МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Российский химико-технологический университет
имени Д. И. Менделеева

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ СИ

Москва

2014

Составители: Н. А. Федосова, А. В. Женса

УДК 004.78
ББК 3281
М 54

Рецензент:
Кандидат технических наук, доцент
Российского химико-технологического университета им. Д. И. Менделеева
Б. Б. Богомолов

Федосова Н. А., Женса А. В.

М 54 Основы языка программирования Си: учебное пособие / Н. А. Федосова, А. В. Женса. − М.: РХТУ им. Д. И. Менделеева, 2014. − 150 с.

Учебное пособие предназначено для студентов, обучающихся по очной форме обучения по дисциплине "Информатика" в 1-ом семестре бакалавриата:

- – по направлению подготовки 230400 "Информационные системы и технологии" (объем 128 часов: 72 аудиторных часов, из них 36 часов лекции и 36 часов лабораторных занятий);
- – по направлению подготовки 230100 "Информатика и вычислительная техника " (объем 126 часов: 72 аудиторных часов, из них 36 часов лекции и 36 часов лабораторных занятий).

Пособие представляет раздел курса лекций по дисциплине "Информатика". В пособие изложены характер языка и структура программ на языке программирования Си. В пособии рассмотрены следующие разделы дисциплины: структура программы; типы данных; операции; операторы; функции; одномерные и многомерные числовые массивы; строки и массивы строк; указатели и работа с динамической памятью; структуры и массивы структур; функции стандартных библиотек. Пособие включает значительное количество подробно рассматриваемых примеров и позволяет приобрести практические навыки программирования на языке Си.

Учебная информация представлена в пособии в текстовой, графической и табличной формах и структурирована по главам.

Представленное учебное пособие рассчитано на использование студентами при изучении теоретических частей вышеуказанных дисциплин, на практических занятиях и при подготовке к лабораторным работам.

УДК 004.78
ББК 3281

Введение

Язык Си – это стандартизированный процедурный язык программирования, разработанный в 1972 году сотрудником компании AT&T Bell Laboratories Деннисом Ритчи (англ. Dennis Ritchie) в качестве развития языка Би. Он был создан как основа при проектировании операционной системы UNIX. Сейчас язык Си используется во множестве операционных систем и является самым распространенным и широко используемым языком программирования (по данным на 2013 год).

Несмотря на то, что изначально язык Си не разрабатывался как язык для обучения программированию (в отличие, например, от Pascal и Qbasic), он активно используется в этом направлении. Синтаксис этого языка стал основой для множества других языков программирования.

Особенностями этого языка являются:

- небольшое количество ключевых слов;
- наличие сложных типов данных (структуры, объединения);
- возможность использования указателей, работа с памятью;
- наличие внешних стандартных библиотек;
- компиляция программного кода в бинарный код;
- использование макропроцессора.

Сравнение языка Си с другими языками:

- язык является прародителем C++, Objective C, C#;
- язык сильно повлиял на Java, Perl, Python;
- низкий уровень языка Си предоставляет выигрыш в скорости исполнения кода;
- недостатки по сравнению с другими языками:
    - отсутствие исключений (exceptions);
    - отсутствие проверки диапазонов (range-checking);
    - отсутствие автоматической сборки мусора;
    - Си не является объектно-ориентированным языком;
    - отсутствие полиморфизма.

Краткая история развития языка программирования Си:
1972 – разработка языка Си.
1978 – опубликование языка Си; появление первой спецификации.
1989 – появление стандарта C89 (известен как ANSI C или Standard C).
1990 – ANSI C, адаптированный под ISO, известен как стандарт C90.
1999 – появление стандарта C99 (этот стандарт имеет обратную совместимость со стандартом C89 и полностью реализован во многих компиляторах).
2011 – выход стандарта C11 (добавлена поддержка многопоточности).

В данном пособии рассматривается синтаксис ANSI / ISO C (C89/C90).

1. Общие сведенья

1.1.Алфавит языка Си

Алфавит языка составляют:

- прописные и строчные буквы латинского алфавита: A-Z, a-z;
- цифры: 0-9;
- специальные знаки: . , ; : ? ! ‵ " + - * / % ( ) [ ] { } > = < \ & # _ ~ ^;
- символы пробела: пробел, табуляция, символ конца строки.

Таблица 1.1. Наименования символов языка Си.

Символ	Наименование	Символ	Наименование
,	запятая	+	Плюс
.	Точка	-	Минус (дефис)
;	Точка с запятой	( )	Круглые скобки
:	Двоеточие	{ }	Фигурные скобки
?	Вопросительный знак	[ ]	Квадратные скобки
!	Восклицательный знак	<	Меньше
‵	Одинарная кавычка (апостроф)	>	Больше
"	Двойная кавычка	~	Тильда
|	Вертикальная черта	#	Решетка
/	Дробная черта (слеш)	%	Процент (знак целочисленного деления)
\	Обратная черта (обратный слеш)	&	Амперсанд

1.2.Лексемы

Лексемы – это наименьшие неделимые элементы языка, из которых составляются остальные конструкции.

Классы лексем:

- ключевые слова;
- идентификаторы;
- константы;
- строковые литералы;
- операторы;
- разделители и пунктуаторы.

Ключевые слова зарезервированы в качестве служебных слов и не могут использоваться в другом смысле.

Таблица 1.2. Ключевые слова языка Си.

auto	double	int	break
else	long	switch	typedef
char	extern	return	case
float	unsigned	default	signed
union	do	if	volatile
continue	while	enum	

Идентификаторы – это символические имена, которыми обозначают переменные, функции, типы данных, метки и другие объекты. Идентификатор может состоять из латинских букв (прописные и строчные), цифр и символа подчеркивания. При этом цифра не может быть первым символом идентификатора.

Язык Си, в отличие от некоторых других языков, чувствителен к регистру. Это значит, что идентификаторы `variable`, `Variable` и `VariAble` являются тремя различными идентификаторами.

Константы служат для предоставления постоянных, неизменяемых значений.

Строковые литералы – это любая последовательность символов, которая заключена в двойные кавычки.

Операторы – это знаки арифметических операций или операций сравнения (+ ++ = == > < >= <=).

Разделители и пунктуаторы осуществляют функции группировки и упорядочивания кода (* = ( ) [ ] { } , ; : … #).

1.3.Комментарии

Многострочный комментарий начинается с символов /* и заканчивается символами */. Между звездочкой и слешем не должно быть никаких пробелов. Любой текст, расположенный между начальными и конечными символами комментария, игнорируется компилятором.

```
/* это
многострочный
комментарий */
```

Комментарии могут находиться в любом месте программы, за исключением середины ключевого слова или идентификатора. Также не следует размещать комментарии в середине выражений. Чаще всего их используют перед телом функций, например, для уточнения их роли в программе.
Многострочные комментарии не могут быть вложенными. То есть в одном комментарии не может находиться другой.

Однострочный комментарий начинается с символов `//` и заканчивается в конце строки.

```
// это однострочный комментарий
```

Однострочные комментарии особенно полезны тогда, когда нужны краткие, не более чем в одну строку пояснения.
Однострочный комментарий может находиться внутри многострочного комментария. Например, следующий комментарий является вполне допустимым:

```
/* это // вложенный однострочный комментарий
в многострочный комментарий */
```

Комментарии должны находиться там, где требуется пояснить логику алгоритма. Чем сложнее ваша программа, тем больше комментариев она требует.
Кроме обычного назначения, комментарии часто помогают при выявлении ошибок в коде. Поочередно комментируя части кода, можно локализовать место ошибки.

1.4. Структура программы

Программа на языке Си имеет строгую структуру. В начале программы располагается блок подключения заголовочных файлов стандартной библиотеки. Второй блок состоит из объявления символьных констант, глобальных переменных, глобальных констант, объявление прототипов функций. Третий блок составляет функция `main()`. Четвертый блок – это последовательное описание второстепенных функций.

Общий порядок следования:

1. заголовочные файлы стандартной библиотеки;
2. заголовочные файлы динамических библиотек;
3. символьные констаты, макросы;
4. глобальные переменные и константы;
5. объявление прототипов вспомогательных функций;
6. описание функции `main();`
7. описание вспомогательных функций.

1.5.Компиляция и запуск программы

После написания кода программы ее необходимо сохранить в файле с расширением «.c» (например, my_program.c). Далее следует компиляция программы, которая состоит из трансляции и компоновки.

Трансляция – это перевод программы на высокоуровневом языке на низкоуровневый язык, близкий к машинному коду. При трансляции происходит образование объектного файла my_program.o.

Компоновка – это линковка (связывание) объектного файла с подключенными в программе библиотеками и последующая сборка итогового исполняемого файла.

В итоге компиляция – это процесс передачи компилятору исходного файла и получение нового исполняемого файла my_program.exe (my_program.out). После формирования исполняемого файла становится возможным запуск программы.

На этапе компиляции также происходит проверка кода на ошибки. Если таковые будут найдены, то компиляция приостанавливается и иполняемый файл не создается.

2. Работа с данными

2.1.Типы данных

Простые базовые типы данных представлены на Рисунке 2.1.

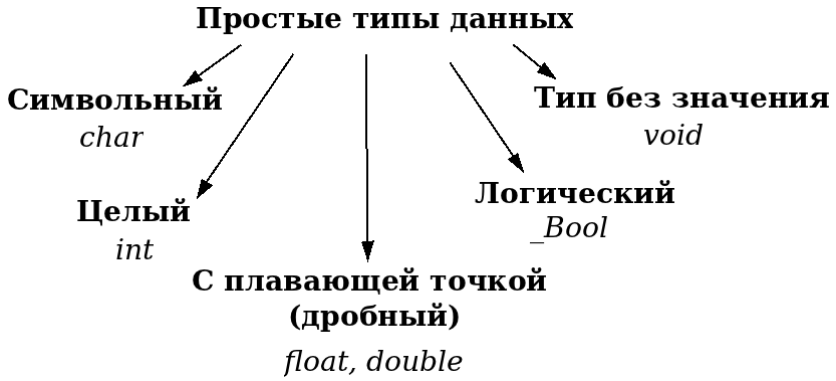


Рисунок 2.1. Простые типы данных языка Си.

Символьный тип данных (`char`) содержит данные в виде единичных символов, букв, цифр или знаков. При написании кода программы единичные символы заключают в одинарные кавычки.

Числовые типы данных (`int, float, double`) включают в себя целые и дробные числа.

- **`int`** – целые числа
- **`float`** – дробные числа (одинарная точность `4` знака после запятой)
- **`double`** – дробные числа (двойная точность `8` знаков после запятой)

Логический тип данных (`_Bool`) позволяет хранить значения:

- `1` (`true`)

- `0` (`false`)

Тип без значения (`void`) служит для объявления функций, которые не возвращают значения, и создания универсальных указателей.

Модификации простых базовых типов

Базовые типы данных (кроме `void` и `_Bool`) могут иметь спецификаторы длины и спецификаторы знака, предшествующие им в тексте программы (`signed`, `unsigned`, `short`, `long`). Спецификатор изменяет значение базового типа так, чтобы он более точно соответствовал своему назначению в программе.

Символьный и целый типы могут быть знаковыми (`signed`) или беззнаковыми (`unsigned`). Целые числа со знаком и без знака отличаются интерпретацией нулевого бита числа. Если целая переменная объявлена со знаком, то компилятор считает, что нулевой бит содержит знак числа. Если в нулевом бите записан `0`, число считается положительным, а если `1` – отрицательным.

Спецификаторы длины (`short`, `long`) определяют диапазоны значений для типа и сколько оперативной памяти будет выделено для хранения переменных данного типа.

Сложные типы данных

На основе базовых типов можно образовать сколь угодно сложные и многоуровневые типы данных:

- массивы;
- структуры;
- перечисления;
- объединения;
- битовые поля.

2.2.Переменные

Переменная – это идентификатор, скрывающий за собой область памяти с хранящимися там данными. Иначе говоря, это имя области памяти.

У каждой переменной есть тип, который соответствует тому, какой тип данных хранит переменная. Данные, которые хранит переменная (значение переменной), могут изменяться в пределах типа переменной. Это значит, что если переменная хранит данные одного типа, то она никогда не сможет хранить данные другого типа. То есть в течение жизни переменной она может сколько угодно раз менять свое значение, но ни разу не может поменять свой тип.

Имена переменных

Имя переменной может состоять из записанных в любом порядке символов латинского алфавита, цифр и символа подчеркивания. При этом первым символом имени переменной не может быть цифра.

При выборе имен переменных желательно придерживаться правила о том, что переменные должны называться в соответствии со своим смыслом. Например, если переменная хранит результат суммирования, то логичнее было бы назвать ее `sum`, `amount`, `count` или `summa`, нежели просто `x`, `k` или `s1`. Для имен переменных рекомендуется использовать английские слова.

При написании программы очень важно стараться выбирать такие названия для переменных и других конструкций, которые будут понятны не только вам, но и другим программистам. Подавляющее большинство программистов работают в команде, и написание понятного кода – это важнейшее условие плодотворного взаимодействия.

Объявление переменных

Перед тем как использовать переменные в программе, их необходимо объявить. Во время объявления переменных происходит выделение памяти для их хранения. Количество выделяемой памяти соответствует типу переменных.

Объявление переменной может быть расположено в трех местах программы: внутри функции, вне всех функций и при определении параметров функции. Это места объявлений соответственно локальных переменных, глобальных переменных и формальных параметров функций.

```
тип_данных имя_переменной;
```

При объявлении переменных сначала записывается тип данных, который будет содержать переменная, затем через пробел – имя переменной. Если требуется объявить несколько переменных одного типа, то это можно сделать через запятую.

- Объявление по одной переменной

```
int number;
double length;
char word;
```

- Объявление сразу нескольких переменных

```
int i, j, age, mass;
double cat, ball, weight;
char symbol, some_letter;
```

Инициализация переменных

Для инициализации переменной после ее объявления необходимо поставить знак равенства и указать начальное значение. При этом инициализировать можно все переменные или только некоторые из них.

- Инициализация переменных при объявлении

```
int y = 5;
double var, age_kitten = 1.7, g = 2.467;
char k, letter = '?', name;
```

Неинициализированные локальные переменные до первого присвоения имеют произвольное значение. Неинициализированные глобальные переменные автоматически обнуляются.
Глобальные переменные инициализируются только один раз в начале работы программы. Локальные переменные инициализируются каждый раз при входе в блок (область между ближайшими фигурными скобками), в котором они объявлены.
Следует сказать о том, что приобретение привычки всегда инициализировать переменные поможет вам избежать ошибок в программе. Недопустимо, чтобы переменные содержали мусор на любом этапе их жизни.

2.3.Константы

Константа – это фиксированное значение, которое не может быть изменено программой. Она может относиться к любому базовому типу.

Константы выполняют несколько функций:

- использование констант как размерности массивов (может не поддерживаться некоторыми версиями компиляторов);
- слежение за тем, чтобы случайным образом не изменить величину какой-либо переменной;
- для удобства пользователя и разграничения числовых значений в программе.

Квалификатор `const`

Для того чтобы обозначить, что идентификатор является константой, нужно добавить квалификатор **`const`** перед типом идентификатора. Присваивать значение константе можно только при объявлении.

```
const тип_константы имя_константы = значение;
```

Отличием константы от переменной является то, что ее идентификатор может стоять слева от знака = только один раз за всю программу (при объявлении константы)

```
const int a = 11, b = 16;
const double dlina = 3.9, shirina = 67.45, count = 567.1;
const char word = '}';
const char letter = word;
```

Директива #define

Директива **#define** определяет идентификатор и последовательность символов, которая во время компиляции программы будет подставляться вместо идентификатора каждый раз, когда он встретится. Идентификатор называется макросом, а сам процесс замены – макрозаменой. В общем виде директива выглядит таким образом:

```
#define имя_макроса  последовательность_символов
```

Обратите внимание, что в этом выражении в конце нет точки с запятой. Если попробовать поставить точку с запятой в конце строки с директивой **#define**, то это может привести к ошибке, так как при макрозамене вместо макроса подставляется вся последовательность символов до конца строки вместе с точкой с запятой.

Имена макросов часто используются для определения имен так называемых "магических чисел". Рекомендуется избегать употребления чисел в программах и заменять их макросами (**#define**) или константами (**const**). Например, если в нашей программе предполагается использовать отрезок [10, 20], то вместо значения 10 можно использовать LEFT, а вместо значения 20 – RIGHT.

```
#define LEFT 10
#define RIGHT 20
```

После определения имени макроса его можно использовать в определениях других имен макросов.

```
#define ONE 1
#define TWO ONE + ONE
#define THREE ONE + TWO
```

Если последовательность символов не помещается в одной строке, то ее можно продолжить на следующей строке, поместив в конце предыдущей обратную косую черту (обратный слеш):

```
#define LONG_STRING "это пример очень \
длинной строки"
```

3. Операции

Язык Си содержит большое количество встроенных операций. Их роль значительно больше, чем в других языках программирования. Существует четыре основных класса операций: арифметические, логические, поразрядные и операции сравнения.

3.1. Арифметические операции

Все используемые в языке Си арифметические операции перечислены в таблице 3.1. Операции делятся на унарные, бинарные, аддитивные и мультипликативные.

В унарных арифметических операциях участвует только один операнд (переменная или константа).

Арифметические операции называются бинарными, если в них участвуют два операнда (переменные, константы или числовые значения).

Аддитивные арифметические операции – это операции сложения и вычитания.

Мультипликативные операции – это операции умножения, деления и нахождения остатка от деления нацело.

Таблица 3.1. Арифметические операции языка Си.

Оператор	Операция	Класс операций	
++	инкремент (a++ ++a)	аддитивные	унарные
--	декремент (a-- --a)		
+	сложение (a + b)		бинарные
-	вычитание (a - b)		
-	унарный минус (-a)	мультипликативные	унарная
*	умножение (a * b)		бинарные
/	деление (a / b)		
%	нахождение остатка от деления нацело (a % b)		

Нахождение остатка от деления % в языке Си работает так же, как и в других языках, его результатом является остаток от целочисленного деления.

```
int x = 5, y = 2;
int z = x % y;
```

Здесь переменная z будет равна 1, так как при делении 5 на 2 получается 2 целых и 1 в остатке. Этот оператор, однако, нельзя применять к типам данных с плавающей точкой, так как компьютер не может эти числа поделить нацело.

Инкремент и декремент

Кроме обычных арифметических операторов в языке Си определены два очень полезных оператора для упрощения часто употребляемых операций. Это инкремент ++ (плюс-плюс) и декремент -- (минус-минус). Оператор инкремента увеличивает значение операнда на 1, а декремента - уменьшает на 1.

Таблица 3.2. Варианты инкремента и декремента.

Обычная запись выражения	Префиксный инкремент/декремент	Постфиксный инкремент/декремент
`x = x + 1;`	`++x;`	`x++;`
`x = x - 1;`	`--x;`	`x--;`

Инкремент и декремент могут иметь префиксную (`++x; --x;`) и постфиксную (`x++; x--;`) форму, то есть могут предшествовать операнду (целой переменной) или следовать за ним. Если оператор инкремента или декремента предшествует операнду, то сама операция выполняется до использования результата в выражении. Если же оператор следует за операндом, то в выражении значение операнда используется до выполнения операции инкремента или декремента. То есть для выражения эта операция как бы не существует, она выполняется только для операнда. Например,

```
x = 10;
y = ++x;
```

здесь сначала переменной x присваивается значение 11, а затем переменной y присваивается значение x, которое уже равно 11. Однако если написать

```
x = 10;
y = x++;
```

то сначала переменной y будет присвоено значение 10, а потом уже прибавится единица к переменной x. В обоих случаях переменной x будет присвоено значение 11, разница только в том, когда именно это случится: до или после присваивания значения переменной x переменной y.

Большинство компиляторов языка Си генерируют для инкремента и декремента очень быстрый и эффективный объектный код, значительно лучший, чем для соответствующих операторов присваивания. Поэтому везде, где это возможно, рекомендуется использовать инкремент и декремент.

Приоритет выполнения арифметических операций

В языке программирования Си есть определенный порядок выполнения арифметических операций. Он схож с тем порядком, который используется в математике.

Приоритет выполнения операций:

1. операции в скобках;
2. функции стандартной математической библиотеки `math.h`;
3. унарные операции;
4. умножение, деление, нахождение остатка от деления;
5. сложение, вычитание.

Операции с одинаковым приоритетом выполняются слева направо. Используя круглые скобки, можно изменить порядок вычислений. В языке Си круглые скобки придают операции (или последовательности операций) наивысший приоритет. Для избегания ошибок необходимо всегда явно указывать приоритет операций с помощью скобок.

3.2. Операции сравнения и логические операции

Операции сравнения – это операции, в которых значения двух переменных или выражений сравниваются друг с другом. А с помощью логических операций реализуются операции формальной логики. Между логическими операциями и операциями сравнения существует тесная связь: результаты операций сравнения часто являются операндами логических операций.

В операциях сравнения и логических операциях в качестве операндов и результатов операций используются значения `ИСТИНА` (`true`) и `ЛОЖЬ` (`false`). В языке Си значение `ИСТИНА` представляется любым числом, отличным от нуля. Значение `ЛОЖЬ` представляется нулем.

Таблица 3.2. Операторы сравнения и логические операторы.

Операторы сравнения	
Оператор	**Операция**
>	больше
>=	больше или равно
<	меньше
<=	меньше или равно
==	равно

!=	не равно
Логические операторы	
Оператор	**Операция**
&&	И
||	ИЛИ
!	НЕ

Приоритет логических операций и операций сравнения

1. !
2. > >= < <=
3. == !=
4. &&
5. ||

Как операции сравнения, так и логические операции имеют низший приоритет по сравнению с арифметическими. То есть, выражение `10 > 1 + 12` интерпретируется как `10 > (1 + 12)`. Результат равен `ЛОЖЬ`.

В одном выражении можно использовать несколько операций:

```
10 > 5 && !(10 < 9) || 3 < 4
```

В языке Си не определена операция `"исключающего ИЛИ"` (`exclusive OR`, или `XOR`). Однако с помощью логических операторов можно описать функцию, выполняющую эту операцию `(a || b) && !(a && b)`.

Как и в арифметических выражениях, для изменения порядка выполнения операций сравнения и логических операций можно использовать круглые скобки. Например, выражение:

```
!0 && 0 || 0
```

равно `ЛОЖЬ`. Однако если добавить скобки, то результатом будет `ИСТИНА`:

```
!(0 && 0) || 0
```

3.3.Явные и неявные преобразования типов данных

Явное и неявное преобразования типов

В Си различают явное и неявное преобразование типов данных. Неявное преобразование типов данных выполняет компилятор, а явное преобразование данных задается программистом.

Неявное преобразование типов при арифметических операциях

При неявном преобразовании типов данных результат любого вычисления будет преобразовываться к наиболее точному типу данных из тех, которые участвуют в вычислении. Если вы делите целое число на другое целое число, то в ответе вы тоже получаете целое число. Но если хотя бы одно из чисел (делитель или делимое) является дробным числом, то в результате получается дробное число, так как оно обладает большей точностью нежели целое число.

```
int x = 5, y = 2;
int z = x / y;        // z = 2
```

здесь `z = 2`, так как это целочисленное деление (вся дробная часть отбрасывается).

Целое число делим на дробное число:

```
int x = 5;
double y = 2.0;
double z = x / y;     // z = 2.5
```

Дробное число делим на целое число:

```
int y = 2;
double x = 5.0;
double z = x / y;     // z = 2.5
```

Дробное число делим на дробное число:

```
double x = 5.0;
double y = 2.0;
double z = x / y;     // z = 2.5
```

Во всех трех случаях значение дробной переменной `z` получается одинаковым `z = 2.5`, так как в процессе деления участвует хотя бы одно дробное число.

Неявные преобразования типов при операции присваивания

Неявное преобразование типов при присваивании происходит тогда, когда справа и слева от операнда присваивания стоят переменные или выражения различных типов. В этом случае выражение с правой стороны операнда неявно преобразуется к типу выражения с левой стороны.

```
double f = 50;
```

Дробной переменной `f` присваивается значение целого числа `50`. При этом число `50` преобразуется компилятором языка в дробное число `50.0`, а затем происходит операция присваивания.

```
int var = -34;
double new_var2 = var1;
```

Целое значение переменной `var` преобразуется компилятором в дробное значение `-34.0` (преобразуется только скопированное значение переменной `var` сама переменная `var` остается целого типа, и в ней ничего не меняется). А затем происходит присвоение переменной `new_var` дробного значения `-34.0`.

```
double age = 9.8;
int class = age - 7;
```

Если целой переменной присвоить дробное значение, то произойдет отбрасывание дробной части. Здесь сначала выполнится вычитание `age – 7`. Результат такого выражения будет дробным `2.8`. Перед присвоением отбрасывается дробная часть, и получается целое число `2`, которое присваивается переменной `class`.

```
double u = 7.34;
const int p = u = 90;
```

Дробной переменной `u` присвоили дробное значение `7.34`. Затем этой же переменной присвоили целое значение `90`, предварительно преобразовав его в дробное `90.0`. Теперь `u = 90.0`. И, наконец, целой переменной `p` присвоили дробное значение `90.0`, предварительно преобразовав его в целое значение `90`. Теперь `p = 90`.

Явные преобразования типов.

Для того чтобы произвести явные преобразования типа переменной или результата вычислений, используются операции преобразования типа:

```
(тип, к которому нужно преобразовать) выражение
```

```
double x;
x = (int)3.74;
```

Дробное число `3.74` преобразуется к целому путем отбрасывания дробной части. Затем получившаяся величина `3` преобразуется в дробную величину 3.0, чтобы записать ее в дробную переменную `x`. В итоге получается, что `x = 3.0`.

```
double x;
x = (int)7.9 / 2;
```

Число `7.9` преобразуется к целому типу `7` и делится на целое число `2`. В результате целочисленного деления `7` на `2` получается целое число `3`, которое затем преобразуется в дробное `3.0` и записывается в дробную переменную `x`.

Явное преобразование типов необходимо, если вам нужно найти среднее арифметическое двух целых переменных:

```
int x = 5, y = 10;
double average = (double)(x + y) / 2;
```

Переменные складываются, сумма переменных преобразуется в дробное число, а затем делится на `2`. При этом мы делим дробное число на целое и получаем дробный результат, который уже присваивается дробной переменной `average`.

Стоит отметить, что результатом вычислений всех математических функций (кроме **`abs()`**) являются дробные числа. Это стоит иметь ввиду для учета неявных преобразований типов.

4. Операторы

4.1. Одиночный оператор

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

```
func();          // вызов функции
a = b + c;       // оператор присваивания
b++;         // оператор инкремента
;            // пустой оператор
```

Первый оператор выполняет вызов функции, второй – присваивание. Третий оператор увеличивает значение переменной на единицу. Последний оператор – это пустой оператор, который не выполняет никаких действий.

4.2. Блок операторов

Блок операторов – это последовательность операторов, заключенных в фигурные скобки. Операторы, составляющие блок, логически связаны друг с другом и рассматриваются как одна программная единица. Блок всегда начинается открывающейся фигурной скобкой `{` и заканчивается закрывающейся фигурной скобкой `}`. Чаще всего блок используется как составная часть какого-либо оператора, выполняющего действие над группой операторов, например, **`if`** или **`for`**.

```
for (x = 0; x < 5; x += 2) {
    x = y + z;
    t = k * m;
}
```

4.3.Оператор присваивания

Оператор присваивания может присутствовать в любом выражении языка Си. Этим Си отличается от большинства других языков программирования, в которых присваивание возможно только в отдельном операторе.

Общая форма оператора присваивания:

```
имя_переменной = выражение;
```

Выражение может быть сколь угодно сложным выражением. Оператором присваивания служит знак `"="`. Адресатом (левой частью оператора) присваивания должен быть объект, способный получить значение, например переменная.

Множественное присваивание

В одном операторе присваивания можно присвоить одинаковое значение многим переменным. Для этого используется оператор множественного присваивания:

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

Операции присваивания выполняются справа налево. Таким образом, сначала выполнится операция `z = 0`, затем `y = z`, и, наконец, `x = y`.

Составное присваивание

Составное присваивание — это разновидность оператора присваивания, в которой запись сокращается и становится более удобной в написании. Например, оператор `a = a + 35;` можно записать как `a += 35;`

Оператор `"+="` сообщает компилятору, что к переменной `a` нужно прибавить `35`. Составные операторы присваивания существуют для всех бинарных операций.

Таблица 4.1 Составные операторы присваивания.

Обычный оператор присваивания	Составной оператор присваивания
`x = x + 3;`	`x += 3;`
`x = x - 3;`	`x -= 3;`
`x = x * 3;`	`x *= 3;`
`x = x / 3;`	`x /= 3;`

`x = x % 3;`	`x %= 3;`

4.4.Операторы ветвления

Тернарный оператор

Оператор ?: Называют тернарным оператором (от лат. ternarius — «тройной»).
Общий вид операции:

```
(Условие) ? Оператор1 : Оператор2;
```

Если условие выполняется, то исполнятся `Оператор1`, если нет — то `Оператор2`.

```
max = (x > y) ? x : y;
printf("Наибольшее число равно %d", max);
```

Тернарную операцию можно включать в другие операции.

```
printf("Наибольшее число = %d", (x > y) ? x : y);
```

Здесь переменная `max` не обязательна. Вычисление наибольшего из `x` и `y` происходит прямо в функции **`printf`**`()`, без использования промежуточной переменной.
Тернарные операции так же могут быть вложенными. В качестве примера можно рассмотреть поиск наибольшего из трех различных чисел.

- В две строки (без вложенности):

```
max = (a > b) ? a : b;
max = (max > c) ? max : c;
```

Сначала находим наибольшее из `a` и `b` и записываем его в `max`. Затем в `max` записываем максимальное из `max` и `c`. Таким образом находим максимальное из трех чисел.

- В одну строку (вложенность):

```
max = (((a > b) ? a : b) > c) ? ((a > b) ? a : b) : c;
```

Такой вид записи условия возможен, но представляет значительную трудность для понимания. В случае сложных условий следует использовать оператор условия **if-else**

Оператор if-else

Общая (полная) форма оператора **if-else** следующая:

```
if(выражение/условие) {
    блок операторов;
}
else {
    блок операторов;
}
```

Если выражение/условие в скобках истинно (любое значение, отличное от нуля), то выполняется блок операторов, следующий за **if**. В противном случае выполняется блок операторов, следующий за **else**. Необходимо помнить, что выполняется либо оператор, связанный с **if**, либо связанный с **else**, но оба — никогда!

Условный оператор может иметь неполную форму, при которой **else** и последующий блок операторов отсутствуют:

```
if (выражение/условие) {
    блок операторов;
}
```

Если у вас не одно, а несколько условий, то каждое из них рекомендуется заключить в круглые скобки. При этом не забывайте об общих круглых скобках.

```
if ((условие1) || (условие2) && (условие3))
```

Условное выражение должно иметь скалярный результат. Это значит, что результатом должно быть целое число, символ, указатель или число с плавающей точкой, но им не может быть массив или структура. В выражении/условии оператора **if** результат типа с плавающей точкой используется редко, потому что это существенно замедляет вычислительный процесс. Объясняется это тем, что для выполнения операций над переменными дробного типа необходимо выполнить больше команд процессора, чем для выполнения операций над целыми числами или

символами. Кроме того, следует избегать сравнения различных типов данных. В большинстве случаев оно исполняется некорректно.

Пример. Вывести на экран модуль целого числа.

```
int x = 0;
printf("Введите x:");
scanf("%d", &x);
if (x < 0) {
  x *= -1;
}
printf("модуль x = %d\n", x);
```

Если мы ввели с клавиатуры отрицательное число, то для вычисления модуля нужно умножить его на минус единицу. Если же число не отрицательно, то, ни на что умножать не надо, его модуль будет равен самому числу. Поэтому отсутствует часть **else**.

Пример. Вычислить значение переменной.

```
int x = 5, y = 7;
double z = 0;
if ((x * y > 50) || (x / y < 10)) {
  z = (x + y) * 10.4;
}
else {
  z = (35 * x - 12 * y) * 0.5;
}
printf("z=%lf", z);
```

Вычисление значения переменной z в зависимости от условий. Переменная вычисляется по одной из двух формул и результат вычисления выводится на экран.

Операторами для **if** и **else** могут быть так же условные операторы. При этом вложенность может быть многократная.

Пример. Найти наибольшее из трех чисел.

```
int a = 0, b = 0, c = 0;
scanf("%d %d %d", &a, &b, &c);
if (a > b) {
  if (a > c) {
    printf("a – наибольшее число");
  }
  else {
```

```
      printf("c – наибольшее число");
    }
  else {
    if (b > c) {
      printf("b – наибольшее число");
    }
    else {
      printf("c – наибольшее число");
    }
  }
```

Условный оператор множественного выбора (переключатель) `switch`.

```
switch (селектор){
  case значение1:
                  блок операторов;
                  break;
  case значение2:
                  блок операторов;
                  break;
  case значение3:
                  блок операторов;
                  break;
  case значение4:
                  блок операторов;
                  break;
  default:
                  блок операторов;
                  break;
}
```

Селектор — это переменная или выражение, которое должно иметь целочисленный или символьный тип. Оператор **`switch`** имеет две и более ветвей исполнения. Каждая ветвь начинается с ключевого слова **`case`**, за ним следует значение селектора, при котором должна выполняться данная ветвь. Чтобы после завершения кода ветви произошел выход из оператора переключателя, используется специальная команда **`break`**. Если такой команды в ветви нет, после исполнения кода выбранной ветви начнется исполнение кода всех следующих за ней ветвей. Ветвь **`default`** исполняется тогда, когда среди прочих ветвей не нашлось ни одной подходящей.

```
printf("Введите оценку ученика");
scanf("%d", &mark);
switch(mark){
  case 2:
          printf("Неуспевающий ученик");
          break;
  case 3:
          printf("Слабый ученик");
          break;
  case 4:
          printf("Хорошист");
          break;
  case 5:
          printf("Отличник");
          break;
  default:
          printf("Неверная оценка");
          break;
}
```

Выбор условного оператора

При написании программы выбор условного оператора следует делать, исходя из задач и сложности реализуемой программы. Если условие, которое нужно реализовать, достаточно мало, то стоит попробовать обратиться к тернарной операции. Если же ветвление вашей программы идет в зависимости от вполне конкретных вариаций переменной целого или символьного типа, то оперировать стоит условным оператором `switch()`. На практике чаще всего используется условный оператор `if-else`. Этот оператор можно использовать для всех случаев ветвления программ.

4.5. Операторы цикла

Операторы цикла

Циклические операции являются часто употребляемыми операциями. Они служат для многократного выполнения последовательности операторов до тех пор, пока не выполниться некоторое условие. Условие может быть установлено заранее или меняться при выполнении тела цикла.

Цикл `while`

Общая форма цикла `while` имеет вид:

```
while (условие) {
    блок операторов;
}
```

Тело цикла может быть пустым, состоять из единственного оператора или блока операторов. Условие цикла может быть любым допустимым в языке Си выражением. Цикл **while** использует предусловие. Оно считается истинным, если значение условного выражения не равно нулю. Если условие выполняется, то и цикл выполняется. Если условие принимает значение ЛОЖЬ, то программа выходит из цикла, и выполняется оператор, следующий за циклом.

Пример. Вывести на экран числа от 0 до 10.

```
#define BEGIN 0
#define MAX 10
int number = BEGIN;
while (number <= MAX) {
  printf("%3d", number);         // вывод на экран
  number++;           // переход к следующему числу
}
```

Пример. Вычислить через сколько минут переполнится дырявая бочка с водой (250 литров), если каждую минуту (кроме шестой) в нее вливается по 3 литра воды, а каждую шестую минуту единовременно вытекает 5 литров.

```
#define MAX_VOLUME 250
int volume = 0, time = 0;
while (volume < MAX_VOLUME) {
  time++;
  if (time % 6 == 0) {
    volume -= 5;
  }
  else {
    volume += 3;
  }
}
printf("Бочка переполнится через %d минут\n", time);
```

Введем макрос для максимального объема бочки и две переменные для текущего объема воды в бочке и времени. Предварительно обнулим переменные. Далее в цикле

будем прибавлять минуты и прибавлять или отнимать литры в зависимости от того, какая идет минута. После выполнения цикла на экран выводится количество минут до переполнения бочки.

Цикл do-while

Общая форма цикла **do-while** имеет вид:

```
do {
    блок операторов;
} while();
```

В отличие от цикла **while**, в котором условие проверяется до выполнения, цикл **do-while** сначала выполняет тело цикла, а затем проверяет условие. Таким образом, цикл **do-while** выполняется по крайней мере один раз, а цикл **while** может не выполниться ни разу.

Пример. На какой раз выпадет число ноль при бросании случайного числа в пределах [0; 1].

```
int count = 0;
double number = 0;
do {
  number = rand() / (double)RAND_MAX;
  count++;
} while (number != 0);
printf("Ноль выпадет на %d раз\n", count);
```

Объявляем одну целую перемененную для подсчета количества бросков и дробную переменную для записи случайного дробного числа от нуля до единицы. В теле цикла **do-while** будет генерироваться случайное число. После генерации числа счетчик количества генераций увеличивается на единицу и проверяется условие цикла. Если условие выполняется, то цикл повторяется.

Цикл **do-while** потенциально может приводить к ошибкам, так как тело цикла в обязательном порядке выполнится хотя бы один раз, даже если условие выполнения цикла окажется неверным. Следует использовать этот вариант циклического оператора с осторожностью и везде, где это возможно, заменять его на цикл **while**

Цикл `for`

Общая форма цикла `for` имеет вид:

```
for (инициализация; условие; приращение) {
    блок операторов;
}
```

- Инициализация — это присваивание начального значения переменной, которая называется параметром цикла.
- Условие представляет собой условное выражение, определяющее, следует ли выполнять в очередной раз тело цикла.
- Оператор приращения осуществляет изменение параметра цикла при каждой итерации.

Эти три оператора обязательно разделяются точкой с запятой. Блок операторов цикла `for` выполняется, если условие принимает значение `ИСТИНА`. Как только условие примет значение `ЛОЖЬ`, то программа выходит из цикла и выполняется оператор, следующий за телом цикла `for`.

Пример. Вывести на экран все целые числа от `0` до `100`.

```
int x = 0;
for (x = 0; x <= 100; x++) {
    printf("%4d", x);
}
```

В этом примере параметр цикла `x` инициализирован числом `0`, а затем перед каждой итерации сравнивается с числом `100`. Пока переменная `x` меньше или равна `100`, вызывается функция `printf()`. После каждого выполнения тела цикла переменная `x` увеличивается на единицу и снова проверяется условие выполнения цикла.

Пример. Вычислить и вывести на экран координаты точек функции $y=7x-5x^2$ при значениях `x [-15; 30]` с шагом `3`. Формат вывода координат на экран [x; y].

```
int x = 0, y = 0;
for (x = -15; x <= 30; x += 3) {
    y = 7 * x - 5 * x * x;
    printf("[%3d; %5d]\n", x, y);
}
```

Параметр цикла изменяется от `-15` до `30`. Тело цикла состоит из вычисления значения переменной `y` и оператора вывода значений переменных `x` и `y` на экран в столбик.

Множество параметров цикла `for`

В качестве параметров для цикла `for` может служить несколько переменных. В этом случае они могут указываться через запятую.

Пример. Сколько потребуется итераций для того, чтобы сумма двух переменных стала больше либо равной `100`. Начальные значения переменных: `1, 0`. Шаги изменения переменных: `-1, +7`.

```
int y = 1, z = 0, iteration = 0;
for (y = 0; y + z < 100; y--, z += 7) {
  iteration++;
  printf("x = %4d\ty = %4d\tz = %4d\n", x, y, z);
}
printf("\nПонадобилось %d итераций\n", iteration);
```

В этом примере можно заметить, что приращение можно выполнять не только с теми переменными, которым задаются в цикле начальные значения. Приращение можно осуществлять с любым количеством любых переменных. Инициализацию также можно проводить с несколькими переменными. Условие выполнения цикла должно быть одно, но оно может быть комбинированным, то есть содержать в себе несколько условных выражений, соединенных логическими операторами (например, `(x > 5) || (y < 7))`.

Кроме того, что в секциях цикла `for` может присутствовать несколько элементов, еще одной его особенностью является возможность полного отсутствия одной, двух или даже всех трех секций.

Вложенные циклы

Тело цикла может содержать в себе различные операторы. Это могут быть операторы присваивания, ветвления, циклические операторы и т.д. Примеры циклов, содержащих условные операторы, были представлены выше. Рассмотрим примеры циклов, которые содержат в себе другие циклы.

Пример. Вывести на экран таблицу умножения.

```
int i = 0, j = 0;
for (i = 1; i <= 9; i++) {
  for (j = 1; j <= 9; j++) {
    printf("%3d", i * j);
  }
  printf("\n");
}
```

На экране получим:

```
  1   2   3   4   5   6   7   8   9
  2   4   6   8  10  12  14  16  18
  3   6   9  12  15  18  21  24  27
  4   8  12  16  20  24  28  32  36
  5  10  15  20  25  30  35  40  45
  6  12  18  24  30  36  42  48  54
  7  14  21  28  35  42  49  56  63
  8  16  24  32  40  48  56  64  72
  9  18  27  36  45  54  63  72  81
```

Внешний цикл отвечает за номер выводимой строки, а внутренний цикл за номер выводимого результата умножения в текущей строке. Номера строк также являются числами, для которых мы составляем таблицу умножения, поэтому они участвуют в вычислении произведения перед его выводом на экран.

Пример. Сгенерировать и вывести на экран случайное количество строчек (`1..25`) из случайного количества (`1..50`) случайных чисел (`-50..50`).

```
int num_str = 1 + rand() % 25, num_numb = 0;
int i = 0, j = 0;
for (i = 1; i <= num_str; i++) {
  num_numb = 1 + rand() % 50;
  for (j = 1; j <= num_numb; j++) {
    printf("%4d", -50 + rand() % 101);
  }
  printf("\n");
}
```

Внешний цикл **for** отвечает за строки, то есть параметр цикла `i` инициализируется со значением `1`, что, в свою очередь, символизирует первую строку. После каждой итерации к параметру будет прибавляться единица до тех пор, пока номер текущей

строки не превысит значение переменной `num_str` (`i <= num_str`), которая содержит значение сгенерированного количества строк. Тело внешнего цикла состоит из трех операторов:

- первый оператор обеспечивает генерацию случайного количества чисел, которые будут сгенерированы для данной строки, и запись этого значения в переменную `num_numb`.
- второй оператор — это внутренний цикл, который отвечает за генерацию и вывод нужного количества случайных чисел на экран. Параметр цикла `j` отвечает за номер выводимого случайного числа. Он изменяется от `1` до величины `num_numb` с шагом `1`. Тело внутреннего цикла состоит из функции **`printf`**`()`, которая обеспечивает вывод на экран случайного числа, которое генерируется непосредственно внутри функции.
- третий оператор — это функция **`printf`**`()`, которая обеспечивает переход на новую строку.

Таким образом, мы имеем три этапа работы тела внешнего цикла:

1. генерация количества чисел в текущей строке
2. генерация и вывод чисел в строку
3. переход на новую строку

Бесконечные циклы

Для создания бесконечных циклов можно использовать любой из трех операторов цикла. Большинство программистов выбирают для этого цикл **`while`**. «Всегда истинное» условие реализуется при указании в условии целого числа, отличного от нуля (чаще всего указывают единицу).

Пример. Вывод на экран некоторого множества случайных чисел.

```
int count = 1 + rand() % 30;
printf("%d случайных чисел будет выведено на экран\n", count);
while (1) {
  printf("%5d", - 500 + rand() % 1000);
  count--;
  if (count == 0) {
    break;
  }
}
```

Самый простой способ создания бесконечного цикла **for** — это оставить пустыми все три секции. Если условие в цикле отсутствует, то предполагается, что оно – ИСТИНА. В оператор можно добавить инициализацию и/или приращение каких-либо параметров цикла, хотя обычно все же используют пустую конструкцию.
Для «контроля над бесконечностью» таких циклов существует оператор прерывания **break**. В случае, если программа встретит слово **break** в обычном или в бесконечном цикле, то она сразу же выйдет из него и продолжит работу, выполняя операторы, следующие за циклом. Неконтролируемые бесконечные циклы очень опасны, они могут привести к утечкам памяти, созданию файлов гигантский размеров, заполнения всего дискового пространства и другим различного рода сбоям.

Пример. Найти первое положительное число, прибавляя к числу `-389` по `5`.

```
int number = -389;
for ( ; ; ) {
    if (number > 0) {
        printf("Первое положительное число – %d\n", number);
        break;
    }
    number += 5;
}
```

Пример. Осуществить ввод символов с клавиатуры до тех пор, пока не будет введен знак «!».

```
char letter;
for ( ; ; ) {
    scanf("%c", &letter);
    if (letter == '!') {
       break;
    }
}
```

Здесь для выхода из бесконечных циклов используется оператор перехода **break**. Этот оператор, как и оператор **continue,** используется для управления циклическими операциями.

4.6.Операторы перехода

Оператор `break`

Этот оператор применяется в двух случаях:

1. для немедленного выхода из цикла;
2. для прекращения работы оператора условия **`switch.`**

Оператор **`break`** прерывает выполнение условного оператора **`switch`** или любого циклического оператора и передает управление следующему за ними оператору.

Пример. Вывести на экран 15 строчек чисел от 1 до 10.

```
for (t = 0; t < 15; t++) {
  count = 1;
  for ( ; ; ) {
    printf("%3d", count);
    count++;
    if (count == 10) {
      break;
    }
  }
}
```

Стоит заметить, что если у вас присутствуют вложенные циклы, то **`break`** прервет работу только одного из них. Для полного выхода из обоих циклов следует использовать два оператора **`break`**.

Если оператор **`break`** присутствует внутри оператора **`switch`**, который вложен в какие-либо циклы, то он относится только к **`switch`**, и выхода из цикла не происходит. Таким образом, оператор **`break`** влияет только на тот цикл или условный оператор **`switch`**, к которому он непосредственно относится.

Оператор `continue`

Оператор **`continue`** прерывает текущую операцию цикла и осуществляет переход к следующей итерации цикла. При этом пропускаются все операторы тела цикла, которые стоят после **`continue`**.

Пример. Осуществить ввод десяти целых чисел с клавиатуры. Вывести на экран только положительные из введенных чисел.

```
int count =  0, number = 0;
while (count < 10) {
  scanf("%d", &number);
  count++;
  if (number <= 0) {
    continue;
  }
  else {
    printf("%d ", number);
  }
}
```

Оператор **continue** используется при реализации сложных алгоритмов.

Оператор goto

Оператор **goto** – это один из операторов перехода. В отличие от остальных он может находиться практически в любом месте программы, а не только в циклах как **break** и **continue**. И переходить с его помощью можно также практически в любое место в пределах текущей функции. Но, несмотря на эти, казалось бы, сильные преимущества, программисты стараются избегать употребления этого оператора именно из-за его “неограниченности”.

Так как переход может осуществляться куда угодно, то не всегда понятно, куда же именно он осуществится. При большом количестве строк кода и множестве переходов **goto** придется сильно постараться, прежде чем понять, как работает программа. В результате чрезмерного использования операторов **goto** программы чрезвычайно плохо читаются. То есть оператор **goto** весьма непопулярен, более того, считается, что в программировании не существует ситуаций, в которых нельзя обойтись без оператора **goto**. Но в некоторых случаях его применение все же уместно. Иногда, при умелом использовании, этот оператор может оказаться весьма полезным, например, если нужно покинуть глубоко вложенные циклы.

Для оператора **goto** всегда необходима метка. Метка – это идентификатор с последующим двоеточием. Метка должна находиться в той же функции, что и оператор **goto**, переход в другую функцию невозможен.

Общая форма оператора **goto** следующая:

```
goto метка;
/* ---------
---------- */
метка:
```

```
/* ---------
---------- */
```

или

```
/* ---------
---------- */
метка:
/* ---------
---------- */
goto метка;
/* ---------
---------- */
```

Метка может находиться как до, так и после оператора **goto**

Оператор return

Оператор **return** используется для выхода из функции. Отнесение его к категории операторов перехода обусловлено тем, что он заставляет программу перейти в точку вызова функции. Оператор **return** может иметь значение, которое при выполнении данного оператора возвращается в качестве значения функции. В функциях типа **void** оператор **return** не используется вообще или используется без значения.

Общая форма оператора **return**:

```
return выражение;
```

4.7.Оператор-выражение

Ранее в пособии было рассмотрено множество различных выражений (фундаментальных элементов языка Си). Стоит еще раз сказать, что любое выражение, которое оканчивается точкой с запятой, является оператором (См. глава 4.1 Одиночный оператор).

5. Функции

Функция – это самостоятельная единица программы, созданная для решения конкретной задачи. Примером могут служить функции стандартной библиотеки (**printf**(), **rand**(), **abs**() и др.). Кроме использования уже готовых функций возможно создание новых функций.

```
тип_возвращаемого_значения имя_функции(список_параметров) {
    тело функции;
    return возвращаемое_значение;
}
```

Каждая функция имеет имя, принимаемые параметры и тип возвращаемого значения.

- Имя для функции следует выбирать в соответствии с ее назначением. Например, если ваша функция что-то суммирует, то лучше выбрать для нее название `summa()`.
- Параметры функции – это необязательный атрибут. В качестве параметров в функцию можно передавать значения переменных, которые требуются для работы функции. Любая функция может, как совсем не принимать параметров, так и принимать их практически неограниченное количество.
- Тип возвращаемого значения соответствует типу значения, которое возвращается в качестве результата работы функции. Если при работе функция не возвращает никакого значения, то за тип возвращаемого значения принимается тип **`void`**. В этом случае у функции отсутствует оператор `return`.

```
void имя_функции(список_параметров) {
    тело функции;
}
```

Объявление функции

Объявление функции сообщает компилятору, что далее в программе будет описана и вызвана объявляемая функция. Функция, как и переменная, должна быть объявлена до своего непосредственного использования. При объявлении следует указать имя функции, тип параметров, которые принимает функции и тип значения, которое функция возвращает.

```
int summa(int, double);
double calculations(double, double, int);
void print_array();
```

Описание функции

Под описанием функции понимается описание действий функции при ее вызове. Описания функций следует располагать последовательно друг за другом после функции main(). Это позволяет лучше ориентироваться в больших программах. При

описании функции параметры функции должны включать не только типы, но и имена переменных. Тело функции описывается в фигурных скобках.

```
тип_возвращаемого_значения имя_функции(список_параметров){
    --------------
    тело функции
    --------------
    return возвращаемое_значение;
}
```

Нижеприведенные функции позволяют находить среднее арифметическое двух целых чисел и наибольшее из двух целых чисел.

```
double average (int a, int b) {
  return (double) (a + b) / 2;
}

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

Возможно использование нескольких операторов **return**.

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

Если функция имеет тип **void** и не возвращает значения, то оператор **return** отсутствует.

```
void print_array(int * arr) {
  int i = 0;
  for (i = 0; i < N; i++) {
    printf("%5d", arr[i]);
  }
}
```

Вызов функции

Функция может быть вызвана из любой другой функции, описанной в программе, даже из самой себя. Если функция в процессе работы вызывает сама себя, то это называется рекурсией. Важной особенностью вызова функции является то, что параметры, передаваемые в функцию, должны быть указаны в правильном порядке в соответствии с их типом так же, как они указаны при объявлении функции. При этом тип параметров указывать не нужно.

Если функция возвращает значение, то результат ее работы может быть присвоен какой-либо переменной или использоваться в выражении.

```
int maximum = max(a, b);
double average = summ1(x, y, a, b) / 4;
printf("%d + %d = %d", a, b, summa(a, b));
```

Если функция не возвращает значения, то ее вызов осуществляется без последующего использования результата. При этом вызов такой функции происходит простой записью имени функции и передачей туда параметров.

```
print_array();
calculations(num1, num2);
```

Связь между объявлением, описанием и вызовом функции

Объявление, описание и вызов функции должны располагаться в определенном порядке:

- объявление – в начале программы, до функции main();
- описание – после функции main();
- вызов – в любом месте программы, но после описания;

```
double count (double, int); //объявление функции

int main(void) {
  double a = 0;
  int b = 0;
  scanf("%lf %d", &a, &b);
  double sum = count(a, b); //вызов функции
  printf("сумма %.2lf и %d равна %.2lf", a, b, sum);
  return EXIT_SUCCESS;
```

```
}

double count (double x, int y) { //описание функции
  return x + y;
}
```

При передаче в функцию происходит копирование значений переменных a и b в переменные x и y. Внутри функции count() переменные a и b не существуют, но вместо них существуют переменные x и y со значениями переменных a и b.

5.1. Работа с функциями

Пример. Задать с клавиатуры длину ребра куба. Вывести на экран объем куба со стороной, равной этому числу. Расчет произвести с помощью функции.

```
int cube_volume(int);

int main(void) {
  int side = 0;
  scanf("%d", &side);
  int volume = cube_volume(side);
  printf("объем куба равен %d", volume);
  return EXIT_SUCCESS;
}

int cube_volume(int a) {
  return pow(a, 3);
}
```

Пример. Задав значения переменных с клавиатуры, вычислить результат с помощью функции. $t = x + \frac{\ln(z-16)}{\cos(x+y)}$

```
double calculations(double, double, double);

int main(void) {
  double a = 0, x = 0, f = 0, t = 0;
  scanf("%lf %lf %lf", &a, &x, &f);
```

```
  if (((f - 16) > 0) && (cos(a + x) != 0)) {
    t = calculations(a, x, f);
    printf("Результат вычислений равен %.4lf", t);
  }
  else {
    printf("Неверные входные данные");
  }
  return EXIT_SUCCESS;
}

double calculations(double x, double y, double z) {
  return x + log(z - 16) / cos(x + y);
}
```

Пример. Задав три целых числа с клавиатуры, с помощью функций определить, какое из них является средним по значению.

```
int compare(int, int, int);

int main(void) {
  int first = 0, second = 0, third = 0;
  scanf("%d %d %d", &first, &second, &third);
  int average = compare(first, second, third);
  printf("Средняя по значению переменная равна %d", average);
  return EXIT_SUCCESS;
}

int compare(int a, int b, int c) {
  int max_1 = (a < b)? a : b;
  int min_1 = (max_1 == a)? b : a;
  int min_2 = (max_1 < c)? max_1 : c;
  return (min_1 < min_2)? min_2 : min_1;
}
```

5.2. Рекурсия

Алгоритм называется рекурсивным, если он вызывает сам себя в качестве вспомогательного. В основе рекурсивных алгоритмов лежит принцип «рекурсивного

определения»: сведения сложного понятия к аналогичному, но менее сложному понятию.

Классический пример рекурсии – это вычисление факториала числа:

0! = 1

1! = 1 = 0! * 1

2! = 1 * 2 = 1! * 2

3! = 1 * 2 * 3 = 2! * 3

4! = 1 * 2 * 3 * 4 = 3! * 4

5! = 1 * 2 * 3 * 4 * 5 = 4! * 5

6! = 1 * 2 * 3 * 4 * 5 * 6 = 5! * 6

---

n! = 1 * 2 * 3 * ... * (n - 2) * (n - 1) * n = (n - 1)! * n

Получив формулу для n!, можно выделить два основных правила его вычисления:

- если `n = 0`, то `n! = 1`;
- если `n > 0`, то `n! = (n - 1)! * n`.

Любое рекурсивное определение состоит из двух частей. Эти части принято называть «базовой» и «рекурсивной».

- **Базовая часть** является нерекурсивной и задает определение для некоторой фиксированной части объектов (если `n = 0`, то `n! = 1`).
- **Рекурсивная часть** определяет понятие через него же и записывается так, чтобы при цепочке повторных применений она сводилась бы к базовой части (если `n > 0`, то `n! = (n - 1)! * n`).

На основе этих двух правил можно выстроить рекурсивную функцию вычисления факториала:

```
int factorial(int n){
  if(n == 0)
    return 1;
  else
    return factorial(n – 1) * n;
}
```

Если представить работу программы графически, то она будет представлять собой «матрешку» (Рисунок 1).

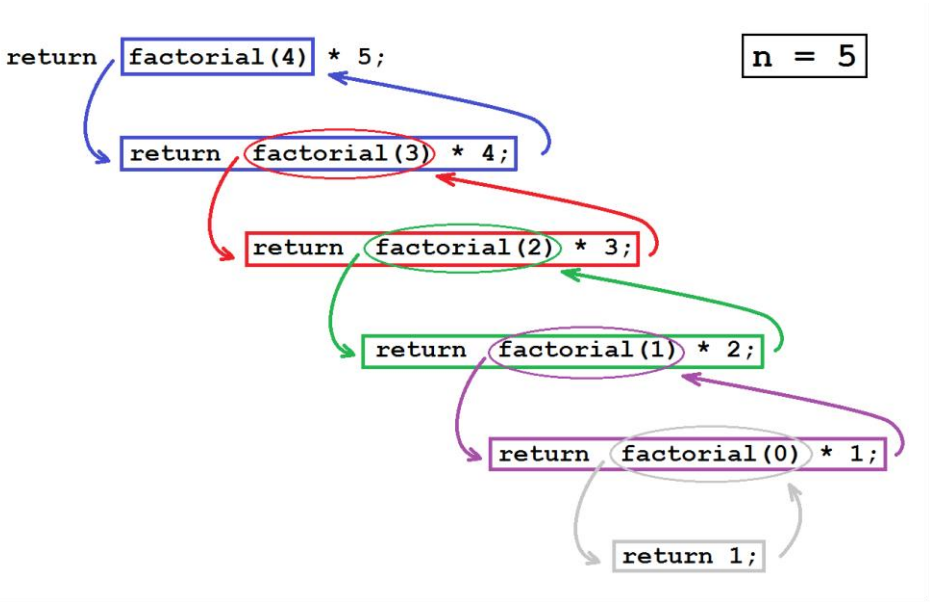


Рисунок 5.1. Графическое представление вычисления факториала.

Спускаясь на уровень вниз, происходит вызов функции `factorial()` с параметром, меньшим на единицу. «Спуск вниз» происходит до тех пор, пока передаваемый параметр не станет равным нулю, что приведет к достижению конца рекурсии и «обратному ходу». До этого момента все вызванные функции существуют одновременно и независимо друг от друга. То есть в каждой из вызванных функций `factorial()` существует своя собственная локальная переменная `n` со своим уникальным значением. Во время «обратного хода рекурсии» функции завершают свою работу в обратном порядке (чем позже была вызвана функция, тем раньше она закончит работу), возвращая результат тем функциям, которые их вызвали. Таким образом, мы возвращаемся к самой первой вызванной функции, и она тоже заканчивает работу, возвращая свой результат.

Рекурсивная функция не может бесконечно вызывать свои клоны с различными параметрами. При каждом таком вызове происходит выделение памяти в стеке и количество вызванных (одновременно находящихся в памяти) функций зависит от величины стека на используемом программном обеспечении. Как правило, стек позволяет произвести около 60000 вызовов функции. При чрезмерно большой глубине рекурсии может произойти переполнение стека вызовов и программный сбой. Поэтому рекомендуется избегать рекурсивных программ с большой глубиной рекурсии.

Для преодоления проблемы с ограниченностью глубины рекурсии имеется специальный тип рекурсии, называемый «хвостовая рекурсия». Этот тип рекурсивных программ позволяет избежать заполнения стека вызванными функциями, так как при таком типе рекурсии вызов функцией самой себя является ее последней операцией.

То есть функция заканчивает свою работу вызовом самой себя. Такая рекурсия при компиляции программы заменяется на итерационный алгоритм.

Таблица 5.1. Пример обычной и хвостовой рекурсии для вычисления факториала числа.

Обычная рекурсия	Хвостовая рекурсия

`int fact (int n){`
`  if(n == 0)`
`  return 1;`
`  else`
`  return fact(n – 1)*n;`
`}`
	`int fact (int n) {`
	`  return fact_r(n, 1);`
	`}`
	
	`int fact_r(int n, int result) {`
	`  if (n == 0)`
	`  return result;`
	`  else`
	`  return fact_r(n - 1, result * n);`
	`}`

Пример. Числа Фибоначчи.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
В этой последовательности каждое последующее число равно сумме двух предыдущих чисел. Достаточно знать первые 2 числа, чтобы вычислить любое другое число.

```
F(0) = 0
F(1) = 1
F(2) = 1 = F(1) + F(0)
F(3) = 2 = F(2) + F(1)
F(4) = 3 = F(3) + F(2)
F(5) = 5 = F(4) + F(3)
F(6) = 8 = F(5) + F(4)
--------------------------
F(n) = F(n - 1) + F(n - 2)
```

Запишем базовую и рекурсивную части:

- базовая: `F(0) = 0`;
- базовая: `F(1) = 1`;
- рекурсивная: `F(n) = F(n - 1) + F(n - 2)`.

Составим рекурсивную функцию:

```
int fibonachchi (int n) {
  if (n == 0)
    return 0;
  else
    if (n == 1)
      return 1;
    else
      return fibonachchi(n - 1) + fibonachchi(n - 2);
}
```

В этом примере каждая функция вызывает не одну свою копию, а две.

Пример. Вывести на экран целые числа от `N` до `K` без использования циклических операций.

Воспользуемся рекурсивным представлением последовательности чисел. Рассмотрим каждую последовательность как первое число из последовательности и всю остальную последовательность.

```
 От 1 до 10   это    1  и   от 2 до 10
 От 2 до 10   это    2  и   от 3 до 10
 От 3 до 10   это    3  и   от 4 до 10
 ---------------------------
 От 8 до 10   это    8  и   от 9 до 10
 От 9 до 10   это    9  и  от 10 до 10
От 10 до 10   это   10
```

Отсюда получаем базовую и рекурсивную части

- базовая: если `n = k`, то печать `k;`
- рекурсивная: если `n < k`, то печать `n` и вызов функции для последовательности `от n - 1 до k`.

```
void print_numbers (int n, int k) {
  if (n == k)
    printf(" %d", k);
  else {
    printf(" %d", n);
    print_numbers(n - 1, k);
  }
}
```

если учесть, что при `k = n`, можно напечатать как `n`, так и `k`, то можно упростить функцию:

```
void print_numbers (int n, int k) {
  printf(" %d", n);
  if (n < k)
    print_numbers(n - 1, k);
  }
}
```

Пример. Определение количества цифр в числе `K`.

Для определения количества цифр в каком-либо числе достаточно последовательно целочисленно делить его `10` до тех пор, пока не получится ноль.

```
K = 82441
82441 /10 = 8244        k = 8244
8244 / 10 = 824         k = 824
824 / 10 = 82           k = 82
82 / 10 = 8             k = 8
8 / 10 = 0              k = 0
```

поделили `5` раз, значит, число содержало `5` цифр.

Составим базовую и рекурсивные части:

- базовая: если `k = 0`, то вывод результата
- рекурсивная: если `k > 0`, целочисленно поделить на `10`, прибавить единицу к счетчику цифр и вызвать функцию заново.

```
void number(int k, int count) {
  if ((k == 0) && (count > 0))
    printf("в числе %d цифр", count);
  else {
    k /= 10;
    count++;
    number(k, count);
  }
}
```

Пример. Подсчет суммы элементов в одномерном массиве.

```
S(0) = 0;
S(1) = array[0] = S(0) + array[0];
S(2) = array[0] + array[1] = S(1) + array[1]
S(3) = array[0] + array[1]+ array[2] = S(2) + array[2]
S(4) = array[0] + array[1]+ array[2] + array[3] = S(3) + array[3]
-----------------------------------------------------------------
S(n) = array[0] + array[1] + ... + array[n-2] + array[n-1] =
= S(n-1) + array[n-1]
```

Составим базовую и рекурсивные части:

- базовая: если `n = 0`, то `sunma = 0`;
- рекурсивная: если `n > 0`, то `summa = S(n-1) + array[n-1]`.

```
int summa(int n) {
  if (n == 0)
    return 0;
  else
    return summa(n - 1) + array[n - 1];
}
```

6. Работа с файлами и директориями

Перед тем как начать изучение файловой системы языка Си, необходимо объяснить, в чем разница между потоками и файлами. В системе ввода/вывода для программ поддерживается единый интерфейс, не зависящий от того, к какому конкретному устройству осуществляется доступ. То есть в этой системе между программой и устройством находится нечто более общее, чем само устройство. Такое обобщенное устройство ввода или вывода (устройство более высокого уровня абстракции) называется потоком, в то время как конкретное устройство называется файлом. Впрочем, файл – тоже понятие абстрактное.

Очень важно понимать, каким образом происходит взаимодействие потоков и файлов. Основная цель разграничения потоков и файлов – это обеспечить единый интерфейс для выполнения всех операций ввода/вывода.

6.1. Потоки

Файловая система языка Си предназначена для работы с самыми разными устройствами, в том числе терминалами, дисководами и даже накопителями на магнитной ленте. Если какое-то устройство сильно отличается от других,

буферизованная файловая система все равно представит его в виде логического устройства, которое называется потоком. Все потоки ведут себя похожим образом. И так как они в основном не зависят от физических устройств, то та же функция, которая выполняет запись в дисковый файл, может ту же операцию выполнять и на другом устройстве, например на консоли.

6.2. Файлы

В языке Си файлом может быть все что угодно, начиная с дискового файла и заканчивая терминалом или принтером. Поток связывают с определенным файлом, выполняя операцию открытия. Как только файл открыт, можно проводить обмен информацией между ним и программой.

Но не у всех файлов одинаковые возможности. Например, к дисковому файлу прямой доступ возможен, в то время как к некоторым принтерам – нет. Таким образом, мы пришли к одному важному принципу, относящемуся к системе ввода/вывода языка Си: все потоки одинаковы, а файлы – нет.

У каждого потока, связанного с файлом, имеется управляющая структура, содержащая информацию о файле, она имеет тип `FILE`. При чтении из файла (или записи в него) каждого символа указатель текущей позиции (курсор) увеличивается, обеспечивая тем самым продвижение по файлу. Файл отсоединяется от определенного потока (т.е. разрывается связь между файлом и потоком) с помощью операции закрытия.

Указатель файла

Ввод/вывод информации с использованием файлов и потоков осуществляется с помощью функций стандартной библиотеки (заголовочный файл `stdio.h`).

Для работы используют переменные типа `FILE *`, которые называются указателями файла. Указатель файла – это указатель на структуру типа `FILE`. Он указывает на структуру, содержащую различные сведения о файле, например: его имя, статус и указатель текущей позиции. В сущности, указатель файла определяет (содержит описание) конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода. Чтобы выполнять в файлах операции чтения и записи, программы должны использовать указатели соответствующих файлов.

```
FILE * file = 0;
```

Теперь переменная `file` будет символизировать файловый поток. Сейчас этот поток пуст.

Открытие файла

Чтобы в файловый поток записать какой-либо файл, нужно открыть этот файл и присвоить потоку его содержимое. За открытие файла отвечает функция **fopen()**. Эта функция открывает поток и связывает с этим потоком определенный файл. Максимальное число одновременно открытых файлов определяется встроенной символической константой FOPEN_MAX. Это значение не меньше 8, но чему оно точно равняется неизвестно — для каждого компилятора свое значение.

```
file = fopen("example.txt","w");
```

Функция принимает два параметра: первый – это название файла (если он лежит в текущем каталоге) или полный путь к файлу, а второй – идентификатор, отвечающий за те действия, которые мы собираемся совершить с этим файлом.

Откроем файл, имя которого задано с клавиатуры:

```
char file_name[250];
scanf("%s", file_name);
file = fopen(file_name,"w");
```

Таблица 9.1. Режимы открытия файла.

Идентификатор	Режим открытия файла
r	Чтение. Курсор в начале файла
r+	Чтение и запись. Курсор в начале файла. Запись накладывается сверху на уже написанный текст.
w, w+	Запись. Курсор в начале файла. Удаление содержимого файла перед записью. Создание файла в случае его отсутствия.
a, a+	Запись (добавление). Курсор в конце файла. Создает файл в случае его отсутствия.

Если файла, который мы пытаемся открыть для чтения, не существует, то функция **fopen()** вернет ноль и файловый поток будет равен нулю.

```
FILE * file = 0;
file = fopen("example.txt","r");
if (file == 0) {
  puts("Такого файла не существует");
}
```

```
else {
  puts ("Файл успешно открыт");
}
```

При работе с файлами для гарантии корректной работы с файлами будем использовать конструкцию:

```
FILE * file;
if ((file = fopen("example.txt","r")) == 0) {
  printf("Ошибка при открытии файла example.txt.\n");
  exit(1);
}
```

При такой записи, если файл не будет обнаружен, программа немедленно завершит свою работу. Завершение работы произойдет при выполнении оператора перехода `exit()`.

Описанный метод помогает при открытии файла обнаружить любую ошибку, например защиту от записи или полный диск, причем обнаружить еще до того, как программа попытается в этот файл что-либо записать. Для гарантии отсутствия ошибок необходимо вначале получить подтверждение, что функция `fopen()` выполнилась успешно, и лишь затем выполнять с файлом необходимые операции.

Закрытие файла

После завершения работы файловым потоком следует его закрыть. За закрытие файловых потоков отвечает функция `fclose()`. Она записывает в файл все данные, которые еще оставались в дисковом буфере и проводит закрытие файла на уровне операционной системы. Так как количество одновременно открытых файлов ограниченно, то иногда приходится закрывать один файл, прежде чем открыть другой.

```
fclose(file);
```

Переименование файла

Чтобы изменить имя файла используется функция `rename()`.

```
rename(старое_имя, новое_имя);
```

В качестве аргументов указываются текущее имя файла и его новое имя. Также возможно указание полного пути к файлу.

```
rename("my_homework.txt", "my_classwork.txt");

char old_name[250] = "green_file.txt";
char new_name[250] = "red_file.txt";
rename(old_name, new_name);
```

Удаление файла

Чтобы удалить файл используется функция **remove()**, аргументом для которой выступает имя удаляемого файла. Стоит обратить внимание, что эта функция использует именно имя файла, а не файловый поток.

```
remove("input.txt");

char file_name[250] = "my_file.txt";
remove(file_name);
```

Возвращение курсора в начало файла

При чтении файла курсор проходит путь от его начала до конца файла. Двигаться курсор может только в одном направлении. Поэтому при возникновении необходимости вернуться в начало файла необходимо либо закрыть файл и открыть его заново, либо воспользоваться функцией `rewind()` которая позволяет переместить курсор из любой точки файла в его начало.

```
rewind(указатель файлового потока)
```

Использование временного файла

При необходимости воспользоваться временным файлом можно с помощью функции `tmpfile()`. Она позволяет создать врменный файл, который будет удален сразу после его закрытия функцией **fclose()**.

```
FILE *temp;
if(!(temp=tmpfile())) {
  printf("Cannot open temporary file.\n");
  exit(1);
}
```

Работа с файлами

В общем случае работа с файлами включает в себя следующие этапы:

1. открытие файла и проверку, что файл открыт;
2. действия над файлом;
3. закрытие файла.

```
FILE * file;
if ((file = fopen("example.txt","r")) == 0) {
    printf("Ошибка при открытии файла example.txt.\n");
    exit(1);
}

// действия
// над
// файлом

fclose(file);
```

6.3.Каталоги

Для создания, удаления и других операций, затрагивающих не файлы, а целые каталоги в языке Си используется стандартная библиотека с заголовочными файлами `<direct.h>` и `<dir.h>`.

Создание каталога

Функция `mkdir()` создает новый каталог, заданный передаваемым ей параметром.

```
mkdir(имя_каталога);
```

Если `имя_каталога` задается как полный путь, то только последняя его компонента должна быть новой, а все предшествующие должны быть существующими каталогами.

```
mkdir("new_folder");
mkdir("C:\\my_folder");
```

Функция `mkdir()` возвращает `0` в случае успеха и `-1` в случае ошибки.

```
int result = mkdir("temp");
if (result == 0) {
  printf("Каталог temp создан\n");
}
```

```
else {
  printf("Не удалось создать каталог temp\n");
}
```

Удаление каталога

Функция `rmdir()` удаляет каталог, заданный передаваемым ей параметром.

```
rmdir(имя_каталога);
```

Каталог `имя_каталога` должен быть пустым и не может быть корневым или текущим рабочим каталогом. В случае указания полного пути к удаляемому каталогу все промежуточные каталоги тоже должны существовать.

```
rmdir("new_folder");
rmdir("C:\\my_folder");
```

Функция `rmdir()` возвращает `0` в случае успеха и `-1` в случае ошибки.

```
int result = rmdir("temp");
if (result == 0) {
  printf("Каталог temp удален\n");
}
else {
  printf("Не могу удалить каталог temp\n");
}
```

Переименование каталога

Для изменения имен каталогов используется такая же функция, что и для переименования файлов – `rename()`.

```
rename(старое_имя, новое_имя);
```

В качестве аргументов указываются текущее имя каталога и его новое имя. Также возможно указание полного пути к каталогу.

```
rename("Ann_folder", "Kate_folder");
```

```
char old_name[250] = "professors_folder";
char new_name[250] = "students_folder";
rename(old_name, new_name);
```

Получение имени текущего каталога

Функция `getcwd()` возвращает полное описание пути к текущему каталогу и записывает его в указанную строку. При вызове функции необходимо указать, в какую строку записать путь и максимальное количество символов, которое можно записать в эту строку.

```
getcwd(строка_для_пути, мах_длина_строки);
```

В случае успеха функция `getcwd()` возвращает строку с полным адресом текущей папки, а в случае ошибки она возвращает `NULL`.

```
char path_name[250];
if (getcwd(path_name, 250) == NULL) {
  printf("Ошибка getcwd");
}
else {
  printf("Текущий каталог - %s", path_name);
}
```

Смена текущего каталога

Это системный вызов, который делает текущим каталог по специфицированному аргументом маршруту `(path)`. Последний должен приводить к существующему каталогу.

```
chdir(новый_текущий_каталог);
```

Функция `chdir()` возвращает `0` в случае успеха и `-1` в случае ошибки.

```
char new_path[250] = "new_directory";
int result = chdir(new_path);
if(result == 0) {
  printf("Текущим стал каталог %s\n", new_path);
}
else {
```

```
    printf("Не могу перейти к каталогу %s\n", new_path);
}
```

7. Массивы

Информация вокруг нас часто собрана в некоторые группы, например числа в календаре или множество букв в текстовом файле. Информация группируется и сортируется для более быстрой обработки, встречаются группы и таблицы данных. Для работы с такими сгруппированными однотипными данными используются массивы.

Массив – это набор однотипных элементов, имеющих одно имя, расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу. Индексация массивов в
языке Си начинается с нуля, и, соответственно, номер элемента и его индекс различны.

Как и другие переменные, массив должен быть объявлен прежде, чем он будет использован в программе. Все элементы массива имеют один и тот же тип, который не может меняться в процессе работы программы. Если вы объявляете массив из целых чисел, то никаким образом нельзя записать туда что-либо другое, кроме целых чисел.

Размер массива – это целая неотрицательная величина, которая соответствует количеству элементов в массиве. Размер массива может задаваться числом, целочисленной константой или макросом. Если размерность массива известна заранее, то рекомендуется использование макроса для обозначения его размера, если размер заранее неизвестен и определяется в процессе работы программы – используется динамическая память. Кроме типа переменных, размер массива также остается постоянным на всем протяжении времени его жизни, поэтому переменные не нельзя указывать в качестве размера массива.

7.1. Одномерные числовые массивы

Объявление одномерного массива:

```
тип имя_массива [количество элементов];
```

```
#define F 120
#define SIZE 45
int array[6];
double a[F], marks[SIZE];
```

```
int positive_numbers[78], negativ_numbers[12];
```

При объявлении массива можно инициализировать его значения.

```
#define N 6
int array_1[N] = {1, 6, 2, 9, -5, 2};
```

Если указано недостаточное количество значений элементов, то оставшиеся элементы заполняются нулями.

```
#define N 9
int array_2[N] = {0, -5};
```

Для того, чтобы обнулить весь массив при объявлении достаточно написать следующее:

```
int array_2[N] = {0};
```

При этом первый элемент массива задается как ноль, а остальные элементы автоматически обнуляются, так как их значения не указаны.

Обращение к элементу массива осуществляется с помощью имени и индекса. Например, вот так можно присвоить значение 3му элементу массива:

```
array[2] = 34;
```

При этом важно понимать, что, так как нумерация элементов начинается с нуля, то 3й элемент массива — это элемент с индексом 2.

Рассмотрим массив из пяти элементов (`N = 5`):

1. `array[0]`
2. `array[1]`
3. `array[2]`
4. `array[3]`
5. `array[4]`

Индекс последнего элемента в массиве – это индекс на единицу меньший, чем размерность массива (`N - 1`).

Объем памяти, необходимый для хранения массива, определяется типом данных, которые хранит массив. Для одномерного массива количество байтов памяти вычисляется следующим образом:

```
количество_байт = (int)sizeof(тип) × длина_массива
```

Функция **sizeof()** возвращает значения типа size_t, поэтому требуется явно преобразовать эти значения в **int**.

Если вывести на экран адреса элементов массива, то можно увидеть, что они расположены друг за другом и отличаются друг от друга на число занимаемых байт памяти (для целочисленного массива это 4 байт).

индекс	0	1	2	3	4
адрес	0xffe60	0xffe64	0xffe68	0xffe6c	0xffe70

Во время компилирования программы компилятор не проверяет ни соблюдение границ массивов, ни содержимое этих массивов. В область памяти, занятую массивом, может быть записано что угодно (даже программный код). Программист должен сам вести проверку границ индексов, так как при нарушении границ массива может происходить разрушение соседних участков памяти.

7.2.Двумерные числовые массивы

Объявление двумерного массива:

```
тип имя_массива [количество строк][количество столбцов];
```

```
#define N 10
#define M 7
int a[N][M];
double mass[N][5], num[40][12];
```

При объявлении двумерного массива указывается количество строк и столбцов. При этом происходит выделение соответствующего количества памяти для хранения двумерного массива.

Подобно одномерным массивам при объявлении двумерного массива можно инициализировать его значения:

```
int massiv[2][3] = {{0, 9, -6}, {-3, 0, -2}};
```

Если при такой инициализации указано недостаточное количество значений элементов, то оставшиеся элементы заполняются нулями. Обнуление двумерного массива:

```
int massiv[N][M] = {{0}};
```

Для работы с двумерными массивами необходимо пользоваться двумя циклами – по одному циклу на каждый индекс. Соответственно, если вы работаете с трехмерным массивом, то циклов будет три.

```
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++) {
    printf("%5d", array[i][j]);
  }
  printf("\n");
}
```

Здесь происходит вывод массива на экран. Внешний цикл отвечает за индекс строки, а внутренний за индекс столбца выводимого на экран элемента. Для каждого значения внешнего цикла (цикл по строкам) выполняются два действия: циклический вывод в строку элементов текущей строки массива и переход на новую строку. Таким образом, мы получаем прямоугольный массив.

Главная диагональ квадратного массива характеризуется равенством индексов строки и столбца `(i == j)`. Соответственно, элементы ниже главной диагонали имеют индекс строки всегда больший, чем индекс столбца `(i > j)`, а элементы выше главной диагонали – индекс столбца всегда больше, чем индекс строки `(i < j)`.

- главная диагональ `(i == j)`
- выше главной диагонали `(i < j)`
- ниже главной диагонали `(i > j)`

Таким же образом, нетрудно заметить, что для побочной диагонали справедливо:

- побочная диагональ `(i + j == N - 1)`
- выше побочной диагонали `(i + j < N - 1)`
- ниже побочной диагонали `(i + j > N - 1)`

7.3.Работа с массивами

Пример. Заполнить одномерный массив из `25ти` элементов целыми числами так, чтобы каждый элемент равнялся квадрату своего индекса. Результат вывести на экран в строку.

```
#define N 25
int i, array[N] = {0};
for (i = 0; i < N; i++) {
```

```
  massiv[i] = i * i;
}
for (i = 0; i < N; i++) {
  printf("%5d", array[i]);
}
```

Цикл **for** осуществляет обращение ко всем элементам массива от первого с индексом 0 до последнего с индексом 24 и каждому элементу присваивает значение `i * i`. После окончания первого цикла, во втором цикле снова происходит обращение ко всем элементам массива, и они один за другим выводятся на экран в строку.

Пример. Заполнить первую половину одномерного массива с клавиатуры, а вторую половину случайными числами от `-50` до `50`.

```
#define N 25
srand(time(0));
int array[N] = {0}, i = 0;
for (i = 0; i < N / 2; i++) {
  scanf("%d", &array[i]);
}
for (i = SIZE / 2; i < N; i++) {
  array[i] = -50 + rand() % 101;
}
```

Пример. Найти сумму и максимальный элемент одномерного массива из дробных чисел, введенных с клавиатуры.

```
#define N 25
double array[N] = {0};
int i = 0;
for (i = 0; i < N; i++) {
  scanf("%lf", &array[i]);
}
double summa = 0.0;
for (i = 0; i < N; i++) {
  summa += array[i];
}
printf("Сумма всех элементов массива %lf\n", summa);
double max = array [0];
for (i = 0; i < N; i++) {
  if (array [0] > max) {
    max = array[i];
```

```
    }
  }
  printf("Максимальный элемент массива %lf\n", max);
```

При нахождении суммы массива нам понадобится переменная, в которую мы будем суммировать элементы массива. Эту переменную обязательно нужно обнулить. После обнуления переменной, в цикле суммируем все элементы массива и выводим результат на экран.

Для нахождения максимального элемента массива нам нужно в цикле обратиться к каждому элементу массива и сравнить его со значением нашего текущего максимума, который изначально нужно инициализировать со значением первого элемента массива. Если значение какого-либо элемента массива больше текущего максимума, то этот максимум становится равным величине этого элемента. Поиск минимума осуществляется аналогично.

Для инициализации минимума или максимума можно использовать любой элемент массива, из той части массива, в которой идет поиск минимума или максимума. Первый элемент массива выбирается для удобства. Это значит, что если бы мы искали максимум во второй половине массива, то за начальное значение можно было выбрать как `max = massiv[N / 2]`.

Пример. Найти среднее значение всех отрицательных элементов целочисленного одномерного массива, значения элементов которого лежат в диапазоне от -41 до 98.

```
#define N 25
srand(time(0));
int array[N] = {0}, i = 0;
for (i = 0; i < N; i++) {
  array[i] = -41 + rand() % 140;
}
int summa_otr = 0, count_otr = 0;
for (i = 0; i < N; i++)
  if (array[i] < 0) {
    summa_otr += array[i];
    count_otr++;
  }
}
double average_otr = (double)summa_otr / count_otr;
printf("Среднее значение = %lf\n", average_otr );
```

Объявляем массив и заполняем его в цикле случайными значениями. Для того чтобы найти среднее значение отрицательных элементов, нам нужно знать сумму и количество этих элементов. Для этого объявляем две переменные `summa_otr` и `summa_otr`, которые будут содержать сумму отрицательных элементов и их количество соответственно. В цикле обращаемся к каждому элементу массива, и если этот элемент отрицательный, то прибавляем его значение к сумме и увеличиваем счетчик отрицательных чисел на единицу. Для хранения среднего значения объявляем дробную переменную `average_otr` и присваиваем ей значение частного, суммы и количества отрицательных чисел. При этом важно явно преобразовать делитель или делимое к дробному виду, чтобы избежать целочисленного деления.

Пример. Найти сумму положительных четных элементов двумерного прямоугольного массива `(N×M)` целых чисел от `-99` до `99`.

```
int summa = 0;
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++) {
    if ((array[i][j] > 0) && (array[i][j] % 2 == 0)) {
      summa += array[i][j];
    }
  }
}
printf("Сумма равна %d\n", summa);
```

Пример. Выяснить, что больше по абсолютному значению: минимальный элемент выше главной диагонали или максимальный элемент ниже побочной диагонали квадратного двумерного массива (N×N).

```
int min = array[0][N - 1], max = array[N - 1][N - 1];
int ind_imax = 0, ind_jmax = 0, ind_imin = 0, ind_jmin = 0;
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++){
    if ((i < j) && (array[i][j] < min)) {
      min = array[i][j];
    }
    if ((i + j > N - 1) && (array[i][j] > max)) {
      max = array[i][j];
    }
  }
}
```

```
if (abs(max) > abs(min)) {
  printf("Максимальный элемент больше");
}
else {
  if(abs(max) < abs(min)) {
    printf("Минимальный элемент больше");
  }
  else {
    printf("Максимальный и минимальный элементы равны");
  }
}
```

Образование новых массивов в процессе работы программы

При необходимости образования новых массивов из уже существующих (для представления некоторой выборки из исходной информации) этот процесс может происходить по двум сценариям:

- заранее известно количество элементов в новом массиве (прямая зависимость от количества элементов в исходном массиве);
- заранее неизвестно количество элементов в новом массиве (в этом случае используется динамическая память).

Случай, когда количество элементов в новом массиве известно заранее, имеет место при задачах разделения одномерных или двумерных массивов на части.

Пример. Для одномерного массива из 16ти элементов образовать 4 новых массива, разделив исходный массив на 4 равные части.

```
#define N 16
int arr[N] = {0}, i = 0;
for(i = 0; i < N; i++) {
 arr[i] = -15 + rand() % 31;
}

int h = 0;
int  new_1[N/4], new_2[N/4], new_3[N/4], new_4[N/4];
for(i = 0, h = 0; i < N / 4; i++) {
 new_1[h] = arr[i];
 new_2[h] = arr[N / 4 + i];
 new_3[h] = arr[2 * N / 4 + i];
 new_4[h] = arr[3 * N / 4 + i];
```

```
    h++;
  }
```

Вторым примером может служить образование одномерного массива на основе двумерного так, чтобы каждый элемент одномерного массива соответствовал строке или столбцу двумерного массива.

Пример. Для двумерного массива дробных элементов найти минимальный элемент в каждой строке и записать найденные значения в массив.

```
  double arr[N][M] = {{0}};
  int i = 0, j = 0;
  // кол-во минимумов == кол-ву строк
  double min_str[N] = {0};

  // начальное значение минимума i-ой строки - значение первого элемента i-ой строки.
  for (i = 0; i < N; i++) {
    min_str[i] = arr[i][0];
    for (j = 0; j < M; j++) {
      if (arr[i][j] < min_str[i]) {
        min_str[i] = arr[i][j];
      }
    }
  }

  for (i = 0; i < N; i++) {
    printf("Мин %dй строки = %8.2lf\n", i + 1, min_str[i]);
  }
```

Для нахождения минимального элемента каждой строки объявим массив `min_str[]`, который будет содержать столько же элементов, сколько строк в двумерном массиве. Для того чтобы выявить минимальные элементы и записать их значения в массив `min_str[]`, используется конструкция с вложенными циклами. Внешний цикл отвечает за индекс строки двумерного массива `arr[]` и одновременно за индекс одномерного массива `min_str[]`. Таким образом, каждый элемент одномерного массива соответствует строке двумерного. Алгоритм поиска минимума аналогичен рассмотренному ранее алгоритму: сначала задаем начальное значение минимума как значение первого элемента из той области, где мы ищем минимум, затем сравниваем все элементы области с минимумом и в случае, если

какой-либо элемент оказывается меньше минимума, то перезаписываем значение минимума. В конце выводим на экран номер строки и соответствующий минимум.

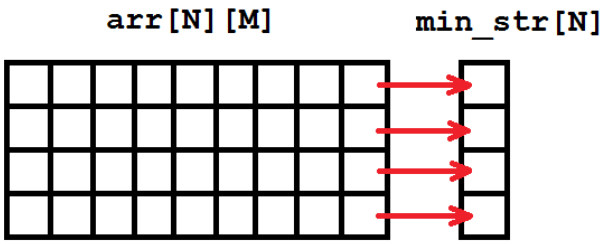


Рисунок 6.1. Получение одномерного массива на основе двумерного. Каждой строке двумерного массива соответствует элемент одномерного.

Пример. Для двумерного прямоугольного массива найти строку, в которой среднее значение четных отрицательных элементов максимально.

```
int arr[N][M] = {{0}}, avr_str[N] = {0};
int sum_str[N] = {0}, count[N] = {0};
for (i = 0; i < N; i++) {
  sum_str[i] = 0;
  for (j = 0; j < M; j++) {
    if ((arr[i][j] < 0) && (arr[i][j] % 2 == 0)) {
      sum_str[i] += arr[i][j];
      count[i]++;
    }
  }
  avr_str[i] = sum_arr[i] / count[i];
}
int max = avr_str[0], index_max = 0;
for (i = 0; i < N; i++) {
  if (avr_str[i] > max) {
    max = avr_str[i];
    index_max = i;
  }
}
printf("Строка = %d ", index_max + 1);
```

Для хранения средних значений в строках будем использовать массив `avr_str[]`. В цикле происходит три этапа расчета среднего значения:

1. изначальное обнуление для последующей записи суммы;
2. суммирование отрицательных четных элементов;

3. деление найденной суммы на количество элементов, входящих в сумму.

После заполнения массива `avr_str[]` значениями находим для него индекс максимального элемента. Это и будет индекс строки с максимальным средним значением.

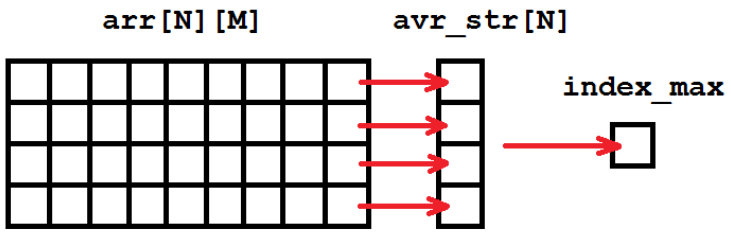


Рисунок 6.2. Получение одномерного массива на основе двумерного. Получение искомого значения на основе одномерного массива.

Задачи с перестановками элементов

Задачи с перестановками — это задачи, в которых какие-либо элементы массива меняются своими значениями.

Рассмотрим обмен значениями двух переменных:

```
int temp = a;
a = b;
b = temp;
```

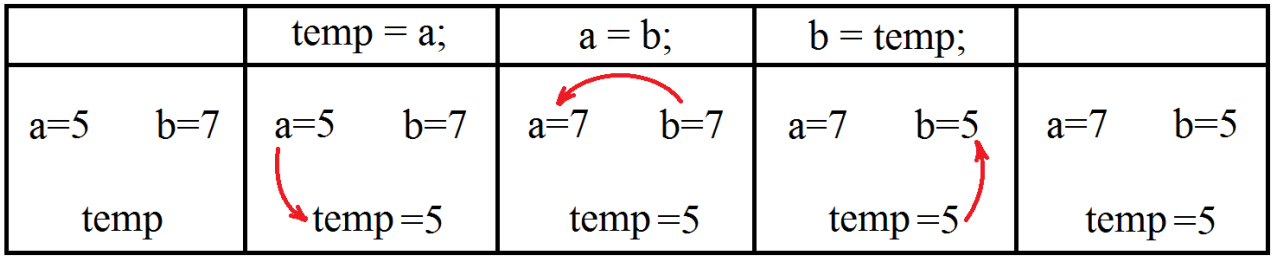


Рисунок 6.3. Иллюстрация обмена значениями двух переменных с использованием промежуточной переменной.

Для решения этой задачи нужно использовать дополнительную переменную, которая позволит временно хранить значение одной из переменных. Если не использовать дополнительную переменную, то уже в первой строчке безвозвратно теряется значение переменой `a`.

```
a = b;
```

```
b = a;
```

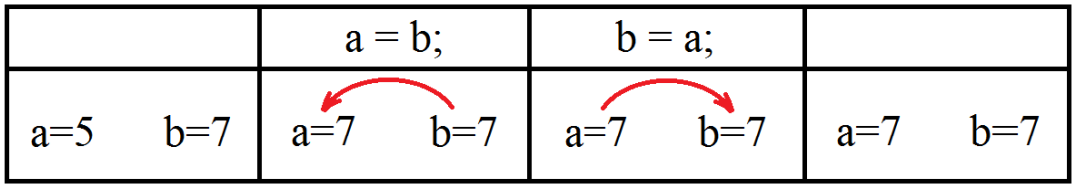


Рисунок 6.4. Иллюстрация обмена значениями двух переменных с потерей значения одной из переменных.

Пример для четырех переменных (`a → b, b → c, c → d, d → a`):

```
int temp = a;
a = d;
d = c;
c = b;
b = temp;
```

При увеличении количества переменных, которые участвуют в обмене значениями, принцип обмена остается таким же (Рисунок 1). Этот принцип сохраняется при любых обменных операциях

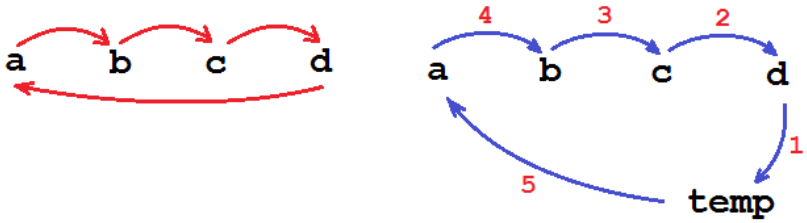


Рисунок 6.5. Принцип обмена переменных значениями

Пример. Поменять местами первую и третью строки массива `arr[N][M]`.

```
for (j = 0; j < M; j++) {
  temp = arr[0][j];
  arr[0][j] = arr[2][j];
  arr[2][j] = temp;
}
```

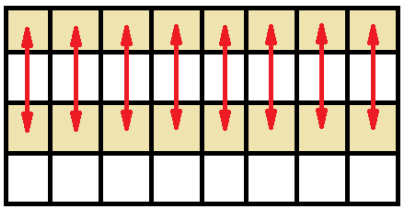


Рисунок 6.6. Иллюстрация обмена первой и третьей строки массива.

Пример. Поменять местами 2ю строку и 4й столбец массива `arr[N][N]`.

```
for (i = 0; i < N; i++) {
  temp = arr[1][i];
  arr[1][i] = arr[i][3];
  arr[i][3] = temp;
}
```

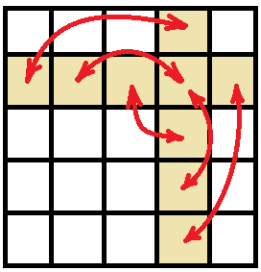


Рисунок 6.7. Иллюстрация обмена строки и столбца двумерного массива.

Для того чтобы произвести обмен значений нужно определить какие элементы необходимо обменять.

```
arr[1][0]    ←→    arr[0][3]
arr[1][1]    ←→    arr[1][3]
arr[1][2]    ←→    arr[2][3]
arr[1][3]    ←→    arr[3][3]
-------------------------------
arr[1][N-1]  ←→  arr[N-1][3]
```

Можно заметить, что индекс столбца в левой колонке меняется так же, как и индекс строки в правой колонке. Поэтому общей формулой можно записать:

```
arr[1][i]    ←→    arr[i][3]
```

Пример. Поменять местами средний столбец и главную диагональ массива `arr[N][N]`.

```
for (i = 0; i < N; i++) {
  temp = arr[i][N / 2];
```

```
        arr[i][N / 2] = arr[i][i];
        arr[i][i] = temp;
    }
```

Здесь:

```
                arr[1][N / 2]     ←→     arr[0][0]
                arr[2][N / 2]     ←→     arr[1][1]
                arr[3][N / 2]     ←→     arr[2][2]
                arr[4][N / 2]     ←→     arr[3][3]
            ----------------------------------------
          arr[N - 1][ N / 2]      ←→   arr[N - 1][ N - 1]
```

Общая формула: `arr[i][N / 2]` ←→ `arr[i][i]`

Пример. Переставить в начало одномерного массива все его положительные элементы. Соблюдать исходный порядок значений не требуется.

```
#define N 25
int arr[N] = {0}, i = 0;
int temp = 0, h = 0;
for (i = 0, h = 0; i < N; i++) {
     if (arr[i] > 0){
          temp = arr[i];
          arr[i] = arr[h];
          arr[h] = temp;
          h++;
     }
}
```

Вводим переменную `h` которая будет отвечать за то, куда мы будем переставлять положительные элементы. Изначально это будет начало массива `(h = 0)`, но после каждой перестановки `h` будет увеличиваться на единицу.

Исходный вид массива:

1	-6	-9	-13	0	2	-8	9	-4	11	5	-3

Конечный вид массива:

1	2	9	11	5	-6	-8	-9	-4	-13	0	-3

Можно заметить, что положительные элементы находятся в начале массива и что они расположены в том же порядке, что в исходном массиве. Однако остальные элементы массива расположены уже в другом порядке. Это произошло по причине того, что при перестановке в начало массива происходил обмен между двумя элементами: положительный элемент менялся с элементом, стоящим в начале массива.

Пример. Переставить в начало одномерного массива все его положительные элементы, так чтобы порядок положительных и неположительных элементов в массиве оставался прежний.

```
#define N 25
int arr[N] = {0}, i = 0;
int temp = 0, h = 0;
for (i = 0, h = 0; i < N; i++) {
  if (arr[i] > 0){
    temp = arr[i];

    for (k = i; k > h; k--) {
      arr[k] = arr[k - 1];
    }
    arr[h] = temp;
    h++;
  }
}
```

Исходный вид массива:

1	-6	-9	-13	0	2	-8	9	-4	11	5	-3

Конечный вид массива:

1	2	9	11	5	-6	-9	-13	0	-8	-4	-3

Здесь неположительные элементы остаются в том же порядке за счет того, что вместо обмена происходит сдвиг всех элементов вправо и перестановка положительного элемента в начало.

```
//Находим положительный элемент
  if (arr[i] > 0){
```

```
//Запоминаем его
    temp = arr[i];

//Сдвигаем на его место элементы слева от него
    for (k = i; k > h; k--) {
      arr[k] = arr[k - 1];
    }

//Ставим элемент на освободившееся слева место в начало массива
    arr[h] = temp;

//Увеличение счетчика перемещенных положительных элементов
    h++;
  }
```

8. Указатели

8.1.Адрес переменных.

У каждой переменной есть место хранения в памяти компьютера, которое характеризуется адресом и размером. Размер памяти, выделяемый под хранение той или иной переменной, зависит от ее типа. Примеры размеров ячеек для хранения некоторых типов переменных (для систем 32-bit):

- `int – 4 байта;`
- `float- 4байта;`
- `double – 8 байт;`
- `char – 1 байт.`

Размер занимаемой памяти для любой переменной или типа переменных можно узнать с помощью функции **`sizeof()`**, передав в нее в качестве параметра объект. При этом функция возвращает значение типа `size_t`, которое необходимо преобразовать к целому типу для корректного вывода на экран или записи в переменную.

```
int size_int = (int)sizeof(int);
int size_double = (int)sizeof(double);
int size_float = (int)sizeof(float);
int h = 0;
int size_h = sizeof(h);
```

Ячейки памяти, в которых хранятся значения переменных, имеют адреса. Чтобы узнать адрес переменных необходимо использовать операцию взятия адреса «&». С помощью функцию **printf()** адрес переменной можно вывести на экран, при этом в строке форматирования адрес переменной указывается как «%p»:

```
int a = 7;
printf("a = %d\n", a);    //значение переменной
printf("address = %p\n", &a);    //адрес переменной
printf("size_a = %d\n", (int)sizeof(a));  // размер в байтах

double b[M] = {0};
printf("b[3] = %lf\n", b[3]);
printf("address = %p\n", &b[3]);
printf("size_b[3] = %d\n", (int)sizeof(b[3]));
```

Указатели

Указатель – это переменная, значение которой содержит адрес другой переменной. Имена указателей принято начинать с буквы «p», а для объявления указателей используют символ «*».

```
тип_указателя * имя_указателя = [значение_указателя];
```

```
int * pa;
double * pvar;
int * pfirst, * parray;
```

Для того чтобы указатель указывал на переменную, в него нужно записать адрес этой переменной. Это можно сделать как при объявлении указателя, так и отдельной операцией:

```
int a = 7;
int * pa_1;

// запись адреса переменной в указатель pa_1
pa_1 = &a;

// запись адреса переменной в указатель pa_2 при объявлении
int * pa_2 = &a;
```

На Рисунке 7.1 представлено схематическое расположение переменной и указателя в памяти компьютера для переменной `a` и указателя на эту переменную `pa`.

```
int a = 7;
int * pa;
pa = &a;
```

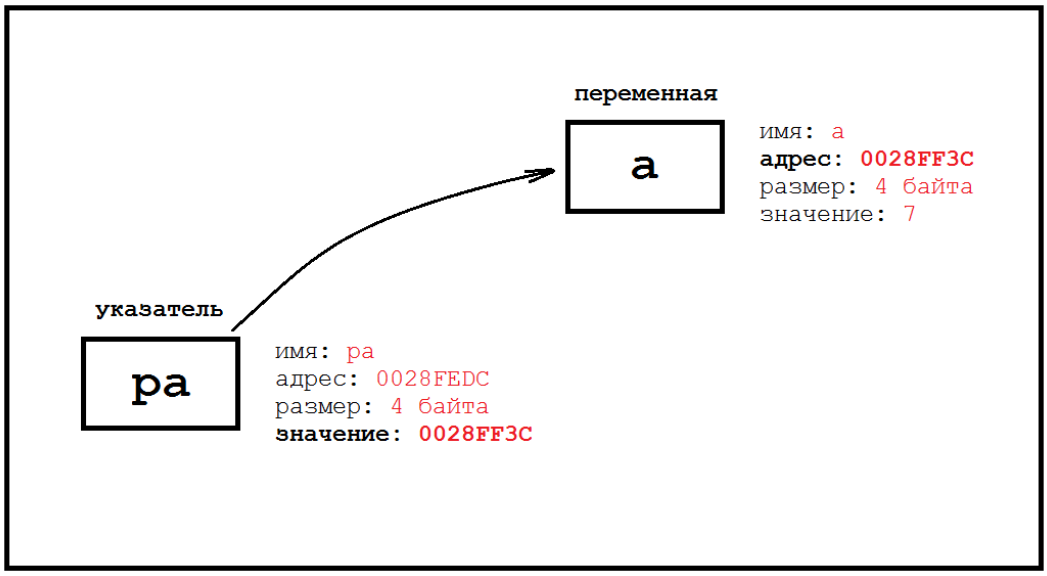


Рисунок 7.1. Схематическое изображение расположения указателя и переменной в памяти. Значение указателя равно адресу переменной.

Указатели предоставляют косвенный способ доступа к переменным через их адрес в памяти компьютера. Для получения такого доступа к переменной используется операция разыменования указателя с помощью символа «*». Например, так можно вывести на экран значение указателя и значение переменной, на которую он указывает:

```
printf("значение указателя = %p", pa);
printf("значение переменной = %d", *pa);
```

Операция разыменования может использоваться для изменения значения переменной, на которую указывает указатель.

```
int a = 7;
int * pa;
pa = &a;
printf("a = %d", a);    //  a = 7

//изменяем значение переменной
//с помощью разыменования указателя
```

```
    *pa = 15;
    printf("a = %d", a);    //  a = 15
```

Если перед указателем поставить знак «*», то мы будем иметь дело не с указателем, а с тем местом, куда он указывает. В нашем случае это переменная `a`.

8.2. Указатели и массивы

Указатели и массивы тесно связаны между собой. Имя любого массива является указателем на первый элемент этого массива. Указателем на каждый последующий элемент массива будет его имя, к которому прибавляется индекс элемента. В памяти компьютера элементы массива расположены следом друг за другом, поэтому, передвигая указатель на позицию вправо (прибавляя единицу) или влево (отнимая единицу), можно обратиться к нужному элементу в массиве. При этом можно заметить, что адреса элементов массива отличаются ровно на то количество байт, которое занимает одна переменная такого же типа как и массив.

Обычная запись	arr[0]	arr[1]	arr[2]	…	arr[N-1]
Запись через указатели	arr	arr + 1	arr + 2	…	arr + N-1
Адреса элементов массива	827280E0	827280E4	827280E8		82728106
Занимаемая память в байтах	4	4	4		4

Чтобы оперировать значениями элементов массива с помощью указателей, используем операцию разыменования. Вывод значений элементов массива с помощью обычной записи и с помощью указателя на элементы массива:

```
//обычный вывод массива на экран
for (i = 0; i < N; i++) {
  printf("%4d", array[i]);
}
//вывод массива с помощью указателей на элементы массива
for (i = 0; i < N; i++) {
```

```
  printf("%4d", *(array + i));
}
//вывод адресов элементов массива с помощью указателей
for (i = 0; i < N; i++) {
  printf("%10p", array + i);
}
```

Указатели и аргументы функций

Передача значений аргументов в функцию может происходить двумя способами: с помощью копирования и через указатели.

Передача значений в функцию через копирование данных в значения аргументов функции рекомендуется использовать для сравнительно небольших объектов, таких как целочисленные и дробные переменные, символы.

Вызов функции	**Описание функции**

`int a = 5, b = 7;`
`int summa = sum(a, b);`
	`int sum(int x, int y) {`
	`  return x + y;`
	`}`

Для массивов, переменных файлового типа, структур и подобных переменных рекомендуется использование указателей при передаче их в функцию. При этом копирование переменной не происходит, а происходит лишь передача информации о местонахождении интересующего объекта в памяти компьютера.

Вызов функции	**Описание функции**

`int array[M] = {0};`
`int summa= sum(array, M);`
	`int sum (int * arr, int size_arr) {`
	`  int summa = 0, i = 0;`
	`  for(i = 0; i < size_arr; i++){`
	`  summa += arr[i];`
	`  }`
	`  return summa;`
	`}`

При передаче массива в функцию используется имя массива и указание размерности массива. Одномерный массив может передаваться двумя способами:

- `int sum (int * array, int n)`
- `int sum (int array[N])`

Оба эти способа обеспечивают передачу одномерного массива в функцию по указателю.

Двумерные и многомерные массивы передаются в функцию единственным способом:

- `double sum (double arr[N][M])`

Пример. С помощью функции вычислить сумму целочисленного одномерного массива из `45ти` случайных чисел в диапазоне `[16; 67]`.

```
#define N 45
int summa(int *, int);
int main(void) {
  int i = 0, arr[N] = {0};
  for (i = 0; i < N; i++) {
    arr[i] = 16 + rand() % (67 - 16 + 1);
  }
  for (i = 0; i < N; i++) {
    printf("%4d", arr[i]);
  }
  printf("\n");
  int sum = summa(arr, N);
  printf("Сумма массива равна %d", sum);
  return EXIT_SUCCESS;
}

int summa(int * array, int size_arr) {
  int i = 0, sum = 0;
  for (i = 0; i < size_arr; i++) {
    sum += array[i];
  }
  return sum;
}
```

Пример. Для двумерного массива, состоящего из `5ти` строк и `9ти` столбцов и заполненного случайными целыми числами от `-55` до `78`, с помощью функций найти и вывести на экран расположения всех элементов равных максимальному.

```
#define N 5
#define M 9
```

```c
int maximum(int [N][M]);
void location(int [N][M], int);
int main(void) {
  int mas[N][M];
  int i = 0, j = 0;
  for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
      mas[i][j] = -55 + rand() % (78 - (-55) + 1);
    }
  }
  for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
      printf("%4d", mas[i][j]);
    }
    printf("\n");
  }
  printf("\n");
  int max = maximum(mas);
  printf("Максимальный элемент массива равен %d\n\n", max);
  location(mas, max);
  return EXIT_SUCCESS;
}

int maximum(int mas[N][M]) {
  int i = 0, j = 0, max = mas[0][0];
  for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
      if (mas[i][j] > max) {
        max = mas[i][j];
      }
    }
  }
  return max;
}

void location(int mas[N][M], int max) {
  int i = 0, j = 0;
  printf("Расположение:\n");
  for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
      if (mas[i][j] == max) {
        printf("%d строка, %d столбец\n", i+1, j+1);
```

```
        }
      }
    }
  }
```

Пример. Для двумерного квадратного массива из 81 элемента, каждый элемент которого равен сумме номера строки и номера столбца, осуществить несколько изменений значений элементов массива и вывести массив на экран после каждого из изменений. Все действия осуществить с помощью функций.

- Умножить каждый четный элемент массива на случайное число в диапазоне от нуля до величины этого элемента.
- Прибавить к соответствующим элементам случайной строки соответствующие элементы случайного столбца.

```
#define N 9
void init_massiv(int [N][N]);
void print_massiv(int [N][N]);
void multiply(int [N][N]);
void add(int [N][N]);
int main(void) {
  srand(time(0));
  int mas[N][N];
  printf("Изначальный массив\n\n");
  init_massiv(mas);
  print_massiv(mas);
  printf("Умножаем четные элементы\n\n");
  multiply(mas);
  print_massiv(mas);
  printf("Прибавляем столбец к строке\n\n");
  add(mas);
  print_massiv(mas);
  return EXIT_SUCCESS;
}

void init_massiv(int mas[N][N]) {
  int i, j;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      mas[i][j] = (i + 1) + (j + 1);
    }
  }
```

```
}
void print_massiv(int mas[N][N]) {
  int i, j;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      printf("%4d", mas[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}
void multiply(int mas[N][N]) {
  int i, j;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      if (mas[i][j] % 2 == 0) {
        mas[i][j] *= rand() % (mas[i][j] + 1);
      }
    }
  }
}
void add(int mas[N][N]) {
  int j;
  int i_rand = rand() % N;
  int j_rand = rand() % N;
  printf("Прибавляем элементы столбца %d к элементам строки
%d\n\n", j_rand + 1, i_rand + 1);

  for (j = 0; j < N; j++) {
    mas[i_rand][j] += mas[j][j_rand];
  }
}
```

8.3.Динамическая память

Переменные, функции, константы записываются и хранятся в памяти компьютера во время выполнения программы. Все эти элементы записываются в стек по мере их появления и удаляются оттуда после окончания своей жизни. Например, во время объявления функции происходит выделение памяти для самой функции и ее параметров. Во время вызова функции все ее переменные по очереди записываются в стек (в том порядке, в котором они объявлены). После окончания работы функции

стек очищается, и все переменные удаляются из него, но уже в обратном порядке. Таким образом, чем позже была объявлена переменная, тем меньше времени она будет существовать в стеке.

Также существует область динамической памяти, которую называют «Куча» (Heap). Если в стеке все объекты располагаются в определенном порядке, то Куча позволяет более свободно работать с памятью. Она позволяет создавать и удалять объекты, освобождать память тогда, когда это требуется. Куча предоставляет возможность удалять переменные и объекты не в строгой последовательности, а по мере необходимости, в нужном программисту порядке.

За выделение и освобождение динамической памяти отвечает стандартная библиотека stdlib.h.

Функция выделения динамической памяти `malloc`()

Функция **`malloc`**() выделяет необходимое количество памяти из Кучи и возвращает указатель на первый байт области памяти, которая была выделена.

```
указатель = (void*)malloc(размер_памяти_в_байтах);
```

- (**`void`***) – это операция приведения типа. При выделении памяти мы заменяем **`void`** на необходимый нам тип.
- размер_памяти_в_байтах находим с помощью функции **`sizeof`**()

```
// выделяем память для одного целого числа
int * p_int = (int*)malloc(sizeof(int));

// выделяем память для пяти целых чисел
int * p_int = (int*)malloc(5 * sizeof(int));

// выделяем память для N целых чисел
#define N 25
int * p_int = (int*)malloc(N * sizeof(int));

//выделяем память для десяти дробных чисел
double * p_double = (double*)malloc(sizeof(double) * 10);
```

Функция освобождения динамической памяти `free`()

Функция **`free`**() освобождает память, на которую указывает указатель, который мы передаем этой функции.

```
free(указатель);
```

```
free(p_int);
free(p_double);
```

Память, которую мы выделяем из Кучи, необходимо обязательно освобождать, так как эта память не очищается после окончания программы. Область динамической памяти очищается только после перезагрузки компьютера, и если ее вовремя не очистить, то это приведет к утечке памяти, замедлению работы компьютера за счет уменьшения свободной оперативной памяти при каждом запуске программы.

Пример. Создать динамический одномерный массив из 50ти элементов. Заполнить этот массив случайными числами и вывести на экран в строку. Графическая иллюстрация представлена на Рисунке 3.

```
//создание указателя
int * arr;
//выделение динамической памяти
arr = (int*)malloc(N * sizeof(int));

//работа с динамическим массивом
for(i = 0; i < N; i++) {
  arr[i] = -15 + rand() % 31;
  printf("%5d", arr[i]);
}

//освобождение динамической памяти
free(arr);
```

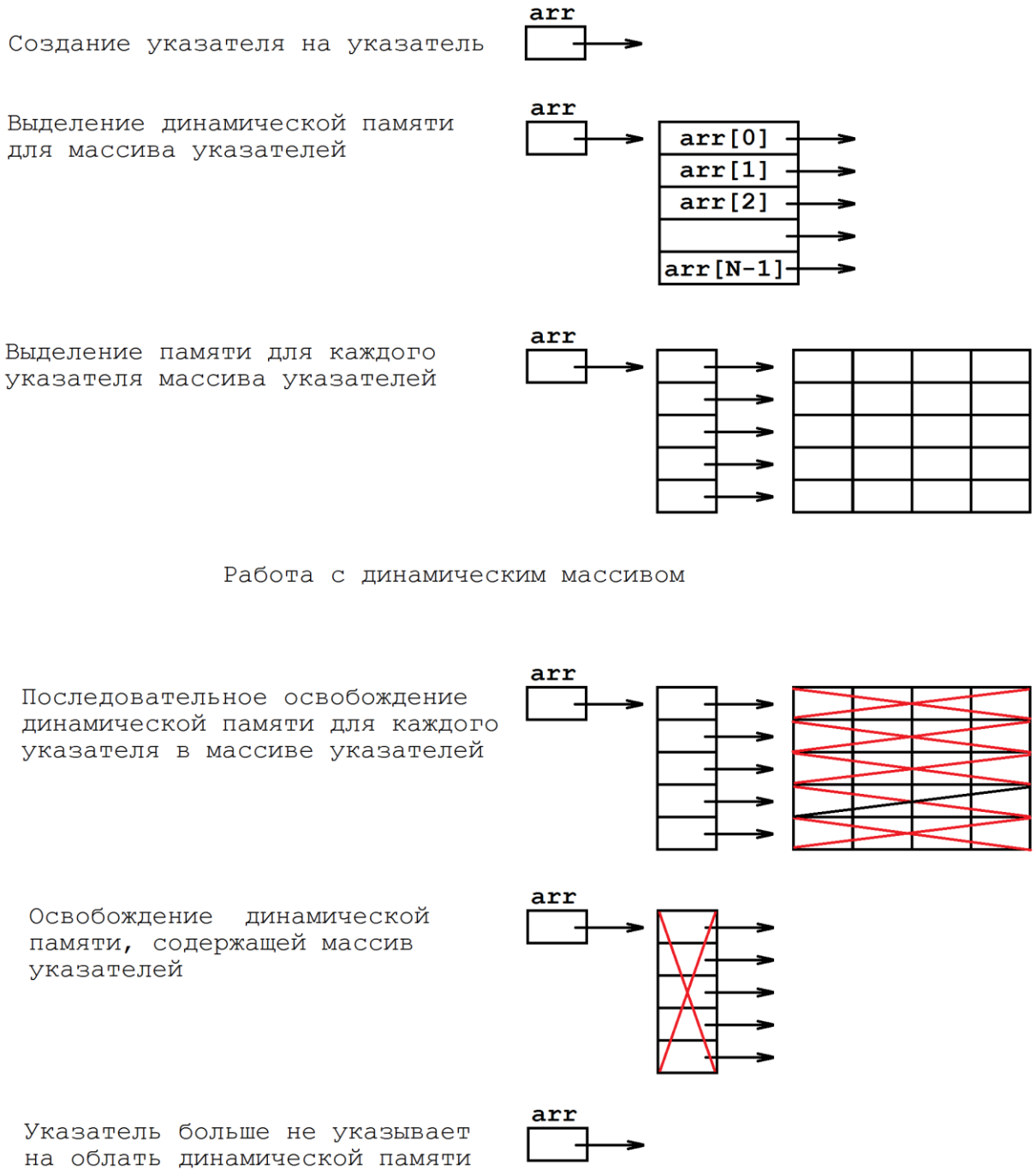
Рисунок 7.1. Жизненный цикл динамического одномерного массива

Пример. Создать динамический двумерный массив из 56ти элементов. Заполнить этот массив случайными числами и вывести на экран в прямоугольном виде. Графическая иллюстрация представлена на Рисунке 4.

```
//создание указателя на указатель
int ** arr;
//выделение динамической памяти для массива указателей
arr = (int**)malloc(N * sizeof(int*));
//выделение памяти для каждого указателя массива указателей
for(i = 0; i < N; i++) {
    arr[i] = (int*)malloc(M * sizeof(int));
}

//работа с динамическим массивом
for(i = 0; i < N; i++) {
  for(j = 0; j < M; j++) {
    arr[i][j] = -15 + rand() % 31;
    printf("%5d", arr[i][j]);
  }
  printf("\n");
}
```

```
    //освобождение памяти для каждого указателя в массиве
указателей
    for(i = 0; i < N; i++) {
        free(arr[i]);
    }
    //освобождение памяти, содержащей массив указателей
    free(arr);
```



Работа с динамическим массивом

Рисунок 7.2. Жизненный цикл динамического двумерного массива

Передача динамического массива в функцию

Одномерные и двумерные динамические массивы передаются в функции только с помощью знака указателя «*».

Для одномерного динамического массива:

```
void init_array(int*, int);
void print_array(int*, int);

int main(void) {
  int i = 0, size_arr = 50;
  int *  arr;
  arr = (int*)malloc(size_arr * sizeof(int));

  init_array(arr, size_arr);
  print_array(arr, size_arr);

  free(arr);
  return EXIT_SUCCESS;
}

void init_array(int * arr, int n) {
  int i = 0;
  for(i = 0; i < n; i++) {
    arr[i] = -15 + rand() % 31;
  }
}

void print_array(int * arr, int n) {
  int i = 0;
  for(i = 0; i < n; i++) {
    printf("%5d", arr[i]);
  }
}
```

Для двумерного динамического массива:

```
void init_array(int **, int, int);
void print_array(int **, int, int);

int main(void) {
  int i = 0, size_n = 4, size_m = 8;
  int ** arr;
```

```
  arr = (int**)malloc(size_n * sizeof(int*));
  for(i = 0; i < size_n; i++) {
    arr[i] = (int*)malloc(size_m * sizeof(int));
  }

  init_array(arr, size_n, size_m);
  print_array(arr, size_n, size_m);

  for(i = 0; i < N; i++) {
    free(arr[i]);
  }

  free(arr);

  return EXIT_SUCCESS;
}

void init_array(int ** arr, int n, int m) {
  int i = 0, j = 0;
  for(i = 0; i < n; i++) {
    for(j = 0; j < m; j++) {
      arr[i][j] = -15 + rand() % 31;
    }
  }
}

void print_array(int ** arr, int n, int m) {
  int i = 0, j = 0;
  for(i = 0; i < n; i++) {
    for(j = 0; j < m; j++) {
      printf("%5d", arr[i][j]);
    }
    printf("\n");
  }
}
```

Пример. В файле `input.txt` на первой строке записано три целых числа: размерность одномерного массива, начало диапазона, конец диапазона значений элементов массива. Прочитать этот файл, создать соответствующий массив и заполнить его случайными значениями из заданного диапазона. Вывести массив на экран.

```
FILE * file;
if((file = fopen("input.txt", "r")) == 0) {
  printf("Файл не найден");
  exit(1);
}
int size = 0, a = 0, b = 0;
fscanf(file, "%d%d%d", &size, &a, &b);
fclose(file);

int * arr;
arr = (int*)malloc(size * sizeof(int));

int i = 0;
for(i = 0; i < size; i++) {
  arr[i] = a + rand() % (b - a + 1);
}
for(i = 0; i < size; i++) {
  printf("%5d", arr[i]);
}
free(arr);
```

Пример. Одномерный целочисленный массив заполнен случайными элементами из диапазона [-34; 87]. Образуйте новый одномерный массив из положительных элементов исходного массива.

```
#define N 70
int i = 0, arr[N] = {0};
for(i = 0; i < N; i++) {
    arr[i] = -34 + rand() % (87 - (-34) + 1);
}

//подсчет количества положительных элементов
int count = 0;
for(i = 0; i < N; i++) {
  if(arr[i] > 0) {
    count++;
  }
}

// выделяем память для нового массива
```

```
int * new_arr;
new_arr = (int*)malloc(count * sizeof(int));

//заполняем новый массив
int h = 0;
for(i = 0, h = 0; i < N; i++) {
  if(arr[i] > 0) {
    new_arr[h] = arr[i];
    h++;
  }
}

//освобождаем память
free(new_arr);
```

Пример. Для одномерного целочисленного массива найти минимальный элемент и на основе его расположения в массиве образовать два новых массива. В первый новый массив войдут элементы слева от минимального элемента исходного массива, а во второй новый массив – элементы справа от минимального элемента исходного массива.

```
#define N 7
int arr[N] = {0}, i = 0;
//находим минимальный элемент и его индекс
int min = arr[0], ind_min = 0;
for(i = 0; i < N; i++) {
  if(arr[i] < min) {
    min = arr[i];
    ind_min = i;
  }
}

//создаем новый динамический массив left_arr
int size_left = ind_min;
int * left_arr;
left_arr = (int*)malloc(size_left * sizeof(int));

//создаем новый динамический массив right_arr
int size_right = N - 1 - ind_min;
```

```
int * right_arr;
right_arr = (int*)malloc(size_right * sizeof(int));

//заполняем новый массив left_arr
for(i = 0; i < size_left; i++) {
  left_arr[i] = arr[i];
}

//заполняем новый массив right_arr
for(i = 0; i < size_right; i++) {
  right_arr[i] = arr[ind_min + 1+ i];
}

//освобождаем память
free(left_arr);
free(right_arr);
```

Пример. Образовать одномерный массив из двумерного массива целых чисел, записав туда все элементы двумерного массива, которые больше среднего значения положительных элементов.

```
int arr[N][M] = {{0}}, sum = 0, count = 0;
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++) {
    if (arr[i][j] > 0) {
      sum += arr[i][j];
      count++;
    }
  }
}
double average = (double)sum / count;

int *  new_arr;
new_arr = (int*)malloc(count * sizeof(int));

h = 0;
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++) {
    if (arr[i][j] > average) {
      new_arr[h] = arr[i][j];
```

```
      h++;
    }
  }
}

free(new_arr);
```

Пример. Для двумерного квадратного массива из 36ти элементов образовать новый одномерный массив из элементов выше главной диагонали. Элементы нового массива должны быть кратны трем.

```
#define N 6
int arr[N][N] = {{0}}, i = 0, j = 0;

// считаем сколько элементов выше главной диагонали кратны трем
int count = 0;
for(i = 0; i < N; i++) {
  for(j = 0; j < N; j++) {
    if((j > i) && (arr[i][j] % 3 == 0)) {
      count++;
    }
  }
}

// выделяем память для нового массива
int * new_arr;
new_arr = (int*)malloc(count * sizeof(int));

// заполняем новый массив
int h = 0;
for(i = 0; i < N; i++) {
  for(j = 0; j < N; j++) {
    if((j > i) && (arr[i][j] % 3 == 0)) {
      new_arr[h] = arr[i][j];
      h++;
    }
  }
}

//выводим на экран новый массив
for(i = 0; i < count; i++) {
```

```
    printf("%5d", new_arr[i]);
}

//освобождаем память
free(new_arr);
```

9. Одномерные и двумерные символьные массивы.

9.1. Строки и массивы строк

Строка - это одномерный массив символов, который заканчивается нулевым символом конца строки. В качестве нулевого символа выступает символ `'\0'`. Таким образом, строка содержит символы, которые ее составляют, и нулевой символ.
При объявлении массива символов, который предназначен для хранения строки, необходимо предусмотреть место для нулевого символа, то есть при его объявлении указать размер этого массива на единицу больше, чем число предполагаемых символов в массиве. Например, объявление массива из 10 символов должно выглядеть следующим образом:

```
#define LEN 10
char name[LEN + 1];
```

При этом величина `LEN + 1` влияет всего лишь на количество выделяемой для него памяти, сам массив может быть меньше этой величины. Таким образом, размер массива при его объявлении – это только максимально возможная длина строки, но, ни в коем случае, не текущий ее размер. Таким образом, в отличие от числовых массивов, длина строкового массива может как бы «изменяться» в пределах его размера.

Рисунок 9.1. Иллюстрация разницы между размером символьного массива и длины строки

При объявлении строки она также может быть сразу инициализирована:

```
#define LEN_1 20
#define LEN_2 29
char str1[LEN_1 + 1] = "строка\n";
char str2[LEN_2 + 1] = {"Test string"};
```

В первом случае выделяется память для 20 символов, а заполняются только семь из них (’\n’ – это один символ). Во втором случае заполняются одиннадцать символов из 29ти выделенных в памяти.
Существует возможность вообще не указывать размер строки при ее объявлении. В этом случае строку нужно инициализировать в обязательном порядке. Размер такой переменной-строки вычисляется, исходя из длины инициализирующей строки.

```
char special_str[] = "Это массив символов";
```

Для данного случая выделится память для хранения 20 символов.

Функции ввода/вывода строк:

- `gets();`
- `puts();`
- `fgets();`
- `fputs();`
- `scanf();`
- `printf();`
- `fscanf();`
- `fprintf();`
- `sscanf();`
- `sprintf()`.

Отметим несколько особенностей функций ввода/вывода строк. Подробнее эти функции рассмотрены в Разделе 11.1.

Функция	**Особенности функции**
`scanf();`	Считывание символов до первого пробела
`fscanf();`	
`gets();`	`'\n'` не дописывается в строку
`fgets();`	`'\n'` дописывается в строку
`puts();`	`'\n'` добавляется после вывода
`fputs();`	`'\n'` не добавляется после вывода

Массив строк – это список из нескольких строк. Например, текстовый файл можно рассматривать как набор отдельных строк, каждую из которых можно записать в массив строк, который будет отображать содержимое текстового файла.

к	о	р	з	и	н	а	\0			
с	\0									
б	у	б	л	и	к	а	м	и	\0	

Рисунок 9.2. Массив строк с одинаковой максимальной длиной

я	г	о	д	н	о	е		п	ю	р	е	\0							
м	о	л	о	ч	н	ы	й		ш	о	к	о	л	а	д	\0			
и	р	и	с	к	и		о	р	е	х	о	в	ы	е	\0				
д	ж	е	м	\0															

Рисунок 9.3. Массив строк с различной максимальной длиной каждой строки

Объявление массивов строк происходит аналогично двумерным числовым массивам. Индекс слева определяет количество строк, а правый – максимальное количество символов в строке. Объявим массив из `25ти` строк с максимальной длиной каждой строки `48` символов:

```
#define LEN_ARR 25
#define LEN_STR 48
char str_array[LEN_ARR][LEN_STR];
```

Так как слово – это тоже строка, то прочитав текстовый файл, можно образовать трехмерный символьный массив таким образом, что первое измерение будет отвечать за номер строки в файле, второе – за номер слова в строке, а третье – за номер символа в слове.

```
#define NUM_STR 15 //количество строк в файле
#define NUM_WRD 10 //количество слов в строке
#define LEN_WRD 100 // длина слова

char words_array[NUM_STR][NUM_WRD][LEN_WRD];
```

В этом примере используется допущение, что в каждой строке одинаковое количество слов. Этого можно избежать при применении динамической памяти.

```
#define NUM_STR 10
int num_wrds[NUM_STR] = {3, 5, 1, 8, 9, 11, 3, 3, 2, 7};
```

```
#define LEN_WRD 120

char ** word_list;
word_list = (char**)malloc(NUM_STR * sizeof(char*));
int i = 0;
for(i = 0; i < NUM_STR; i++) {
  word_list[i] =
              (char*)malloc(num_wrds[i]* sizeof(char*));
}
int j = 0;
for(i = 0; i < NUM_STR; i++) {
  for(j = 0; j < num_wrds[i]; j++) {
    word_list[i][j] =
            (char*)malloc(LEN_WRD * sizeof(char[LEN_WRD]));
  }
}
```

Здесь массив `num_wrds` соответствует количеству слов в строке `i + 1`, для примера массив задан явно при объявлении.

9.2. Функции стандартной библиотеки string.h

Для упрощения работы со строками в языке Си определено множество различных функций, которые содержатся в заголовочном файле библиотеки `string.h` ниже представлены несколько из них.

Функция	**Описание**
`strlen(str)`	Определение длины строки `str`
`strcmp(str_1, str_2)`	Сравнение строки `str_1` и строки `str_2`
`strcat(str_1, str_2)`	Дописать строку `str_2` в строку `str_1`
`strcpy(str_1, str_2)`	Копирование строки `str_2` в строку `str_1`
`strchr(str, symbol)`	Возвращает первую позицию символа symbol в строке str (поиск с начала строки)
`strrchr(str, symbol)`	Возвращает первую позицию символа `symbol` в строке `str` (поиск с конца строки)
`strpbrk(str_1, str_2)`	Возвращает первую позицию любого из символов строки `str_2` в строке `str_1` (поиск с начала строки)
`strstr(str_1, str_2)`	Возвращает первую позицию вхождения строки `str_2` в строку `str_1` (поиск с начала строки)

9.3.Задачи с использованием строк и массивов строк

Пример. Вывести строку на экран, вставляя пробелы между ее буквами.

```
#define LEN 250
char sample_string[LEN];
gets(sample_string);
int i = 0;
for(i = 0; i < strlen(sample_string); i++) {
  printf("%c ", sample_string[i]);
}
```

Пример. Определить совпадают ли два введенных с клавиатуры адреса.

```
#define LEN 25
char address_Ann[LEN], address_Mary[LEN];
gets(address_Ann);
gets(address_Mary);
if (!strcmp(address_Ann, address_Mary))
    printf("Адреса совпадают");
else
    printf("Это разные адреса");
```

Так как функция **strcmp()** возвращает ноль, если строки одинаковы, то для выполнения условия ставим отрицание. Таким образом, если строки равны, то значение функции **strcmp()** ноль будет инвертирован в единицу и условие выполнится.

Пример. Найти длину строки без использования стандартной функции

```
#define LEN 250
char sample_string[LEN];
gets(sample_string);
int i = 0, length = 0;
while(sample_string[i] != '\0') {
  length++;
  i++;
}
printf("%d", length);
```

Пример. Разделить введенную с клавиатуры строку на две строки.

```
#define LEN 250
```

```
char sample_string[LEN], new_str_1[LEN], new_str_2[LEN];
gets(sample_string);
int i = 0, k = 0;

for(i = 0, k = 0; i <strlen(sample_string) / 2; i++, k++) {
  new_str_1[i] = sample_string[i];
}
new_str_1[k] = '\0';


for(i = strlen(sample_string) / 2, k = 0;
                          i < strlen(sample_string); i++, k++) {
 new_str_1[k] = sample_string[i];
}
new_str_1[k] = '\0';
```

Пример. Определить, какое место в стоке является первым вхождением в строку заданного символа.

```
#define LEN 50
char string_1[LEN + 1], letter = 0;
gets(string_1);
letter = getchar();
char * position = strchr(string_1, letter);
if(position == 0) {
  printf("Такого символа нет в строке\n");
}
else {
  printf("Символ найден в строке\n");
  int i = 0, ind = 0;
  for(i = 0; i < strlen(string_1); i++) {
    if ((string_1 + i) == position) {
      ind = i;
      break;
    }
  }
  printf("Указатель буквы %c = %p\n", letter, position);
  printf("Позиция символа %c в строке = %d", letter, ind + 1);
}
```

Пример. Удалить из строки заданную подстроку.

```
#define LEN 250
char smpl_string[LEN], str_to_delit[LEN];
gets(smpl_string);
gets(str_to_delit);

char * position = strstr(smpl_string, str_to_delit);
int i = 0, ind = 0;
for(i = 0; i < strlen(smpl_string); i++) {
  if ((smpl_string + i) == position) {
    ind = i;
    break;
  }
}

int k = 0;
for(i = ind, k = ind + strlen(str_to_delit);
                          k < strlen(smpl_string); i++, k++) {
  smpl_string[i] = smpl_string[k];
}
smpl_string[strlen(smpl_string) - strlen(str_to_delit)] = '\0';
puts(sample_string);
```

Пример. Вводить строки с клавиатуры и записывать их в массив строк `str_arr[10][50]` только тогда, когда длина введенной строки превышает `15` символов.

```
#define LEN_ARR 10
#define LEN_STR 50
char str_arr[LEN_ARR][LEN_STR];
for(i = 0; i < LEN_ARR;  ) {
  gets(str_arr[i]);
  if (strlen(str_arr[i]) > 15) {
    i++;
  }
}
```

Пример. В массиве из `50` строк, найти и вывести на экран самые короткие из них.

```
#define LEN_ARR 50
#define LEN_STR 200
char str_arr[LEN_ARR][LEN_STR];

int i = 0;
```

```
for(i = 0; i < LEN_ARR; i++) {
  gets(str_arr[i]);
}

//начальное значение минимальной длины - длина первой строки
int min_len = strlen(str_arr[0]);

// находим минимальную длину строки
for(i  = 0; i < LEN_ARR; i++) {
  if (strlen(str_arr[i]) > min_len) {
    min_len = strlen(str_arr[i]);
  }
}
//выводим на экран все строки, длина которых равна минимальной
for(i = 0; i < LEN_ARR; i++) {
  if (strlen(str_arr[i]) == min_len) {
    puts(str_arr[i]);
  }
}
```

Пример. Отсортировать массив строк по возрастанию длины строк.

```
#define LEN 250
#define LEN_ARR 25
char string_array[LEN_ARR][LEN], temp_str[LEN];
int i = 0;
for(i = 0; i < LEN_ARR; i++) {
 gets(string_array[i]);
}

// численный массив с длинами каждой строки
int  str_lenght[LEN_ARR];
for(i = 0; i < LEN_ARR; i++) {
  str_lenght[i] = strlen(string_array[i]);
}

// сортировка массива строк одновременно с состриовкой массива
длин строк
int k = 1, tmp = 0;
while(k > 0) {
  k = 0;
  for(i = 0; i < LEN_ARR - 1; i++) {
    if(str_lenght[i] > str_lenght[i + 1]) {
```

```
            k++;
            tmp = str_lenght[i];
            str_lenght[i] = str_lenght[i + 1];
            str_lenght[i + 1] = tmp;

            strcpy(temp_str, string_array[i]);
            strcpy(string_array[i], string_array[i + 1]);
            strcpy(string_array[i + 1], temp_str);
        }
    }
}
```

9.4.Работа с текстовыми файлами

Работа с текстовыми файлами всегда начинается с чтения исходного текстового файла.

Чтение текстового файла	
Чтение до определенного момента	**Чтение до конца файла**
Чтение `N` символов	Чтение всех символов
Чтение `N` слов	Чтение всех слов
Чтение `N` строк	Чтение всех строк

Пример. Прочитать из файла `text_file.txt` и вывести на экран `7` символов, затем `15` слов и еще `9` строк.

```
#define LEN_STR 250
#define LEN_WRD 250
FILE *file;
char symbol = 0, str_wrd[LEN_WRD], str_array[LEN_STR];

file = fopen("text_file.txt", "r");
int i = 0;

// чтение файла по символам
for(i = 0; i < 7; i++) {
 symbol = fgetc(file);
}

// чтение файла по словам
for(i = 0; i < 15; i++) {
```

```
 fscanf(file, "%s", str_wrd);
}

// чтение файла по строкам
for(i = 0; i < 9; i++) {
 fgets(str_array, LEN_STR, file);
}

fclose(file);
```

Символ конца файла EOF

При достижении конца файла функция **fgetc()** возвращает символ EOF. Этот символ означает конец файла (End Of File). В численном значении символ EOF равен минус единице. Поскольку у каждого символа есть свой код, и этот код является положительным числом, то минус единица не может быть кодом какого-либо символа.

Использование символа EOF может быть удобно только при посимвольном чтении файла. Если файл читается с помощью **fgets()** или **fscanf()**, то для определения достижения конца файла удобнее использовать функцию **feof()**. Эта функция универсальна и может использоваться при любом способе чтения файла (в том числе при чтении по одному символу).

Главное отличие **feof()** и EOF состоит в том, что при использовании EOF программа проверяет равенство какой-либо из символьных переменных значению конца файла. При использовании функции **feof()** программа проверяет, не является ли последний прочитанный символ равным значению конца файла.

Пример. Вывести на экран все содержимое файла по одному символу.

```
char letter = 0;
// пока символ не равен символу конца файла
while ((letter = fgetc(file)) != EOF) {
    putchar(letter);
}
fclose(file);
```

Пример. Вывести на экран все строки из файла, пропуская между ними пустую строку.

```
#define LEN_STR 200
char str[LEN_STR];
```

```
// пока последний прочитанный символ не равен концу файла
while (!feof(file)) {
  fgets(str, LEN_STR, file);
  puts(str);
}
fclose(file);
```

Здесь функция `fgets()` считывает из файла строку вместе с символом `'\n'` и при записи в строковую переменную добавляет символ конца строки `'\0'`. Таким образом, при выводе такой строки курсор будет в любом случае переходить на новую строку. Если же выводить ее на экран с помощью функции `puts()`, то переход на новую строку будет двойным.

Все действия с файлами происходят с помощью:

- действия с использованием только исходного файла;
- используется массив строк;
- используется вспомогательный файл.

Использование только исходного файла

Пример. Прочитав файл, определить, сколько слов в этом файле начинается со строчной или прописной буквы «а».

```
#define LEN_STR 50
char word[LEN_STR];
int kol_slov = 0;
while (!feof(file)) {
  fscanf(file, "%s", word); //считываем слово
  if ((word[0] == 'a') || (word[0] == 'A'))
    kol_slov++;
}
printf("Количество слов, начинающихся
                          с буквы\'a\' равно %d\n", kol_slov);
fclose(file);
```

Пример. Прочитав файл, определить, сколько в этом файле слов длиннее 5ти символов и вывести на экран все остальные слова.

```
#define LEN_STR 50
char word[LEN_STR];
int kol_slov = 0;
while (!feof(file)) {
  fscanf(file, "%s", word);
  if (strlen(word) > 5)
```

```
      kol_slov++;
    else
      printf("%s ", word);
  }
  printf("Кол-во слов = %d\n\n", kol_slov);
  fclose(file);
```

Пример. Прочитать из файла 7 строк и записать их в массив строк. Вывести массив строк на экран.

```
#define LEN_ARR 7
#define LEN_STR 50
char str_arr[LEN_ARR][LEN_STR], last_symbol = 0;
int i = 0, len = 0;
FILE * file;
file = fopen("input.txt", "r");
for(i = 0; i < LEN_ARR; i++) {
  //чтение строки
  fgets(str_arr[i], LEN_STR, file);

  //находим чему равен последний символ в строке
  len = strlen(str_arr[i]);
  last_symbol = str_arr[i][len - 1];

  // если последний символ = '\n' меняем его на '\0'
  if (last_symbol == '\n')
    str_arr[i][len - 1] = '\0';
}
fclose(file);

//вывод массива строк на экран
for(i = 0; i < LEN_ARR; i++) {
  puts(str_arr[i]);
}
```

При чтении файла функцией `fgets()` в конце каждой строки (кроме последней строки) записывается символ перехода на новую строку. Передвигая символ конца строки на место символа переноса строки, мы избавляемся от этого нежелательного явления.

Использование массива строк

Пример. Прочитав исходный текстовый файл, отсортировать его строки по убыванию длины.

```
#define LEN_STR 250
FILE * file;
char temp_str[LEN_STR];
int num_str = 0;
//подсчет количества строк в файле
file = fopen("input.txt", "r");
while(!feof(file)) {
  fgets(temp_str, LEN_STR, file);
  num_str++;
}
fclose(file);

//выделение динамической памяти для массива строк
char ** str_array;
str_array = (char**)malloc(num_str * sizeof(char*));

int i = 0;
for(i = 0; i < num_str; i++) {
  str_array[i] = (char*)malloc(LEN_STR * sizeof(char));
}

//заполнение массива строк, символ "\n" удаляем из конца строки
file = fopen("input.txt", "r");
char last_symbol = 0;
for(i = 0; i < num_str; i++) {
  fgets(str_array[i], LEN_STR, file);
  last_symbol = str_array[i][strlen(str_array[i]) - 1];
  if(last_symbol == '\n') {
    str_array[i][strlen(str_array[i]) - 1] = '\0';
  }
}
fclose(file);

// численный массив с длинами каждой строки
int * str_lenght;
str_lenght = (int*)malloc(num_str * sizeof(int));

for(i = 0; i < num_str; i++) {
  str_lenght[i] = strlen(str_array[i]);
}

//сортировка двух массивов методом пузырька
```

```
int k = 1, tmp = 0;
while(k > 0) {
  k = 0;
  for(i = 0; i < num_str - 1; i++) {
    if(str_lenght[i] < str_lenght[i + 1]) {
      k++;
      tmp = str_lenght[i];
      str_lenght[i] = str_lenght[i + 1];
      str_lenght[i + 1] = tmp;

      strcpy(temp_str, str_array[i]);
      strcpy(str_array[i], str_array[i + 1]);
      strcpy(str_array[i + 1], temp_str);
    }
  }
}

free(str_lenght);

//вывод отсортированного массива строк в исходный файл
file = fopen("input.txt", "w");
for(i = 0; i < num_str; i++) {
  fprintf(file, "%s\n", str_array[i]);
}
fclose(file);

for(i = 0; i <num_str; i++) {
  free(str_array[i]);
}
free(str_array);
```

Пример. Прочитав исходный текстовый файл, образовать новый текстовый файл из неповторяющихся слов первого файла.

```
#define LEN_STR 250
char temp_wrd[LEN_STR];
int num_wrd = 0;

FILE * file;
file = fopen("input.txt", "r");
//считаем количество слов в исходном файле
while(!feof(file)) {
```

```
      fscanf(file, "%s", temp_wrd);
      num_wrd++;
    }
    fclose(file);

    //выделяем динамическую память для хранения массива слов
    char ** wrd_array;
    wrd_array = (char**)malloc(num_wrd * sizeof(char*));

    int i = 0;
    for(i = 0; i < num_wrd; i++) {
      wrd_array[i] = (char*)malloc(LEN_STR * sizeof(char));
    }

    //заполняем массив слов
    file = fopen("input.txt", "r");
    for(i = 0; i < num_wrd; i++) {
      fscanf(file, "%s", wrd_array[i]);
    }
    fclose(file);

    //открываем итоговый файл для записи
    file = fopen("output", "w");
    int k = 0, count = 0;

    for (k = 0; k < num_wrd; k++) {

    //считаем сколько раз слово встечается в массиве слов
      count = 0;
      for(i = 0; i < num_wrd; i++) {
        if(!strcmp(wrd_array[i], wrd_array[k])) {
          count++;
        }
      }
    //если слово встретилось только один раз, значит, оно не
повторяется
      if (count == 1) {
        fprintf(file, "%s ", wrd_array[k]);
      }
    }
    fclose(file);
```

```
    for(i = 0; i <num_wrd; i++) {
      free(wrd_array[i]);
    }
    free(wrd_array);
```

Использование промежуточного файла

Пример. Определить сколько слов в каждой строке.

```
    #define LEN_STR 250
    FILE *file, *tmp_file;
    char str_array[LEN_STR], tmp_wrd[LEN_STR];
    int count = 0;
    //открываем исходный файл для чтения
    file = fopen("input.txt", "r");
    int num_str = 0;
    //читаем одну строку из исходного файла и записываем ее во
временный файл
    while(!feof(file)) {
      fgets(str_array, LEN_STR, file);
      num_str++;

    //запись строки во врменный файл
      tmp_file = fopen("tmp.txt", "w");
      fputs(str_array, tmp_file);
      fclose(tmp_file);

    //читаем промежуточный файл по словам и считаем их количество
      tmp_file = fopen("tmp.txt", "r");
      count = 0;
      while(!feof(tmp_file)) {
        fscanf(tmp_file, "%s", tmp_wrd);
        count++;
      }
      fclose(tmp_file);

      printf("В %dй строке %d слов\n", num_str, count);
    }
    fclose(file);
```

Пример. На основе исходного файла создать новый файл таким образом, чтобы в каждой строке слова располагались в обратном порядке.

```
    #define LEN_STR 250
```

```
FILE *file, *tmp_file, *out_file;
char str_array[LEN_STR], tmp_wrd[LEN_STR];
int count = 0;
//открываем исходный файл для чтения, итоговый файл для записи
file = fopen("input.txt", "r");
out_file = fopen("outtput.txt", "w");

char last_symbol = 0;
//читаем исходный файл до конца файла
while(!feof(file)) {
//читаем строку и удаляем в ее конце символ "\n"
  fgets(str_array, LEN_STR, file);
  last_symbol = str_array[strlen(str_array) - 1];
  if(last_symbol == '\n') {
    str_array[strlen(str_array) - 1] = '\0';
  }
  //записываем строку в промежуточный файл
  tmp_file = fopen("tmp.txt", "w");
  fputs(str_array, tmp_file);
  fclose(tmp_file);

//читаем промежуточный файл по словам и считаем их количество
  tmp_file = fopen("tmp.txt", "r");
  count = 0;
  while(!feof(tmp_file)) {
    fscanf(tmp_file, "%s", tmp_wrd);
    count++;
  }
  fclose(tmp_file);

//объявляем динамический массив строк для записи слов
  char ** wrd_array;
  wrd_array = (char**)malloc(count * sizeof(char*));

  int i = 0;
  for(i = 0; i < count; i++) {
    wrd_array[i] = (char*)malloc(LEN_STR * sizeof(char));
  }

//еще раз читаем файл и заполняем массив строк словами
  tmp_file = fopen("tmp.txt", "r");
  for(i = 0; i < count; i++) {
```

```
    fscanf(tmp_file, "%s", wrd_array[i]);
  }
  fclose(tmp_file);

//делаем обратную перестановку в массиве строк
  for(i = 0; i < count / 2; i++) {
    strcpy(tmp_wrd, wrd_array[i]);
    strcpy(wrd_array[i], wrd_array[count - 1 - i]);
    strcpy(wrd_array[count - 1 - i], tmp_wrd);
  }
//выводим инвертированный массив строк в итоговый файл
  for(i = 0; i < count; i++) {
    fprintf(out_file, "%s ", wrd_array[i]);
  }
  fprintf(out_file, "\n");

  for(i = 0; i <count; i++) {
    free(wrd_array[i]);
  }
  free(wrd_array);
}

fclose(out_file);
fclose(file);
```

Пример. Определить в какой из самых длинных строк меньше всего слов.

```
#define LEN_STR 250
FILE *file, *tmp_file;
char str_array[LEN_STR], tmp_wrd[LEN_STR];
char last_symbol = 0;
int max_length = 0;

//определение максимальной длины строки в файле
file = fopen("input.txt", "r");
while(!feof(file)) {
  fgets(str_array, LEN_STR, file);
  last_symbol = str_array[strlen(str_array) - 1];
  if(last_symbol == '\n') {
    str_array[strlen(str_array) - 1] = '\0';
  }
  if(strlen(str_array) > max_length) {
   max_length = strlen(str_array);
```

```
    }
  }
  fclose(file);

  //среди строк с максимальной длиной находим минимальное
количество слов
  file = fopen("input.txt", "r");
  int min_wrd = 1000, count = 0;;
  while(!feof(file)) {
    fgets(str_array, LEN_STR, file);
    last_symbol = str_array[strlen(str_array) - 1];
    if(last_symbol == '\n') {
      str_array[strlen(str_array) - 1] = '\0';
    }
  //строки с максимальной длиной записываются во временный файл
    if(strlen(str_array) == max_length) {
      tmp_file = fopen("tmp.txt", "w");
      fprintf(tmp_file, "%s", str_array);
      fclose(tmp_file);

  //поиск количества слов во временном файле
      count = 0;
      tmp_file = fopen("tmp.txt", "r");
      while(!feof(tmp_file)) {
        fscanf(tmp_file, "%s", tmp_wrd);
        count++;
      }
      fclose(tmp_file);
  //сравнение текущего количества слов с минимальным
      if(count < min_wrd) {
        min_wrd = count;
      }
    }
  }
  fclose(file);

  //еще раз проходим по исходному файлу, находим строки с
максимальной длиной и выбираем из них строки с минимальным
количеством слов
  file = fopen("input.txt", "r");
  while(!feof(file)) {
    fgets(str_array, LEN_STR, file);
```

```
    last_symbol = str_array[strlen(str_array) - 1];
    if(last_symbol == '\n') {
      str_array[strlen(str_array) - 1] = '\0';
    }
    if(strlen(str_array) == max_length) {
      tmp_file = fopen("tmp.txt", "w");
      fprintf(tmp_file, "%s", str_array);
      fclose(tmp_file);

      int count = 0;
      tmp_file = fopen("tmp.txt", "r");
      while(!feof(tmp_file)) {
        fscanf(tmp_file, "%s", tmp_wrd);
        count++;
      }
      fclose(tmp_file);

      if(count == min_wrd) {
        printf("Искомая строка - %s содержит %d слов\n",
                                              str_array, min_wrd);
      }
    }
  }
  fclose(file);
```

10. Структуры

Структура — это совокупность переменных, объединенных под одним именем. С помощью структур удобно размещать связанные между собой элементы информации. Объявление структуры создает шаблон, который можно использовать для создания ее объектов (экземпляров этой структуры). Переменные, из которых состоит структура, называются элементами структуры или полями.

10.1. Простые структуры

Определение структуры

```
struct имя_структуры {
    поле 1;
    поле 2;
    поле 3;
    -------
```

```
    поле N;
};
```

Ключевое слово **struct** сообщает компилятору, что объявляется (декларируется) структура. Далее указывается имя структуры и в фигурных скобках поля, которые будет иметь данная структура. Поля структуры могут быть любого типа. Это могут быть как целые или дробные числа, числовые массивы, строки, массивы строк и даже другие структуры.

Как правило, поля структуры связаны друг с другом по смыслу. Например, информацию о человеке, состоящую из имени, адреса и телефона логично представить в виде структуры.

```
struct person {
    char name[250];
    int age;
    char address[250];
    char telephone[250];
};
```

Обратите внимание, что объявление завершается точкой с запятой, потому что объявление структуры является оператором. Кроме того, имя структуры `person` идентифицирует эту конкретную структуру данных и является спецификатором ее типа.

В данном случае на самом деле никакая переменная не создается. Всего лишь определяется вид данных. Когда вы объявляете структуру, то определяете агрегатный тип, а не переменную. И пока вы не объявите переменную этого типа, то существовать она не будет.

Чтобы объявить переменную (то есть физический объект) типа `person`, напишем следующее:

```
struct person client;
```

В этом операторе объявлена переменная типа `person`, которая называется `client`. Таким образом, `person` описывает вид структуры (ее тип), а `client` является экземпляром (объектом) этой структуры.

Когда объявляется переменная-структура, компилятор автоматически выделяет количество памяти, достаточное, чтобы разместить все ее поля.

Доступ к полям структуры

Доступ к отдельным полям структуры осуществляется с помощью оператора «.», который называют оператором точка или оператором доступа к полю структуры. Например, в следующем выражении полю age в уже объявленной переменной-структуре client присваивается значение 45:

```
client.age = 45;
```

Это отдельное поле определяется именем объекта (в данном случае client), за которым следует точка, а затем именем самого этого члена (в данном случае age). В общем виде использование оператора точка для доступа к члену структуры выглядит таким образом:

```
имя_объекта.имя_поля
```

Таким образом, чтобы вывести на экран значение age, следует написать:

```
printf("%d", client.age);
```

Будет выведено значение, которое находится в поле age переменной-структуры client. Точно так же в вызове gets() можно использовать массив символов client.name:

```
gets(client.name);
```

Так как name является массивом символов, то чтобы получить доступ к отдельным символам в массиве client.name, можно использовать индексы вместе с name. Например, с помощью следующего кода можно посимвольно вывести на экран содержимое client.name:

```
for(i = 0; i < strlen(client.name); i++)
    putchar(client.name[i]);
```

Обратите внимание, что индексируется именно name (а не client). client — это имя всего объекта-структуры, а name — имя поля этой структуры. Таким образом, если требуется индексировать поле структуры, то индекс необходимо указывать после имени этого поля.

Присваивание структур

Информация, которая находится в одной структуре, может быть присвоена другой структуре того же типа при помощи единственного оператора присваивания. Нет необходимости присваивать значения каждого поля в отдельности.

```
    //объявление структуры «мобильный телефон» с полями: фирма, модель, операционная система и цена
    struct mobile_phone {
        char firma_name[20];
        char model_name[20];
        char OS_system[10];
        int price;
    };

    //объявляем две переменные-структуры
    struct mobile_phone phone1, phone2;

    //вводим с клавиатуры значение полей для phone1
    gets(phone1.firma_name);
    gets(phone1.model_name);
    gets(phone1.OS_system);
    scanf("%d", &phone1.price);

    //копируем структуру phone1 в структуру phone2
    phone2 = phone1;

    //теперь обе переменные-структуры имеют одинаковые значения всех полей.
```

Использование ключевого слова `typedef`

Это ключевое слово позволяет упростить работу с некоторыми типами данных. Используя **`typedef`**, можно определять новые имена типов данных. На самом деле таким способом новый тип данных не создается, а всего лишь определяется новое имя для уже существующего типа. Такие выражения могут помочь в самодокументировании кода, позволяя давать понятные имена стандартным типам данных. Общий вид декларации **`typedef`** (оператора **`typedef`**) такой:

```
typedef  тип  новое_имя;
```

- `тип` — это любой тип данных языка Си;
- `новое_имя` — это новое имя указанного типа.

Важно понимать, что новое имя является дополнением к уже существующему имени, а не его заменой.

Например, для **`float`** можно создать новое имя:

```
typedef float new_float;
```

Это выражение дает компилятору указание считать `new_float` еще одним именем **`float`**. Затем, используя `new_float`, можно создать переменную типа **`float`**:

```
new_float number;
```

У нас появилась дробная переменная `number` типа `new_float`, а `new_float` является еще одним именем типа **`float`**.

Теперь, когда имя `new_float` определено, его можно использовать и в другом операторе **`typedef`**. Например, выражение

```
typedef new_float another_new_float;
```

дает компилятору указание признавать `another_new_float` в качестве еще одного имени `new_float`, которое в свою очередь является еще одним именем **`float`**.

Использование операторов **`typedef`** может облегчить чтение кода и его перенос на новую машину. Однако следует помнить, что новый физический тип данных таким способом вы не создадите.

Ключевое слово **`typedef`** может быть использовано при объявлении структуры:

```
typedef struct {
  char firma_name[250];
  char model_name[250];
  char OS_system[250];
  int price;
} mobile_phone;
```

Таким образом, последнее слово `mobile_phone` — это новое имя типа. А тип описывается структурой:

```
struct {
  char firma_name[250];
  char model_name[250];
  char OS_system[250];
  int price;
}
```

Теперь мы можем использовать слово `mobile_phone` как обозначение описанной структуры. При этом объявление экземпляров структуры будет выглядеть следующим образом:

```
mobile_phone phone_list, phone_list2, my_phone;
```

Пример. Создать структуру `Функция` с полями: `x[N]`, `y[N]`, `z[N]`. Заполнить массивы координат x и y значениями:

- по `x`: от `-3` с шагом `2.5`;
- по `y`: от `15` с шагом `1.8`.

Вычислить значения поля `z[N]` по формуле: `z = 8x + 1.5y`.

```
#define N 40

//задаем структуру «функция»
typedef struct {
  double x[N];
  double y[N];
  double z[N];
} function;

int main(void) {

  int i;
  double h;
  function funk; //объявляем переменную-структуру


  //заполнение значениями поля double x[N]
```

```
  for(i = 0, h = -3; i < N; h += 2.5, i++)
    funk.x[i] = h;

  /заполнение значениями поля double y[N]
  for(i = 0, h = 15; i < N; h += 1.8, i++)
    funk.y[i] = h;

  //вычисление значений поля double z[N]
  for(i = 0; i < N; i++)
    funk.z[i] = 8 * funk.x[i] + 1.5 * funk.y[i];

  //вывод на экран значений всех полей
  for(i = 0; i < N; i++)
    printf("x = %7.2lf, y = %7.2lf, z = %7.2lf\n",
                                funk.x[i], funk.y[i], funk.z[i]);

  return EXIT_SUCCESS;
}
```

10.2. Массивы структур

Чтобы объявить массив структур, вначале необходимо определить структуру, а затем объявить переменную массива этого же типа. Например, чтобы объявить 100-элементный массив структур типа mobile_phone, нужно написать следующее:

```
#define N 100
typedef struct {
    char firma_name[20];
    char model_name[20];
    char OS_system[10];
    int price;
} mobile_phone;

mobile_phone phone_list[N];
```

Это выражение создаст 100 переменных, каждая из которых организована так, как определено в структуре mobile_phone.

Чтобы получить доступ к определенной структуре из массива структур, нужно указать имя массива структур с индексом. Например, чтобы вывести поле стоимость третьего элемента массива структур phone_list нужно сделать следующее:

```
printf("%d", phone_list[2].price);
```

В массивах структур индексирование так же начинается с 0.

Чтобы указать определенную структуру, находящуюся в массиве структур, необходимо указать имя этого массива с определенным индексом. А если нужно указать индекс определенного элемента в структуре, то необходимо указать индекс этого элемента. Таким образом, в результате выполнения следующего выражения первому символу поля model_name, находящегося в третьей структуре из phone_list, присваивается значение 'X'.

```
phone_list[2].model_name[0] = 'X';
```

Приведем пример работы с массивом структур, для которого полем является числовой или строковый массив. Структура «мама» состоит из полей: имя, список имен детей, массив, отражающий возраст детей. Рассмотрим три способа заполнения массива структур. Эти способы также подходят и для единичной структуры.

```
#define N 4 //количество мам
#define M 3 //количество детей у каждой мамы
typedef struct {
  char mum_name[20];
  char childrent_name[M][20];
  int childrent_age[M];
} mother;

//три способа заполнить структуру значениями:

//первый способ: задать чему равны поля структур при объявлении (инициализация при объявлении)

mother mammy[N] = {
    {"Anna", {"Katrin","Piter", "Rosa"}, 9, 5, 12},
    {"Rita", {"Misha", "Liza", "Olga"}, 14, 15, 21},
    {"Mary", {"Egor", "Alex", "Mike"}, 20, 7, 15},
    {"Sveta", {"Katrin", "Marta", "Erin"}, 28, 25, 5}
};

//второй способ: задать чему равны поля структур с клавиатуры
for (i = 0; i < N; i++) {
```

```
  gets(mammy[i].mum_name);
  for (j = 0; j < M; j++) {
    gets(mammy[i].childrent_name[j]);
    scanf("%d\n", &mammy[i].childrent_age[j]);
  }
}

//третий способ: задать чему равны поля структур из файла
FILE * file;
if ((file = fopen("mammy_list.txt","r")) == 0) {
  printf("error");
  exit(1);
}
for (i = 0; i < N; i++) {
  fscanf(file, "%s", mammy[i].mum_name);
  for (j = 0; j < M; j++) {
    fscanf(file, "%s %d", mammy[i].childrent_name[j],
                                    &mammy[i].childrent_age[j]);
  }
}
fclose(file);
```

Пример. Задать структуру, описывающую студента и его успеваемость. Создать массив из 25ти студентов. Вывести на экран таблицу со всеми полями получившегося массива структур.

```
#define N 25
//объявляем структуру «студент»
typedef struct {
  char name[100];
  int age;
  int course;
  char group[20];
  double avr_mark;
} student;

int main(void) {
  int i = 0;

  //объявляем массив студентов из N элементов
  student muctr_stud[N];
```

```
    FILE * file;
    if ((file = fopen("student_list.txt","r")) == 0) {
      printf("error");
      exit(1);
    }
      //заполняем поля массива структур из файла
    for (i = 0; i < N; i++) {
      fscanf(file, "%s %d %d %s %lf", muctr_stud[i].name,
&muctr_stud[i].age, &muctr_stud[i].course, muctr_stud[i].group,
&muctr_stud[i].avr_mark);
    }
    fclose(file);


    printf("Таблица всех студентов\n");

    //выводим шапку таблицы
    printf("%10s %5s %5s %10s %15s\n\n","student_name", "age",
"course", "group", "average_mark");

    //выводим строчки таблицы
    for (i = 0; i < N; i++) {
        printf("%10s %5d %5d %10s %15.1lf\n", muctr_stud[i].name,
muctr_stud[i].age, muctr_stud[i].course, muctr_stud[i].group,
muctr_stud[i].avr_mark);
    }
    return EXIT_SUCCESS;
}
```

Пример. Задать структуру, описывающую домашних питомцев. Создать массив из 10ти животных. Вывести на экран таблицу со всеми полями получившегося массива структур. Вывести на экран клички всех питомцев, которые старше среднего возраста всех питомцев.

```
#define N 7
//объявляем структуру «питомец»
typedef struct {
  char type[50];
  char name[100];
  int age;
  char favourite_food [20];
}pet;

int main(void) {
```

```
      int i = 0;
      pet pet_list[N]; //объявляем массив из N питомцев

      FILE * file;
      if ((file = fopen("pets_list.txt","r")) == 0) {
        printf("error");
        exit(1);
      }
      for (i = 0; i < N; i++) {
        fscanf(file, "%10s %10s %5d %10s", pet_list[i].type,
pet_list[i].name, &pet_list[i].age,
pet_list[i].favourite_food);
      }
      fclose(file);

      printf("Таблица всех питомцев\n");

      //выводим на экран шапку таблицы
      printf("%10s %10s %5s %10s\n\n", "pet_type", "name", "age",
"favourite_food");
      //выводим на экран строчки таблицы
      for (i = 0; i < N; i++) {
        printf("%10s %10s %5d %10s", pet_list[i].type,
pet_list[i].name, pet_list[i].age, pet_list[i].favourite_food);
      }

      //суммируем возраст всех питомцев
      int sum_age = 0;
      for (i = 0; i < N; i++) {
        sum_age += pet_list[i].age;
    }

      //расчитываем средний возраст
      double avr_age = (double)sum_age / N;
      printf("Средний возраст питомцев равен %lf", avr_age);

      //выводим на экран имена питомцев с возрастом больше среднего
      printf("Клички питомцев, чей возраст больше среднего\n\n");

      for (i = 0; i < N; i++) {
        if(pet_list[i].age > avr_age) {
          printf("%10s\n", pet_list[i].name);
        }
```

```
  }

  return EXIT_SUCCESS;
}
```

10.3. Вложенные структуры

Рассмотрим ситуацию, когда одним из полей структуры является другая структура. В этом случае сначало необходимо определить вложенную структуру. А затем использовать ее при определении основной структуры.

Рассмотрим пример, когда существует структура Мама с полями: Имя, Возраст, Массив детей; где Массив детей – это массив структур Ребенок с полями Имя, Пол, Возраст.

```
#define N 4 //количество мам
#define M 3 //количество детей у каждой мамы

typedef struct {
  char child_name[20];
  int child_age;
} child;

typedef struct {
  char mum_name[20];
  int mum_age;
  child child_list[M];
} mother;

int main(void) {
  mother mum_list[N];
  FILE * file, *file2;
  int i = 0, j = 0;
  file = fopen("mum_file", "r");
  file2 = fopen("child_file", "r");
  for(i = 0; i < N; i++) {
    fscanf(file, "%s %d", mum_list[i].mum_name,
                                  &mum_list[i].mum_age);
    for(j = 0; j < M; j++) {
      fscanf(file2, "%s %d",
                  mum_list[i].child_list[j].child_name,
                     &mum_list[i].child_list[j].child_age);
    }
  }
  fclose(file);
  fclose(file2);
  return EXIT_SUCCESS;
}
```

В случае, если у каждой из мам разное количество детей, то при определении структуры невозможно указать размер массива структур типа `child`. В таком случае необходимо использовать динамическую память для работы с массивом типа `child`.

```
#define N 4 //количество мам

typedef struct {
  char child_name[20];
  int child_age;
} child;

typedef struct {
  char mum_name[20];
  int mum_age;
  int child_num;
  child * child_list; //указатель на массив детей
} mother;

int main(void) {
  mother mum_list[N];
  FILE * file, *file2;
  int i = 0, j = 0;
  file = fopen("mum_file", "r");

//считываем Имя, Возраст и Количество детей у каждой мамы
  for(i = 0; i < N; i++) {
    fscanf(file, "%s %d %d", mum_list[i].mum_name,
           &mum_list[i].mum_age, &mum_list[i].child_num);
}
  fclose(file);

//Выделяем динамическую память для массива детей
  for(i = 0; i < N; i++) {
    mum_list[i].child_list =
        (child*)malloc(mum_list[i].child_num * sizeof(child));
  }

  file2 = fopen("child_file", "r");

//второй цикл от нуля до количества детей у каждой мамы
  for(i = 0; i < N; i++) {
    for(j = 0; j < mum_list[i].child_num; j++) {
      fscanf(file2, "%s %d",
                  mum_list[i].child_list[j].child_name,
                     &mum_list[i].child_list[j].child_age);
    }
  }
  fclose(file2);
```

```
//освобождаем память
  for(i = 0; i < N; i++) {
    free(mum_list[i].child_list);
  }

  return EXIT_SUCCESS;
}
```

11. Стандартная библиотека языка Си

11.1. stdio.h

Заголовочный файл стандартной библиотеки **stdio.h** (**st**andar**d** **i**nput **o**utput) содержит в себе функции для ввода и вывода, функции файловых операций и несколько символьных констант.

Функции файловых операций

Функция	Описание
fopen(имя файла, режим открытия)	Открытие файла для чтения или записи
fclose(указатель файлового потока)	Закрытие файла
remove(путь к файлу)	Удаление файла
remame(старое имя, новое имя)	Переименование файла
rewind(указатель файлового потока)	Возврат курсора в начало файла
tmpfile()	Создание и открытие временного файла, который удаляется при его закрытии

Перечисленные функции подробнее рассмотрены в Разделе 9.2.

Функции ввода/вывода

Функция	Описание
Работа с отдельными символами	
getchar()	Консольный ввод одного символа
putchar()	Консольный вывод одного символа
fgetc()	Файловый ввод одного символа
fputc()	Файловый вывод одного символа
Работа со строкой символов	
gets()	Консольный ввод строки символов
puts()	Консольный вывод строки символов
fgets()	Файловый ввод строки символов
fputs()	Файловый вывод строки символов

Работа с любыми данными	
`scanf()`	Консольный ввод
`printf()`	Консольный вывод
`fscanf()`	Файловый ввод
`fprintf()`	Файловый вывод
Чтение и запись в строку	
`sscanf()`	Ввод из строки символов
`sprintf()`	Вывод в строку символов
Другие функции и символьные константы	
`feof()`	Функция определяет, не является ли последний прочитанный символ символом конца файла
`fflush()`	Принудительное очищение буфера ввода/вывода
`EOF`	Символьная константа для обозначения конца файла
`FOPEN_MAX`	Символьная константа для обозначения максимального возможного числа одновременно открытых файлов (не менее 8ми)

Ввод/вывод отдельных символов

Консольный ввод/вывод символов

```
char symbol = 0;
symbol = getchar();  // считывание символа
putchar(symbol);     // вывод на экран
```

Файловый ввод/вывод символов

```
char symbol = 0;
FILE * file;
file = fopen("input.txt", "r");
symbol = fgetc(file);        // считывание символа из файла
fclose(file);

file = fopen("output.txt", "w");
fputc(symbol, file);         // вывод символа в файл
fclose(file);
```

Ввод/вывод строк

Консольный ввод/вывод строк

```
#define LEN 500
char sample_str[LEN];
gets(sample_str); // считывание символа из файла

puts(sample_str); // вывод символа в файл
```

Файловый ввод/вывод строк

```
#define LEN 500
char sample_str[LEN];
FILE * file;
file = fopen("input.txt", "r");
fgets(sample_str, LEN, file);   // считывание строки из файла
fclose(file);

file = fopen("output.txt", "w");
fputs(sample_str, file);        // вывод строки в файл
fclose(file);
```

Универсальные функции ввода/вывода

Функция консольного вывода `printf()`

Функция `printf()` выводит информацию на экран с учетом выбранного форматирования. На вход функции подается строка форматирования и список параметров.

```
printf(" строка форматирования ", список параметров);
```

- Строка форматирования – это специальная последовательность символов, которая отображает, как именно вы хотите записать число или символ. Рассмотрим ее более подробно.

```
"%[флаг][ширина][.точность]тип"
```

- Знак процента (%) указывает на то, что в это место строки форматирования будет вставлен один из параметров.

- Флаги

Флаги представлены в Таблице 5.

Таблица 11.1. Флаги строки форматирования.

Знак	Название	Значение	Отсутствие знака
-	дефис	Выровнять по левому краю	По правому краю
+		Указать знак числа	Указать знак только для отрицательных чисел
пробел	пробел	Поместить перед результатом пробел	--
0	ноль	Дополнить нулями до ширины поля	Дополнить пробелами до ширины поля

- Ширина – это ширина выделяемого поля для записи. Если выделяется недостаточное по ширине поле, то оно автоматически расширяется до нужного размера. Например, если для записи числа 4711 выделяется поле шириной в 1, 2 или 3, то ширина автоматически увеличивается до 4.

- Точность соответствует количеству знаков после запятой, которые нужно вывести на экран. При этом происходит округление до нужного знака. Например, если вы выводите число 2.385 с точностью до двух знаков, то на экране появится число 2.39.

- Тип – это условное обозначение, которое соответствует тому типу переменных, которые мы выводим на экран. Некоторые типы представлены в Таблице 6.

Таблица 11.2. Обозначения типа аргумента в строке форматирования.

Обозначение	Значение	Тип аргумента
`d`	`double`	указатель на `int`
`f`	`float`	указатель на `float`
`lf`	`long float`	указатель на `double`
`c`	`char`	указатель на `char`
`s`	`string`	указатель на `строку`

- В список параметров могут входить символы, строки, целые и дробные числа, переменные, выражения.

- Ширина и точность — это необязательные параметры. Если они не указаны, то срабатывает форматирование по умолчанию. Например, для целых чисел это означает, что будет выделена минимально допустимая ширина поля, а для дробных чисел — что точность устанавливается на `6` знаков после запятой, а ширина поля так же становится минимально возможной.

1	printf ("%d", 3);	3						
2	printf ("%4d", 3);				3			
3	printf ("%-4d", 3);	3						
4	printf ("+7d", 3);						+	3
5	printf ("+-3d", 3);	+	3					
6	printf ("0+5d", 3);	+	0	0	0	3		
7	printf ("0+-5d", 3);	+	3					
8	printf ("%2d", 3675);	3	6	7	5			

1. В первом случае мы указываем только тип данных, которые мы выводим, поэтому все остальные параметры выбираются по умолчанию: выравнивание по правому краю, знак указывать только для отрицательных чисел, ширина поля равна количеству цифр.
2. Указана только ширина поля, значит все остальные параметры выбираются по умолчанию. Получаем параметры форматирования: выравнивание по правому краю, знак указывать только для отрицательных чисел, ширина поля равна `4`.
3. Флаг «дефиса» указывает на выравнивание по левому краю. Форматирование: выравнивание по левому краю, знак указывать только для отрицательных чисел, ширина поля равна `4`.
4. Флаг «плюс» требует выводить знак числа для положительных и отрицательных чисел. Форматирование: выравнивание по правому краю, знак указывать для всех чисел, ширина поля равна `7`.
5. Сочетание флага «дефис» и «плюс». Форматирование: выравнивание по левому краю, знак указывать для всех чисел, ширина поля равна `3`.
6. Флаг «ноль» заполняет пустые пробелы в начале нулями. Форматирование: выравнивание по правому краю, знак указывать для всех чисел, лишние пробелы заполнить нулями, ширина поля равна `5`.
7. Флаги «плюс», «дефис», «ноль». При сочетании флагов «дефис» и «ноль» флаг «ноль» не учитывается, так как при выравнивании по левому краю нет лишних

пробелов в начале. Форматирование: выравнивание по левому краю, знак указывать для всех чисел, ширина поля равна 5.

8. Если указанная ширина поля меньше, чем минимально необходимое поле для записи числа, то оно расширяется. В нашем случае поле указано равным 2, а число состоит из 4x цифр, поэтому ширина поля автоматически становится равной 4. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, ширина поля равна 4.

Примеры вывода на экран дробных чисел (double):

1	printf ("%lf", 2.157);	2	.	1	5	7	0	0	0
2	printf ("%5.1lf", 2.157);			2	.	2			
3	printf ("%5lf", 2.157);	2	.	1	5	7	0	0	0
4	printf ("%.1lf", 2.157);	2	.	2					
5	printf ("%7.5lf", 2.157);	2	.	1	5	7	0	0	
6	printf ("%+6.2lf", 2.157);		+	2	.	1	6		
7	printf ("%-7.0lf", 2.157);	2							
8	printf ("%+07.2lf", 2.157);	+	0	0	2	.	1	6	
9	printf ("%+-07.2lf", 2.157);	+	2	.	1	6			

1. Если в строке форматирования дробного числа указан только тип, то по умолчанию устанавливается: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность — 6 знаков после запятой, ширина поля равна количеству цифр в целой части числа плюс еще 7 (6 — для дробной части и 1 для запятой).
2. Если точность меньше, чем количество знаков после запятой, то число округляется до указанной точности. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 1, ширина поля равна 5.
3. Если указана только ширина поля, то точность выставляется по умолчанию равной 6. При этом ширина поля увеличивается. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 6, ширина поля равна 8.
4. Указана только точность, ширина поля подстраивается автоматически. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 1, ширина поля равна 3.
5. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 5, ширина поля равна 7.

6. Флаг «плюс» указывает на то, что следует выводить знак числа. Форматирование: выравнивание по правому краю, знак указывать для всех чисел, точность равна 2, ширина поля равна 6.
7. Флаг «дефис» устанавливает выравнивание по левому краю. Если точность равна нулю, то число округлится до целого. Форматирование: выравнивание по левому краю, знак указывать только для отрицательных чисел, точность равна 0, ширина поля равна 7.
8. Флаг «ноль» выводит нули вместо лишних пробелов. Флаг «плюс» указывает знак числа. Форматирование: выравнивание по правому краю, знак указывать для всех чисел, точность равна 2, ширина поля равна 7, между знаком числа и числом выводим нули.
9. Флаги «плюс», «дефис», «ноль». При сочетании флагов «дефис» и «ноль» флаг «ноль» не учитывается, так как при выравнивании по левому краю нет лишних пробелов в начале. Форматирование: выравнивание по левому краю, знак указывать для всех чисел, точность равна 2, ширина поля равна 7.

Если мы заранее не знаем, какая нам нужна точность и/или ширина поля, то мы можем указать эти величины в качестве параметров, а в строке форматирования использовать знаки «*».

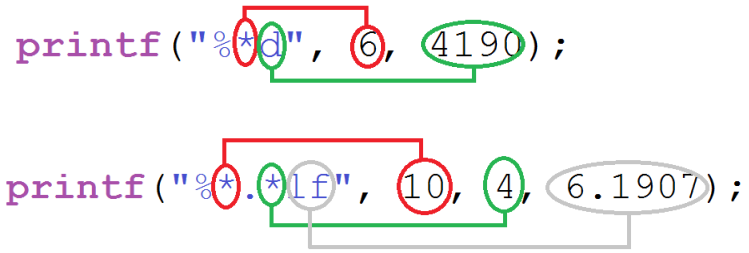


Рисунок 11.1. Схема подстановки при использовании знака «*» для целых и дробных аргументов.

Примеры использования знака «*»

printf ("%d", 5, 123);			1	2	3		
printf ("%*.*lf", 7, 2, 2.157);				2	.	1	6
printf ("%4.*lf", 1, 2.157);		2	.	2			
printf ("%*.2lf", 6, 2.157);			2	.	1	6	

Примеры вывода символов и строк.

printf ("%5c", 'h');					h								
printf ("%-3c", 'k');	k												
printf ("%s","hello!");	h	e	l	l	o	!							
printf ("%s", "I like cookies");	I		l	i	k	e		c	o	o	k	i	e	s

Вывод нескольких значений разных типов. Сложное форматирование.

Если вы хотите вывести не одно, а сразу несколько значений, то это нужно отразить и в строке форматирования, и в списке параметров. Например,

```
printf("%d %lf %d %c", 45, 95.1, -72, 'f');
```

выведет нам в строку через пробел: «**45 95.100000 -72 f**»

Если же мы не поставим пробелы в строке форматирования

```
printf("%d%lf%d%c", 45, 95.1, -72, 'f');
```

то увидим: «**4595.100000-72f**»

Если не ставить пробелы в строке форматирования, то и на экране их тоже не будет.

Во многих случаях нам нужно не просто увидеть цифры или буквы на экране, а знать, что каждая из них значит. Чаще всего на экран выводят не абстрактные числа, а переменные, которые содержат в себе результаты вычислений.

```
x = 45;
y = 95.1;
z = -72;
printf("x = %d y = %.1lf z = %d", x, y, z );
```

на экране появится: «**x = 45 y = 95.1 z = -72**»

```
int x = 27, y = 10;
double z = x / y;
printf("результат вычисления x/y = %lf", z);
```

на экране появится: «**результат вычисления x/y = 2.000000**»

Можно вывести результат вычисления более подробно:

```
int x = 27, y = 10;
double z = x / y;
printf("результат вычисления %d/%d = %.2lf", x, y, z);
```

тогда вы увидите: «**результат вычисления 27/10 = 2.00**»

В случае если нам нужно вывести информацию на нескольких строчках, это можно сделать с помощью управляющих последовательностей и экранирования.
Экранирование (\ ) – это операция, позволяющая заменять в тексте управляющие символы на соответствующие текстовые подстановки.

Управляющие последовательности не требуется отделять пробелами. Они пишутся слитно с текстом.

```
    printf("Строка1\nСтрока2\nСтрока3");
```
«Строка1
Строка2
Строка3»

```
    printf("Это обычная строка\n\tA это красная строка");
```
«Это обычная строка
   А это красная строка»

```
    printf("Одинарная кавычка \' используется редко.");
```
«Одинарная кавычка ' используется очень редко.»

```
    printf("Прямая речь заключается в двойные кавычки \" \".");
```
«Прямая речь заключается в двойные кавычки " ".»

```
    printf("Ставка по кредиту не более %d%% годовых.", 5);
```
«Ставка по кредиту не более 5% годовых.»

```
    printf("x = %d\ny = %.2lf\n\tz = %c", 5, 6.2, 'j');
```
«x = 5
y = 6.20

z = j»

Функция консольного ввода scanf()

Функция **scanf()** считывает с клавиатуры символы, строки, числа и записывает их в адрес указанных переменных. Синтаксис функции очень похож и даже практически идентичен функции **printf()**.

```
scanf("строка форматирования", адреса переменных);
```

Строка **scanf("%d", &x);** означает, что мы считываем с клавиатуры целое число и записываем его в адрес **(&)** переменной **x**. Фактически переменная **x** станет равной числу, которое мы введем с клавиатуры.

&(амперсанд) – это операция взятия адреса.

```
scanf("%lf", &y); // дробное значение для переменной y
scanf("%d %lf", &a, &b); // целое a и дробное b через пробел
scanf("%c %c", &str1, &str2); // два символа через пробел
```

Так же можно использовать сложное форматирование:

```
scanf("часы = %d минуты = %d", &hour, &min);
```

здесь мы должны учесть форматирование, то есть мы должны **напечатать** «часы = 12 минуты = 34», соблюдая все слова и пробелы. Таким же образом можно вводить даты:

```
scanf("%d.%d.%d", &day, &month, &year);// печатаем «15.05.2023»
```

```
scanf("мимо пробегали %d розовых слоников", &elephants);
```

здесь тоже печатаем фразу целиком: «мимо пробегали 8 розовых слоников». Проще говоря, как написано в строке форматирования, так и нужно вводить данные с клавиатуры.

Файловый ввод/вывод информации с помощью fscanf() и fprintf()

Универсальные функции файлового ввода/вывода, соответствующие функциям консольного ввода/вывода имеют только одно отличие – первым параметром функции указывается используемый файловый поток.

```
#define LEN 500
int int_var = 0;
double dbl_var = 0;
char symbol = 0, str[LEN];

// чтение из файла целого и дробного числа, символа и строки
FILE * file;
file = fopen("input.txt", "r");
fscanf(file, "%d %lf %c %s", &int_var, &dbl_var, &symbol, str);
fclose(file);

// вывод в файл целого числа, дробного числа, символа и строки
file = fopen("output.txt", "w");
fprintf(file, "%d %lf %c %s", int_var, dbl_var, symbol, str);
fclose(file);
```

Стоит обратить внимание на то, что амперсант ставится только перед переменными. Так как строка – это массив символов, то амперсанд перед имененм массива не ставится, так как имя массива это и еть его адрес. Кроме этого следует отметить еще одну особенность работы функции **fscanf()** со строками: чтение происходит до символа пробел. То есть эта функция может прочитать и записать в строку слово из файла, но не может прочитать сразу несколько слов и целую строку.

Вод/вывод данных в строку

В случае если необходимо извлечь информацию из строки или же наоборот записать ее туда используются функции **sscanf()** и **sprintf()**.

Чтение из строки

```
#define LEN 500
char str[LEN], month[LEN];
int day = 0;
gets(str);    // ввод "23 сентября"

//чтение из строки и запись в переменные
sscanf(str, "%d %s", &day, month); //day=23, month = "сентября"

// Вывод "Сегодня 23 сентября"
printf("Сегодня %d %s", day, month);
```

Запись в строку

```
#define LEN 500
char str[LEN], month[LEN];
int day = 0;

//ввод day = 23, month = "сентября"
scanf("%d %s", &day, month);

//запись в строку слова "Сегодня" и переменных в строку
sprintf(str, "Сегодня %d %s", day, month);

puts(str);   // Вывод "Сегодня 23 сентября"
```

Управление потоком данных

Функция `fflush()` сбрасывает в указанный поток все данные, находящиеся в буфере. Это часто необходимо для оперативного вывода информации в консоль. Например, при использовании ввода с клавиатуры и предварительного «пригласительного вывода».

```
int variable = 0;
printf("Введите  значение переменной variable\n");
scanf("%d", &variable);
```

В этом случае при запуске программы сначала вам необходимо будет задать значение переменной, а потом только появится надпись «Введите значение переменной variable». Это происходит из за того, что при выводе данные накапливаются в буфере, и, пока он не окажется переполнен, они не выводятся в соответствующий поток.

Для того, чтобы принудительно очистить буфер и вовремя напечатать необходимую информацию используется функция `fflush()`.

```
int variable = 0;
printf("Введите  значение переменной variable\n");
fflush(stdout);
scanf("%d", &variable);
```

В качестве параметра функции указывается название потока в который необходимо записать данные из буфера. В данном случае это стандартный поток вывода на консоль.

11.2. math.h

Заголовочный файл содержит функции для выполнения простых математических операций. Все функции работают с типом **double**, если не определено иначе.

Таблица 11.3. Некоторые математические функции заголовочного файла стандартной библиотеки math.h

Математический вид	math.h	Описание функции
|x|	abs(x)	модуль целого числа
|x|	fabs(x)	модуль дробного числа
$\sqrt{x}$	sqrt(x)	квадратный корень
a^x	pow(a, x)	возведение в степень
e^x	exp(x)	экспонента
ln(x)	log(x)	натуральный логарифм
lg(x)	log10(x)	десятичный логарифм
sin)x)	sin(x)	синус
cos(x)	cos(x)	косинус
tg(x)	tan(x)	тангенс
	round(x)	округление до целого
	ceil(x)	округление до большего целого
	floor(x)	округление до меньшего целого

Стоит заметить, что тригонометрические функции принимают значения не в градусах, а в радианах. Например, `cos(60)` не будет равен `0.5`, потому что мы посчитали синус угла `60` радиан, а не `60` градусов.

11.3. stdlib.h

Этот заголовочный файл содержит в себе функции выделения памяти, функции контроля за работой программы, преобразования типов, генерации псевдослучайных чисел, а также несколько символьных констант.

Функции преобразования типов

Функция	Описание
`atof(строка)`	Преобразованиестроки в дробное число **double** или **float**
`atoi(строка)`	Преобразованиестроки в целое число **int**

```
double x = atof("4.5");
int y = atoi("50");
printf("%d %lf", y, x);
```

Генерация псевдослучайных чисел

Псевдослучайные числа (далее – случайные числа) генерируются с помощью функции **rand**(). Эта функция возвращает целое число в диапазоне от нуля до RAND_MAX. Случайное число генерируется алгоритмом, который возвращает последовательность внешне не связанных цифр при каждом вызове. Этот алгоритм использует серии вызовов, которые должны быть инициализированы значением с помощью функции **srand**().

RAND_MAX - это константа, определенная в заголовочном файле **<stdlib.h>**. Ее значение может варьироваться в зависимости от используемой реализации языка Си, но она не меньше, чем 32767.

Рассмотрим несколько примеров для понимания принципов работы со случайными числами и функцией **rand**().

Пример. Сгенерировать целое случайное число от 0 до 9.

```
int x = rand() % 10;
```

Функция **rand**() генерирует число от нуля до RAND_MAX, а затем находится остаток от деления этого числа на 10. Рассмотрим, почему при этом образуется диапазон целых чисел от 0 до 9.

При нахождении остатка от деления одного числа на другое результатом могут быть только определенные целые числа, диапазон этих чисел строго ограничен. Например, если какое-либо число целочисленно делится на 3, то остатком от деления могут быть только три числа:

- ноль (если делится без остатка);
- один (если в остатке единица);
- два (если в остатке двойка).

Других чисел в остатке быть не может.

Соответственно, если находить остаток от деления на 10, то в результате можно получить все целые числа в диапазоне от 0 до 9, вне зависимости от того, какое число мы делим на 10. Таким образом, какое бы число не сгенерировала функция **rand**() при нахождении остатка от деления на 10, получится одно из чисел диапазона [0; 9].

Пример. Сгенерировать целое число в диапазоне от `-328` до `1485`.

```
int x = -328 + rand() % 1814;
```

Сначала подсчитаем, сколько чисел нам нужно сгенерировать. `328` чисел в отрицательную сторону по оси абсцисс, плюс еще 1485 чисел в положительную сторону и плюс число ноль. Всего получается, что заданный диапазон содержит `1814` чисел. Такое количество чисел можно сгенерировать, если найти остаток от деления результата функции **`rand()`** на `1814` (**`rand()`** `% 1814`). Так получатся числа от нуля до `1814`. Чтобы получить нужный диапазон `[-328; 1485]`, нужно сдвинуть полученный диапазон `[0; 1814]` на `-328`. Получаем `-328 +` **`rand()`** `% 1814`.
Вторым способом подсчитать количество цифр между двумя любыми числами является вычитание из большего числа меньшего числа и прибавление единицы:

```
int x = -328 + rand() % (1485 - (-328) +1);
```

Пример. Сгенерировать дробное случайное число в диапазоне от нуля до единицы.

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

Для генерации дробных чисел используется обычное деление. В данном случае деление на число `RAND_MAX`, приведенное к дробному типу, для избежания целочисленного деления. Функция **`rand()`** генерирует случайное число в диапазоне от `0` до `RAND_MAX`. Если мы разделим его на `RAND_MAX`, то минимально получаем ноль (`0 / RAND_MAX`), а максимально – единицу (`RAND_MAX / RAND_MAX`).

Пример. Сгенерировать дробное случайное число в диапазоне от A до B.

```
double x = a + rand() * (b - a) / (double)RAND_MAX;
```

Функция **`rand()`** генерирует случайное число в диапазоне от `0` до `RAND_MAX`. Если сгенерируется ноль, то мы в результате получим число `(A + 0)`, если сгенерируется число `RAND_MAX`, то мы получаем число `(A + (B - A) = B)`. Таким образом, мы получаем дробные числа в заданном диапазоне `[A; B]`.

Инициализация генератора случайных чисел.

Функция **srand()** отвечает за инициализацию генератора случайных чисел. Генератор случайных чисел инициализируется с помощью аргумента, передаваемого в качестве параметра функции. Если параметром задавать одно и то же число, то при различных запусках программы происходит генерация случайных чисел в одинаковом порядке. Например, если в 1й запуск была сгенерирована последовательность 38, 2, -9.., то и во все последующие запуски будет сгенерирована та же последовательность.

Для того чтобы при каждом запуске последовательности случайных чисел были разными, необходимо, чтобы при каждом запуске программы параметрами были различные числа. Для этого можно использовать функцию **time()** из заголовочного файла <**time.h**>. Эта функция возвращает текущее календарное время системы (для UNIX систем это количество секунд, прошедших с полночи 1 января 1970 года по Гринвичу), которое при каждом запуске будет различным. Функция **time()** так же должна содержать в себе параметр, в качестве которого обычно используется ноль.

Таким образом, получаем, что вызов функции **srand()** выглядит так:

```
srand(time(0));
```

Функция **srand()** может вызываться в программе несколько раз. При этом генератор случайных чисел будет инициализироваться заново. Например, следующий код выведет на экран три целых числа, первое и последнее из которых будут одинаковыми, так как эти числа генерируются с одинаковым параметром инициализации генерации случайных чисел, и они одинаковые по порядку генерации.

```
srand(1);
printf("First number: %d\n", rand() % 100);
srand(time(0));
printf("Random number: %d\n", rand() % 100);
srand(1);
printf("Again the first number: %d\n", rand() % 100);
```

Функции выделения и освобождения памяти

Функция	Описание
malloc(количество памяти)	Выделение памяти из Кучи
free(указатель)	Освобождение выделенной в Куче памяти

Более подробно функции рассмотрены в Разделе 7.3.

Функции контроля за работой программы. Функция перехода exit()

Функция `exit()` не является оператором перехода языка Си, но с ее помощью можно осуществить переход. Эта функция используется для прекращения работы программы, то есть с помощью нее управление переходит к оператору `return` главной функции `main()`. Проще говоря, эта функция вызывает немедленное прекращение работы программы и передачу управления операционной системе.

```
exit (код_возврата);
```

Значение кода возврата передается вызвавшему программу процессу. Обычно в качестве этого процесса выступает операционная система. Нулевое значение кода возврата обычно используется для указания нормального завершения работы программы. Другие значения указывают на характер ошибки. В качестве кода возврата можно использовать макросы `EXIT_SUCCESS` и `EXIT_FAILURE` (выход успешный и выход с ошибкой).

11.4. string.h

Заголовочный файл содержит функции для работы со строками.

Функция	Описание
`strlen(str)`	Определение длины строки `str`
`strcmp(str_1, str_2)`	Сравнение строки `str_1` и строки `str_2`
`strcat(str_1, str_2)`	Дописать строку `str_2` в строку `str_1`
`strcpy(str_1, str_2)`	Копирование строки `str_2` в строку `str_1`
`strchr(str, symbol)`	Возвращает первую позицию символа symbol в строке str (поиск с начала строки)
`strrchr(str, symbol)`	Возвращает первую позицию символа `symbol` в строке `str` (поиск с конца строки)
`strpbrk(str_1, str_2)`	Возвращает первую позицию любого из символов строки `str_2` в строке `str_1` (поиск с начала строки)
`strcpy(str_1, str_2)`	Возвращает первую позицию вхождения строки `str_2` в строку `str_1` (поиск с начала строки)

Определение длины строки

```
#define LEN 50
char example_string[LEN + 1];
```

```
gets(example_string);
int lenght = strlen(example_string); // поиск длины строки
printf("Длина строки = %d", lenght);
```

Сравнение двух строк

```
#define LEN 50
char string_1[LEN + 1], string_2[LEN + 1];
gets(string_1);
gets(string_2);
// определение идентичности строк
if (!strcmp(string_1, string_2))
  printf("Строки идентичны\n");
}
else {
  printf("Строки не идентичны\n");
}
```

Слияние двух строк

```
#define LEN 50
char string_1[LEN + 1], string_2[LEN + 1];
gets(string_1);
gets(string_2);
// дописывание строки string_2 в конец строки string_1
strcat(string_1, string_2);
printf("Слияние двух строк в одну = %s", string_1);
```

Копирование строк

```
#define LEN 50
char string_1[LEN + 1], string_2[LEN + 1];
gets(string_1);
gets(string_2);
// копирование строки string_2 в строку string_1
strcpy(string_1, string_2);
printf("Теперь обе строки идентичны\n");
```

Поиск позиции символа в строке (указатель на символ)

```
#define LEN 50
char string_1[LEN + 1], letter = 0;
gets(string_1);
letter = getchar();
// поиск символа с начала строки
```

```
char * position = strchr(string_1, letter);
if (position == 0) {
  printf("Такого символа нет в строке\n");
}
else {
  printf("Символ найден в строке\n");
  printf("Указатель на символ %c при поиске
                    с начала строки = %p\n", letter, position);
}

#define LEN 50
char string_1[LEN + 1], letter = 0;
gets(string_1);
letter = getchar();
// поиск символа с конца строки
char * position = strrchr(string_1, letter);
if(position == 0) {
  printf("Такого символа нет в строке\n");
}
else {
  printf("Символ найден в строке\n");
  printf("Указатель на символ %c при поиске
                     с конца строки = %p\n", letter, position);
}

#define LEN 50
char string_1[LEN + 1], letters[LEN + 1];
gets(string_1);
gets(letters);
// поиск символов из массива letters в строке string_1
char * position = strpbrk(string_1, letters);
if(position == 0) {
  printf("В строке string_1 не найдено
                                символов из строки letters\n");
}
else {
  printf("Символы найдены\n");
  printf("Указатель на один из всимволов
                          строки %s = %p\n", letters, position);
}
```

Поиск подстроки в строке (указателя на начало подстроки)

```
#define LEN 50
char string_1[LEN + 1], sub_string[LEN + 1];
gets(string_1);
gets(sub_string);
// поиск подстроки sub_string в строке string_1
char * position = strstr(string_1, sub_string);
if(position == 0) {
  printf("В строке string_1
                        не найдено подстроки sub_string\n");
}
else {
  printf("Подстрока найдена\n");
  printf("Указатель на начало подстроки %s = %p\n",
                                        sub_string, position);
}
```

11.5. time.h

Заголовочный файл содержит функции для работы с датой и временем. При этом некоторые функции могут работать с «местным временем», которое может отличаться от календарного времени (в связи с часовыми поясами).

Функция	Описание
`clock()`	Возвращает время, измеренное процессом в тактах от начала выполнения программы. Тип возвращаемого значения `clock_t`
`time()`	Возвращает текущее календарное время Тип возвращаемого значения `time_t`

Для пересчета значения типа `clock_t` в секунды используется символьная константа `CLOCKS_PER_SEC`. Она определяет количество тактов системных часов в секунду.

11.6. direct.h и dir.h

Заголовочные файлы содержат функции для работы с директориями файловой системы.

Функция	Описание
`mkdir(имя каталога);`	Создание нового каталога
`rmdir(имя каталога);`	Удаление указанного каталога
`rename(старое имя, новое имя);`	Переименование каталога
`getcwd(строка для сохранения пути, max длина строки);`	Получение имени текущего каталога

`chdir(новый текущий каталог);`	Смена текущего каталога

При указании имени каталога в случае, если искомый каталог лежи в тойже папке что и программа (проект), указывается только имя каталога. Если каталог находится вне текущей директории, то указывается полный путь к каталогу.
Более подробно использование функция описывается в Разделе 9.3.

Учебное издание

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ СИ

Составители:
ФНДОСОВА Наталья Алексеевна,
ЖЕНСА Андрей Вячеславович

Редактор: Е. В. Копасова
Подписано в печать 15.04.2014 г. Формат 60x84 1/6
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз. Заказ

Российский химико-технологический университет имени Д. И. Менделеева
Издательский центр
Адрес университета и издательского центра:
125047 Москва, Миусская пл., 9