

Функция `fopen()` открывает поток и связывает его с файлом. Затем она возвращает указатель на этот файл. Наиболее часто файлом считается физический файл, расположенный на диске. Прототип функции `fopen()` имеет следующий вид:

```
FILE *fopen(const char *имя_файла, const char *режим)
```

Здесь параметр `имя_файла` представляет собой указатель на строку символов, которая задает допустимое имя файла и может включать в себя описание пути к нему. Строка, на которую ссылается указатель `режим`, определяет предназначение файла. Допустимые значения параметра `режим` представлены в табл. 79.2.

Таблица 79.2

Значение	Смысл
<code>r</code>	Открыть текстовый файл для чтения
<code>w</code>	Создать текстовый файл для записи
<code>a</code>	Добавить записи в конец текстового файла
<code>rb</code>	Открыть бинарный файл для чтения
<code>wb</code>	Создать бинарный файл для записи
<code>ab</code>	Добавить записи в конец бинарного файла
<code>r+</code>	Открыть текстовый файл для чтения и записи
<code>w+</code>	Создать текстовый файл для чтения и записи
<code>a+</code>	Добавить записи в конец текстового файла или создать текстовый файл для чтения и записи
<code>r+b</code>	Открыть бинарный файл для чтения и записи
<code>w+b</code>	Создать бинарный файл для чтения и записи
<code>a+b</code>	Добавить записи в конец бинарного файла или создать бинарный файл для чтения и записи

Если во время открытия файла произойдет ошибка, функция `fopen()` вернет нулевой указатель:

```
FILE *f;
if (!(f = fopen(name, "r"))) ... // Ошибка
```

Хотя большинство значений параметра `режим` имеют вполне очевидный смысл, некоторые из них требуют дополнительных разъяснений. Если при попытке открыть файл только для чтения выяснится, что такого файла на диске нет, функция `fopen()` вернет признак ошибки. Если попытаться открыть несуществующий файл для добавления записей, он будет создан. Кроме того, если для

добавления открыть существующий файл, все новые записи будут записаны в его конец. Исходное содержание файла останется неизменным. Если несуществующий файл открывается для записи, он также создается. Если для записи открывается существующий файл, его содержимое уничтожается и заменяется новыми данными. Разница между режимами `r+` и `w+` заключается в том, что при попытке открыть несуществующий файл в режиме `r+` новый файл не создается, в отличие от режима `w+`. К тому же, если файл ранее существовал, то в режиме `w+` его содержимое будет уничтожено, а в режиме `r+` – нет.

Как следует из табл. 79.2, файл можно открыть как в текстовом, так и в бинарном режиме. Как правило, команды возврата каретки и прогона бумаги в текстовом режиме переводятся в символы перехода на новую строку. При выводе все происходит наоборот: символы перехода на новую строку интерпретируются как команды перевода каретки и прогона бумаги. При работе с бинарными файлами такие преобразования не выполняются.

Количество одновременно открытых файлов определяется константой `FOPEN_MAX`. Как правило, она равна 8, но ее точное значение следует искать в документации, сопровождающей компилятор.

79.4. Закрытие файла

Функция `fclose()` закрывает поток, открытый ранее функцией `fopen()`. Она записывает все оставшиеся в буфере данные в файл и закрывает его, используя команды операционной системы. Ошибка, возникшая при закрытии файла, может породить множество проблем, начиная с потери данных и разрушения файлов и заканчивая непредсказуемыми последствиями для программы. Кроме того, функция `fclose()` освобождает управляющий блок файла, связанного с потоком, позволяя использовать этот блок повторно. Операционная система ограничивает количество одновременно открытых файлов, поэтому, прежде чем открыть один файл, следует закрыть другой.

Прототип функции `fclose()` выглядит следующим образом:

```
int fclose(FILE *fp)
```

Здесь параметр `fp` представляет собой указатель файла, возвращенный функцией `fopen()`. Если функция вернула нулевой указатель, значит, файл закрыт успешно. Значение EOF является признаком ошибки. Для распознавания и обработки возникших проблем можно использовать стандартную функцию `ferror()`. Как правило, функция `fclose()` выполняется неверно, если диск был преждевременно вынут из дисковода или переполнен.

Занятие 80

80.1. Запись символа

Для вывода символов предназначены две функции: `putc()` и `fputc()`. Эти эквивалентные функции просто сохраняют совместимость со старыми версиями языка C.

Функция `putc()` записывает символ в файл, открытый с помощью функции `fopen()`. Прототип этой функции выглядит так:

```
int putc(int символ, FILE *fp)
```

Здесь параметр `fp` представляет собой указатель файла, возвращенный функцией `fopen()`, а аргумент является символом, подлежащим выводу. Указатель файла сообщает функции `putc()`, в какой именно файл следует произвести запись. Несмотря на то, что параметр имеет тип `int`, в файл записывается лишь младший байт.

Если функция `putc()` выполнена успешно, она возвращает символ, записанный ею в файл. В противном случае она возвращает константу `EOF`.

80.2. Чтение символа

Ввод символа осуществляется двумя эквивалентными функциями: `getc()` и `fgetc()`. Наличие одинаковых функций позволяет сохранить совместимость со старыми версиями языка C.

Функция `getc()` считывает символ из файла, открытого функцией `fopen()` в режиме чтения. Прототип функции `getc()` имеет следующий вид:

```
int getc(FILE *fp)
```

Здесь параметр `fp` является указателем файла, возвращенным функцией `fopen()`. Функция `getc()` возвращает целочисленную переменную, младший байт которой содержит введенный символ. Если возникла ошибка, старший байт этой переменной равен нулю. Если при чтении обнаруживается конец файла, функция `getc()` возвращает константу `EOF`. Следовательно, для считывания данных до конца файла можно применять следующий фрагмент:

```
do {
    ch = getc(fp);
} while (ch != EOF);
```

Однако функция `getc()` возвращает константу `EOF` и при возникновении ошибок.

80.3. Минимальный набор функций

Функции `fopen()`, `getc()`, `putc()` и `fclose()` образуют минимальный набор процедур для работы с файлами. Приведенная ниже программа иллюстрирует применение функций `putc()`, `fopen()` и

`fclose()`. Она считывает символы с клавиатуры и записывает их в файл, пока пользователь не введет символ `$`:

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    if (argc != 2) {
        printf("Вы забыли указать имя файла.\n") ;
        exit(1);
    }
    if ((fp = fopen(argv[1], "w")) == NULL) {
        printf ("Невозможно открыть файл!\n");
        exit(1);
    }
    do {
        ch = getchar();
        putc(ch, fp);
    } while (ch != '$');
    fclose(fp);
}
```

Следующая программа выполняет противоположные операции: она считывает произвольный текстовый файл и выводит его содержимое на экран:

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    if (argc != 2) {
        printf("Вы забыли указать имя файла.\n") ;
        exit(1);
    }
    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf ("Невозможно открыть файл!\n");
        exit(1);
    }
    ch = getc(fp);
    while (ch != EOF) {
        putchar(ch);
        ch = getc(fp);
    }
    fclose(fp);
}
```

Занятие 81

81.1. Конец файла

Как указано выше, обнаружив конец файла, функция `getc()` возвращает константу `EOF`. Однако это не самый лучший способ для распознавания конца файла. Во-первых, операционная система может работать как с текстовыми, так и с бинарными файлами. Если файл открыт для бинарного ввода, из него может быть считано целое число, равное константе `EOF`. Следовательно, конец файла, распознаваемый функцией `getc()`, может не совпадать с реальным концом файла. Во-вторых, функция `getc()` возвращает константу `EOF` не только по достижении конца файла, но и при возникновении любой другой ошибки. По этой причине константу `EOF` невозможно интерпретировать однозначно. Для решения этой проблемы в языке C предусмотрена функция `feof()`, распознающая конец файла. Ее прототип имеет следующий вид:

```
int feof (FILE *fp)
```

Обнаружив конец файла, функция `feof()` возвращает истинное значение, в противном случае она возвращает число 0. Таким образом, приведенная ниже процедура считывает бинарный файл, пока не обнаружит его конец:

```
while (!feof(fp)) ch = getc(fp);
```

Разумеется, этот способ можно применять как для бинарных, так и для текстовых файлов.

Следующая программа, копирующая текстовые или бинарные файлы, иллюстрирует применение функции `feof()`:

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
    FILE *in, *out;
    char ch;
    if (argc != 3) {
        printf("Вы забыли ввести имя файла.\n");
        exit(1);
    }
    if ((in = fopen(argv[1], "rb")) == NULL) {
        printf("Невозможно открыть исходный файл.\n");
        exit(1);
    }
    if ((out = fopen(argv[2], "wb")) == NULL) {
        printf("Невозможно открыть файл-результат\n");
        exit(1);
    }
    while (!feof(in)) {
```

```
        ch = getc(in);
        if (!feof(in)) putc(ch, out);
    }
    fclose(in);
    fclose(out);
}
```

81.2. Функция `rewind()`

Функция `rewind()` устанавливает курсор в начало файла, заданного в качестве ее аргумента. Иными словами, она "перематывает" файл в начало. Ее прототип имеет следующий вид:

```
void rewind(FILE *fp);
```

Здесь параметр `fp` является допустимым указателем файла.

81.3. Функция `ferror()`

Функция `ferror()` определяет, возникла ли ошибка при работе с файлом. Ее прототип имеет следующий вид:

```
int ferror(FILE *fp);
```

Здесь параметр `fp` является допустимым указателем файла. Функция возвращает истинное значение, если при выполнении операции возникла ошибка, в противном случае она возвращает ложное значение. Поскольку с каждой операцией над файлом связан определенный набор ошибок, функцию `ferror()` следует вызывать сразу же после выполнения операции, в противном случае ошибка может быть упущена.

Следующая программа иллюстрирует работу функции `ferror()`. Она удаляет из файла символы табуляции, заменяя их соответствующим количеством пробелов, которое задается константой `tab_size`. Обратите внимание на то, что функция `ferror()` вызывается после выполнения каждой файловой операции. При запуске программы укажите в командной строке имена входного и выходного файлов:

```
#include <stdio.h>
#include <stdlib.h>
#define TAB_SIZE 8
#define IN 0
#define OUT 1
void err(int e);
void main(int argc, char *argv[])
{
    FILE *in, *out;
    int tab, i;
    char ch;
```

```

if (argc != 3) {
    printf("необходимы два параметра\n");
    exit(1);
}
if ((in = fopen(argv[1], "rb")) == NULL) {
    printf("Невозможно открыть %s.\n", argv[1]);
    exit(1);
}
if ((out = fopen(argv[2], "wb")) == NULL) {
    printf("Невозможно открыть %s.\n", argv[2]);
    exit(1);
}
tab = 0;
do {
    ch = getc(in);
    if (ferror(in)) err(IN);
    if (ch == '\t') {
        for (i = tab; i < 8; i++) {
            putc(' ', out);
            if (ferror(out)) err(OUT);
        }
        tab = 0;
    } else {
        putc(ch, out);
        if (ferror(out)) err(OUT);
        tab++;
        if (tab == TAB_SIZE) tab = 0;
        if (ch == '\n' || ch == '\r') tab = 0;
    }
} while (!feof(in));
fclose(in);
fclose(out);
}
void err(int e)
{
    if (e == IN) printf("Ошибка при вводе.\n");
    else printf("Ошибка при выводе.\n");
    exit(1);
}

```

81.4. Очистка потока

Для того чтобы очистить поток вывода, следует применять функцию `fflush()`, имеющую следующий прототип:

```
int fflush(FILE *fp);
```

Эта функция записывает содержимое буфера в файл, связанный с указателем `fp`.

Занятие 82

82.1. Работа со строками

Кроме функций `getc()` и `putc()` файловая система языка C содержит функции `fgets()` и `fputs()`, выполняющие чтение символьных строк из файла и запись их в файл соответственно. Эти функции аналогичны функциям `getc()` и `putc()`, однако они считывают и записывают строки, а не отдельные символы. Вот как выглядят их прототипы:

```
int fputs(const char *строка, FILE *fp);
```

```
char *fgets(char *строка, int длина, FILE *fp);
```

Функция `fputs()` записывает в заданный поток строку, на которую ссылается указатель строка. При возникновении ошибки она возвращает константу `EOF`.

Функция `fgets()` считывает строку из указанного потока, пока не обнаружит символ перехода или не прочтёт длина – 1 символ. В отличие от функции `getc()` символ перехода считается составной частью строки. Результирующая строка должна завершаться нулём. В случае успеха функция возвращает указатель на введенную строку, в противном случае она возвращает нулевой указатель.

Следующая программа иллюстрирует работу функции `fputs()`. Она считывает строку с клавиатуры и записывает ее в файл с именем `TEST`. Для прекращения работы программы следует ввести пустую строку. Поскольку функция `gets()` не позволяет вводить символ перехода, при записи в файл он явно дописывается перед каждой строкой, что облегчает чтение файла.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    char str[80];
    FILE *fp;
    if ((fp = fopen("TEST" , "w")) == NULL) {
        printf("Невозможно открыть файл.\n");
        exit(1);
    }
    do {
        printf("Введите строку (выход - <ENTER>):\n");
        gets(str);
        strcat(str, "\n");
        fputs(str, fp);
    } while (*str != '\n');
}

```

82.2. Функции fprintf() и fscanf()

Кроме уже упомянутых функций ввода-вывода, в языке C предусмотрены функции fprintf() и fscanf(). Эти функции совершенно аналогичны функциям printf() и scanf(), за исключением одного – они работают с файлами. Прототипы функций fprintf() и fscanf() имеют следующий вид:

```
int fprintf(FILE *fp, const char *управляющая_строка, ...);
int fscanf(FILE *fp, const char *управляющая_строка, ...);
```

Здесь параметр fp представляет собой указатель файла, возвращенный функцией fopen(). Функции fprintf() и fscanf() выполняют операции ввода-вывода с файлом.

Рассмотрим в качестве примера программу, считывающую с клавиатуры строку и целое число, а затем записывающую эту информацию в файл с именем TEST. Затем программа читает этот файл и выводит считанные данные на экран. Проверьте содержимое файла TEST после выполнения программы и убедитесь, что его можно понять:

```
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
void main(void)
{
    FILE *fp;
    char s[80];
    int t;
    if ((fp = fopen("test", "w")) == NULL) {
        printf("Невозможно открыть файл.\n");
        exit(1);
    }
    printf("Введите строку и целое число: ");
    fscanf(stdin, "%s%d", s, &t);
    fprintf(fp, "%s %d", s, t);
    fclose(fp);
    if ((fp = fopen("test", "r")) == NULL) {
        printf("Невозможно открыть файл.\n");
        exit(1);
    }
    fscanf(fp, "%s%d", s, &t);
    fprintf(stdout, "%s %d", s, t);
}
```

Хотя функции fprintf() и fscanf() часто обеспечивают наиболее простой способ чтения и записи данных, они не всегда эффективны. Поскольку данные, записанные в файл с помощью формата ASCII, точно соответствуют своему представлению на экране, каждый вызов функций сопряжен с дополнительными расходами.

Занятие 83

83.1. Функции fread() и fwrite()

Для чтения и записи данных, размер типа которых превышает 1 байт, файловая система языка C содержит две функции: fread() и fwrite(). Эти функции позволяют считывать и записывать блоки данных любого типа. Их прототипы имеют следующий вид:

```
size_t fread(void *буфер, size_t количество_байтов,
              size_t количество_блоков, FILE *fp)
size_t fwrite(const void *буфер, size_t количество_байтов,
              size_t количество_блоков, FILE *fp)
```

В прототипе функции fopen() параметр буфер представляет собой указатель на область памяти, в которую записываются данные, считанные из файла. В прототипе функции fwrite() параметр буфер представляет собой указатель на область памяти, содержащую данные, которые должны быть записаны в файл. Значение параметра количество_блоков определяет, сколько блоков подлежит считыванию или записи. Длина блоков задается параметром количество_байтов.

Функция fread() возвращает количество считанных блоков. Оно может быть меньше параметра количество_блоков, если обнаружен конец файла или возникла ошибка. Функция fwrite() возвращает количество записанных блоков. Если не произошло никаких ошибок, это значение равно параметру количество_блоков.

83.2. Применение функций fread() и fwrite()

Если файл был открыт в бинарном режиме, его чтение и запись можно осуществлять с помощью функций fread() и fwrite().

Буфер, как правило, представляет собой обычную область памяти, предназначенную для хранения переменных. В реальных программах значения, возвращаемые функциями fread() и fwrite(), следует анализировать, распознавая возможные ошибки.

В большинстве случаев функции fread() и fwrite() используются для считывания сложных данных, тип которых определен пользователем, особенно структур. Например, допустим, что в программе определена следующая структура:

```
struct struct_type
{
    float balance;
    char name[80];
} cust;
```

Следующий оператор записывает содержимое переменной cust в файл, на который ссылается указатель fp:

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

Занятие 84

84.1. Функция fseek() и файлы с произвольным доступом

С помощью функции fseek() можно осуществлять операции ввода и вывода данных, используя файлы с произвольным доступом. Эта функция устанавливает курсор файла в заданную позицию. Ее прототип имеет следующий вид:

```
int fseek(FILE *fp, long int смещение, int начало);
```

Здесь параметр fp представляет собой указатель файла, возвращенный функцией fopen(). Параметр смещение указывает количество байтов, на которое следует переместить курсор файла от точки, заданной параметром начало. Значение параметра начало задается одним из макросов: SEEK_SET – начало файла; SEEK_CUR – текущая позиция; SEEK_END – конец файла.

Таким образом, чтобы переместить курсор файла, находящийся в начале файла, на количество байтов, заданное параметром смещение, следует применять макрос SEEK_SET. Если точкой отсчета является текущая позиция курсора, используется макрос SEEK_CUR, а если курсор файла установлен на последнюю запись, следует применять макрос SEEK_END. Если функция fseek() выполнена успешно, она возвращает значение 0.

Функцию fseek() можно применять для перемещения на заданное количество байтов, умножая размер соответствующего типа на количество элементов. Допустим, например, что список рассылки состоит из структур, имеющих тип list_type. Для перехода на десятый адрес, записанный в файле, используется следующий оператор:

```
fseek(fp, 9*sizeof (struct list_type), SEEK_SET);
```

84.2. Текущая позиция

Текущую позицию файлового курсора можно определить с помощью функции ftell(). Ее прототип имеет следующий вид:

```
long int ftell(FILE *fp);
```

Данная функция возвращает текущее положение курсора файла, связанного с указателем fp. Если возникла ошибка, она возвращает значение -1.

Как правило, произвольный доступ нужен лишь при работе с бинарными файлами. Причина довольно проста. Поскольку текстовые файлы могут содержать управляющие символы, которые интерпретируются как команды, количество байтов, на которое следует выполнить смещение курсора, может не соответствовать ожидаемому символу. Следует знать, что текстовый файл (содержащий текст), можно открыть в бинарном режиме.

Занятие 85

85.1. Удаление файла

Функция remove() удаляет указанный файл. Ее прототип имеет следующий вид:

```
int remove(const char *имя_файла);
```

Если функция выполнена успешно, она возвращает нуль, в противном случае она возвращает ненулевое значение.

Следующая программа удаляет файл, заданный в командной строке. Однако сначала она дает пользователю возможность передумать. Такие утилиты могут быть полезными для новичков:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
void main(int argc, char *argv[])
{
    char str[80];
    if (argc != 2) {
        printf("Вызов: xerase <filename>\n");
        exit(1);
    }
    printf("Стереть %s? (Y/N): ", argv[1]);
    gets(str);
    if (toupper(*str) == 'Y')
        if (remove(argv[1])) {
            printf("Невозможно удалить файл\n");
            exit(1);
        }
}
```

85.2. Переименование файла

Функция rename() заменяет имя файла. Возвращает ненулевое значение в случае, если попытка изменить имя оказалась неудачной. Ее прототип имеет следующий вид:

```
int rename(const char *старое_имя, const char *новое_имя);
```

Первый параметр задает старое имя, второй – новое.

85.3. Создание временного файла

Функция tmpfile() создает временный файл с режимом доступа "wb+", который автоматически удаляется при его закрытии или обычном завершении программой своей работы. Ее прототип имеет следующий вид:

```
FILE *tmpfile(void);
```

Функция возвращает поток, а если не смогла создать файл – NULL.

Занятие 86

86.1. Стандартные потоки

В начале выполнения любой программы, написанной на языке C, автоматически открываются три потока: `stdin` (стандартный поток ввода), `stdout` (стандартный поток вывода) и `stderr` (стандартный поток ошибок). Обычно эти потоки связаны с консолью, однако операционная система может перенаправлять их на другие устройства.

Поскольку стандартные потоки представляют собой указатели файлов, их можно использовать для консольного ввода-вывода. Например, функцию `putchar()` можно определить таким образом:

```
int putchar(char c)
{
    return putc(c, stdout);
}
```

Как правило, поток `stdin` используется для ввода с клавиатуры, а потоки `stdout` и `stderr` используются для записи на экран дисплея.

Потоки `stdin`, `stdout` и `stderr` можно применять как файловые указатели в любой функции, использующей указатели типа `FILE*`. Например, функцию `fgets()` можно применять для ввода строки с консоли, используя следующий вызов:

```
fgets(str, 80, stdin);
```

Фактически такое использование функции `fgets()` довольно полезно. Как упоминалось ранее, применение функции `gets()` может привести к переполнению массива, в который записываются символы, поскольку она не предусматривает проверки возможного выхода индекса массива за пределы допустимого диапазона. При использовании потока `stdin` функция `fgets()` представляет собой разумную альтернативу, поскольку она выполняет проверку индекса массива и предотвращает его переполнение. Остается лишь одно неудобство: функция `fgets()`, в отличие от функции `gets()`, не удаляет из файла символ перехода на новую строку, поэтому его необходимо удалять вручную.

Потоки `stdin`, `stdout` и `stderr` не являются переменными в общепринятом смысле этого слова и им ничего нельзя присвоить с помощью функции `foren()`. Кроме того, они открываются и закрываются автоматически, поэтому не следует пытаться их закрывать самостоятельно.

86.3. Связь с консольным вводом-выводом

Между консольным и файловым вводом-выводом существует небольшое различие. Функции консольного ввода-вывода работа-

ют с потоками `stdin` или `stdout`. По существу, функции консольного ввода-вывода просто являются специальными версиями своих файловых аналогов и включены в язык лишь для удобства программистов.

Консольный ввод-вывод можно осуществлять с помощью функций файловой системы, а файловые операции ввода-вывода можно выполнять с помощью консольных функций, например функции `printf()`. Это возможно благодаря тому, что консольные функции ввода-вывода работают с потоками `stdin` и `stdout`. Если операционная среда позволяет перенаправлять потоки, потоки `stdin` и `stdout` можно связать не с клавиатурой и экраном, а с другими устройствами.

86.3. Перенаправление стандартных потоков

Для перенаправления стандартных потоков можно применять функцию `freopen()`. Эта функция связывает существующий поток с новым файлом. Таким образом, с ее помощью стандартный поток можно связать с новым файлом. Ее прототип имеет следующий вид:

```
FILE *freopen(const char *имя_файла,
              const char *режим,
              FILE *поток)
```

Параметр `имя_файла` является указателем на строку, содержащую имя файла, связанного с заданным потоком. Файл открывается в режиме, заданном параметром `режим`, который может иметь те же значения, что и параметр `режим` в функции `foren()`. В случае успешного выполнения функция возвращает указатель файла `поток`, в противном случае она возвращает нулевой указатель.

В следующей программе функция `freopen()` перенаправляет стандартный поток `stdout` в файл `OUTPUT`:

```
#include <stdio.h>
void main(void)
{
    char str[80];
    freopen("OUTPUT", "w", stdout);
    printf("Введите строку: ");
    gets(str);
    printf(str);
}
```

Как правило, перенаправление стандартных потоков с помощью функции `freopen()` оказывается полезным в особых ситуациях, например, при отладке. Однако этот способ менее эффективен, чем применение функций `fread()` и `fwrite()`.