Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. Р. Е. АЛЕКСЕЕВА»
(НГТУ)

Э.С. СОКОЛОВА, Д.В. ДМИТРИЕВ, Д.А.ЛЯХМАНОВ

ПРОГРАММИРОВАНИЕ НА С++

Рекомендовано Ученым советом Нижегородского государственного технического университета им. Р.Е. Алексеева
в качестве учебного пособия для студентов всех форм обучения, включающих элементы дистанционных технологий,
по направлениям 09.03.01 «Информатика и вычислительная техника», 09.03.02 «Информационные системы и технологии»

Нижний Новгород 2015

УДК 004.4’242(075.8)
ББК 32.973.26-018.1я73
С 594

Р е ц е н з е н т
доктор физико-математических наук, профессор,
заведующий кафедрой численного и функционального анализа
ННГУ им. Н.И. Лобачевского *Д.В. Баландин*

Соколова Э.С., Дмитриев Д.В., Ляхманов С.Н.
С 594 Программирование на языке С++. Часть 1. Введение в программирование на С/С++: учеб. пособие / Э.С. Соколова, Д.В. Дмитриев, Д.А. Ляхманов; НГТУ им. Р.Е. Алексеева. – Нижний Новгород, 2015. – 160 с.

Учебное пособие предназначено для студентов направлений 09.03.01 «Информатика и вычислительная техника» и 09.03.02 «Информационные системы и технологии».

В 1-й части учебного пособия дан вводный курс в программирование на языках С/С++ - сведения об основных типах данных, операциях над переменными, операциях ввода/вывода данных, операторах языка, сложных структурах данных, в том числе динамических списках. Большое внимание уделено работе с указателями, способах передачи аргументов в функции, работе с файлами. Пособие насыщено примерами, демонстрирующими возможности языка. Особое внимание уделено грамотности использования конструкций языка, приведены примеры возможных ошибок. Коды программ достаточно полно прокомментированы.

Рис. 13. Табл. 6. Библиогр.: 10 назв.

УДК 004.4’242(075.8)
ББК 32.973.26-018.1я73

ISBN ……

Оглавление

1. Основные понятия языка С++

1.1. История создания языка С++

В 70-х годах прошлого столетия Дэннису Ритчи, сотруднику компании Bell Laboratories, занимающемуся разработкой ОС UNIX, необходим был краткий язык программирования, во-первых – обеспечивающий эффективное управление аппаратными средствам (что присуще языкам низкого уровня – ассемблерам); во-вторых - не привязанный к определенному типу процессора (обладающему свойством переносимости с одной платформы на другую); и в третьих – позволяющий создавать компактные, быстро работающие программы.

В результате им был создан язык программирования Си, сочетавший в себе эффективность доступа к аппаратным средствам и общий характер используемых языков высокого уровня (ЯВУ). Этот универсальный язык стал эффективно применяться в задачах системного программирования – при разработке трансляторов, ОС, экранных интерфейсов, инструментальных средств, а также широко использоваться при программировании алгоритмов в различных прикладных областях. Программы на языке Си были сравнимы по скорости с программами, написанными на ассемблере, при этом более просты и наглядны.

Язык Си основан на парадигме структурного программирования, цель которой – улучшить понятность, надежность программ, при этом основной акцент делается на программирование алгоритмов. Программа в этом случае представляет собой набор функций, в которых скрыты разработанные алгоритмы, а обмен данными между функциями осуществляется через их параметры.

С течением времени мощности языка Си стало не хватать для разработки больших и сложных приложений. В начале 80-х годов Бьярном Страуструпом был создан язык С++, в котором он реализовал объектно-ориентированный подход (ООП) к программированию. Страуструп решил две основные задачи: сделал С++ совместимым со стандартом языка Си и расширил язык Си конструкциями объектно-ориентированного программирования, основанными на концепции классов из языка Simula.

Основная идея ООП – приспособить язык программирования для решения задач, создавая формы данных (классы), соответствующих специфике задачи. Таким образом, в отличие от процедурных языков, каким являлся Си, акцент в новом языке был сделан не на разработку алгоритмов, а на разработку данных для конкретной задачи. В результате в С++ разработано большое количество библиотек классов, что позволяет

включать в разрабатываемые программы существующие классы и легко адаптировать и повторно использовать хорошо проверенные коды программ.

Язык С++ поддерживает четыре парадигмы программирования: процедурную, модульную (язык Си), объектно-ориентированную (классы языка С++), обобщенную (шаблоны классов языка С++). В С++, по отношению к языку Си, введены новые понятия: объекты, классы, инкапсуляция, сокрытие данных, наследование, полиморфизм.

С++ — язык для профессиональных программистов, решающих сложные прикладные задачи, и начинать писать на нем программы новичку достаточно сложно. В данном пособии упор сделан на синтаксис, семантику, структуры данных, работу с указателями, файлами, источники возникновения ошибок, особое внимание уделено структурному и модульному программированию, взаимодействию функций.

При изучении языка можно использовать компиляторы Turbo С++ и Borland С++— программное обеспечение, разработанное компанией Borland
International, или Visual Studio С++ — программный продукт, разработанный для программирования на С++ компанией Microsoft.

1.2. Краткая характеристика языка С++

Большой набор операторов и операций С++ позволяет писать компактные и эффективные программы. Однако такие мощные средства требуют от нас осторожности, аккуратности и хорошее знание языка со всеми его преимуществами и недостатками.

Основные особенности языка С++: он в полном объеме учитывает современную технологию программирования, имеет управляющие структуры, позволяющие создавать хорошо структурированные программы.

В нем предусмотрены:

- блочная структура программы;
- оператор ветвления ***if***;
- операторы повторения с предусловием ***while***, ***for***;
- оператор повторения с постусловием ***do***;
- оператор выбора варианта ***switch*.**

Язык использует стандартные типы данных:

- целый ***int*;**
- символьный ***char*;**
- вещественный ***float*, *double*.**

Кроме того, в нем предусмотрены такие типы данных, как:

- массивы, структуры, файлы;
- возможность создания более сложных структур – списков очередей, стеков, деревьев и т.д.

Язык С++ достаточно мал по объему. В нем отсутствуют многие средства, встроенные в другие языки:

- нет встроенных средств ввода/вывода (подобно процедурам *read* и *write* в Паскале);
- нет встроенных средств для работы с символьными строками;
- нет встроенных средств динамического распределения памяти и т.д.

Все эти действия реализованы в библиотеках стандартных функций, входящих в системное окружение языка С++.

Программа на С++ состоит из одной или нескольких программных единиц – функций, которые могут быть скомпилированы вместе или отдельно. Аргументы в функции передаются по значению (посредством копирования, когда функция не может изменить фактический аргумент в вызывающей программе), или по адресу – через указатель и по ссылке (когда функция меняет фактическое значение).

В С++ хорошо реализован механизм указателей на переменные и функции. Указатель - это переменная для хранения машинного адреса некоторой переменной или функции. Поддерживая арифметику указателей, язык С++ позволяет осуществлять доступ к адресам памяти и создавать высокоэффективные программы. Очевидно, что при этом требуется особая осторожность при работе с адресами.

Еще одна особенность С++ — слабый контроль типов, который позволяет создавать высокоэффективные программы. От программиста требуется аккуратность при написании выражений. Вольное обращение с переменными недопустимо. Компилятор в ряде случаев не сообщает об ошибках несовместимости типов переменных в выражениях, автоматически приводя их к одному типу (например, вещественной переменной можно присвоить символьное значение, а целой – вещественное). Поиск таких алгоритмических ошибок достаточно сложен, а необнаруженные в программе ошибки приводят к тяжелым последствиям.

Надеемся, что приведенное краткое указание особенностей языка С++ настроит вас на то, что прежде, чем приступать к разработке программ, следует внимательно ознакомиться с идеологией С++. Тогда ваши, пусть сначала и небольшие победы, придадут вам уверенность в том, что вы можете стать хорошим программистом.

1.3. Синтаксис языка

Файл, содержащий программу на языке C++, создается с помощью текстового редактора среды разработки и записывается на диск с заданным нами именем и типом «.cpp».

При разработке программы в качестве разделителей используются символы табуляции, пробелы, перевода строки и перевода формата.

В программу при необходимости включают комментарии в двух видах: – если комментарий занимает несколько строк, то он заключается в символы

```
/* текст
      комментария  */
```

– если комментарий убирается в одну строку, то он предваряется символами

```
// текст   комментария
```

В алфавит языка входят символы:

- цифры 0…9
- латинские строчные и прописные буквы a…z, A…Z
- знаки операций *, /, +, -, !, ^, &, %, ?, :, ~
- скобки ( ), [ ], { }
- другие символы . , \ _ ; и т.д.

Для обозначения имен переменных, функций и типов данных используются идентификаторы (последовательность букв, цифр и знаков подчеркивания, не начинающаяся с цифры). Имена переменных и функций должны быть мнемоническими, т.е. сообщать об их содержании.

1.4. Основные типы данных

К основным типам данных относят типы целых чисел (int), символов (char) и чисел с плавающей точкой (float, double).

```
char a, b, c;
int x, y_10;
float f1, f2;
double eps;
```

На их основе строят производные типы данных, например, с помощью модификаторов signed (знаковое), unsigned (беззнаковое), long (длинное), short (короткое). Тип int может модифицироваться с помощью каждого из них. Тип char с помощью signed или unsigned. Тип double с помощью long.

Отметим, что базовые числовые и символьные типы по умолчанию являются знаковыми (согласно стандарту ANSI/ISO) и явное применение signed не обязательно.

Если в объявлении присутствует модификатор без явно указанного типа данных, предполагается, что тип данных int.

Необходимо помнить, что стандарт ANSI/ISO устанавливает минимальный диапазон значений типа, а не размер в байтах, который может отличаться на разных машинах в зависимости от архитектуры процессора.

Всегда можно определить конкретный размер типа, используя *sizeof* – операцию определения размера типа или переменной с помощью следующих выражений - *sizeof (short)* или *sizeof (ch).*

В C++ реализован еще один специальный тип – *void* (пустой), который используется только при объявлении функций (когда они не возвращают значений) или указателей.

Обратите внимание! Нельзя объявить переменную типа void и использовать описатель void в операциях приведения типа!

1.5. Константы

Константа – это число, символ или строка символов. Константы используются в программировании для задания постоянных величин.

В языке C++ различают 5 типов констант.

1. **Целые константы** представляются в трех видах:
- десятичные (используются цифры *0…9*);
- восьмеричные (цифры *0…7*, первой цифрой должен быть 0);
- шестнадцатеричные (цифры *0…9* и буквы *a…f* или *A…F* для значений *10…15,* должны начинаться с символов *0x* или *0X*).

Примеры разных константных значений: *123*, *0123*, *0x123*.

Если требуется константа, значение которой превышает наибольшее машинное целое (со знаком в случае десятичных констант и без знака в случае восьмеричных и шестнадцатеричных), то она представляется как длинное целое.

2. **Длинные целые константы** явно определяются латинской буквой *l* или *L* , стоящей в конце константы. Длинные целые константы могут быть десятичные, восьмеричные и шестнадцатеричные: *10L*, *012L*, *0XAL*, *0xal*, *0XaL*, *123456*.

Функция вывода на экран двух константных значений *23L,0xAL* типа *long* в формате ***%d*** - десятичное целое:

printf ***("%d, %d",23L,0xAL); //Результат: 23, 10***

3. **Вещественные константы** по умолчанию представляются числами с плавающей точкой двойной точности, т.е. как имеющие тип *double* (8 байт), и состоят из следующих частей - целой части, десятичной точки, дробной части, символа экспоненты *e* или *E*, экспоненты в виде целой части (может быть со знаком).

Любая часть из нижеследующих пар (но не обе сразу) может быть опущена:

- целая или дробная часть;
- десятичная точка или символ *E(e)* и экспонента в виде целой константы. Например: *.75*; *.75e2*; *25.*; *2.e2*; *15e-3*.

Константу с плавающей точкой можно описать, используя тип *float* (4 байта).

4. **Символьные константы** состоят из одного символа *ASCII*-кода, заключенного в апострофы. Значение символьной константы равно коду представляемого ею символа.

Ряд символов кодов *ASCII* не имеет графического изображения, например, перевод строки или знаки табуляции. Для представления непечатаемых управляющих символов в языке C++ имеются два способа:

– используется сам код *ASCII*:

'\007' - код звукового сигнала
'\012' - переход к новой строке
'\015' - возврат каретки
' \033' - клавиша *ESC*

Обратите внимание! Номер символа должен быть записан в восьмеричном виде, т.е. коду символа предшествует символ нуля (заметим, что нули в первых позициях можно опустить, но сам код символа – восьмеричное число).

– используются специальные управляющие последовательности для записи некоторых символьных констант, приведенные в табл.1.

Таблица 1

	Символьное значение	Имя	8-ричный код	10-тичный код
Звуковой сигнал	'\a'	BEL	7	7
Перевод строки	'\n'	LF	12	10
Горизонтальная	'\t'	HT	11	9
табуляция	'\v'	VT	13	11
Вертикальная	'\b'	BS	10	8
табуляция	'\r'	CR	15	13

Возврат на шаг	'\f'	FF	14	12
Возврат каретки	'\\'	\	134	92
Перевод формата	'\"	‘	140	96
Обратная косая черта	'\'"	“	42	34
Апостроф	'\0'	NULL	0	0
Кавычки				
Нулевой символ				

Обратите внимание! При записи трех символов (обратной косой черты, апострофа и кавычек) перед ними ставится символ \ .

Вторая форма записи управляющих констант предпочтительней, т.к. во-первых, она нагляднее, а во-вторых, такие программы обладают свойством переносимости.

Таким образом, одно и то же символьное значение можно представить разными способами:

char ch1='A'; ***//эквивалентные значения***

char ch2='\101';

5. Строковые константы представляются последовательностью символов, заключенных в кавычки, например, *"stroka"*.

Строковые переменные представляют массив символов, тип их описывается как массив символов *char[ ]*, при этом в квадратных скобках указывается максимальный размер возможного значения строки. В конце строкового значения компилятор автоматически помещает нулевой символ *'\0'*, который служит признаком конца строки. Таким образом, размер массива хотя бы на единицу должен быть больше возможного числа символов в строке (нулевой символ, завершающий строку, также является элементом массива).

Объявление строк с их инициализацией имеет вид:

char s1[10]="Stroka"; ***//выделено 10 байт памяти***
char s2[]="Stroka"; ***//автоматически выделено 7 байт памяти***

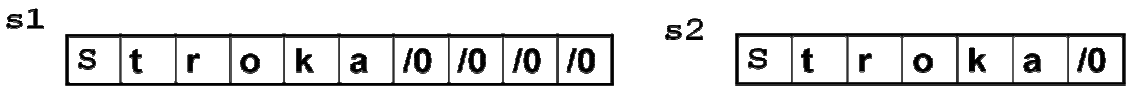


Рис.1. Выделяемая при инициализации память и ее содержимое

Напомним, что константные выражения, содержащиеся в тексте программы, обрабатываются на этапе компиляции, а не при ее работе. Таким образом, инициализация значений в операторах объявления сокращает время работы программы.

1.6. Объявления переменных

Объявления устанавливают связи между величинами и их идентификаторами и служат для компилятора источником информации о свойствах величин, используемых в программе.

Следует различать объявление и определение. Объявление (декларация) объявляет имя и тип объекта.

Определение выделяет для объекта участок памяти, соответственно инициализируя объект явно или неявно. Один и тот же объект может быть объявлен неоднократно, но определиться он должен только один раз. В большинстве случаев объявление является определением (не является, например, при объявлении функции, объявлении со спецификатором extern или объявлении формальных параметров функции).

Объявление переменной может включать следующее:

[спецификатор][квалификатор][модификатор] тип имя [инициализатор]

при этом части, заключенные в квадратные скобки, могут отсутствовать.

Тип переменной определяет размер выделяемой памяти, вид хранящейся в ней информации, диапазон значений, а также совокупность операций над значениями переменной.

Имя переменной – идентификатор, причем *символы верхнего и нижнего регистров рассматриваются как разные*.

Переменные можно инициализировать в инструкциях объявления, используя знак равенства или круглые скобки:

```
int max=100, a(1);          //два вида инициализации целого
char ch1= 'A', ch2(65);  //два вида инициализации символа
char newstr='\n';     //инициализация символом новой строки
char er[]="error";       //инициализация массива
                                  //er[0] er[1] er[2] er[3] er[4] er[5]
                  //'e'   'r'   'r'   'o'   'r'   '\0'
float x=1.25;               //инициализация вещественного значения
```

Спецификатор типа это класс памяти, определяющий:

- *область действия переменной* – ту часть программы, где переменную можно использовать (т.е. существует доступ к связанной с этой переменной областью памяти);

- *время жизни* может быть постоянным (в течение выполнения всей программы) и временным (в отдельном блоке);
- *область видимости идентификатора* – та часть программы, в которой есть доступ к области памяти, связанной с этим идентификатором. Обычно область видимости и область действия совпадают.

Подробно на классах памяти остановимся далее в главе 9.

К квалификаторам относятся:

const – делает переменную константой.

restrict – применяется к указателям, сообщает компилятору о том, что на данный участок памяти указывает единственный указатель, к которому применен квалификатор (на этот участок памяти могут указывать так же указатели основанные на исходном, т.е. указатель на указатель и т.д.).

volatile – сообщает компилятору о том, что значение переменной может поменяться в любой момент извне. Знание таких подробностей компилятором важно, т.к. большинство компиляторов оптимизирует код, предполагая при этом неизменность переменной.

Если имена локальных переменных внутри некоторого блока совпадают с именами глобальных переменных, то при обращении к глобальным переменным используют операцию *::* (доступ к области видимости глобальных переменных).

```
int x;   //внешняя переменная  x  невидима в функции fun()
         //доступ к ней внутри функции – с помощью операции ::

void fun();
{   int y;
    int x;      //объявление локальной переменной x
    x=2;        //присваивание значения локальной переменной
    ::x=3;      //присваивание значения глобальной переменной
    return;
}
```

1.7. Операции и выражения

Выражение состоит из одного или нескольких операндов и символов операций. Операнды – это переменные, константы, подвыражения, к которым применяются операции. Операндом может быть вызов функции, имеющей тип, т.е возвращающей значение через механизм *return*.

Каждое выражение в С++ имеет значение. Чтобы определить это значение, нужно выполнить операции в порядке их приоритетов. Например:

(x=7)+(c=10+x) ***// значение выражения равно 24 (7+17)***

Выражение со знаком *"="* имеет то же значение, что и переменная, стоящая слева от знака *"="*.

Выведем на экран значения выражений двумя способами, реализованными в С++ - с помощью объекта *cout* и функции форматного вывода *printf*:

```
cout<<"15>20="<<(15>20);
printf("%d,%d",15>20,5<10);
```

Экран

```
15>20 = 0
0,1
```

Реализации ввода/вывода данных в С++ посвящена глава 2.

Обратите внимание! Как в С++ отличить выражение со знаком присвоить (=) от оператора присвоить? Оператор в С++ заканчивается символом точка с запятой (;), это часть оператора (в отличие от Паскаля, где точка с запятой – разделитель операторов). Таким образом, x=1 – выражение, а x=1; – оператор.

1.7.1. Операции над основными типами данных

В языке С++ имеется следующие виды операций:

- **Арифметические операции:**

  \+ – сложение (2 операнда);
  \- – вычитание (2 операнда) или изменение знака (1 операнд);
  * – умножение (2 операнда);
  / – деление (2 операнда);
  % – деление по модулю (2 операнда).

  Отметим, что при делении целых чисел результат – целое, например:

```
int i , j, k;
i=5; j=3;
k=i/j;          //дробная часть отбрасывается,  k==1

                //где == означает тождественное равенство
```

Результат деления по модулю – остаток от деления величины слева от знака % на величину справа. Имеет смысл только для целых чисел:

*(x=5)%2 //**результат выражения равен 1***

Кроме того, в С++ реализованы унарные (выполняемые над одним операндом) операции инкремента (++) и декремента (--):

++ – увеличение значения на 1. Префиксная форма: *++i* – значением этого выражения является значение *i* после увеличения на *1*. Постфиксная форма: *i++* – значением выражения является значение *i* до увеличения на *1*;

-- – **уменьшение значения на 1**. Префиксная форма : *--i* , постфиксная *i--* .

int i=1, j=1;
*int n=i++; //**n==1,i==2 (знак == означает равно)***
*int m=++j;//**m==2, j==2***

Операции имеют приоритет выполнения. Например, выражение *i+++j* эквивалентно *(i++)+j* , т.е. приоритет операции инкремента больше приоритета операции сложения. Если *i=1*, *j=5* , то результат выражения равен *6,* а значение *i* равно 2.

Операции ++ и -- удобно использовать для организации счетчиков.

Обратите внимание! Если в результате операции ++ полученный результат не умещается в ячейку памяти переменной, автоматически происходит сброс (заполнение нулями всех разрядов) и продолжается новое заполнение ячейки, начиная с младшего разряда. Приведем пример бесконечного цикла (в С++ значения, отличные от 0, есть истина):

int i=0; while (1) i++;

- **Побитовые операции**

К переменным целого типа применимы побитовые операции, выполняемые над значениями битовых разрядов:

~ (тильда) – побитовое отрицание (дополнение до единицы, один операнд):

mask=12;
*cout<<~mask; //**результат выражения -13***

Значение выражения *~mask* содержит *1* во всех разрядах, в которых *mask* содержит *0*, и *0* во всех разрядах, в которых *mask* содержит *1*.

Обратите внимание! Для хранения знака числа выделяется старший левый разряд, 15-й при двухбайтовом или 31-й при

четырехбайтовом типе *int*. Положительное число кодируется 0, отрицательное -1 (минус 1).

>> - сдвиг вправо (2 операнда**):**

k>>i – двоичное представление *k* сдвигается вправо на *i* разрядов.

Если *k* объявлено как беззнаковое целое (*unsigned*), то сдвиг будет логическим, т.е. освобождаемые слева разряды будут заполняться *0*. Для чисел со знаками результат зависит от типа ЭВМ. Скорее всего, сдвиг будет арифметическим, т.е. освобождающиеся слева разряды будут заполняться значением знакового разряда.

<< - сдвиг влево (2 операнда**):**

k<<i - двоичное представление *k* сдвигается влево на *i* разрядов. Освобождающиеся при этом правые разряды заполняются нулями.

Операции сдвига выполняют эффективное умножение и деление на степени числа 2:

k<<5 – эквивалентно операции умножения: $k*2^5$

k>>5 – эквивалентно операции деления: $k:2^5$, если число *k>0*.

int x=10<<2; ***//x==40,*** ***$x=10*2^2=40$***

int x=10>>2; ***//x==2,*** ***$x=10:2^2=2$***

Можно провести аналогию для 10-й системы счисления - сдвиг десятичной точки при умножении и делении числа на 10.

Еще раз отметим, что если при побитовом сдвиге в старший знаковый разряд попадает единица, получаемый результат – отрицательное число.

& - логическое побитовое «И» (2 операнда):

x&y – значение выражения содержит 1 в разрядах, в которых *x* и *y* содержат 1, и 0 во всех остальных разрядах .

| - логическое побитовое «ИЛИ» (2 операнда):

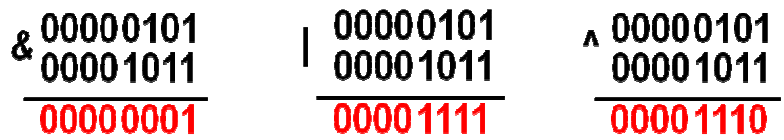
x|y – значение выражения содержит 1 во всех разрядах, в которых *x* или *y,* либо оба, содержат 1, и 0 во всех остальных .

^ - (код 136_8) побитовое исключающее ИЛИ:

x^y – значение выражения содержит 1 в тех разрядах, в которых *x* и *y* содержат разные двоичные значения, и 0 в остальных разрядах.

Проиллюстрируем значения следующих выражений:

5&11==1 *5 |11==15* *5 ^11==15*



Обратите внимание! Операнды логических операций &, |, ^ должны иметь целый тип. Бит знака, если он есть, также участвует в операции. Над операндами выполняются преобразования типов к целому по умолчанию.

1.7.2. Операции присваивания

В C++ семантика операции присвоить имеет вид:

<Леводопустимое выражение> = <Выражение>.

Обратите внимание! Слева от операции присвоить в C++ может стоять некоторое выражение, называемое леводопустимым.

В операции присваивания *x=b* имя переменной *x* является частным случаем леводопустимого выражения. В общем случае это ссылка на некоторый объект. Леводопустимое выражение обеспечивает получение или изменение значения объекта. К леводопустимым выражениям относятся:

- имена скалярных и индексированных переменных;
- имена указателей;
- имена полей структурированных данных, содержащих операции точка (.) или косвенной адресации (->);
- ссылки на объекты;
- выражения с операцией разыменовывания (*);
- вызовы функций, возвращающих ссылки на объекты.

Приведем примеры операций присваивания леводопустимым выражениям (*, ->, & - операции):

p=j/i;* *//слева – выражение разадресации указателя***
pst->ball=5; ***//имя поля структуры с косвенной разадресацией***
int &alt=x; ***//присваивание значения ссылке на объект***

Приведем примеры праводопустимых выражений, которые нельзя использовать в левой части оператора присвоить (они являются

значениями): имя функции и имя массива (это адреса их расположения в памяти), имя константы, вызов функции, не возвращающей ссылки.

Операции присвоить выполняются справа налево:

x=y=z=10 ***//сначала z=10, затем y=(z=10), т.е. y=10, далее x=10***

В C++, кроме операции =, имеются следующие комбинированные операции присваивания:

+= , -= , *= , /= , %= , >>= , <<= , &= , ^= , |= .

Примеры использования комбинированных операций присвоить:

x+=2	эквивалентно	*x=x+2*
x-=z	~	*x=x-z*
y=(2+x)*	~	*y=y*(2+x)*
y%=10	~	*y=y%10*
y/=10	~	*y=y/10*
z>>=1	~	*z=(z>>1)*
m<<=mask	~	*m=(m<<mask)*
b^=a	~	*b=(b^a)*
b|=(a%2)	~	*b=(b|(a%2))*

При присваивании тип правого операнда преобразуется к типу левого операнда. Специфика этого преобразования зависит от обоих типов и будет подробно рассмотрена в п. 1.8.

Обратите внимание! В C++ операция присваивания вырабатывает значение, которое может быть использовано далее в вычислении выражения.

Например, при вычислении выражения

(sum=sum+count)<max

скобки обязательны, иначе в соответствие с приоритетом операций вычисляется совсем другое выражение. В кружках показан порядок выполнения операций при вычислении выражения со скобками и без скобок:

Приоритеты выполнения операций приведены в табл.2 на стр. 20.

1.7.3. Операции отношения и логические операции

Значениями выражений, содержащих операции отношений и логические операции, являются *истина* и *ложь*. Значение *ложь* представляется нулевым целым значением, а *истина* – любым ненулевым значением. Тип результата логического выражения в C++ *int*.

- **операции отношения**

= = (тождественное равенство), **! =** (неравенство), **<** , **>** (меньше, больше), **<=** , **>=** (меньше или равно, больше или равно).

- **логические операции**

! (логическое отрицание), **||** (логическое ИЛИ), **&&** (логическое И).

Операнды логических операций могут иметь целый, плавающий тип или быть указателями. Типы первого и второго операндов в операциях отношения и логических операциях могут различаться. Над операндами производятся преобразования типов по умолчанию в соответствие определенным правилам преобразования типов (п. 1.8).

Обратите внимание! В выражениях, содержащих логические операции, сначала вычисляется левый операнд. Если его значения достаточно для определения результата операции, то правый операнд не вычисляется. Например:

```
x==y || x==z   //если x == y, то значение всего выражения
               // истина (1), и второй операнд не вычисляется
x<y && y<z     //если x>=y, то значение всего выражения
               //ложь (0). Второй операнд не вычисляется
```

Скобки в данных выражениях не нужны, т.к. приоритет операций отношения (==) выше приоритета логических операций.

1.7.4. Дополнительные операции

- **Операция определения размера объекта в байтах** применима к одному операнду**,** выполняется на этапе компиляции и имеет вид:

sizeof выражение или ***sizeof (тип)***

Результат операции – число байт, требуемых для размещения значения выражения или размещения объектов указанного типа. Результат операции *sizeof* имеет тип *unsigned*.

Обратите внимание! Если операция определения размера применяется к типу данных, то круглые скобки обязательны!

Операция *sizeof* определяет размер типа выражения, а не его значение, например:

```
sizeof  5.4==8          //размер типа выражения - double
sizeof (int)==4         //размер типа int
```

- **Операция преобразования типа** имеет один операнд:

(тип) выражение

Значение выражения преобразуется к типу, указанному в скобках:

```
int x;
float y=2.8;
x = 2*(int)y     //результат  x==4
```

Если опустить операцию преобразования типа, то для оператора *x=2*y*; производится преобразование типов по умолчанию. Значение выражения с плавающей точкой *2*y* преобразуется к типу *int* отбрасыванием дробной части. В результате получим значение *x=5*.

- **Условная операция ?:** имеет три операнда

<выражение 1>?<выражение 2>:<выражение 3>

и выполняется следующим образом. Если выражение 1 истинно (имеет ненулевое значение), то вычисляется значение выражения 2, иначе вычисляется значение выражения 3. Результат выполнения условной операции - вычисленное значение одного из выражений – 2 или 3. Отметим, что всегда вычисляется только одно из выражений 2 или 3.

Примеры вычисления абсолютного и максимального значений:

```
i=(i>0)?i:-i;
max=(x>y)?x:y;
```

- **Операция запятая** имеет два операнда:

<выражение1>, <выражение2>

Сначала вычисляется выражение 1, затем выражение 2. Результат выполнения операции имеет значение и тип выражения 2, например:

y=(i=1),(j=i+1); //результат y==2

Операция запятая обычно используется в цикле *for*, позволяя включать в его спецификацию несколько инициализирующих или корректирующих выражений.

for (int i = 0 , j = 0 ; i + j < 50 ; i++ , j += 2) printf("Hello!\n");

В данном примере цикл for имеет две управляющие переменные, объявленные внутри цикла.

- **Операция взятия адреса &**

Результат этой унарной операции – адрес операнда, перед которым стоит знак &.

*int *p, x; //p - указатель на целое,x - целое*
p = &x; //p получает значение адреса x

- **Операция разадресации *** (обращение по адресу)

Результат операции – содержимое по указанному адресу, например:

*int *p, x, y;*
p = &x;
*y = *p; //y получает значение по адресу p, т.е. y==x*

Все рассмотренные операции разбиваются на 16 групп по приоритетам их выполнения (табл. 2). Операции в группе с меньшим номером имеют более высокий приоритет над операциями из групп с большими номерами. Если в выражении присутствуют операции из одной группы и отсутствуют скобки, то, в зависимости от группы, они могут выполняться как слева направо, так и справа налево (свойство ассоциативности).

Таблица 2

При-оритет	Обозначение	Операция	Ассо-циативность
1	() [] →	Вызов функции Ссылка на элемент массива Ссылка на элемент структуры с	 →

	. ::	помощью указателя Ссылка на элемент структуры Указание области видимости	
2	! ~ - ++ -- & * sizeof (тип) new delete	Логическое отрицание Побитовое отрицание Унарный минус Увеличение на единицу Уменьшение на единицу Взятие адреса Обращение по адресу Определение размера в байтах Преобразование типа Динамическое выделение памяти Освобождение динамической памяти	←
3	.* ->*	Разыменование указателя на компоненту класса Доступ к компоненту класса через указатель	→
4	* / %	Умножение Деление Деление по модулю	→
5	+ -	Сложение Вычитание	→
6	>> <<	Побитовый сдвиг вправо Побитовый сдвиг влево	→
7	< <= > >=	Меньше Меньше или равно Больше Больше или равно	→
8	== !=	Равно Не равно	→
9	&	Побитовое И	→
10	^	Побитовое исключающее ИЛИ	→
11	|	Побитовое ИЛИ	→
12	&&	Логическое И	→
13	||	Логическое ИЛИ	→
14	?:	Условная операция	←
15	= *= /= %=	Присваивание	

	+= ~= <<= >>= &= ^= |=		←
16	,	Операция запятая	→

Отметим некоторые особенности обработки выражений.

Порядок обработки операндов в рамках одной операции может быть разным в зависимости от компилятора, однако для 4-х операций *(&& || ?: ,)* гарантируется, что 1 операнд будет обрабатываться первым. В связи с этим, если в каком-либо выражении имеется присваивание значения переменной и повторное ее использование, то результат может оказаться разным для разных компиляторов, поэтому подобных случаев следует избегать. Например, результат выполнения следующих кодов в Microsoft Visual Studio далеко не очевиден:

v=(i=2)+(i+5); ***//результат v==9, хотя из записи выражения***
//очевидней ожидать результат v==7

v=(i=2)+(++i); ***//результат v==6,***
//(очевидный результат v==5)

Таким образом, следует избегать в выражениях использования одной и той же переменной, если к ней в подвыражении применяются операции, изменяющие ее значение.

1.8. Преобразование типов

Разнообразие типов данных в C++, с одной стороны, предоставляет широкие возможности программисту, но, с другой стороны, усложняет задачу обработки данных. Сложение двух чисел типа *short* может выполняться с помощью иных машинных команд, чем сложение чисел типа *long*. C++ допускает наличие в выражении операндов разных типов. А так как в C++ имеется 11 типов целочисленных значений и три типа с плавающей точкой, компьютеру приходится обрабатывать множество различных ситуаций при вычислении выражений.

Компилятор использует набор правил для автоматического преобразования типов:

1. Операнды типа *char* и *short* преобразуются в значение типа *int*, операнды типа *unsigned char* или *unsigned short* - в значение типа *unsigned int*, а операнды типа *float* - в значение типа *double*.

2. Если операция выполняется над данными двух различных типов, обе величины приводятся к "высшему" из двух типов. Этот процесс называется "повышением" типа.

3. Последовательность имён типов, упорядоченных от "высшего" типа к "низшему", выглядит так: *double, float, long, int, short, char*. Применение ключевого слова *unsigned* повышает ранг соответствующего типа данных со знаком.

4. В операторе присваивания конечный результат вычисления выражения в правой части приводится к типу переменной, которой должно быть присвоено это значение. Данный процесс может привести как к "повышению", так и к "понижению" типа, когда величина приводится к типу данных, имеющему более низкий приоритет.

"Повышение" типа обычно происходит гладко, в то время как "понижение" может повлечь за собой потерю информации. Причина этого проста: полученное значение целиком может не поместиться в элементе данных низшего типа. Например, в переменную типа *unsigned char* нельзя поместить значение, большее 255.

Приведем пример обработки данных с автоматическим "повышением" и "понижением" их типов:

```
#include <stdio.h>      //подключение библиотеки для ввода/вывода
int main(void)
{
   char ch;
   int i;
   float f;

   f = i = ch= 'A';                                  //строка 8
   printf("ch=%c, i=%d, f=%2.2f \n", ch, i, f);          //строка 9
   ch += 1;                                          //строка 10
   i = f+2*ch;                                       //строка 11
   f = 2.0*ch+i;                                        //строка 12
   printf("ch=%c, i=%d, f=%2.2f \n", ch, i, f);     //строка 13
   ch =2.0e30;                                          //строка 14
   printf("Теперь ch=%c \n", ch);                            //строка 15
   return 0;
}
```

На экране мы увидим следующие результаты работы программы:

ch=A, i=65, f=65.00

ch=B, i=197, f=329.00
Теперь ch=

Строка 8: величина *'A'* присваивается символьной переменной *ch*. Переменная *i* получает целое значение, являющееся преобразованием символа *'A'* в целое число, т. е. *65*. Переменная *fl* получает значение *65.00*, являющееся преобразованием числа *65* в число с плавающей точкой.

Строка 9: реализует вывод переменных ch, *i*, *f* в формате символа *%c*, целого *%d* , вещественного *%2.2f* с точностью двух цифр дробной части.

Строка 10: значение символьной переменной *'A'* преобразуется в целое число *65*, к которому добавляется *1*. После этого получившееся число *66* преобразуется в код символа *'B'* и помещается в переменную *ch*.

Строка 11: при умножении на *2* значение переменной *ch* преобразуется в целое число *66*. При сложении с величиной переменной *f* получившееся в результате число *132* преобразуется в число с плавающей точкой. Результат *197.00* преобразуется в число целого типа и присваивается переменной *i*.

Строка 12: перед умножением на *2.0* значение переменной *ch ('B')* преобразуется в число с плавающей точкой. Перед выполнением сложения величина переменной *i (197)* преобразуется в число с плавающей точкой, и результат выражения *329.00* присваивается переменной *f*.

Строка 14: здесь производится попытка осуществить процесс "понижения" типа - переменной *ch* присваивается довольно большое число. В результате "усечения" 8-байтового значения *2.0e30* до *1* байта переменной типа *ch* мы получили код некоторого символа, не имеющего графического изображения.

Лучше всего вообще избегать преобразования типов, особенно его "понижения". Если это все же необходимо, то следует явно использовать операцию преобразования типа:

```
int  m1, m2;
m1 = 1.5+1.8;                  //m1==3
m2 = (int)1.5+(int)1.8;              //m2==2
```

В 1-м операторе присваивания типы преобразуются *автоматически*. В результате сложения чисел *1.5* и *1.8* получаем *3.3*, отбрасыванием дробной части это число приводится к типу *int* переменной *m1*.

Во 2-м операторе присваивания числа *1.5* и *1.8 явно* преобразуются в целые числа отбрасыванием дробной части, и переменная *m2* получает значение *2*.

2. Операторы ввода/вывода

Ввод/вывод в C++ реализуется либо с помощью функций, унаследованных от библиотек Си, либо с помощью потоков C++. Смешивать эти два способа в одной программе не рекомендуется.

Функции ввода/вывода делятся на три класса:

1. *Ввод/вывод верхнего уровня*, обеспечивающий *буферизацию* работы с файлами, когда обмен данными между программой и файлом осуществляется через промежуточный буфер, расположенный в оперативной памяти. Буферизация ввода/вывода выполняется автоматически, она позволяет ускорить выполнение программы за счет уменьшения количества обращений к сравнительно медленно работающим внешним устройствам. Для пользователя файл, открытый на верхнем уровне, представляется как последовательность считываемых или записываемых байтов, что отражает понятие "*поток*" (англ. stream). Функции верхнего уровня одинаково работают в различных операционных средах, с их помощью можно писать переносимые программы.

2. *Ввод/вывод для консольного терминала*, когда возможности функций ввода/вывода верхнего уровня распространяются на соответствующий класс устройств, добавляя новые возможности - читать или записывать на консоль (терминал) или в порт ввода/вывода (например, порт принтера). Для ввода или вывода с консоли устанавливаются некоторые дополнительные режимы, например, ввод с эхо-печатью символов или без нее, установка окна вывода, цветов текста и фона. Функции для консоли и порта являются уникальными для компьютеров, совместимых с IBM-PC.

3. *Ввод/вывод низкого уровня*, где функции низкого уровня из стандартной библиотеки применяются при разработке собственных подсистем ввода/вывода. Большинство функций этого уровня переносимы в рамках некоторых систем программирования на Си, в частности, относящихся к операционной системе Unix или совместимым с ней.

Для использования функций ввода/вывода высокого уровня в стиле Си необходимо подключить к программе (до функции main) заголовочный файл *stdio.h* или *cstdio* с помощью директивы компилятору:

#include <stdio.h>

В языке C++ средства ввода/вывода данных реализованы также в библиотеках *iostream.h* или *iostream* (библиотеки классов в ООП).

Познакомимся с работой функций ввода/вывода библиотеки языка Си *stdio.h* и объектами ввода/вывода библиотеки языка C++ *iostream.*

2.1. Функции ввода/вывода высокого уровня библиотеки stdio.h

Каждой программе при работе с библиотекой *stdio.h* предоставляются три стандартных потока, соединенных по умолчанию с консолью. Указатели на эти потоки возвращают макрокоманды *stdin, stdout, stderr:*

- *stdin* – стандартный поток ввода с клавиатуры;
- *stdout* – стандартный поток вывода на экран;
- *stderr* – поток вывода сообщений об ошибках.

В библиотеке *stdio.h* находятся три группы функций ввода-вывода:

- *scanf, printf* – форматный ввод/вывод числовых и смешанных значений;
- *getchar, putchar* – ввод/вывод символов;
- *gets, puts* – ввод/вывод строк;
- *perror* – вывод сообщения об ошибке в *stderr.*

2.1.1. Функции форматного ввода/вывода

Данные функции перешли в C++ из языка Си. Они обеспечивают гибкость при вводе/выводе данных, особенно смешанных типов, но при этом ненадежны как раз из-за отсутствия контроля типов. Тем не менее, они успешно применяются при разработке приложений, поэтому остановимся на них подробнее.

Функция форматного ввода данных имеет вид**:**

scanf("Управляющая строка", <список ввода>);

Управляющая вводом строка (формат ввода) - строка символов, показывающая, в каком виде должны быть введены переменные *списка ввода*. Она содержит только *спецификаторы формата*, распознаваемые по символу процент (%), и пробельные символы. Список ввода — адреса переменных, в которые вводятся значения.

Спецификаторы формата – это знак % и специальный символ:
% d, %i - десятичное целое;
% c – символ;
% s - строка символов;
% e - число с плавающей точкой, экспоненциальная запись;
% f - число с плавающей точкой, десятичная запись;

% g - используется вместо записей %e или %f; выбирается наиболее короткая из них;
% u - десятичное целое без знака *(unsigned)*;
% o - восьмеричное целое без знака (используется, в том числе, для указателей);
% x - шестнадцатеричное целое без знака.

Например, ввод двух целых значений в переменные *x* и *y* с помощью функции форматного ввода:

scanf("%d %d", &x, &y);

Функция форматного вывода данных имеет вид:

printf ("Управляющая строка", <список вывода>);

Управляющая выводом строка (формат вывода) содержит, в отличие от управляющей строки функции *scanf,* в том числе текстовые символы, выводимые на экран в тех же позициях, в которых они стоят в управляющей строке, а также спецификаторы формата списка вывода. *Список вывода* — переменные и выражения, значения которых выводятся на экран. Вывод на экран значений двух целых переменных через пробел (между спецификаторами формата *%d* вставляем для этого пробел):

printf("%d %d", x, y);

Вывод на экран значений переменных и текстовой информации:

printf("%d hours and %d minutes", hour, minutes);

"%d часов и %d минут" - это *управляющая строка*; *hour, minutes – аргументы вывода*. Результат работы функции *printf* отображается на экране в виде:

12 hours and 30 minutes

Обратите внимание! Количество спецификаторов формата (%управляющий символ) в управляющей строке должно совпадать с количеством вводимых или выводимых значений списков ввода/вывода.

В функции *printf* может отсутствовать список аргументов, тогда в управляющей строке отсутствуют также спецификаторы формата, а содержатся только текстовые символы, выводимые на экран:

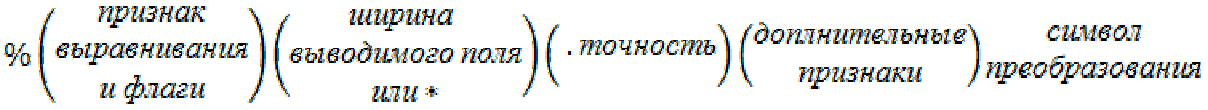
printf("What is it?");

Если необходимо вывести на печать символ %, то в управляющей строке его нужно указать дважды:

```
procent=3*9;
printf("%d%% изделий изготовлено к сроку", procent);
                                    //27% изделий изготовлено к сроку
```

Между знаком формата % и специальным символом в управляющей строке могут стоять модификаторы формата, обеспечивающие гибкость вывода информации.

Общий вид спецификации преобразования для функции *printf*:



- По умолчанию при отсутствии *флага* результат выравнивается по *правой* границе выводимого поля.
- Виды *флагов*:
  "–" (минус) - выравнивание по *левому* краю;
  "+" - печать знака, если аргумент знакового типа (не для *unsigned*);
  "#" - для формата *%o* и *%x* - печать лидирующих символов ***0*** или ***0x***;
  для формата *%f* и *%e* - печать десятичной точки.
- *Ширина выводимого поля* определяет минимальное число выводимых символов; если число символов явно *превышает* заданную *ширину*, то поле для печати будет расширено; если символов *меньше,* то недостающие символы будут заполнены пробелами. Символ * или отсутствие явно заданной ширины обозначает, что число выводимых символов будет определяться текущим значением переменной.
- *Точность* определяет число печатаемых цифр после точки для форматов *%f* и ***%e*** и число печатаемых символов для строк.

Приведем примеры функций вывода значений разных типов в указанном формате:

```
int x=10, ch='S';
float y=10;
char a[]="the first programm";
      //1-я строка экрана
printf("%5d,%5o,%#5x\n",x,x,x);
```

Экран

```
   10,   12,  0xa
   65 0101 0x41
S,S    ,    S,
10.00, 1.000000e+001
the first programm
the first programm
       the f
the f       !
```

```
        //2-я строка экрана
printf("%5d%#5o%#5x\n",65,65,65);
        //3-я строка экрана
printf("%c,%-5c,%5c,\n",ch,ch,ch);
        //4-я строка экрана
printf("%2.2f,%e\n",y,y);
        //вывод строкового значения в 4-х разных форматах
printf("%12s\n%-12s\n%12.5s\n%-12.5s!\n",a,a,a,a);
```

Обратите внимание! При работе функции *printf* используется *стековая память*. Стек работает по принципу: ***L***ast ***I***n - ***F***irst ***O***ut ("последним пришел – первым вышел"), т.е. доступ к *первой* помещенной в стек компоненте осуществляется в самую *последнюю* очередь. Вычисление выражений списка вывода при работе *printf* происходит с ***конца.***

Рассмотрим эту особенность работы функции *printf* на следующих примерах (стрелкой указано направление вычисления выражений, а в комментариях - результат работы соответствующей функции):

```
char ch='a';
printf("%c %c %c %c\n", ch, ch+1, ch+2, ch+3);//a b c d
```


Сначала вычисляется выражение *ch+3* и помещается в стек. Затем по такому же принципу вычисляются остальные выражения. На экран значения выражений выводятся в обратном порядке (сначала *ch*, затем *ch+1* и т.д.).

В следующих примерах явно видно использование стека при работе функции *printf:*

```
char ch='a';
printf("%c %c %c %c\n", ch,ch+1,ch+2,ch=ch+3);//d e f d
```


Сначала переменной *ch* присваивается новое значение - она увеличивается на 3 (т.е. *ch*=='*d*'). Все дальнейшие вычисления происходят с новым значением переменной.

```
char ch='a';
printf("%c %c %c %c\n",ch,ch++,ch++,ch++); //d c b a
```

Значение четвёртого выражения равно *'a'*, после чего *ch* увеличивается на 1. Аналогично значение третьего выражения равно *'b'*, второго - *'c'*, первого - *'d'*.

```
char ch='a';
printf("%c %c %c %c\n",ch,++ch,++ch,++ch); //d d c b
```

Здесь *ch* увеличивается на 1 *перед* вычислением четвёртого выражения, поэтому его значение равно *'b'*. Аналогично значение третьего выражения равно *'c'*, второго – *'d'*, первого - *'d'*.

Рассмотренные примеры позволяют сделать вывод, что не следует использовать в качестве аргументов функции *printf* выражения, в которых переменные меняют свои значения - это может привести к ошибкам!

Функция *printf* имеет тип *int* и возвращает число успешно выведенных значений. В случае возникновения ошибки при выводе *printf* возвращает отрицательное значение.

Функция *scanf* используется в основном при вводе смешанных типов данных в некоторой стандартной форме или для ввода чисел с плавающей точкой.

Общая форма спецификации преобразования для *scanf*:

$$\%(*)(\textit{ширина})\begin{pmatrix}\textit{дополнительные}\\ \textit{признаки}\end{pmatrix}\begin{matrix}\textit{символ}\\ \textit{преобразования}\end{matrix}$$

- * – признак подавления присваивания, вводимое значение, определённое данной спецификацией, не присваивается никакой переменной;
- *ширина* – целое число, определяющее максимальное количество символов, считываемых из входного потока по данной спецификации.

Т.к. аргументами функции *scanf* являются *указатели на соответствующие типы*, то при формировании списка аргументов функции необходимо помнить два правила:

– если нужно ввести некоторое значение и присвоить его переменной одного из основных типов, то перед именем переменной требуется писать символ **&** (взятие адреса);

– при вводе значения строковой переменной символ **&** перед её именем не ставится, т.к. строка – это массив символов, а *имя массива по определению в C++ является указателем*, т.е. адресом его размещения в памяти.

Приведем примеры функции *scanf:*

```
int a, b;
//на вход подаются данные в виде строки:    135/85
scanf("%4d%*c%4d", &a, &b); //%*c – пропуск символа '/'
```

В результате *a==135, b==85,* символ '/' не присвоен никакой переменной.

```
char str_1[15], str_2[15] ;
unsigned e;
//на вход подаются данные в виде:  Высота=15мм
scanf("%6s%*c%d%2s", str_1, &e, str_2);
```

В результате *str_1=="Высота", e==15, str_2=="мм",* символ '=' не присвоен никакой переменной.

Функция *scanf* имеет тип *int* и возвращает число элементов, успешно получивших значение при вводе. Если при вводе произошла ошибка, *scanf* возвращает значение *EOF* (***E**nd **O**f **F**ile* - признак конца файла). Признак конца файла определён в файле *stdio.h* языка Си с помощью директивы *define* следующим образом:

```
#define   EOF    -1
```

т.е. *EOF* – константное значение, равное -1. Чтобы не зависеть от конкретного значения этой константы в разных реализациях языка, к этой константе принято обращаться по имени *EOF*.

Подробнее директива *define* будет рассмотрена в п. 17.2 учебного пособия.

2.1.2. Функции ввода-вывода символов

Для ввода/вывода символов используются функции *getchar* и *putchar* библиотеки языка Си *stdio.h.* Они реализуют простейший ввод символа с клавиатуры и вывод его значения на экран. Объявления данных функций (их прототипы) имеют вид:

```
int getchar(void); //ввод символа,функция без параметров
int putchar(int);  //вывод символа, параметр типа  int
```

Функция *getchar* позволяет читать символы из входного потока, пока не будет достигнут признак конца файла:

```
int i;
while ((i=getchar())!=EOF)
  {
          // обработка i
  }
```

Функция *getchar* не имеет параметров и возвращает значение через входящий в тело данной функции оператор *return <выражение>*. Для сохранения возвращаемого значения вызов функции должен быть праводопустимым выражением оператора присваивания: *int c=getchar;* При этом тип переменной, получающей возвращаемое функцией значение, совпадает с типом функции *getchar*.

Функция *putchar* возвращает значение через аргумент типа *int*. Пример ввода-вывода информации с использованием этих функций:

```
char ch;
ch=getchar();
while (ch!='\n')
       { if (ch!=' ')  putchar(ch);
          ch=getchar();
       }
```

Экран

```
a b c d
abcd
```

Следующий фрагмент определяет количество введенных символов:

```
char ch;
int i;
puts("Enter a line of symbols");
for (i=0; (ch=getchar( ))!='\n'; i++);
printf("\n %d symbols are entered\n", i);
```

В цикле *for* реализован подсчет (*i*++) значений выражения *ch=getchar( )*, отличных от символа перехода на новую строку *'\n'*, формируемого нажатием клавиши <enter>.

Оператор *putchar(getchar());* позволяет дублировать вводимые данные на экране, причем вводимый символ выводится на экран, не сохраняя своего значения в программе.

Если в задачах необходимо определить, какого вида символ мы ввели, следует подключить заголовочный файл *ctype.h*, который содержит макроопределения (п. 17.2), позволяющие проверять, к какому классу

принадлежат вводимые символы. Например, в нем содержится макроопределение

#define isdigit(x) ((x)=='+'||(x)=='-'? 1 : 0)

которое можно использовать для распознавания символов цифр:

```
c=getchar();
if (isdigit(c)) printf(" Цифра\n");
```

Перечислим ряд функций, включенных в файл *ctype.h*, которые возвращают *0*, если результат проверки на соответствующий символ отрицательный:

isalpha(c) проверка на букву;
isdigit(c) на цифру;
islower(c) на строчную букву;
isspace(c) на пустой символ(пробел , табуляцию, новая строка);
isupper(c) на прописную (заглавную) букву;
isalnum(c) на цифру или букву;
isgraph(c) на псевдографический символ (исключая пробел).

Например, псевдографические символы позволяет различать следующее макроопределение, содержащееся в файле *ctype*:

#define isgraph(c) ((c)>=0x21 &&(c)<=0x7e)

2.1.3. Функции ввода-вывода строк

Для ввода/вывода строк используются функции *gets, puts* библиотеки *stdio.h* языка Си, которые предназначены для ввода/вывода текстов с клавиатуры на экран.

Объявления функций ввода/вывода строк (их прототипы) имеют вид:

```
char *gets(char *);
int puts(char*);
```

Тип *char* * определяется как указатель на тип *char*. Функция *gets* в качестве параметра получает адрес, по которому вводит строковое значение, а также возвращает этот адрес через механизм return.

Ввод строки и вывод ее значения на экран имеет следующий вид:

```
char str[80];      //объявлен массив из 80 символов
```

```
gets(str);
puts(str);
```

Функция *gets* помещает текст, введенный с клавиатуры до символа новой строки, формируемый нажатием клавиши <*enter*>, по указанному адресу *str,* (напомним, что имя массива в C++ - адрес его размещения в памяти). Функция *puts* выводит строковое значение, хранимое по указанному в качестве параметра адресу *str*, на экран. Функция puts в случае успешного завершения возвращает последний выведенный символ и EOF в случае ошибки.

В функции *gets* реализованы два механизма возврата значения - через параметр и через оператор возврата *return*. Это дает возможность вызова ее следующим образом:

```
char str[80], *p;
p=gets(str);
puts(p);
```

где переменная *p* получает значение адреса переменной *str* , и значение *p* можно использовать при работе с массивом *str.*

Обратите внимание! Нумерация элементов массива в C++ начинается с нуля, т.е. можно записать выражение *p==&str[0].*

Обработка текстов, включая их ввод/вывод, будет подробно рассмотрена в главе 8.

2.2. Объекты ввода/вывода библиотеки iostream

Заголовочный файл *iostream (*поток ввода/вывода ***I****nput/****O****utput* ***S****tream*) языка программирования C++ содержит классы, функции и переменные для организации ввода-вывода. Он включён в стандартную библиотеку языка C++. Файл *iostream* при управлении вводом-выводом использует объекты *cin, cout, cerr* и *clog* для передачи информации в и из стандартных потоков ввода, вывода, ошибок (без буферизации) и ошибок (с буферизацией) соответственно. В нем определены операции помещения в поток << и чтения из потока >>. Являясь частью стандартной библиотеки C++, эти объекты также являются частью стандартного пространства имён - *std*.

Если ваш компилятор поддерживает старую библиотеку *iostream.h* языка C++, где используется расширение *.h* для заголовочных файлов, то программа вывода строки текста *"Hello!"* имеет вид:

```
#include <iostream.h>
int  main()
{  cout<<"Hello!";     //строка "Hello!" вставляется
                       //в выходной поток (на экран)
   return 0;
}
```

Если вы используете библиотеку *iostream*, тот же фрагмент кода имеет вид:

```
#include <iostream>
using namespace std;

int  main()
{   cout<<"Hello!";
    return 0;
}
```

Отметим, что практически все стандартные библиотеки C++ используют пространство имен *std*, поэтому вторая строка программы – включение пространства имен *std.* Пространство имен (*namespace*) — это способ объединения логически связанных объявлений под общим именем (глава 11). Если не включить директиву пространства имен *std*, придется каждый вызов функции из этих библиотек предварять префиксом пространства имен: *std::cout <<"Hello!";*

Обратите внимание! Операции извлечения из потока и помещения в поток имеют то же обозначение, что и побитовые операции сдвига, применимые к целым числам. Это возможно потому, что в классах *istream* и *iostream* операции извлечения из потока << и помещения в поток >> определены путем перегрузки операций сдвига.

Перегрузка операций – это переопределение их действия. При этом сохраняются количество операндов (свойства унарности, бинарности операций), их приоритет и направление выполнения.

Компилятор различает перегруженные операции по семантике их использования. Вопросы перегрузки функций будут рассмотрены в п. 10.5.

Обратите внимание! В следующих примерах выводимые выражения необходимо заключить в скобки:

```
cout << (i<j);     //приоритет операции < меньше приоритета <<
```

cout << (i<<j); ***//правая операция << означает сдвиг,***
//иначе вывод двух значений i и j

2.3. Функции ввода/вывода с консольного терминала

Функции ввода/вывода для консоли используют специфические особенности IBM-совместимого компьютера, такие как наличие специального видеоадаптера, и не являются переносимыми на другие типы компьютеров. Прототипы функций содержатся в файле *conio.h.*

Функции для работы с окном консоли, аналогичные библиотеке Crt Паскаля, приведены в табл. 3.

Таблица 3

Функция	Прототип и описание
window	*void window(int left, int top, int right, int bottom);* устанавливает текущее окно консоли по указанным координатам
clrscr	*void clrscr(void);* очищает текущее окно
clreol	*void clreol(void);* очищает текущую строку окна от позиции курсора до конца
delline	*void delline(void);* удаляет строку окна, в которой установлен курсор
insline	*void insline(void);* вставляет пустую строку в позиции курсора
gotoxy	*void gotoxy(int x, int y);* перемещает курсор в указанные столбец *x* и строку *y* окна
textbackground	*void textbackground(int newcolor);* устанавливает указанный фоновый цвет окна номерами 0-15 или названиями, определенными в *conio.h* (*BLUE, GREEN* и т. д.)
textcolor	*void textcolor(int newcolor);* устанавливает указанный цвет вывода текста в окне аналогично параметрам функции *textbackground*
wherex	*int wherex(void);* возвращает номер столбца окна, в котором находится курсор
wherey	*int wherey(void);* возвращает номер строки окна, в которой находится курсор

Кроме того, в файле *conio.h* описаны прототипы ряда специфичных функций:

- чтение символа с консоли без отображения (с отображением) его на экране, возвращает код символа:

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

- проверка буфера клавиатуры на наличие символов:

int kbhit(void);

Если буфер клавиатуры не пуст, *kbhit* возвращает ненулевое значение. В этом случае программа может прочитать символы из буфера клавиатуры при помощи фукнкций *getch* и *getche*. Если буфер клавиатуры пуст, функция возвращает нулевое значение. Например:

//бесконечный цикл, заканчивается нажатием любой клавиши:

```
while (!kbhit());
```

Функцию *kbhit* можно использовать для обработки данных в бесконечном цикле, пока не будет нажата какая-либо клавиша, выводя пользователю на экран сообщение о том, что программа работает:

```
while (!kbhit())
{
//обработка данных
}
cout<<"Is working";
```

3. Структура программы на языке С++

Программа на языке С++ представляет собой набор функций, среди которых одна должна иметь имя *main (главная).* Операционная система передает управление в программу пользователя на функцию с именем *main*, тем самым начиная выполнение программы. От других функций в программе функция *main* отличается тем, что ее нельзя вызвать изнутри программы, а ее параметры, если они существуют, обычно задаются ОС, хотя это необязательно.

В общем виде структура файла программы показана на рис.2 и состоит из директив препроцессору, объявлений и функций.

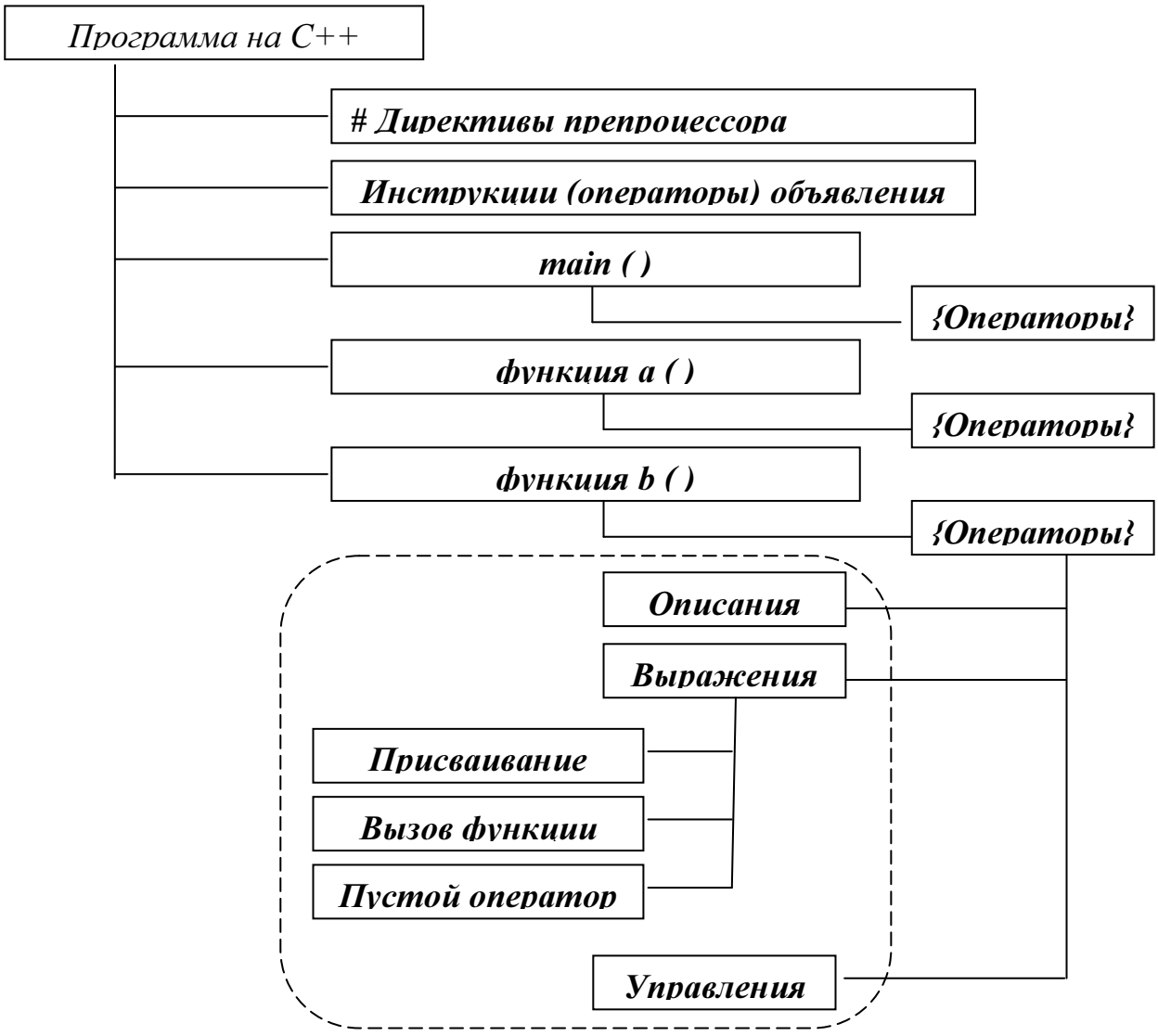


Рис.2. Общая структура программы

3.1. Директивы препроцессора

Если в качестве первого символа в строке программы используется символ #, то эта строка является *командной строкой препроцессора* C++.

Этапы обработки программы пользователя включают следующие шаги:

- подготовка исходного файла (*name.cpp*);
- компиляция и получение объектных файлов (*name.obj*);
- компоновка (сборка) объектных файлов для получения исполняемого файла (*name.exe*).

Компилятор C++ выполняет дополнительный шаг, выполняемый компьютерной программой, называемой ***препроцессором***. Эта программа обрабатывает исходный программный код, после чего он подается на компиляцию. Директивы препроцессору могут включаться в любом месте программы - их действие остаётся в силе до конца файла.

Директивы препроцессора позволяют изменить текст программы, например, заменить некоторые лексемы в тексте, вставить текст из другого файла, запретить компиляцию части текста и т.п.

Обратите внимание! После директив препроцессора точка с запятой не ставится.

Язык C++ не имеет встроенных операторов ввода-вывода, средств работы с математическими функциями, средств обработки строковых значений, функций распределения памяти и т.д. Эти средства содержатся в специальных файлах, содержимое которых становится доступным программе в результате включения в нее директивы препроцессору

#include <имя файла>

Файлы, включаемые в директиву *include*, называются заголовочными файлами (так как включаются в начало файла программы), и поддерживают разные семейства программных средств. Это могут быть файлы стандартных библиотек языков Си, C++, или файлы, созданные разработчиками для различных приложений. Например, файл *math.h* поддерживает различные математические функции языка Си, функции ввода/вывода сгруппированы в библиотеках, интерфейсы которых объявлены в файлах *stdio.h*, *io.h* и *conio.h*, *iostream*, *istream* и некоторых других. Расширение *.h* зарезервировано для старых заголовочных файлов языка Си, которые так же используются и в C++. Заголовочные файлы языка C++ не имеют никакого расширения.

Для работы в программе средств ввода/вывода в нее необходимо включить строки одного из двух приведенных далее вариантов:

#include <stdio.h>

#include <iostream>
using namespace std;

Директива *include* указывает компилятору, что в этом месте в текст программы нужно включить информацию, содержащуюся в соответствующем файле.

Напомним, что файл *stdio.h* перешел в C++ из языка Си и содержит макроопределения, прототипы функций, используемые в библиотеке ввода/вывода, информацию о средствах взаимодействия с файловой системой. Он содержит определения функций ввода/вывода *printf, scanf, putchar, getchar, puts, gets,* определение типа *FILE* (структуры, хранящей всю информацию о файле, с которым работает программа), определения некоторых констант – *NULL* (аналог нуля), *EOF*, *BUFSIZE* (признак конца файла и размер буфера потока в байтах) и т.д. Аббревиатура *stdio.h* - ***st***andar***d i***nput/***o***utput ***h***eader - стандартный заголовок ввода/вывода.

Отметим, что имя файла в директиве можно задавать в двух видах:

#include <stdio.h> ***//имя файла в угловых скобках***
#include "myfile.h" ***//имя файла в двойных кавычках***

Угловые скобки сообщают препроцессору, что файл следует искать в стандартных системных каталогах. Кавычки говорят о том, что сначала нужно искать файл в текущем каталоге, а затем в стандартных.

Практически все программы используют директивы препроцессора, указывающие компилятору выполнить то или иное действие в момент компиляции. С другими директивами мы познакомимся в главе 17.

3.2. Инструкции объявления

Все объекты в программе должны быть объявлены до их использования. Объявления устанавливают соответствие имени и атрибутов переменной, функции или типа.

Все внешние используемые объекты (константы, переменные, типы, функции) должны быть объявлены перед функцией *main*. Кроме того, существует возможность сделать переменные видимыми вне файла, в котором они объявлены, для этого в других файлах их тип и имя объявляется с использованием ключевого слова (спецификатора) *extern* без инициализации.

Перед использованием функции она также должна быть объявлена. С использованием функций в языке С++ связаны три понятия – определение функции (описание действий, выполняемых функцией), объявление функции (ее прототип) и вызов функции.

Определение функции – это ее заголовок и тело операторов. Заголовок задает *тип возвращаемого значения*, *имя функции*, *типы* и *число формальных параметров*. После заголовка точка с запятой не ставится. Тело функции определяет ее действие – это заключенные в фигурные скобки объявления переменных и операторы.

Объявление функции (прототип) содержит тип возвращаемого функцией значения, ее имя и типы параметров.

Обратите внимание! В объявлении можно не указывать имена параметров, здесь важны их типы, количество и последовательность объявлений. Объявление функции заканчивается точкой с запятой.

3.3. Функция main

Любая программа на языке С++ состоит из описания одной (*main*) или нескольких функций. Программа начинает выполняться с функции *main*, которая для выполнения необходимых действий вызывает другие функции.

Если программа работает с глобальными объектами (переменными, константами), которые должны быть доступны в любой функции файла, то их описывают перед функцией *main*. Если внутри какой-либо функции объявлена переменная, совпадающая с именем глобальной переменной, то доступ к глобальной переменной в функции осуществляется с помощью операции доступа к области видимости глобальной переменой (::).

Функция *main* по умолчанию имеет тип *int*. Ее выполнение заканчивается оператором *return*, который может иметь вид: *return 0;* или *return 1;* , или достижением закрывающей тело функции фигурной скобки. Возвращаемое функцией *main* значение не сохраняется.

Компилятор идентифицирует функции по круглым скобкам, внутри которых описываются параметры функции.

Функция *main*, как любая другая функция, может иметь параметры. В языке Си заданы два встроенных аргумента функции *main: argc* и *argv*. Заголовок функции *main* с параметрами в этом случае имеет вид:

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

Очевидно, встает вопрос – каким образом функция *main* получает значения параметров для своей работы? Их задает пользователь в

командной строке. Подробнее с данным механизмом мы познакомимся в главе 12.

Тело функции - это последовательность операторов, заключённых в фигурные скобки, причём один оператор может занимать одну и более строк, а также два и более операторов могут быть расположены в одной строке.

Оператор заканчивается символом "**;**", являющимся принадлежностью, а не разделителем операторов, как, например, в языке Паскаль.

a=5 - выражение. a=5; - оператор.

4. Операторы языка C++

Тело функции представляет собой последовательность операторов (исполняемых инструкций), различающихся по действию. Их основные типы представлены на рис. 2 (заключены в пунктирную рамку).

4.1. Простейшие операторы

- *Операторы описания (объявления)* объектов (типов, переменных, констант) должны стоять в теле функции перед их использованием. Все переменные, используемые в программе, должны быть описаны. Компилятор в соответствии типу переменных выделяет память для их хранения значений.
- *Оператор-выражение* - это любое выражение, заканчивающееся символом ´***;***´.
- *Оператор присваивания* в общем виде имеет вид

идентификатор = выражение;

но может быть записан и неявно. Например, в следующей программе:

```
#include <stdio.h>
int main(void )
{
   int n, x=6, y=7, j;
   n=(x-y)/(x+y);
   j++;
   printf("%d, %d, %d", n, j);
   return 0;
}
```

++j; - неявный оператор присваивания. Он эквивалентен явному оператору присваивания *j=j+1;*. Такой способ записи оператора присваивания допустим только для операций ++ и —.

- *Оператор вызова функции имеет* вид:

имя функции (список аргументов);

Он приводит к выполнению последовательности операторов, образующих тело функции.

В следующем примере первая строка – вызов стандартной функции форматного вывода *printf*, результат работы которой – вывод на экран строковой константы. Во второй строке вызывается пользовательская функция *input*, например, ввода данных в массив *mas1* размером *m* на *n*. При таком вызове результат ее работы возвращается в вызывающую функцию через параметры (вводимые значения размещаются по адресу *mas1*). В третьей строке вызывается пользовательская функция *sum* вычисления суммы элементов массива, результат работы которой сохраняется в переменной *k*. В этом случае возврат функцией значения реализован через механизм *return*.

```
//вызов стандартной функции printf
printf("The beginning of work of the program");

//вызовы пользовательских функций с аргументами mas1,m,n
input(mas1,m,n);
k = sum(mas1,m,n);
```

- *Пустой оператор* состоит только из "**;**" и используется для обозначения пустого тела управляющего оператора. Примеры его использования приведены далее.

4.2. Управляющие операторы языка C++

Программа обрабатывает исходные данные и преобразует их в соответствии некоторому алгоритму в результат. Для этого программа использует инструментальные средства – управляющие операторы, позволяющие принимать решения и выполнять повторяющиеся действия. Для этих действий в языке C++ реализованы операторы ветвления *if* и *switch*, операторы цикла *for, while, do while* и операторы прерывания их выполнения *continue* и *break***.** Поведение управляющих операторов может меняться с помощью логических выражений.

Начнем знакомство с управляющими операторами с операторов ветвления, позволяющих выбрать один из взаимоисключающих маршрутов обработки данных.

4.2.1. Условный оператор

Условный оператор имеет две формы – сокращенную *if* и полную *if else*.Сокращенная форма**:**

if (выражение) оператор

Выражение обязательно заключается в круглые скобки и может иметь арифметический тип (быть отношением сравнения или константой) или тип указателя.

Если значение выражения истинно, то выполняется *оператор*, в противном случае управление передаётся на следующий за *if* оператор.

Обратите внимание! Выражение считается *истинным*, если его значение *отлично от нуля*. В языке C++ все ненулевые величины имеют значение "истина" и только 0 имеет значение "ложь". Это иллюстрирует фрагмент программы, где все выражения имеют значение "истина":

```
if (56)  printf("56  - it is true \n");
if (-90) printf("-90 - it is true \n");
if (5>2) printf("5>2 - it is true \n");
```

Полная форма условного оператора:

```
if  (выражение) оператор1
else оператор2
```

Если выражение истинно, то выполняется *оператор1*, а *оператор2* пропускается. В противном случае программа пропускает *оператор1* и выполняет *оператор2*. Например, поиск максимума имеет вид:

```
if (x>y) max=x;
else max=y;
```

Отметим, что оператор *if* в данном случае может быть заменен условной операцией (код представлен на стр. 19).

```
//проверка четности
    if(10%2) printf(" odd\n");     // нечетно
    else printf(" even\n");        // четно
```

Оператор *if* может быть вложенным:

```
   if (выражение1)  оператор1
else  if (выражение2)  оператор2
     else  оператор3
```

Если *выражение1* истинно, выполняется *оператор1*. Если *выражение1* ложно, но *выражение2* истинно, выполняется *оператор2*. В случае, когда оба выражения ложь, выполняется *оператор3*. Например:

```
int num;
scanf("%d", &num);
if (num>=10) printf("Number > or = 10\n");
else if (num<=5) printf("Number < or = 5\n");
  else printf("Number in the range (5,…,10)\n");
```

Обратите внимание! В следующем примере программа работает противоположно тому, что должно было бы быть на первый взгляд:

```
int main()
{
   if(3<2<1) printf("3<2<1  -  CORRECT!!!");
   else      printf("3<2<1  -  NONSENSE!!!");

}
```

Выполните данную программу на компьютере и попробуйте объяснить полученный результат.

Обратите внимание! Достаточно часто выражение в операторе *if* представляет собой сравнение на равенство значений переменных и констант:

```
if (x==5)  оператор
```

Однако часто по невнимательности выражение записывают в виде:

```
if (x=5)  оператор
```

где *x=5* не сравнение на равенство, а присваивание переменной значения *5*.

В этом случае выражение всегда истинно, т.е. всегда выполняется *оператор*. Начинающему работать в С++ программисту бывает трудно обнаружить эту логическую ошибку. В случае сравнения переменной и константы таких ошибок можно избежать, если поменять местами операнды:

```
if (5==x)  оператор
```

Тогда в случае неверного написания операции сравнения ***(5=x)*** компилятор сообщит об ошибке, т.к. константа не может быть леводопустимым выражением в операторе присвоить.

4.2.2. Оператор – переключатель switch

Если необходимо реализовать обработку данных, например, по пяти взаимоисключающим вариантам, то при использовании оператора *if* строится конструкция из четырех вложенных *if else*.

В таких задачах удобнее использовать оператор *switch*, который действует как переключатель на нужный вариант вычислений, и имеет вид:

```
switch (целочисленное выражение)
{
  case константа 1: оператор 1
  case константа 2: оператор 2
  . . .
  default: оператор n
}
```

Оператор *switch* сравнивает значение *целочисленного выражения* с *константами* (метками) операторов, и управление передаётся оператору, у которого метка совпадает со значением *выражения*. *Константы* являются целочисленными значениями, каждая из которых должна встречаться в теле оператора *switch* только один раз.

Оператор, следующий за *default*, выполняется, если ни одна из *констант* варианта не равна значению *выражения*. Вариант *default* не обязательно должен быть последним. Он может отсутствовать, тогда если значение *выражения* не совпало ни с одной константой, никаких действий не выполняется, и управление передаётся следующему оператору программы.

Обратите внимание! Оператор *switch* существенно отличается от аналогичного оператора *case* языка Паскаль. Конструкции *case* и *default* в C++ действуют как метки строки, с которой начинают последовательно выполняться операторы оставшихся вариантов в *switch*, не замечая меток *case* и *default*, т.е. не реализован автоматический останов на следующем варианте. Чтобы прервать выполнение оператора *switch* в конце исполняемой группы операторов выбранного варианта, необходимо поставить оператор *break,* который передаст управление за пределы тела оператора *switch*.

Приведем пример выбора функции меню работы с файлом:

```
    int n;
    cin >> n;
    switch (n)
    {
   case 1: {Creat_file(f);    break;}
   case 2: {Read_file(f);     break;}
   case 3: {Correct_file(f); break;}
   case 4: {Copy_file(f);     break;}
   case 5: {Del_inf_File(f);}
}
```

Оператор в теле *switch* может быть помечен несколькими метками *case*. Фрагмент программы распознавания четного, нечетного или нулевого значения, если *x* принимает одно из возможных значений диапазона *0÷9*:

```
    switch (x)
    {  case 1:
   case 3:
   case 5:
   case 7:
   case 9:

            {puts ("x – Odd numbers");   //нечетные числа
      break;
     }
   case 2:
   case 4:
   case 6:
   case 8:
     {puts ("x – Even numbers"); //четные числа
      break;
     }
   default: puts ("x = 0");
}
```

Обратите внимание! Довольно распространена ошибка, когда забывают включить оператор *break* в оператор *switch* для прерывания его выполнения.

Отметим, что оператор *break* используется также для прерывания работы операторов циклов *while, do while, for* , с которыми мы познакомимся ниже.

4.2.3. Оператор пошагового цикла

Для программирования неоднократно повторяющихся действий можно использовать оператор цикла *for*, который в С++ имеет вид:

for(выражение1; выражение2; выражение3) оператор

Покажем смысловые значения выражений в операторе *for*:

for(инициализация; условие продолжения цикла;
обновление переменных цикла) оператор

Выражение1 (инициализация параметров цикла) вычисляется один раз при входе в цикл. *Выражение2* служит для проверки условия продолжения или окончания цикла и вычисляется перед каждой итерацией цикла. Если значение *выражения2* становится ложным (в общем случае равным нулю), цикл завершается. *Выражение3* служит для изменения переменных цикла и вычисляется в конце каждой итерации цикла.

Приведем пример программы, выводящей на экран числа от *1* до *5* и их квадраты в формате *5* позиций под целое десятичное значе

```
int k;
for (k=1; k<=4; k++)
   printf ("%5d %5d\n", k, k*k);
```

Экран

```
    1     1
    2     4
    3     9
    4    16
```

Из первой строки цикла *for* известна вся информация о параметрах цикла: начальное и конечное значение переменной *k*, а также насколько увеличивается значение переменной на каждой итерации цикла.

Цикл *for* можно использовать для реализации временной задержки с целью согласования скорости реагирования (в данном случае замедления) машины с возможностями восприятия человека:

```
for (volatile int n=0; n<100000000; n++);
```

Никаких действий в этом цикле не производится, пустой оператор не выполняет никаких действий. Но второе и третье выражения вычисляются

100 миллионов раз, что требует определенного машинного времени. Стоит остановить внимание на том, что компиляторы чаще всего оптимизируют код на этапе компиляции и циклы подобного рода они могут вообще пропустить. В данном случае, что бы этого не произошло необходимо объвить переменную n с квалификатором volatile.

Приведем различные примеры оператора *for*:

- использование операции декремента для счёта в порядке убывания:

```
for (n=15; n>0; n--)
{ printf ("%d seconds to start!", n);
  for (volatile int j=1; j<10000000; j++);          //временная
задержка(пауза)
system(“cls”);

}
```

- изменение переменной цикла на 11 единиц:

```
   for (n=3; n<60; n+=11)
            printf("%d \n", n);
```

- использование переменной типа char в качестве параметра цикла:

```
 for (ch='a'; ch<='z'; ch++)
            printf("Code ASCII for %c is %d \n", ch, ch);
```

- использование в качестве переменной цикла вещественного числа:

```
 for (n=100.0; n<2300.0; n*=1.8)
       printf("Your savings: $%3.f\n", n);//Ваши накопления
```

- можно пропустить в цикле одно или даже все три выражения (при этом символы ";" должны остаться):

```
      u=3;
      for (v=5; u<100; )
            u*=v;         //третье выражение стало оператором цикла
```

- цикл выполняется бесконечное число раз, т.к. пустое условие всегда считается истинным:

```
for( ; ; )
        printf("Good morning! \n");
```

- бесконечный цикл можно завершить выполнением в его теле операторов break, goto, return, например:

```
for (i=1; ; )
        {i++;
         cout << i;
         if (i>5) break;
}
```

Экран

```
2  3  4  5  6
```

- первое выражение не обязательно должно инициализировать переменную, например, это может быть оператор printf. Необходимо помнить, что первое выражение вычисляется только один раз перед входом в цикл. Следующая программа запрашивает символы до тех пор, пока не будет введён символ 'k':

```
char sym=getchar();
for (printf("Enter a symbol:\n"); //первое выражение
    sym!='k';          //второе выражение
    sym=getchar()      //третье выражение
    );
printf("Has guessed!");     //Угадал!
```

Экран

```
Enter a symbol:
a
c
k
Has guessed!
```

4.2.4. Оператор цикла while

Цикл *while* (с предусловием), в отличие от цикла *for*, не содержит инициализации и обновления переменной цикла в явном виде:

while (выражение) оператор

Если *выражение* истинно, выполняется *оператор*, затем снова вычисляется *выражение*. Эта последовательность действий повторяется до тех пор, пока *выражение* не станет ложным (равным нулю). Затем управление передаётся следующему оператору. Цикл с предусловием может не выполниться ни разу при определенных данных.

В теле цикла должны быть операторы, изменяющие значение *выражения* так, чтобы в конце концов оно стало ложным. В противном случае получим бесконечный цикл, выход из которого при определенных значениях данных должен быть принудительным (*break)*.

В качестве примера вычислим сумму *n* натуральных чисел:

```
    unsigned n, s=0, i=1;
printf("Enter number n= ");          //введи число n=
scanf("%u", &n);
while (i<=n)
{   s+=i;
    i++;
}
printf("The sum is equal: %u", s); //сумма равна:
```

Приведём примеры эквивалентных операторов ***while*** и *for:*

- ***while (выражение) оператор***
  for (;выражение;) оператор

- ***for (выражение1;выражение2;выражение3) оператор***
  выражение1;
  while (*выражение2)*
  { оператор
  выражение3;
  }

4.2.5. Оператор цикла do while

Оператор цикла с постусловием имеет вид:

do оператор while (выражение);

где *выражение* вычисляется после выполнения каждой итерации цикла.

Тело цикла *do while* всегда выполняется по *крайней мере один раз*, поскольку проверка выхода из цикла осуществляется только после его завершения. Использовать цикл *do while* лучше всего в тех случаях, когда должна быть выполнена хотя бы одна итерация.

Вычислим сумму *n* натуральных чисел с помощью оператора *do while*:

```
    unsigned n, s=0, i=1;
printf("Enter number n: ");
scanf("%u", &n);
do
{   s+=i;
```

```
    i++;
}
    while (i<=n);
    printf("The sum is equal: %u", s);
```

4.2.6. Оператор продолжения continue

Этот оператор может входить в состав циклов *while, do while, for*. Как и в случае оператора *break*, он приводит к изменению характера выполнения цикла. *Continue* вызывает пропуск оставшейся части итерации и переход к следующей итерации.

Следующий фрагмент программы определяет количество введенных в строке символов, отличных от букв латинского алфавита нижнего регистра:

```
    char sym;
    int s=0;
while ((sym=getchar())!='\n')
{  if (sym>='a'&& sym<='z')continue;
   s++;
}
    printf("%d", s);
```

4.2.7. Оператор возврата return

В функции с возвращаемым значением (имеющей тип, отличный от void), должен присутствовать оператор *return*, который заканчивает работу функции и передает значение в вызывающую функцию:

return выражение;

Обратите внимание! Тип *выражения* должен совпадать с типом функции.

Следующая функция возвращает сумму ряда натуральных чисел от *1* до *n*:

```
int sum(int n)
{ unsigned s=0, i=1;
 while (i<=n)
       { s+=i;
         i++;
```

```
        }
        return s;
    }
```

Вызов функции *sum* может иметь один из следующих видов, где возвращаемое значение:

```
int s=sum(10);                  //сохраняется в переменной
printf("%d", sum(10)); //выводится на экран
```

Для функции, имеющей тип *void*, выражение в операторе *return* должно отсутствовать, и *return* прекращает ее работу, ничего не возвращая в точку вызова.

В следующей программе в функции *main* выводится на экран модуль значения в результате вызова функции *module*, прототип которой объявлен перед функцией *main*, а определение приведено после функции *main*:

```
#include <stdio.h>
unsigned module(int);          //объявление функции module

int main(void)
{   int x, y;
   printf("Enter two integers: ");  //введи два целых
   scanf("%d %d", &x, &y);
   printf("Their absolute values:\n"); // Их абсолютные
                                                  // значения
   printf ("%d, %d \n", module(x), module(y));
   return 0;
}

unsigned module(int x)              //определение функции module
{
   if (x<0) return (-x);
   else return(x);
}
```

Обратите внимание, что работу функции *main* также заканчивает оператор возврата *return 0*; тип константного выражения при этом совпадает с типом функции *int*.

5. Указатели в языке C++

Упрощенная схема организации памяти машины – это массив последовательно пронумерованных (проадресованных) ячеек, с которыми можно работать по отдельности или связными кусками.

Указатель – это переменная (как правило, четырехбайтовая), которая хранит адрес некоторого объекта (группы ячеек памяти) - переменной, массива или функции (с методами классов ситуация несколько иная). Используя указатель, мы получаем косвенный доступ к объекту.

Указатели позволяют реализовать эффективную обработку массивов, структур, вызовов функций. Указатели необходимы при работе с динамической памятью, позволяя в процессе работы программы создавать новые объекты, и освобождать память, когда они больше не нужны.

Значением указателя является шестнадцатеричное целое – адрес определенного участка памяти. Если указатель не ссылается на объект, ему может быть присвоен нулевой адрес (NULL).

Неправильное использование указателей (в C++ используется ручное управление памятью) может привести к программным ошибкам с тяжелыми последствиями, в том числе зависанию или завершению приложения или даже операционной системы. Причем ошибка в самом указателе себя не проявляет, например, до попытки обратиться к объекту с помощью этого указателя или двойного применения операции *delete* к указателю. О таких ошибках и методах их устранения мы скажем далее. В C++ необходимо помнить о постоянной проверке корректности использования указателей. В этой связи развитие языков программирования привело к созданию новых языков, в которых практически полностью отказались от использования указателей (например, в языках Java и C#).

5.1. Объявление указателей

При объявлении переменной-указателя необходимо определить тип объектов данных, адрес которых будет содержать указатель, и перед именем указателя поставить символ звездочка (*). Следующий код создает указатели *px* и *py* на целые, указатели *pa* и *pb* на вещественные и указатель *pch* - на символьные значения:

```
int    *px, *py, x, y;
float  *pa, *pb, a, b;
char   *pch, ch;
```

Префикс *"p"* принято использовать в именах переменных-указателей. Тип переменной-указателя определяется по типу объекта, с которым будет связан указатель.

Обратите внимание! Синтаксис описания любого объекта фактически имитирует синтаксис выражения, в котором этот объект может появляться. В данном случае объявление *char **pch**;* говорит о том, что выражение **pch* имеет тип *char*, т.е. его можно использовать как леводопустимое выражение: **pch='A';*

Можно объявить указатель на тип *void*. Далее будут приведены примеры работы с таким указателем.

5.2. Операции над указателями

Как и любые переменные, указатели в процессе работы программы получают значения. Чтобы связать указатель с некоторой переменной, необходимо использовать унарную операцию получения адреса &, в результате чего вместо значения переменной будет возвращен ее адрес:

```
px=&x; py=&y; pch=&ch;
```

Обратите внимание! Важно понимать разницу между *px* и *&x*: *px* - это переменная, а *&x* – константа. Указатель *px* может указывать на любой объект целого типа, а *&x* - это адрес места расположения в памяти целой переменной *x*, который в процессе выполнения программы не меняется. Очевидно, что операция получения адреса не применима к выражениям или константам:

```
&(x+y), &25      //ошибка
```

Чтобы в выражениях использовать вместо переменной указатель на нее, применяют операцию разыменовывания – звездочку (*):

```
int x=20, y, *px=&x;   //инициализация переменной и указателя
y = *px;               //эквивалентно  y=x
```

Оператор *y=*px*; присваивает переменной *y* содержимое памяти, на которую указывает *px*. Операция * разыменовывает указатель, т.е. превращает его в саму переменную.

Указатели могут входить в выражения, при этом унарные операции *&* и * по уровню приоритета входят в группу 2 (табл. 2, стр. 20), совместно с такими операциями, как *!, ~, -, ++, --, (тип), sizeof.*

Ссылки на указатели могут появляться и в левой части оператора присваивания, например, если *px* присвоено значение px=&x;, можно выполнить оператор:

```
*px+=1;          // увеличение значения переменной x на 1
```

Поскольку указатели являются переменными, с ними можно обращаться, как и с остальными переменными, например:

```
py=px;           //оба указателя ссылаются на одну переменную
px=0xff;         //ошибка, несоответствие типов
px=(int *)0xff;  //правильный оператор
```

Последний пример показывает, что *px* теперь будет указывать не на переменную, а на ячейку памяти, имеющую адрес *0xff* (в результате чего можно несанкционированно испортить данные).

К указателям применяются операции отношения >, >=, !=, = =, <=, < и указатели можно использовать в выражениях в качестве операндов операций отношений. При этом сравнивать указатели допустимо только с другими указателями того же типа или с константой *NULL*, обозначающей значение условного нулевого адреса.

Приведем пример, в котором используются операции над указателями и выводятся на экран получаемые значения (рис.3).

Обратите внимание! Для вывода значений указателей в форматной строке функции *printf* используется спецификация преобразования *%p.*

При печати разности значений указателей (адресов) в байтах в функции *printf* использована спецификация преобразования *%d* - вывод знакового десятичного целого:

```
#include <stdio.h>
#include <conio.h>                  //содержит функцию getch
int main( )
{  //описание массива с его инициализацией
   float x[]={10.0, 20.0, 30.0, 40.0, 50.0};
   float *p1, *p2;
   int i;
   printf("\n Addresses of pointers: &p1=%p,&p2=%p\n",&p1,&p2);

      printf("\n Addresses of elements of array:\n");
   for(i=0; i<5; i++) printf("&x[%d]=%p, ", i, &x[i]);
```

```
   printf("\n\n Values of elements of array:\n");
   for  (i=0; i<5; i++) printf(" x[%d]=%5.1f ", i, x[i]);

    printf("\n\n Access to array elements by the use of
          pointer:\n"); //доступ к элементам массива через указатели
   for(p1=&x[0],p2=&x[4]; p2>=&x[0]; p1++,p2--)
   {
     printf("\n *p1=%f, p1=%p, *p2=%f, p2=%p", *p1,p1,*p2,p2);
     printf("\n p2-p1=%d",p2-p1);    // Difference of addresses
   }
   getch();
   return 0;
}
```

```
Addresses of pointers: &p1=0022F800,&p2=0022F7F4

Addresses of elements of array:
&x[0]=0022F80C, &x[1]=0022F810, &x[2]=0022F814, &x[3]=0022F818, &x[4]=0022F81C,


Values of elements of array:
x[0]= 10.0  x[1]= 20.0  x[2]= 30.0  x[3]= 40.0  x[4]= 50.0

Access to array elements by the use of pointer:

*p1=10.000000, p1=0022F80C, *p2=50.000000, p2=0022F81C
p2-p1=4
*p1=20.000000, p1=0022F810, *p2=40.000000, p2=0022F818
p2-p1=2
*p1=30.000000, p1=0022F814, *p2=30.000000, p2=0022F814
p2-p1=0
*p1=40.000000, p1=0022F818, *p2=20.000000, p2=0022F810
p2-p1=-2
*p1=50.000000, p1=0022F81C, *p2=10.000000, p2=0022F80C
p2-p1=-4
```

Рис. 3. Экран результатов работы с элементами массива через указатели

Переменная, объявляемая как указатель на тип *void*, может быть использована для ссылки на объект любого типа. При выполнении операции над такими указателями или над объектами, на которые они указывают, необходимо явно определить тип объектов с помощью операции приведения типов:

```
int n,*p_int;
void *p=&n;
*(int*)p=10;              //n присвоили значение 10
p=new int [10];  //выделили память под массив из 10 целых
p_int=(int*)p;     //указатель на целое связали
```

```
                                    //с динамически выделенной памятью
for(int i=0;i<10;i++) //динамическому массиву целых
   cin>>*p_int[i];        //присвоили значения
```

Над указателями нельзя выполнять следующие операции:

- изменение знака (минус);
- операции умножения и деления *, /, %, побитовые операции и соответствующие им операции присваивания;
- логические операции за исключением логического отрицания (!).

5.3. Указатели на указатели

Указателю можно присвоить адрес другого указателя на переменную некоторого типа. Объявим переменную как указатель на указатель на целое:

```
int **pp_int;
```

Здесь *pp_int* - указатель на указатель, который указывает на целое число.

Число уровней косвенной адресации можно неограниченно увеличивать.

Объявим следующие три переменные и приведем операторы присваивания этим переменным значений:

```
int **pp_int, *p_int, x;
x=123;
p_int=&x;
pp_int=&p_int;
```

Соответственно в выражениях можно использовать следующие подвыражения:

```
**pp_int    //значение переменной x
 *pp_int          //адрес целой переменной x
  pp_int          //адрес указателя p_int на целое
```

Указатели на указатели используются при работе с многомерными массивами – для доступа к элементам статического массива, создания и работы с динамическими массивами, а также в качестве параметров функций, обрабатывающих массивы.

В программах указатели на функции в основном используются для сохранения адресов программ обработчиков прерываний и для передачи функций в качестве формальных параметров другим функциям.

В языке C++ существует сильная взаимосвязь между массивами и указателями. В следующем разделе, после знакомства с типом массива в C++, мы остановимся подробно на механизме работы с элементами массива через указатели.

6. Массивы в языке C++

Массив – это структура данных, позволяющая хранить как единое целое (под одним именем) последовательность элементов одинакового типа. Объявление массива определяет его имя, тип элементов и число элементов в массиве.

В инструкциях объявления массив идентифицируется по квадратным скобкам (функция – по круглым).

Свойства массивов в языке C++:

- нумерация элементов массива начинается с *0*, т.е. последний элемент массива имеет индекс на *1* меньше, чем число элементов в массиве;
- имя массива является константой и содержит адрес первого элемента массива;
- обращение к элементам массива реализовано не только по индексам, но и через указатели.

6.1. Объявления массивов

Объявим одномерные массивы целых, символьных и вещественных значений:

```
int a[10], b[5];
char str[80];
float MyArray[50];
```

Первый элемент массива *MyArray* имеет индекс *0*, последний – *49*.

Объявление двухмерного массива предполагает наличие двух его размеров, заключенных в квадратные скобки:

```
char page [24][80];
```

Элементы массивов располагаются последовательно в ячейках памяти по возрастанию адресов. Элементы многомерного массива запоминаются построчно (для массива *page* 80 элементов *1*-й строки, затем *80* элементов *2*-й строки и т. д.).

Двухмерный массив *int a[3][4]* можно схематично представить следующим образом:

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]

a[2][0]	a[2][1]	a[2][2]	a[2][3]

Количество байт памяти, которое необходимо для хранения двухмерного массива, вычисляется по формуле:

Кол-во байт=(размер типа данных)(кол-во строк)*(кол-во столбцов).*

Если адрес первого элемента массива равен *a*, то адрес последнего *a+sizeof(int)*3*4-sizeof(int).*

Выведем на экран значения адресов первого и последнего элемента массива *a*, используя операцию взятия адреса:

```
printf ("%x, %x \n", &a[0][0], &a[2][3]);
```

При последовательной работе с элементами массива, чем правее стоит индекс, тем быстрее он изменяется, т.е. элементы массива хранятся в памяти в порядке возрастания самого правого индекса.

Индекс в массиве может быть любым целым выражением, которое может включать как целые переменные, так и целые константы.

Обратите внимание! При работе с массивом компилятор C++ не делает проверку выхода индексов массива за объявленные границы, поэтому необходимо аккуратно определять начальное и конечное условия выполнения циклов обработки массивов. Ошибки такого рода приводят к аварийному завершению программы.

6.2. Инициализация массивов

Массив можно инициализировать, т.е. присвоить элементам массива начальные значения при его описании, указав в фигурных скобках список инициализаторов:

```
float farr [6] = {1.1, 2.2, 3.3, 4.0, 5, 6};
```

Многомерные массивы, в том числе и двумерные массивы, можно инициализировать, рассматривая их как массив массивов. Следующие инициализации эквивалентны:

```
int a[3][5]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int a[3][5]={{1,2,3,4,5}, {6,7,8,9,10}, {11,12,13,14,15}};
```

Такая форма записи эквивалентна набору операторов:

a[0][0]=1; a[0][1]=2; a[0][2]=3; a[0][3]=4;
a[0][4]=5; a[1][0]=6; a[1][1]=7; и т.д.

Количество инициализаторов не обязательно должно совпадать с количеством элементов массива. Если в объявлении массива присутствует инициализатор хотя бы одного элемента, все, что не инициализируется, обнуляется.

int A[3][5]={1,2,3,4,5,6,7,8,9,10,11};
int B[3][5]={{1,2,3},{4,5,6,7,8},{9,10,11}};

Соответствующие массивы будут заполнены следующим образом:

Массив A:

6	7	8	9	10
11	0	0	0	0

Массив B:

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

В первом случае все элементы массива инициализируются подряд и зануляются последние значения массива, во втором случае – зануляются последние элементы одномерных массивов, для которых список инициализации меньше второй (правой) размерности.

Обратите внимание! При инициализации многомерных массивов можно опустить значение самого левого индекса, компилятор сам определит его по количеству выделенных во внутренние фигурные скобки инициализирующих списков:

int mas[ ][3]={{1}, {2}};
//размерность массива будет 2 на 3, инициализированы 2 элемента

Подсчет количества элементов компилятор осуществляет следующим образом:

int n = sizeof mas/sizeof (int);

Следующая инициализация не допустима:

int a[ ][ ]={{1,2,3},{4,5,6,7,8},{9,10,11}};

Компилятор выдаст сообщение об ошибке, т.к. количество элементов в одномерном массиве не определено, а список его

инициализации может быть меньше его размерности, (что показано в примере выше).

Ввод данных в массив с клавиатуры с помощью функций и операций ввода имеет вид:

```
int x[n][m],i,j;
for(i=0; i<n; i++)
   for(j=0; j<m; j++)
    scanf("%d",&x[i][j]);
```

```
int x[n][m],i,j;
for(i=0; i<n; i++)
   for(j=0; j<m; j++)
    cin >> x[i][j]);
```

Вывод данных на экран с помощью функций и операций вывода:

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

```
for(i=0; i<n; i++)
   {for(j=0; j<m; j++)
      cout >> x[i][j]);
    cout >> endl;
   }
```

6.3. Указатели на массивы

Указатели позволяют повысить эффективность программ при связывании их с массивами. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей, при этом вариант с указателями обычно оказывается более быстрым.

Связать указатель с массивом можно следующим образом:

*int a[50], *pa;*
*pa=&a[0]; **//указатель содержит адрес элемента a[0] массива***

Так как имя массива является адресом его местоположения в памяти, то имя массива является синонимом местоположения нулевого элемента массива. Следовательно, операторы присваивания *pa=&a[0];* и ***pa=a;*** эквивалентны.

Обратите внимание! Объявления *int pa[]* и *int *pa;* идентичны по действию - оба объявляют *pa* указателем. Но необходимо учитывать следующее отличие: указатель является переменной (второе объявление), а имя массива (первое объявление) - константой, то есть конструкция *a=pa* будет незаконной. При этом первое объявление не является инструкцией объявления переменной программы (так как явно не указан размер массива), так массив можно объявить как формальный параметр функции в ее заголовке.

Компилятор при анализе инструкций описания, в том числе массивов, автоматически выделяет поименованные области памяти для

хранения значений переменных в соответствии с их типом. Напомним, что не указывать размер в крайних левых скобках массива при описании можно только в случае его инициализации.

Итак, если *pa* указывает на некоторый элемент массива *a*, то по определению *pa+1* указывает на следующий элемент, (*pa+i)* – на элемент, стоящий на *i*-й позиции после элемента, на который указывает *pa*, а (*pa-i)* - на элемент, стоящий на *i*-й позиции до элемента, на который указывает *pa*.

Если *pa=&a[i];* то *pa+1==&a[i+1]*
pa-1==&a[i-1]

Если *pa=&a[0];* то ** pa == a[0]*
**(pa+1) == a[1]*
**(pa+i) == a[i]*

Отсюда следует, что арифметические операции над указателями выполняются особым образом - с масштабированием на величину, определяемую как размер памяти (в байтах), выделяемый под объект, на который указывает указатель.

Это означает, что в выражении *pa+i* перед прибавлением *i* умножается на *sizeof*(*int*), (2 или 4 в зависимости от компилятора), т.е. на количество байт, отводимое под хранение данных целого типа. В случае символьного массива *i* будет умножаться на *1*. Масштабирование выполняется по правилу: *i*sizeof(тип элемента массива)* – адрес *i*-го элемента массива.

Приведем следующие тождественные равенства для нашего примера:

pa+2 == &a[2]
**(pa+2) == a[2]*
**pa+2 == a[0]+2*

Приведем алгоритмы присваивания элементам массива значений тремя разными способами.

```
int x[2][3], i, j;
i=0;
while (i<2)
{ j=0;
  while(j<3)
     x[i][j++]=i*3+j+1;
  i++;
}
```

Операции приведения типов во 2-м и 3-м примерах обязательны в соответствие требованиям C++ к совпадению в выражениях типов указателей, иначе компилятор сообщит об ошибке:
*'initializing' : cannot convert from 'int [2][3]' to 'int *'*

Элементы массива во всех примерах получают значения *1, 2, 3, 4, 5, 6*.

Как мы видим, обрабатывать элементы массива можно двумя способами: с помощью индексов и с помощью указателей. Второй способ работает, как правило, быстрее.

```
int x[2][3], i, j;
i=0;
while (i<2*3)
  *((int*)x+i++)=i+1;
```

```
int x[2][3],  i,
*px=(int*) x;
 i=0;
 while (i<2*3)
    *px++ = ++i;
```

6.4. Указатели и многомерные массивы

Многомерные массивы хранятся в последовательных ячейках памяти, при этом чем правее индекс, тем быстрее он изменяется.

Важным свойством языка C++ является то, что многомерные массивы рассматриваются как массивы массивов. Для двухмерного массива это приводит, в частности, к следующему:

*int a[2][4], *pa;*

тогда *a[0]==&a[0][0]* ***//a[0] - имя первого одномерного массива***
a[1]==&a[1][0] ***//a[1] - имя второго одномерного массива***

Имена элементов двухмерного массива – его одномерные массивы *a[0]*, *a[1]* и т.д., также являются адресами их месторасположения в памяти. Поэтому имя одномерного массива *a[0]* синоним адреса элемента двухмерного массива *a[0][0]*.

Таким образом, для получения адреса первого элемента массива, мы можем использовать следующие выражения: *a, a[0]* или *&a[0][0]*.

Для получения адреса первого элемента второй строки массива можно использовать следующие выражения: *a+1, a[1]* или *&a[1][0]* (рис.4):

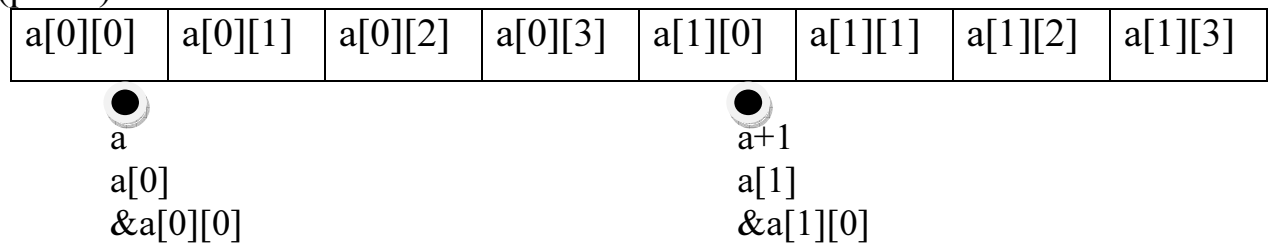


Рис. 4. Адреса одномерных массивов в двухмерном массиве

Таким образом, при прибавлении единицы к целой переменной ее значение увеличивается на единицу. При прибавлении единицы к указателю его числовое значение увеличивается на размерность типа, на который он указывает.

Таким образом, с точки зрения синтаксиса языка С++ указатели *a, a[0], a[1]* являются константами-адресами и их значения нельзя изменять во время выполнения программы.

Доступ к элементу массива *a[i][j]* через указатель можно получить следующим образом:

a – адрес двухмерного массива, или адрес его нулевого одномерного массива, или сам одномерный нулевой массив, или адрес элемента *a[0][0]* массива;

a+i – адрес *i*-го одномерного массива (*a[i])* , или адрес нулевого элемента *a[i][0]* в *i*-м одномерном массиве;

**(a+i)* – *i*-й одномерный массив, или его же адрес;

**(a+i)+j* – адрес *j*-го элемента в одномерном массиве *a[i]* ;

***(*(a+i)+j)* – элемент массива *a[i][j].*

Алгоритм вычисления суммы элементов двухмерного массива через указатель можно представить следующим образом:

```
int x[n][m],i,j,sum=0;
for(i=0; i<n; i++)
        for(j=0; j<m; j++)
                sum+=*(*(x+i)+j);
```

Можно создавать массивы указателей, в том числе инициализировать их в инструкциях объявления:

```
int data[6];                          //массив целых
int *p[6];                            //массив указателей на целые
int *pd[]={&data[0],&data[4],&data[2]}; //инициализация
                                      //массива указателей на целые
```

В последней строке кода создается одномерный массив указателей на целое из трех элементов, где первый элемент ссылается на нулевой элемент массива *data*, второй - на его четвертый элемент, а третий – на его второй элемент.

Приведем еще один пример работы с элементами массива через указатели:

```
int arr[3][3]={1,2,3,4,5,6,7,8,9};
int sum=**arr+*(*arr+1)+arr[0][2];
cout<<"sum="<<sum<<endl;
```

Экран

```
sum=6
123
```

*cout<<**arr<<*(*arr+1)<<arr[0][2];*

6.5. Алгоритм сортировки двухмерного массива с использованием указателей

Интересные возможности массива указателей появляются в тех случаях, когда его элементы адресуют либо элементы другого массива, либо указывают на начало одномерных массивов соответствующего типа. В таких случаях эффективно решаются задачи сортировки сложных объектов с разными размерами.

Если с двумерным массивом связать несколько одномерных массивов указателей, то можно упорядочить строки матрицы одновременно по разным правилам, например, одновременно по возрастанию и убыванию, "добираясь" до строк с помощью разных массивов указателей.

В качестве примера рассмотрим программу одновременного упорядочения и по возрастанию, и по убыванию одномерного массива без перестановки его элементов.

В программе с помощью перестановки значений элементов массива указателей *pmin* определяется последовательность просмотра элементов массива *array* в порядке убывания их значений. Массив указателей *pmax* решает такую же задачу, но для просмотра массива *array* в порядке возрастания их значений. Рис. 5 иллюстрирует исходную и результирующую адресации элементов массива *array* элементами массивов указателей. Для "обмена" значений элементов массивов *pmax* и *pmin* введен вспомогательный указатель *float *e*.

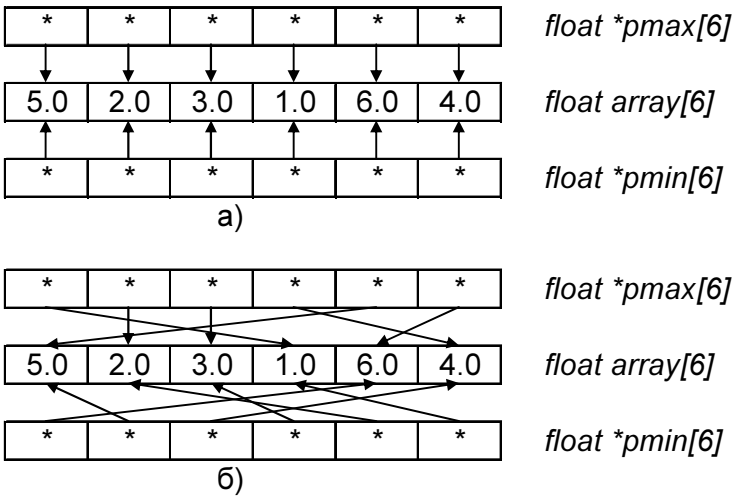


Рис. 5. Массивы указателей в задаче упорядочения: *а* - до упорядочения; *б* - после упорядочения

Код программы имеет вид:

```
#include <stdio.h>
#include <conio.h>
#define N 6                    //см.п. 17.2
main()
{
float array[ ]={5.0, 2.0, 3.0, 1.0, 6.0, 4.0};
float *pmin[N], *pmax[N], *e;
int i, j ;
for (i=0; i<N; i++)
     pmin[i]=pmax[i]=&array[i];

for (i=0; i<N-1; i++)
  for (j=i+1; j<N; j++)
     { if (*pmin[i]<*pmin[j])
            {
              e=pmin[i] ;
              pmin[i]=pmin[j];
              pmin[j]=e;
            }
       if (*pmax[i]>*pmax[j])
            {
              e=pmax[i] ;
              pmax[i]=pmax[j] ;
              pmax[j]=e;
            }
     }
printf("\n <--------------  \n");
for (i=0; i<N; i++)
     printf("\t%5.3f",*pmin[i]);
printf("\n -------------> \n");
for (i=0; i<N; i++)
     printf("\t%5.3f",*pmax[i]) ;
getch();
}
```

Результаты выполнения программы:

```
<--------------
       6.000   5.000   4.000   3.000   2.000   1.000
-------------->
       1.000   2.000   3.000   4.000   5.000   6.000
```

7. Динамическое распределение памяти

При обработке данных часто заранее неизвестно, сколько понадобится памяти для их хранения. Размер данных становится известным только во время выполнения программы. Кроме того, статической памяти бывает недостаточно для решения сложных задач.

Операторы управления динамической памятью обеспечивают возможность занимать и освобождать области памяти в процессе выполнения программы.

При динамическом выделении памяти:

- в программе размещается не сам объект, а только вызов функции, выделяющей для него память, что делает файл задачи заметно короче;
- размер объекта может быть вычислен внутри программы, можно также изменить размер существующего объекта, не изменяя его значения;
- если данный объект больше не нужен, память, которую он занимает, можно освободить.

В C++ существуют два способа динамического выделения памяти. Первый способ, унаследованный от языка Си, использует стандартные библиотечные функции библиотек *malloc.h*, *stdlib.h* для создания динамических объектов.

В объектно-ориентированном программировании, где создаются и уничтожаются динамические объекты (экземпляры классов), функции динамического выделения памяти не работают. Для этих целей в C++ введены две операции для работы с динамической памятью:

- *new* - выделение памяти;
- *delete* - освобождение памяти.

7.1. Функции работы с динамической памятью

C++ наследовал из языка Си следующие функции работы с динамической памятью:

*1. void * malloc(unsigned size);*

Функция *malloc* выделяет блок памяти размером *size* байт и возвращает указатель на него, выровненный по границе слова, т.е. размер блока может быть больше, чем *size* байт в результате выравнивания. При неудачном завершении функция возвращает значение *NULL*. Содержимое выделенного блока не определено (не обнулено).

2. *void * calloc(unsigned n, unsigned m);*

Функция *calloc* возвращает указатель на начало блока обнуленной динамической памяти, выделенной для размещения *n* элементов по *m* байт каждый. При неудачном завершении возвращает значение *NULL*.

3. *void * realloc(void *ptr, unsigned size);*

Функция *realloc* изменяет размер блока ранее выделенной динамической памяти до размера *size* байт, где *ptr* - адрес начала изменяемого блока. Функция возвращает адрес, возможно, новой области памяти. Если *ptr==NULL* (память ранее не выделялась), то функция работает как *malloc*. Если *ptr!=0,* а *size==0*, *realloc* действует как функция *free*. Если *realloc* не может выделить требуемое пространство, возвращаемое значение равно *NULL*, а содержимое блока, на которое указывает *ptr*, сохраняется.

4. *void * free(void * ptr);*

Функция *free* освобождает ранее выделенный блок динамической памяти, адрес первого байта которого равен значению *ptr*. Если *ptr==NULL*, то *free* ничего не выполняет. Если *ptr* не является значением одной из функций, предварительно выделивших область памяти, поведение функции *free* не определено.

Рассмотрим примеры использования перечисленных функций.

Следующий код создает одномерный массив вещественных чисел, выводит их в обратном порядке с указанием индексов элементов, и освобождает память:

```
{
 float *t;
 int i,n;
 scanf("%d",&n);
 t=(float*)malloc(n*sizeof(float));
 for(i=0; i<n; i++)
            scanf("%f",&t[i]);
 for(i=n-1; i>=0; i--)
      printf("t[%d]=%4.2f\n", i+1, t[i]);
 free(t);
}
```

Экран

```
5
1  2  3  4  5
t[5]= 5.00
t[4]= 4.00
t[3]= 3.00
t[2]= 2.00
t[1]= 1.00
```

Обратите внимание! Перед функцией *malloc* обязательно должна стоять операция приведения возвращаемого ею значения к типу переменной, получающей это значение (*float** - указателю на вещественное значение, как объявлена переменная *t)*.

Приведем код создания двухмерного массива размером *5* на *80* символов:

```
char *mas[5];
for (i=0; i<5; i++)
  if (mas[i]=(char *)malloc(80))
      gets(mas[i]);
  else  ; //обработка ошибки
for (i=0; i<5; i++) puts(mas[i]);
```

В результате переменная *mas* (массив указателей на *char*) получит значения адресов выделенных блоков памяти размером *80* байт для хранения строк текста. Функция *gets* присваивает значения этим блокам выделенной памяти. Если значение выражения в операторе *if* станет равно *NULL* (ошибка выделения памяти), переход на ветку *else*.

Функцию *realloc* следует использовать следующим образом. Чтобы не потерять данные вследствие неуспешного завершения функции выделения блока памяти нового размера, ее результат следует сохранить в дополнительной переменной-указателе:

```
int *p1, *p2;
p1 = (int *)malloc(100); //выделение памяти под массив
//заполнение массива и его обработка
. . .

//увеличить блок памяти под массив до размера new_size байт
p2 = (int *)realloc(p1,new_size);
if(p2!=NULL)  p1=p2;   // если realloc  успешен
. . .

// работа с массивом через указатель p1, как и раньше
```

7.2. Операторы работы с динамической памятью

В C++ для работы с динамической памятью введены два оператора:

- *new* - выделение памяти, в случае ошибки возвращается нулевой указатель;

- *delete* - освобождение памяти, в зависимости от компилятора указатель обнуляется или его значение не определено. Явное обнуление указателя после освобождения памяти позволит избежать незамеченных ошибок.

Можно сначала выделить память и затем поместить туда значение, или сразу инициализировать выделенную область памяти:

```
int *p1=new int;     // память под целое
int *p2=new int(5);  //память под целое с его инициализацией
cout<<*p2<<endl;            //вывод значения
delete p2;                  //освобождение памяти
p2=0;                       //явное обнуление указателя

int *p3=new int[10]; //память под массив 10 значений типа int
for(i=0;i<10;i++)
   cin>>p3[i];              //заполнение массива

int (*mas)[3][4];           //указатель на двухмерный массив
mas = new int [2][3][4]; //память под 3-х мерный массив
```

Ошибки в использовании операции *new*:

```
p1  = new int[];                 // размер памяти не известен
mas = new int[][3][4]; // размер памяти не известен
```

Выделим память под двухмерный динамический массив, имея следующие объявления:

```
int **px, n,m;
cin>>n>>m;                          //размер массива
px=new int*[n];             //память под массив указателей
for (i=0;i<n;i++)
      px[i]=new int[m];     //память под массив целых
for (i=0;i<n;i++)
  for (j=0;j<m;j++)
       cin>>px[i][j];           //заполнение массива
```

Здесь *px* - указатель на указатель на тип *int*. В результате создали массив из *n* указателей на одномерные массивы *px[i]* целых чисел.

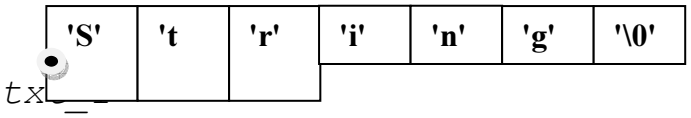
8. Символьные строки

Символьная строка в C++ - это одномерный массив символов, заканчивающийся нулевым символом *'\0'*. Этот символ соответствует нулевому байту, который является первым символом кодировке ASCII. Этот байт используется функциями ввода/вывода строк в качестве признака конца строки. Значение строковой константы заключается в кавычки (двойной апостроф). Зададим вопрос: в чем отличие констант *'A'* и *"A"*? В том, что первая – символ, а вторая – строка, заканчивающаяся нулевым символом *'\0'*. Для хранения первой выделяется *1* байт памяти, для второй – *2* байта, при этом размер строкового значения равен *1*.

8.1. Объявление строк

Строки можно инициализировать следующим образом:

char txt_1[]="String";



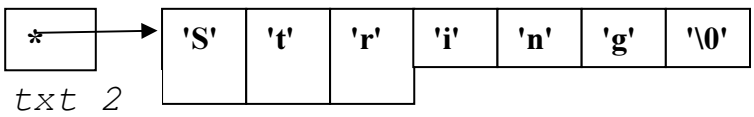
Под переменную *txt_1* выделяется *7* байт памяти (*sizeof txt_1==7)*, которые на этапе компиляции инициализируются строковым значением.

Здесь, как обычно, имя массива *txt_1* является указателем на первый элемент массива, т.е.

**txt_1= ='S'* или *txt_1[0]= ='S'*
**(txt_1+6)= ='\0'* или *txt_1[6]= ='\0'*

Для создания строки можно использовать указатель на тип *char*:

*const char *txt_2="String";*



Здесь объявлен указатель на тип *char*, под него выделяется *sizeof txt==4* байта памяти, и значением этой памяти является указатель на первый символ константного значения *"String"***.**

Обратите внимание! Сходства определения *txt_1* и *txt_2*:

txt_1 и *txt_2* являются указателями на тип *char,* через них получаем доступ к строке символов. В том и другом случае сама строка

инициализации определяет размер памяти, необходимой для её размещения.

Различия их определения:

Первое объявление вызывает создание в статической памяти массива из 7 элементов, и имя *txt_1* является синонимом адреса нулевого элемента массива. При этом *txt_1* является *константой*, т.е. значение *txt_1* нельзя изменять, например, с помощью операции инкремента *txt_1++,* т.к. это означало бы изменение положения (адреса) массива в памяти. Можно применять операции, например, *txt_1+6* для идентификации очередного элемента массива.

Второе объявление тоже вызывает создание в статической памяти массива из 7 элементов для хранения строк, но, кроме того, будет выделена дополнительная ячейка памяти для размещения переменной *txt_2*, являющейся указателем на этот массив. Первоначальное значение этой переменной - адрес начала строки, но её значение в процессе выполнения программы может изменяться. Поэтому к ней можно, например, применять операцию инкремента. После выполнения операции txt_2+=5 указатель будет указывать на 6-й символ массива - *'g'*. Формально в этом случае инициализируется не массив, а переменная типа указатель.

Во втором варианте желателен спецификатор const, поскольку компилятор может разместить константы в памяти только для чтения, и попытка их изменить приведет к сбою. При этом не всякая версия среды разработки выдаст соответствуюшее предупреждение на этапе разработки. Поэтому всегда объявляйте указатели, в которые вы собираетесь записывать адреса строковых литералов как const char*. В этом случае компилятор не позволит модифицировать данные и диагностирует ошибку, что поможет вам исправить логику программы.

Следующий код иллюстрирует разницу инициализации:

Экран

```
4,  7
```

```
char *str1="string";
char str2[ ]="string";
printf("%d, %d",sizeof(str1),sizeof(str2));
```

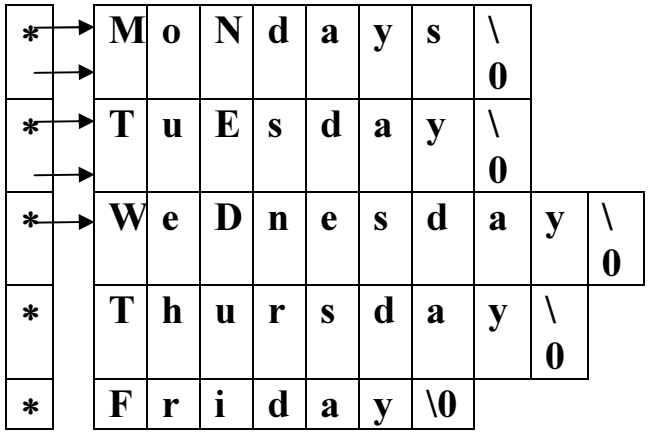
Массив символьных строк (двухмерный массив) может быть инициализирован двумя способами:

```
char week1[5][10]=
{"Monday","Tuesday","Wednesday","Thursday","Friday"};

const char *week2[5]=
```

{"Monday","Tuesday","Wednesday","Thursday","Friday"};

M	**O**	**n**	**d**	**a**	**y**	**s**	**\0**	**\0**	**\0**
T	**U**	**e**	**s**	**d**	**a**	**y**	**\0**	**\0**	**\0**
W	**E**	**d**	**n**	**e**	**s**	**d**	**a**	**y**	**\0**
T	**H**	**u**	**r**	**s**	**d**	**a**	**y**	**\0**	**\0**
F	**R**	**i**	**d**	**a**	**y**	**\0**	**\0**	**\0**	**\0**



Во втором случае описан массив указателей на строки. Оба примера показывают инициализацию двухмерного массива, причем в строку с номером *0* заносится строковая константа *"Monday"*, в строку с номером *1* - *"Tuesday"* и т.д.

Кроме различий, указанных ранее (т.е. второй вариант требует пять дополнительных ячеек памяти для хранения массива указателей), имеется ещё одно существенное различие. Первое описание за счёт наличия второго индекса *[10]* создаёт "прямоугольный" массив, в котором все строки имеют одинаковую длину - *10* символов (неиспользованные элементы инициализируются нулями), а второе описание определяет "рваный" массив, где длина каждой строки определяется размером соответствующего элемента в списке инициализации. Таким образом, при хранении текстов с разной длиной строк второй вариант экономит память.

Объявление массива символов из шести (*sizeof mas1==6*) элементов типа *char:*

*char mas1**[]={**'S', 'T', 'R', 'I', 'N', 'G'**};***

Следующее объявление формирует символьную строку из семи символов (имеется еще нулевой байт, *sizeof mas2==7)*:

*char mas2**[]={**"STRING"**};***

Следует чётко различать эти два случая и использовать тот или иной в зависимости от поставленной задачи.

8.2. Ввод - вывод символьных строк

При работе с библиотекой *stdio.h* можно использовать все три группы функций ввода/вывода – форматный, посимвольный и строками. Эффективно ввод/вывод строк реализован в функциях *gets, puts.*

Прототипы функций:

```
char *gets(char *str);
int puts(char *str);
```

Функция *gets* для работы получает адрес *str*, по которому будет расположено строковое значение. Она читает и размещает по указанному адресу символы до тех пор, пока не встретится символ новой строки *'\n'*, заменяет его нулевым символом *'\0'*, и, кроме того, возвращает введенную строку вызывающей программе посредством оператора *return*. В следующем коде в результате работы функции *gets* значения получают две переменные - *str* и *p*:

```
char *p, str[80];
p=gets(str);
puts(str);
puts(p);
```

Функция *puts* записывает в стандартный вывод с использованием буфера символы строки, начиная с указанного адреса до тех пор, пока не встретит нулевой символ '\0', заменяет его символом новой строки *('\n')* и осуществляет физический вывод на экран, освобождая буфер. В приведенном ранее коде представлены два способа вывода строки – через ее имя *str* и через переменную-указатель на нее *p.*

Функция *puts* имеет целый тип и возвращает значение последнего выведенного символа, который всегда является символом *'\n'*. В случае ошибки *puts* возвращает значение *EOF*.

```
printf ("%d", puts("Working function puts( )"));
                                    //работает функция puts( )
```

В результате на экран будет выведено число *10* - десятичный код символа *'\n'*.

Выведем на экран значения строк с указанного адреса:

```
const char *s1="pointer initialization ";//инициализация указателя
char s2[]="array initialization "; //инициализация массива
```

```
    puts(s1);
    puts(s2);
    puts(s1+8);
    puts(&s2[6]);
```

Экран

```
pointer initialization
array initialization
initialization
initialization
```

Функция *gets* при возникновении ошибки или при достижении конца файла (*EOF*) возвращает нулевой адрес *NULL.*

В следующем коде дублируются введенные строки текста до тех пор, пока не встретится символ конца файла:

Экран

```
one
one
two
two
three
three
^Z
```

```
//вводим через <enter> строки
//one,two,three и в конце
// комбинацию клавиш <ctrl>/<z>
    char str[20];
    while (gets(str))
      puts(str);
```

Наличие в функции *gets* механизма возврата через *return* позволяет использовать *gets* следующим образом:

while (gets(str)!=NULL)… или *while (gets(str))…*

Далее приведен код, в котором реализована функция *mylen_str* определения длины строки:

```
int mylen_str(char []); //прототип функции mylen_str
int main()
{ char *str=new char[80]; //выделение памяти под массив
  gets(str);
  cout<<mylen_str(str);
}
int mylen_str(char str[]) //определение функции mylen_str
{ int i=0;
  while(*str)                    //выход из цикла при *str=='\0'
    { i++;
      str++;
    }
  return i;
}
```

Остановимся на типичных ошибках при работе со строками текста:

```
char *ps;   //объявлен указатель, память под строковое
gets(ps);   //значение не выделена

const char *ps="string";
gets(ps);  //использование константы в качестве аргумента

char ps1[]="string1", ps2[]="string2";
ps2=ps1;    //попытка присвоить значение константе ps2
```

```
char str1[10], str2[10];
str1="Hello";  //имя массива (константа)нельзя
 str2=str1;    //использовать в левой части
                    //оператора присваивания
```

```
char str1[10]= "Hello";
char* str2;
str2=str1;
str2[1]='h';      //меняется также значение str1
```

В последнем фрагменте ошибка в коде только в том случае, если значение *str1* не предполагалось менять.

Отметим, что часть ошибок приведет к сбою на этапе компиляции, что позволит исправить ошибки. Гораздо опаснее «идеологические» ошибки (первый и два последних фрагмента кода), которые в результате выполнения программы приводят к непредвиденным ситуациям.

Таким образом, со строками (массивами символов и указателями на них) следует работать аккуратно.

Ввод/вывод строк можно осуществить с использованием объектов ввода/вывода библиотеки *iostream*:

```
char str[80];
cin>>str;  //ввод данных из потока
cout<<str; //вывод данных в поток
```

Результат работы следующего кода отображен на экране:

```
char st1[]="String1", st2[]="String2";
cout<<st1<<st2;
cout<<endl<<st1<<endl<<st2<<endl;
cout<<st1<<'\n'<<st2<<'\n';
```

Экран

```
String1 String2
String1
String2
String1
String2
String1   0096781C
```

cout<<st1<<'\t'<<(int)st1<<endl;*

где 0096781C – адрес переменной *st1*,а ее значение - *"String1"*

Ошибки при использовании объектов ввода/вывода строк:

```
char str[12];
cin>>str;
cout<<str;
```

Экран

```
First line
First
```

Принцип работы *cin* – для ограничения строки используются пробельные символы, поэтому *str* получает значение *"First"*.

Если в следующем фрагменте год и адрес вводить через *<enter>* (с новой строки):

```
int year;
char adress[80];
cin>>year;          //адрес вести не успеем!
cin>>adress;              //будет прочитан символ конца строки!
cout<<year<<'\n';
cout<<adress<<'\n'; //будет выведен год и пробел
```

Кроме того, после вывода строки, превышающей размер массива, программа может «зависнуть».

Файл *istream*, которому принадлежит объект *cin*, содержит строчно-ориентированные функции-элементы (методы) - *getline.* Обращение к этому методу:

```
cin.getline(str,20);
```

Читается не более *19* символов, включая символ новой строки, который удаляется, а в конец добавляется нулевой байт:

Экран

```
char s1[20],s2[20];
cin.getline(s1,20);
cin.getline(s2,20);
cout<<s1<<'\n'<<s2;
```

```
String1
string2
string1
string2
```

Можно то же самое записать короче, объединив методы:

```
cin.getline(s1,20).getline(s2,20);
```

Ввод/вывод строк можно реализовать с использованием функций форматного или посимвольного ввода/вывода. Функции *printf* и *scanf* не эффективны, если требуется выводить только строковые переменные, они занимают больше места в оперативной памяти и работают медленнее, чем специальные функции, предназначенные для ввода/вывода строк. В следующем примере форматный вывод строк с помощью функции *printf* привидит к размещению их в одной строке, т.к. спецификатор *%s* не преобразует символ ´\0´ в символ новой строки ´\n´.

```
char s1[10], s2[10], s3[10];
printf("Enter 3 string values:\n");
gets(s1); gets(s2); gets(s3);
printf("%s, %s, %s", s1, s2, s3);
```

Экран

```
Enter 3 string
values:
str1
str2
str3
str1, str2, str3
```

Необходимо явно указать в *printf*, что вы хотите выводить значение с новой строки:

```
printf("%s\n", str);
```

8.3. Функции обработки строк

Рассмотрим работу приведенной ниже программы. В данном коде при компиляции программы в результате работы директивы *define* идентификатор *ANSWER* будет замещен строкой *"Pascal"*:.

```
#include  <stdio.h>
#define ANSWER "Pascal"
void main( )
{
  char tryAns[20];
    // Кто создал первую суммирующую машину?
  puts("Who created the first computers?");
  gets(tryAns);
  while (tryAns!=ANSWER)
  {
    puts("Wrong! Try again!"); //Неверно! Повторите попытку
    gets (tryAns);
  }
    puts ("Correctly!Правильно!"); //Правильно!
}
```

В результате анализа кода программа должна заканчивать работу при правильном ответе – вводе с клавиатуры строки текста – *Pascal*.

Однако, сколько бы мы не вводили правильные данные (*Pascal*), мы будем получать на экране сообщение – «*Wrong! Try again!*». Наша программа работает не верно!

Ошибка в том, что *tryAns* и *ANSWER* на самом деле являются указателями (это массивы, а имя массива в C++ - адрес его местоположения в памяти), поэтому сравнение *(tryAns!=ANSWER)* спрашивает не о том, одинаковы ли эти две строки, а одинаковы ли адреса, на которые ссылаются *try* и *ANSWER*. Адреса *tryAns* и *ANSWER* различны, поэтому ваша программа представляет бесконечный цикл.

В языке C++ сравнение строк реализовано с помощью библиотечной функции *strcmp*. Чтобы использовать эту и множество других функций обработки строк, необходимо в программу включить файл *string.h*, который содержит прототипы функций работы со строками.

Приведем наиболее часто используемые функции:

- *Функция strcmp* (***STR****ing* ***C****o****MP****arasion) сравнения строк*

int strcmp ***(const*** *char* ****str1,const*** *char* ****str2);***

использует два указателя строк в качестве аргументов и возвращает значение 0, если значения строк одинаковы. В противном случае возвращает знаковое целое, равное разности первых несовпадающих символов сравниваемых строк:

```
#include<string.h>
int main( )
{  printf("%d\n", strcmp("A", "A"));                  //0
   printf("%d\n", strcmp("A", "B"));                  //-1
   printf("%d\n", strcmp("B", "A"));                  //1
   printf("%d\n", strcmp("C", "A"));                  //2
   printf("%d\n", strcmp("apples", "apple"));   //115
   return 0;
}
```

Функция возвращает отрицательное число, если первая строка предшествует второй в алфавитном порядке (т.е. из кода символа первой строки вычитается код символа второй строки). В последнем примере "*apples*" и "*apple*" совпадают в первых пяти символах, а шестой символ

первой строки *'s'* сравнивается с шестым символом второй строки, который является нулевым символом (*0* в *ASCII*). Возвращаемое значение

's'-'\0'=115-0=115,

где *115* является кодом буквы *'s'* в *ASCII* последовательности кодов.

Исправим ошибку в программе поиска ответа на вопрос о создателе вычислительной машины. Выражение в цикле *while* будет иметь вид:

while (strcmp(tryAns, ANSWER))

Т.к. ненулевые значения всегда интерпретируются как "истина", то мы можем в операторе *while* записать выражение (*strcmp(try, ANSWER*)) без его сравнения с нулем.

- *Функция strlen (**STR**ing **LEN**gth) определения длины строки*

*int strlen (const char *str);*

определяет количество значащих символов до первого нулевого символа *'\0'*и, не учитывая его, возвращает длину строки в байтах:

```
char s[80];
int  len = strlen(gets(s));
printf("Фактическая длина строки %d символов \n", len);
```

- *Функция srtcat (**STR**ing con**CAT**enation) объединения двух строк*

char strcat(char *str1, const char *str2);*

присоединяет копию второй строки к концу первой, и это объединение становится новым значением первой строки. Вторая строка не меняется. Функция возвращает указатель на первую строку:

```
char str1[80]= "What is";
char str2[80]= "your name?";
strcat(str1,str2);
puts(str1);                          //результат What is your name?
```

Здесь все же обратим внимание на возможную ошибку, которую можем сделать, если знаем язык Паскаль:

```
    char str1[80]= "What is";
char str2[80]= "your name?";
    char str3= str1+str2;          //это сложение указателей!
```

Данная ошибка обнаруживается компилятором.

Обратите внимание! Функция *strcat не проверяет*, умещается ли объединение строк в первом массиве символов. Никогда не забывайте контролировать используемую память. Если памяти не хватает, часть данных просто теряется!

- *Функция strcpy (**STR**ing **CoPY**) копирования строк*

```
char * strcpy (char *str1,const char *str2);
```

копирует посимвольно строку, на которую указывает *str2*, в строку, на которую указывает *str1*, и возвращает адрес *str1*.

Второй указатель, ссылающийся на исходную строку, может быть объявленным указателем, именем массива или строковой константой. Первый указатель, ссылающийся на копию, должен ссылаться на массив или часть массива, имеющего размер, достаточный для размещения строки:

```
char mas[30], *ptr="Hello";
strcpy(mas,ptr);
puts(ptr);          // Hello
puts(mas);          // Hello
```

Обратите внимание! Функция *strcpy* также *не проверяет*, умещается ли вторая строка в первом массиве символов, и в случае ошибки последствия могут быть непредсказуемы.

Следующий код копирует значение *mas2* в *mas1* и выводит *mas1* в обратном порядке:

```
char mas1[20], mas2[20], *p;
int i;
gets(mas2);                    //если ввели   a b c d e
p=strcpy(mas1, mas2);
    p+=strlen(mas1)-1;
for(i=strlen(mas1); i>=0; p--, i--)
```

*putchar(*p);* ***//результат e d c b a***

- *Функция strchr поиска символа в строке*

  char strchr(**const** char * str, int ch);*

ищет символ *ch* в строке и возвращает указатель на первое местоположение символа *ch* в строке с адресом *str*. Символ может быть нуль-символом *´\0´*, тогда функция возвращает его адрес.

- *Функция strstr поиска подстроки в строке*

  char strstr(const char* str1,const char* str2);*

ищет подстроку *str2* в строке *str1* и возвращает указатель на первое вхождение подстроки. Функция возвращает *NULL*, если вхождение подстроки не найдено.

- *Функция strncat объединения двух строк*

*char * (char * str1,**const** char * str2, int n);*

причем из строки *str2* копируется в конец *str1* не более *n* символов.

- *Функция strncmp сравнения первых n символов двух строк*

*int strncmp(**const** char * str1,**const** char * str2, int n);*

в случае равенства возвращает ноль, в противном случае разность *ASCII* кодов первых несовпадающих символов.

- *Функция strncpy копирования не более n символов строки s2*

  *char * strncpy(char *str1,const char *str2, int n);*

- *Функция, возвращающая указатель на последнее вхождение символа ch в строку, и NULL в случае его отсутствия*

  *char * strrchr(const char *str, int);*

- *Функция, возвращающая указатель на первое вхождение любого из символов строки str2 в строку str1*

  *char * strpbrk(**const** char *str1**, const** char *str2**);***

- *Функция, возвращающая следующую лексему из str1, отделенную любым из символов из набора str2*

  char * strtok**(**char *str1**, const** char *str2**);**

8.4. Функции преобразование символьных строк

Часто необходимо преобразовать число в строку и наоборот. В C++ реализовано целое семейство функций, выполняющих данные преобразования. Для их использования требуется подключить стандартные заголовки *stdlib.h* или *cstdlib.*

При вводе чисел числовые данные можно считывать как строки и затем использовать функцию преобразования строки в числовое значение. Функции *atof, atoi* и *atol* преобразуют строку в число типа *float, int* или *long*. Функции *itoa* и *ltoa* выполняют обратное преобразование.

Используя эти функции, можно избежать ряда ошибок, которые могут возникнуть при использовании функции *scanf* и *printf* при вводе/выводе данных.

Прототипы функций:

```
int atoi (const char *str);   //преобразует строку в целое число
long atol  (const char *str); //преобразует строку в длинное целое
double atof (const char *str); //преобразует строку в вещественное
                    //число двойной точности
```

Следующие фрагменты показывают результаты работы функций преобразования символьных строк:

```
printf("%d, %ld, %f\n", atoi("1234"), atol("123456"),
   atof("123.45"));                  //1234, 123456, 123.450000
printf("%d, %f, %f", atoi("123a4"), atof("12e2"),
   atof("a"));                       //123, 1200.000000, 0.000000
```

Результаты работы функций приведены в комментариях. Функции прекращают преобразования, если встречается не цифровой символ, и возвращают сформированное к данному моменту значение.

Отметим, что пробелы и знаки табуляции перед цифрами игнорируются, а знаки ´+´ или ´-´ преобразуются в знаки числа:

```
printf("%d, %d\n", atoi("+123"), atoi("-123"));
                                        //123,  -123
```

Функции, преобразующие целое значение в строковое представление чисел:

```
char *itoa (int value, char *str, int r);
char *ltoa (long value, char *str, int r);
```

Целое значение *value* преобразуется в строку символов, завершающуюся *'\0',* и помещается по адресу *str*. Целый аргумент *r* определяет систему счисления.

Преобразование числа в строковое представление в десятичной системе счисления:

```
char str[10];
str = itoa (1234, str, 10);       //str= =”1234”
```

Напомним, что для преобразования числа в строковое значение требуется выделить необходимый блок памяти для хранения строки.

9. Классы памяти и область действия объектов

В данном случае речь не идет об объектах классов, т.е. об объектно-ориентированном программировании на C++, а об унаследованном из языка Си понятии *объекта* – области памяти компьютера, которая может использоваться программой. Атрибуты объектов – это имя, тип, область действия и время жизни объекта.

Мы уже знаем, что

- объект (переменные, константы, функции) – это некоторая область памяти, поименованная с помощью идентификатора;
- тип данных указывает компилятору, сколько байт памяти нужно выделить для размещения значения, интерпретирует смысл значения и определяет правила его использования, а также задает совокупность допустимых над ним операций;
- в соответствие с типом при определении значения переменной в соответствующую ей область памяти помещается некоторый код. Если переменная получает значение при компиляции, то говорят о ее инициализации, если при выполнении программы – то о присваивании значения.

Каждая переменная, кроме типа, принадлежит к некоторому *классу памяти*, который определяет, как компилятор размещает переменную в памяти: в сегменте данных, стеке или в регистре, и соответственно время ее существования.

Таким образом, *класс памяти* определяет:

- область действия переменной;
- время, в течение которого переменная может сохранять свое значение.

По классам памяти переменные делятся на автоматические (внутренние), внешние, статически внутренние, статически внешние и регистровые. Для описания классов памяти используются специальные ключевые слова: *auto, extern, static, register*.

9.1. Описание классов памяти переменных

Класс памяти зависит от того, где переменная объявлена и какое ключевое слово (если оно есть) используется.

1. *Автоматические переменные*

Переменные, которые объявлены внутри функции, а также аргументы функции называются автоматическими или внутренними (локальными) переменными.

```
int fun_A()
{
int x,y;    //по умолчанию x,y являются автоматическими
auto int z; //явно подчеркнуто с помощью auto
. . .
}
```

Автоматические переменные имеют локальную область действия, она ограничена блоком { }, в котором объявлена. Автоматические переменные создаются при каждом входе в функцию и пропадают после выхода из неё.

2. Регистровые переменные

Если переменная объявлена с помощью ключевого слова *register*, делается попытка хранить соответствующий объект в быстродействующей памяти – регистрах процессора, которые могут быть использованы для хранения данных, к которым требуется многократный доступ, например, счетчиков:

```
register int index;
```

Так как у компьютера число регистров ограничено, то определение объекта как регистрового не гарантирует, что для его хранения будет выделен именно регистр. Если свободных регистров нет, то такой объект будет создан как автоматический.

Если размер памяти, занимаемой переменной, превышает разрядность регистров, то переменная трактуется как автоматическая. Для регистровых переменных неприменима операция получения адреса &.

3. Внешние переменные

Внешний объект объявляется с помощью ключевого слова *extern* и доступен во всех файлах программы, т.е. имеют место внешний тип компоновки и статическая продолжительность существования объекта. Такая переменная существует и сохраняет своё значение в течение всего выполнения программы.

Таким образом, чтобы сделать переменные видимыми вне файла, в котором они объявлены, необходимо объявить их тип с использованием ключевого слова *extern*.

Обычно создают заголовочный файл, содержащий функции и переменные, для которых требуется обеспечить видимость в программе, и с помощью директивы *#include* его включают в другие файлы.

Модификатор *extern* обычно используется при раздельной компиляции исходных файлов, в дальнейшем объединяемых при

построении исполняемого *.exe* файла. Программа может состоять из нескольких исходных файлов, которые компилируются отдельно, а затем связываются в один файл задачи – файл проекта.

Если необходимо использовать внешнюю переменную в том файле, где она не объявлена, или использовать её до того, как она объявлена, применяется описание *extern*:

```
//файл file1.cpp
int x, y;       //объявление глобальных переменных
int main ()
{   ...
       fun_1();
   int a=x;  //использование внешней переменной после
                          //ее определения в fun_1
}

fun_1()
{   ...
   x = 2;            //определение переменной, ее инициализация
   ...
}

//файл file2.cpp
extern int x;   //описание внешней переменной,
              //создание ссылки на внешний объект
fun_2()
{   ...
   c = x    //использование внешней переменной в том файле,
              //где она не определена
}
```

Обычно объявления глобальных объектов помещаются в заголовочный файл, который включается в модули программы.

Описание *extern* не выделяет дополнительной памяти, оно сообщает компилятору, что переменная с таким именем и данного типа объявлена позже или в другом файле.

Отметим, что внешние переменные существуют всё время работы программы.

Обратите внимание! Все функции в C++ являются внешними объектами, т.е. одна функция не может быть определена внутри другой.

Имена функций не являются переменными, но все правила, касающиеся внешних переменных, справедливы и для имён функций.

4. Внутренние статические переменные

Модификатор *static* сообщает компилятору о необходимости статического выделения памяти для переменной или функции, но по области действия внутренняя статическая переменная аналогична автоматической, т.е. доступна только внутри функции, в которой она объявлена (имеет место внутренний тип компоновки). Такая переменная существует в течение всей программы и сохраняет своё значение между вызовами функции.

В следующем коде переменная *x* инициализируется каждый раз при вызове функции, а статическая переменная *y* – только один раз при компиляции функции:

```
int main ( )
 { int i ;
      for (i =1; i<= 3; i ++)
            fun();
      return 0;
 }

fun()
 { int  x = 1;
      static  int  y = 1;
      printf ("%d, %d\n", x++, y++);
 }
```

Экран

```
1,  1
1,  2
1,  3
```

Т.к. по умолчанию функция имеет внешний класс памяти, это позволяет другим модулям в программе обращаться к ней. Если необходимо ограничить видимость функции ее собственным модулем, ее объявляют статической – *static*, причем ключевое слово *static* используется только в прототипе функции.

5. Внешние статические переменные

Внешняя статическая переменная объявляется с использованием ключевого слова *static* вне функции.

Если обычная внешняя переменная может использоваться функциями из любого файла, то внешняя статическая переменная может использоваться только функциями файла, в которых она объявлена, причем только после определения переменной.

Табл. 4 отражает возможности работы с переменными, имеющими соответствующий класс памяти.

Таблица 4

Класс Памяти	Ключевое слово	Продолжительность существования	Место объявления	Область действия
Автоматический	*Auto*	временно	локальная	внутри блока
Регистровый	*Register*	временно	локальная	внутри блока
Статический внутренний	*Static*	постоянно	локальная	внутри блока
Статический внешний	*Static*	постоянно	глобальная (один файл)	вне блока
Внешний	*Extern*	постоянно	глобальная (все файлы)	вне блока

Рекомендуется в основном применять автоматические переменные и не злоупотреблять внешними, поскольку это противоречит одному из главных принципов «защитного программирования» – «необходимо знать только то, что нужно».

9.2. Инициализация переменных с классом памяти

Мы уже говорили, что если переменной явно не присвоено значение, ее нельзя использовать в выражениях, предполагая, что ее содержимое – «мусор». Рассмотрим, каким образом класс памяти влияет на инициализацию переменных по умолчанию.

1. Постоянная (статическая внутренняя, статическая внешняя и внешняя) переменная или массив инициализируется нулями, если в операторе объявления явно не задано другое значение.

2. Значения временных переменных, не инициализированных в операторе описания, не определены (содержат «мусор»).

Если *массив статический или внешний*, то он по умолчанию инициализируется нулями.

3. Если *только часть* элементов массива *получает значения при инициализации*, то оставшиеся элементы инициализируются нулями соответствующих типов.

4. Если *часть элементов* массива *получает значения в процессе работы* программы, то оставшаяся часть - либо «мусор» (внутренняя переменная), либо нули (внешняя).

Приведем примеры объявлений внешних переменных (они объявлены вне функции *main)*:

```
int x;                          //x=0
float y [5];                    //y[0]=0.0, ...,  y[4]=0.0
float  *pnt;                    //pnt=0
char mas [4]                    //mas[0]= '\0', … , mas[3]= '\0'

main( )
{
        int  y[5];                      //«мусор»
         static int z[5];       //нули
          . . .
}
```

10. Функции

Функция – это поименованная, логически самостоятельная часть программы, которую определяют один раз, а вызывают многократно из разных точек программы, передавая в нее аргументы.

Если функция в результате работы возвращает значение, она может вызываться в выражении. Тип функции должен совпадать с типом возвращаемого значения. Если функция не возвращает значение, она объявляется с типом *void*. В C++ функция может возвращать типы классов, в частности контейнеры, что будет рассмотрено в ООП.

Функции, из которых состоит программа, могут группироваться в отдельные модули (файлы), которые компилируются отдельно, а затем объединяются в исполняемую программу с помощью программы компоновки. Чтобы использовать модуль, необходимо знать его интерфейс – т.е. заголовки всех функций и описание доступных извне типов, объявления переменных и констант. Чтобы использовать функцию, необходимо знать ее интерфейс – это заголовок функции (имя, тип функции, и типы ее параметров). Количество, порядок следования и типы аргументов определяют сигнатуру функции.

10.1. Объявление и определение функций

Прежде чем использовать функцию, ее необходимо объявить.

Объявление функции (прототип) задаёт её имя, тип возвращаемого значения и типы аргументов и заканчивается символом точка с запятой.

Прототип может размещаться как в теле программы (до первого обращения к функции), так и в отдельном файле заголовков. Прототип функции снабжает компилятор информацией о том, аргументы какого типа ожидает функция, и значение какого типа она возвращает.

Определение функции содержит заголовок с указанием списка передаваемых параметров без точки с запятой в конце заголовка, и тело функции (объявления и операторы, заключенные в фигурные скобки).

При *вызове функции* вместо формальных параметров, указанных в ее заголовке, подставляются фактические значения – переменные, выражения или константы. Проверка соответствия типов формальных и фактических параметров осуществляется на этапе компиляции.

В каждой программе на C++ есть специальная функция *main*, которая получает управление при входе в программу и называется главной функцией. С нее начинается выполнение программы. Функция *main*, также как и другие функции, может иметь аргументы. Работа функции *main* с аргументами рассмотрена в главе 12.

Программа на С++ состоит из функций, которые взаимодействуют между собой во время работы программы.

В следующем примере функция *main* вызывает функцию *sum* вычисления суммы двух чисел:

```
float sum(float, float);      //объявление функции
int main()
{  float a,b,c;
   b = 10; c = 20;
   a = sum(b,c);              //вызов функции
}
float sum(float x, float y)       //определение функции
{ return  (x+y);
}
```

Очевидно, что количество, последовательность и типы параметров в прототипе (объявлении) и заголовке функции должны совпадать.

Определение функции создаёт объект данных, то есть определение функции - это исполняемая инструкция для компилятора.

Объявление функции нужно для того, чтобы сообщить компилятору, что функция имеет параметры соответствующих типов и возвращает значение указанного типа.

Выход из функции выполняется инструкцией *return(выражение);* При этом вычисляется значение выражения, которое передается в вызывающую функцию через имя функции. Значение перед возвратом преобразуется к типу функции.

Определим функции получения абсолютного значения и выбора максимума из двух чисел:

```
int abs(int i)
{
   return(i<0 ? -i : i);
}

int max(int x,int y)
{
   return(x>y ? x : y);
}
```

Вызов этих функций, где в качестве фактических аргументов используются константы, имеет вид:

```
int n = abs(-5);
int z = min(10,5);
```

Инструкцию *return* можно опустить, если тип функции *void*.

Если в инструкции *return* не задано выражение, то просто осуществляется возврат управления, как и в случае достижения последней закрывающей фигурной скобки функции.

Примеры объявления функций без параметров:

```
void fun1();   //функция не возвращает значение
char *fun2();  //функция возвращает указатель на символ
char **fun3(); //функция возвращает указатель, который
               //содержит адрес указателя на  символ
```

10.2. Связь между функциями

При использовании функций линейный ход выполнения программы нарушается в момент передачи управления функции (вызова функции).

Для работы функций используется стековая память. При каждом вызове функции в стеке выделяется память для размещения всех параметров и локальных переменных функций, а также адреса возврата. По соглашению о вызовах в С++ параметры в стек помещаются в последовательности справа-налево, после чего в стек помещается адрес возврата. При завершении работы функции эта память освобождается.

При вызовах функций всегда строго соблюдается вложенность, поэтому в вершине стека всегда находятся локальные переменные и параметры активной в данный момент функции. Этот прием делает возможной легкую реализацию рекурсивных процедур. Когда функция вызывает сама себя, для всех ее локальных переменных выделяется новая память в стеке, и вложенный вызов работает с собственным представлением локальных переменных. Когда вложенный вызов завершается, занимаемая его переменными область памяти в стеке освобождается и актуальным становится представление локальных переменных предыдущего уровня.

Связь между функциями осуществляется через их *параметры*, *возвращаемые функциями значения* (механизм *return*) и через *внешние глобальные переменные*.

Передача параметров в функции осуществляется тремя способами: *по значению, по адресу и по ссылке.*

10.2.1. Передача параметров в функцию по значению

При передаче параметров по значению в момент вызова функции ее формальные параметры инициализируются значениями соответствующих фактических параметров. Вызываемая функция получает временные копии фактических параметров, и операторы функции работают с этими копиями. В этом случае функция не может воздействовать на фактический параметр вызывающей функции, т.е. изменить его значение.

Например, функция вычисления факториала получает копию фактического параметра 20, с которой работает, и возвращает значение факториала, сохраняемое в переменной *f*, через инструкцию *return*:

```
long k = fact(20);          //вызов функции

long fact (int n)           //определение функции
{ long f;
  if(n<0) return(n);
  for(f=1; n; --n)
  f*=(long)n;        //(long)- операция приведения типа
  return(f);
}
```

Рассмотрим задачу сортировки трех значений с помощью функции. Функция *sort* получает при вызове значения трех параметров и должна расставить их по убыванию. Посмотрим, что получится, если параметры передавать по значению:

```
void sort (int, int, int);          //прототип

void main()
{  int a, b, c;
   cin >> a >> b >> c;
   sort(a, b, c);                        //вызов функции sort
   cout << a << b << c;                        //вывод результата
   return;
}

void sort(int a, int b, int c)   //определение функции
{  int x;
if (c>a) {x=c; c=a; a=x;}
   if (c>b) {x=c; c=b; b=x;}
   if (b>a) {x=a; a=b; b=x;}
```

```
  return;
}
```

Программа не сортирует данные, на экран выводятся значения в исходном порядке, т.к. функция отсортировала их копии и после завершения ее работы данные потеряны.

Чтобы обойти это ограничение, используется передача параметров в функцию по адресу.

10.2.2. Передача параметров в функцию по адресу

Если функция должна влиять на значения фактических параметров, как в рассмотренном примере сортировки, то в вызываемую функцию необходимо передать адреса фактических параметров, а не их значения. Формальные параметры в этом случае представляют собой локальные (временные) переменные, которые получают значения адресов передаваемых объектов, и функция во время работы оперирует непосредственно с фактическими параметрами, влияя на их значения.

Так как через инструкцию *return* функция может вернуть в вызывающую функцию только одно значение, передача параметров по адресу имеет еще один важный смысл - позволяет функции возвращать через параметры сразу несколько значений тому процессу, из которого она была вызвана.

Передача параметров по адресу часто оправдана в случае их большого размера, например, больших массивов или объектов типа класса. Временные и пространственные расходы на размещение и копирование таких объектов могут оказаться неприемлемыми для реальной программы, и передача параметров по адресу позволяет сэкономить время и память.

Передача параметров по адресу возможна двумя способами:

- объявлением параметров указателями;
- объявлением параметров ссылками.

При объявлении параметров указателями функция s*ort* имеет вид:

```
void sort(int *pa, int *pb, int *pc)
{
  int x;
  if (*pc>*pa) {x=*pa; *pa=*pc; *pc=x; }
  if (*pc>*pb) {x=*pc; *pc=*pb; *pb=x; }
if (*pb>*pa) {x=*pa; *pa=*pb; *pb=x; }
  return;
```

}

а ее вызов: *sort (&a,&b,&c);*

При передаче адреса фактического параметра в функцию с помощью указателя в фактическом выражении используется операция взятия адреса &, а для получения значения в функции этот адрес разыменуется (используется операция разадресации *).

При передаче параметров по адресу функция может несанкционированно изменить внешний по отношению к ней объект. Чтобы защитить фактические параметры, необходимо запретить их изменение внутри функции с помощью модификатора *const*:

*void fun(const int *);*

Неудобство написания кода и его «утяжеление» вызывает необходимость разадресации указателей. Этого можно избежать, передавая параметры в функцию *по ссылке*.

При передаче адреса параметров по ссылке в функцию передаётся адрес указанного при вызове параметра, а внутри функции все обращения к этому аргументу неявно разыменовываются.

Обратите внимание! Ссылка не создает копии объекта. Она, в отличие от указателя, не занимает места в памяти, это другое имя уже существующего объекта.

При работе с функциями ссылки используются в качестве их параметров и типов возвращаемых значений.

Прежде, чем рассматривать передачу параметров по ссылке, приведем примеры объявления ссылок (*alt* и *NewStroka*) в коде:

```
int x;
int  &alt=x;                    //ссылка alt-альтернативное имя для x
const char &NewStroka='\n';     //ссылка на константу '\n'
```

Справа от оператора присвоить стоит существующий объект, под который выделена память в соответствии с его типом. Слева – ссылки на объекты (*alt* и *NewStroka*). Значение ссылки - адрес этого объекта.

Чтобы понять синтаксис объявления и использование ссылок, объявим переменную *point* - указатель на целое, и присвоим ему значение адреса целой переменной *x*:

*int *point = &x;*

В соответствии приведенным ранее описаниям следующие выражения истинны:

**point == alt , point ==&alt.*

Таким образом, операции над ссылкой – это операции над тем объектом, с которым они связаны.

Определение функции *sort* с параметрами ссылками имеет вид:

```
sort(int &a, int &b, int &c)
{
  int x;
  if (c>a) {x=c; c=a; a=x;}
  if (c>b) {x=c; c=b; b=x;}
  if (b>a) {x=a; a=b; b=x;}
  return;
 }
```

а вызов функции соответственно: *sort (a,b,c);*

Применение ссылок улучшает читаемость программы, т.к. не требуется использование операции получения адреса & и разыменовывания *.

Обратите внимание! При использовании ссылок в функцию реально передаются указатели, хотя из синтаксиса этого не видно. Если функция возвращает значение ссылки, то аналогично возвращается указатель на объект, а не сам объект.

При объявлении ссылок можно сделать следующие ошибки:

```
int &p;                  //нельзя не инициализировать ссылку
void &p=x;               //ссылка не может иметь тип void
int & x[5]={1,2,3,4,5};  //нельзя создать массив ссылок
int &&px=x;              //нельзя создать ссылку на ссылку
int &py=new(int&);       //нельзя динамически выделить
                         // память под ссылку
```

10.2.3. Передача в функцию массива

Описание массива, прототип и вызов функции, осуществляющей ввод *n* элементов в массив, имеют вид:

```
int mas[100];                 //описание массива
int input(int [], int);       //прототип функции
input(mas,n);                 //вызов функции
```

При передаче массива в функцию необходимо в прототипе и заголовке функции указать его тип, а также количество элементов в массиве. Вызов функции передает имя массива *mas* в функцию, а также указывает количество элементов *n*, которые необходимо в него ввести.

Обратите внимание! В C++ массивы в функции передаются *только как указатель на его первый элемент!* Имя массива *mas* является адресом его расположения в памяти. Поэтому функции работы с массивами могут, если это нужно, изменить их содержание.

Обратите внимание! Размер массива не является частью типа параметра. Поэтому размер одномерного массива в прототипе функции можно не указывать (пустые квадратные скобки обязательны). Размер массива также можно не указывать в заголовке определения функции *input*.

Поясним, почему размер массива не важен при объявлении его параметром.

Так как массив в языке C++ всегда передается по его адресу, мы должны всегда передавать адрес его первого (по синтаксису нулевого) элемента. Если массив статический – это можно сделать по его имени, указателю на такой же тип массив или по ссылке, если динамический - по указателю на его первый элемент.

Таким образом, следующие объявления функций эквивалентны:

```
int input(int [20]);
int input(int []);
int input(int  *);
```

Функция не знает реального размера передаваемого массива, компилятор C++ также не может это проверить (в C++ не реализована проверка выхода индекса за пределы объявления его размера). Поэтому при передаче массива в качестве параметра необходимо указывать его размер:

```
int input(int [], int);    //прототип функции
```

```
int input(int x[], int n) //определение функции
{
  . . .                          //тело функции
}
```

Если функция для обработки получает многомерный массив, то объявление его параметром функции может не содержать размер значения первого индекса (наличие квадратных скобок обязательно), при этом значения второго и последующих индексов обязательно:

```
int sum1(int [][10]);
int sum1(int (*)[10]);
```

Круглые скобки для операции * во втором объявлении обязательны – объявлен указатель на одномерный массив из 10-ти целых (иначе будет объявлен массив из 10-ти указателей на целые).

Код программы ввода/вывода двухмерного массива имеет вид:

```
#include "conio.h"
#include <iostream>
using namespace std;

void inp(int [][10],int,int);          //прототип функции ввода
void out(int [][10],int,int);          //прототип функци вывода

int main()
{
      int mas[5][10];
      inp(mas,2,3);              //вызов функции ввода
      out(mas,2,3);                    //вызов функции вывода
      getch();
      return 0;
}

void inp(int mas[][10],int n,int m)  //определение
{
      int i,j;                                    //функция ввода
      puts("Введите элементы массива:");
      for (i=0;i<n;i++)
            for (j=0;j<m;j++)
```

```
                cin>>mas[i][j];
}

void out(int mas[][10],int n,int m)   //определение
{
      int i,j;                                    //функция вывода
      for (i=0;i<n;i++)
            for (j=0;j<m;j++)
                  cout<<mas[i][j];
}
```

Для работы с массивом можно использовать передачу не самого массива, а переменной-указателя на него:

```
void in_Ar(int (*p)[10],int n,int m);        //прототип

main()
{
 int mas[5][10],(*p)[10];
 p=mas;
 in_Ar(p,2,3);                              //вызов функции ввода 6-ти
.       .       .                                  //элементов в массив
}
void in_Ar(int (*p)[10],int n,int m) //определение функции
      {
        int i,j;
        for (i=0; i<n; i++)
     for (j=0; j<n; j++)
               cin>>*(*(p+i)+j);
      }
```

Если создан динамический массив в соответствии следующего объявления указателя на него *int **px* (п.7.2), то прототипы функций ввода/вывода будут иметь вид:

```
int **input(int **px,int,int);  //функция ввода
void output(int **px,int,int);          //функция вывода
```

10.2.4. Ссылочный тип функции

Функции могут возвращать значения типа ссылка.

В следующем коде функция возвращает ссылку на максимальный элемент массива. Продемонстрируем вызов функции двумя способами:

```
int &max(int n, int x[]);

int main()
{ int x[]={2, 4, 10, 8};                    //инициализация массива
  int n=sizeof(x)/sizeof(int);    //размер массива
  cout<<"max="<<max(n,x)<<'\n';          //вызов функции max
  max(n,x)=0;          //второй способ вызова функции с заменой
                  //максимального значения нулем
  cout<<"new array:"<<'\n';
  for (int i=0; i<n; i++)
     cout<<"\t"<<x[i];     //вывод нового значения массива
  cout<<"max="<<max(n,x)<<'\n';          //вызов функции max
  return 0;
}

int &max(int n, int x[])
{ int i_max=0;
  for (int i=0; i<n; i++)
     i_max=x[i]>x[i_max]? i : i_max;
  return x[i_max];
}
```

Здесь допустим вызов функции в левой части оператора присвоить (второй способ), при этом изменяется значение `x[2]`, т.к. `max` возвращает ссылку на целое `x[2]`.

Экран

```
max=10
new array:
     2    4    0    8
max=8
```

10.3. Функции с переменным числом параметров

В языке программирования C++ допускается использование функций с переменным количеством параметров. В этом случае в конце списка параметров ставится многоточие (…). Примерами функций с переменным количеством параметров являются стандартные функции *scanf* и *printf*.

Определим функцию для сложения произвольного количества целых чисел следующим образом:

```
int add_num(int n, ...); //n - количество слагаемых
```

Для обработки функций с переменным количеством параметров используется переменная типа *va_list* и три макрокоманды, которые определены в заголовочном файле *stdarg.h*:

```
va_start(va_list ap, last_arg);
va_arg(va_list ap, type);
va_end(va_list ap);
```

Макрокоманда *va_start* инициализирует переменную *ap* для начала извлечения аргументов функции, а параметр *last_arg* должен быть именем последнего заданного параметра функции.

Макрокоманда *va_arg* возвращает следующий аргумент функции, причем параметр *ap* должен совпадать с соответствующим параметром из макрокоманды *va_start*, а параметр *type* должен указывать тип следующего аргумента.

Макрокоманда *va_end* после завершения работы с аргументами функции обеспечивает правильную работу инструкции *return* в функции с переменным количеством параметров.

Приведем пример функции *print*, которая выводит на экран переменное количество целых чисел, передаваемых ей в качестве параметров:

```
#include <stdio.h>
#include <stdarg.h>
int print(int, ...);

int main()
{
  print(0);
  print(1, 2);
  print(2, 4, 6);
  return 0;
}
int print(int n, ...)
{
```

```
 va_list a;
if (n < 1)                    //проверяем, есть ли параметры
   {  printf("There are no numbers.\n");
      return 0;
   }
printf("Number of parameters = %d\n", n);
   va_start(a,n);      //инициализируем a последним аргументом
   while(n)
   {
         int x=va_arg(a,int); //получаем следующий аргумент
         printf("%d\n", x);
         --n;
   }
   va_end(a);          //завершение работы с аргументами
   return 0;
}
```

10.4. Указатели и ссылки на функции

Имя функции без скобок в C++ является указателем на эту функцию, его значение – адрес размещения функции в памяти. Адресом функции является ее точка входа.

Возможны только две операции с функциями: вызов и взятие адреса. Если имя функции присвоить указателю, его можно использовать для вызова функции.

1. Указатели на функции

Объявим указатель на функцию с двумя аргументами типа *int,* возвращающую указатель на тип *int*:

```
int *(*pfun)(int,int);
```

Скобки *(*pfun)* обязательны, без них будет объявлен не указатель *pfun*, а функция с двумя аргументами целого типа, возвращающая указатель на указатель на тип *int*. Ошибка возникнет, так как операция скобки *()* имеет более высокий приоритет, чем операция *, и сначала идет обращение к функции *pfun*, и только затем реализуется разыменовывание результата.

Фрагмент кода, в котором в момент вызова функции инициализируются значения ее аргументов:

```
void print(int);
int main()
{   int y;
    void  (*p)(int)=print;  //описание указателя на функцию
                            //с его инициализацией
    print(y=5);                     //первый способ вызова функции
    (*p)(y=5);                      //второй способ вызова функции
}

void print(int n)           //оределение функции print
{
    printf ("%d/n", n);
}
```

При разработке меню, т.е.функций, управление которыми выполняется с помощью меню, удобно использовать массив указателей на функции.

В следующем коде объявлена функция *fun,* в качестве аргумента получающая значение указателя на функцию типа *void* с аргументом целого типа:

```
typedef  void(*PTF)(int) //объявление нового типа данных PTF
void fun (PTF);                  //объявление функции с параметром
                                        //указатель на функцию
void fun1(int);      //объявление функций с одинаковой сигнатурой
void fun2(int);
void fun3(int);

int main()
{
   int y;
   void (*p[5])(int); //массив из 5 указателей на функции
                      //с целым аргументом без возвращаемого значения


   p[0]=fun1;               //инициализация элементов массива
   p[1]=fun2;               //именами функций: p[i] присваивается
   p[2]=fun3                      //значение адреса функции
   for(int i=0; i<2; i++)
               p[i](i*5);   //вызов в цикле функций через указатель
```

```
                                    //с аргументом  i*5
  fun(p[0]);                //вызов функции fun, получающей
                                    //в качестве аргумента функцию  fun1
}

void fun(PTF  p_fun) //определение функции fun
{
      p_fun(10);          //вызов функции p[0] с параметром 10
}
```

2. Ссылки на функции

Ссылка на функцию обладает всеми правами основного имени функции, т.е. является *синонимом* ее имени. Изменить значение ссылки на функцию невозможно, поэтому указатели на функции имеют большую сферу применения, чем ссылки.

При объявлении ссылки инициализируются именем функции:

```
void (& alt)(int)=fun1;   //два способа инициализации ссылки
void (& alt)(int)(fun1);
```

Инициализирующее значение – это имя уже объявленной функции, имеющей тот же тип и ту же сигнатуру, что и ссылка.

10.5. Перегрузка и шаблоны функций

В языке С++ можно определять несколько функций с одним и тем же именем. Конфликт имен не возникает, так как в процессе компиляции принимается во внимание тип функции, и количество и типы ее параметров. В результате вызывается именно требуемая функция. Каждая из перегруженных функций имеет свое собственное определение. По вызовам функций компилятор выбирает нужное определение функции.

Предоставление компилятору выбора среди нескольких функций называется *перегрузкой.* Повторное использование имени функции удобно, когда решается одна и та же задача, но с различными типами и количеством параметров.

Объявим прототипы функций, имеющих одинаковое имя, но различающихся списками аргументов по типу и по числу:

```
int max(int,int);               //возвращает большее  из 2-х целых
float max(float,float,float); //большее  из 3-х float
```

int max(int,int [20]); ***//максимальное значение массива***
*int max(int,int,char **);* ***//максимальную длину строки текста***

Определим первые две перегруженные функции:

```
int max(int x, int y)
{
    return x>y ? x : y;
}

float max(float x, float y, float z)
{
    return x>y ? (x>z ? x : z) : (y>z ? y : z);
}
```

Вызовы функций имеют вид:

```
int rez1 = max(20, 40);
float rez2 = max(1.5, 2e4, 2.4e-1);
```

Распознавание перегруженных функций при вызове выполняется по их сигнатурам (в зависимости от числа и типа аргументов) в процессе компиляции.

Шаблоны функций позволяют автоматизировать создание функций, обрабатывающих по одному алгоритму разные типы данных.

Для этих целей можно создать несколько перегруженных функций, работающих с различными типами данных, но при этом код программы увеличивается и содержит одинаковые по логике функции.

Шаблон функции определяет набор операторов, с помощью которых программа может создать несколько функций.

При обработке вызова функции на этапе компиляции ей передается тип аргументов, при этом генерируется соответствующий код функции. Таким образом, шаблон – средство параметризации функций. Параметризировать можно тип возвращаемого значения, типы аргументов, при этом количество и порядок аргументов фиксированы.

В определении шаблона семейства функций используется служебное слово *template*, в угловых скобках < > параметризируется список аргументов шаблона, перед каждым аргументом шаблона стоит служебное слово *class*.

Пример шаблона (*Type* – общий тип шаблона) функций для обмена значений двух передаваемых аргументов:

```
template <class Type>
void swap (Type *x, Type *y)
{
        Type z;
   z=*x; *x=*y; *y=z;
}
```

Пусть имеем вызовы функций:

```
int a, b;
float  x, y;
swap (&a, &b);
swap (&x, &y);
```

На этапе компиляции будут сгенерированы следующие определения функций:

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

Таким образом, шаблон функций автоматизирует подготовку определений перегруженных функций. Процесс создания компилятором определения функции называется инстанцированием шаблона.

Т.к. тип аргументов в шаблоне функции один и тот же, при вызове функций следует приводить аргументы к одному типу:

```
int a;
float  b;
swap (&a, &b);                  //ошибка, разные типы аргументов
swap (&(float)a, &b);           //явное приведение типа
```

Определяемая с помощью шаблона функция может иметь непараметризированные параметры, возвращаемое функцией значение также может быть непараметризировано.

Определим шаблон семейства функций, подсчитывающих количество положительных элементов в параметризированном массиве:

```
template <class type>
int count (int, type *);                    //прототип шаблона

int main()
{
   int mas1[]={2,-3,4,-20,0,1};
   int n=sizeof(mas1)/sizeof(mas1[0]);

      //число положительных элементов целочисленного массива
   cout<<"Number of positive elements
                  of an integer array"<<count(n,mas1);
   float mas2[]={1.2,4.5,-2,3.4};
   int n=sizeof(mas2)/sizeof (mas2[0]);

      //число положительных элементов вещественного массива
      cout<<"Number of positive elements
                  of a real array"<<count(n, mas2);
   return 0;
}

template <class type>                       // шаблон функции
int count (int size, type *arr)
{
      int k;
   for(int i=0; i<size; i++)
     if(arr[i]>0) k++;
   return k;
}
```

11. Поименованные области

В C++ можно явным образом задать область определения имен как часть глобальной области с помощью оператора *namespace*.

Объявление поименованной области или пространства имен имеет вид:

namespase [имя области]
{объявления}

и определяет пространство имен функций, классов и переменных, находящихся в отдельной области видимости.

Поименованная область ограничивает доступ к объявленным в ней объектам, препятствуя соответственно конфликту имен объектов, объявленных в разных местах программы.

Объявим две поименованные области *one* и *two*:

```
namespase one
{
        int  i, j;
   float  x, y;
    int   fun1(int, int);
    void  fun2(float *);
}

namespase two
{
        int  x, y;
   float  fun1(float *);
    void  fun2(float *);
}
```

В области *one* видимы переменные *x, y* типа *float*, в области *two* используются переменные *x, y* типа *int*.

Поименованная область может расширяться с помощью неоднократного объявления. Пример расширения поименованной области *one*:

```
namespase one
{
  int k, n, m;                  //объявление новых переменных
  float  fun2(float, float);  //перегрузка функции
```

```
  void   fun3(int *, int);  //объявление новой функции
}
```

В поименованной области могут содержаться также определения объектов, но обычно определения выполняются в нужном месте программы с использованием унарной операции доступа к области видимости ::, которая ставится перед именем объекта, например:

```
float one::fun3 (int *, int)      //определение функции
        {
          .   .   .
        }

one::i = 1;          //определение переменной
one::fun2(1.5, 10);  //вызов функции
```

Можно объявить объект доступным вне своего пространства с использованием оператора *using*, после чего его можно использовать без явного указания имени области:

```
using one::i;
i = 10;
```

Унарная операция указания области видимости используется также в том случае, если необходимо получить доступ к глобальным переменным из функции, которая содержит одноименную локальную переменную:

```
char str[40];
int func()
{
 char str;
  . . .
  str = 'A';
  strcpy(::str,"Hello world!");
}
```

Здесь *str* – обращение к локальной переменной, ::*str*- ссылка на глобальную переменную.

12. Аргументы командной строки

Командная строка - это программный продукт, который обеспечивает прямую связь между пользователем и операционной системой. Запустить данное приложение можно следующим способом:

Пуск -> Все программы -> Стандартные -> Командная строка.

Win+R и в появившемся окне ввести cmd.

Интерфейс командной строки определяет количество введенных с клавиатуры текстовых строк в командной строке, которые в качестве аргументов передаются функции *main*. Командная строка задает параметры компиляции и запускает эту компиляцию.

Запустить программу (функцию *main*) на выполнение можно с помощью командной строки.

Передача данных из командной строки операционной системы программе осуществляется с помощью массива указателей.

Механизм передачи аргументов организован следующим образом.

Пусть имеем программу *copy.cpp*, которая копирует содержимое одного файла в другой. Откомпилируем ее, получив в результате исполняемый файл программы *copy.exe***.** Запустим его на выполнение из командной строки. Получив приглашение (**>c**:), введем имя исполняемого файла (прописав путь) и значения аргументов (имя исходного файла *f1.dat* и имя нового *f2.dat* , в который копируем информацию):

>c:\cpp\copy.exe f1.dat f2.dat

Можно опустить расширение *.exe* исполняемого файла, т.к. запускается на выполнение загрузочный файл:

>c:\cpp\copy f1.dat f2.dat

Следует указать полный путь к файлам *f1.dat* и *f2.dat*, если они не содержатся в текущем каталоге.

Компилятор языка обрабатывает командную строку, разбивая ее на отдельные слова, используя в качестве разделителей слов пробелы. В функцию *main* передается счетчик слов командной строки ***argc*** (*argument count*) и массив указателей на каждое слово ***argv*** (*argument vector)*.

Заголовок функции *main* с аргументами имеет вид:

```
int main (int argc,  char *argv[])
{
        // тело функции
}
```

Пример программы копирования файлов, запускаемой на выполнение приведенной командной строкой:

```
int main(int argc, char *argv)
{
        FILE *f1, *f2;
   char str[80];
   if (argc!=3) //проверка кол-ва слов в командной строке
    {
              printf("Bad command!
              Format command: copy name1 name2\n");
     exit();
    }
   printf(" copy %s  в  %s \n", argv[1], argv[2]);
   f1=fopen(argv[1], "r");
   f2=fopen(argv[2], "w");
   while(fgets(str, sizeof(str), f1))
      fputs(str,f2);
   fclose(f1);
   fclose(f2);
   return 0;
}
```

Работа с файлами и потоками подробно рассмотрена в главе 16, здесь же акцент сделан на работу с аргументами командной строки.

В объявлении аргумента *char *argv[]* функции *main* скобки пусты, т.к. функции *main* не известен размер передаваемого ей массива слов. Напомним, что компилятору при обработке объявления не требуется информация о том, сколько элементов содержится в одномерном массиве.

Содержимое массива *argv* и ассоциированных с ним строк представлено в табл. 5:

Таблица 5

Адрес	**Содержимое**
10000 – 10004	*'c' 'o' 'p' 'y' '\0'*
10005 – 10011	*'f' '1' '.' 'd' 'a' 't' '\0'*
10012 – 10018	*'f' '2' '.' 'd' 'a' 't' '\0'*

Элемент массива *argv[0]* содержит указатель на имя программы *"copy"*, элемент *argv[1]* содержит адрес слова *"f1.dat"*, *argv[2]* - слова *"f2.dat"*.

Приведем еще один откомпилированные код, где файл *letter.exe* запускается на выполнение с помощью командной строки:

>c:\cpp\letter Program begin to work

//файл letter.cpp

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int i;
   for (i=1; i<argc; i++)
      printf("%s ", argv[i]);
   return 0;
}
```

Функция *main* получает значения аргументов, где *argc==5,*а массив *argv* получает значения адресов слов командной строки. Результат работы – вывод на экран введенного в командной строке текста.

Передача аргументов посредством командной строки является удобным способом передачи значений служебным программам.

Есть еще один формат функции *main* с аргументами:

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

где *envp[]* - массив указателей строковых величин, определяющих операционную среду *windows*.

Ниже приведена программа *program.cpp* создания динамического числового массива со строками разной длины («рваный» массив), при этом число строк массива (5) вводится в командной строке:

c:\program.exe 5

После ввода данных в массив реализован его вывод, причем числа строк массива выводятся в обратном порядке:

```
#include <stdio.h>
#include <stdlib.h> // alloc.h – устарел, но работает
int main(int argc, char *argv[])
{
 double ** line;  //line - указатель для массива указателей
 int i, j, n;
 int * m;        //Указатель для блока памяти длин строк

 if (argc!=2)
  {
     puts("Bad command");
     exit();
  }
 n = atoi(argv[1]);       //память под массив указателей
 line = (double**)calloc(n, sizeof(double*)) ;
 m=(int *)malloc(sizeof(int)*n);
 for (i=0; i<n; i++)              //цикл по количеству строк
  {    //Ввод количества элементов i-го одномерного массива
        printf("Enter the number of array elements
                              m[%d]=", i);
    scanf("%d",  &m[i]);
    line[i]=(double*) calloc(m[i], sizeof(double));
              //Ввод  элементов массива
    printf("Enter the array elements: ");
    for(j=0; j<m[i]; j++)
       scanf("%le", &line[i][j]);
 }

              //Результаты обработки
 printf("\nResults:")
 for (i=n-l; i>=0; i--)
  {
              for(j=0;j<m[i];j++)
```

```
        printf("\t%f", line[i][j]);
      free (line [i]);  //Освободить память строки массива
    }
  free(line);    //Освободить память массива указателей
  free(m);       //Освободить память массива int
  return 0;
}
```

Приведенным результатам выполнения программы соответствует рис. 6, где условно изображены все массивы, динамически формируемые в процессе выполнения программы.

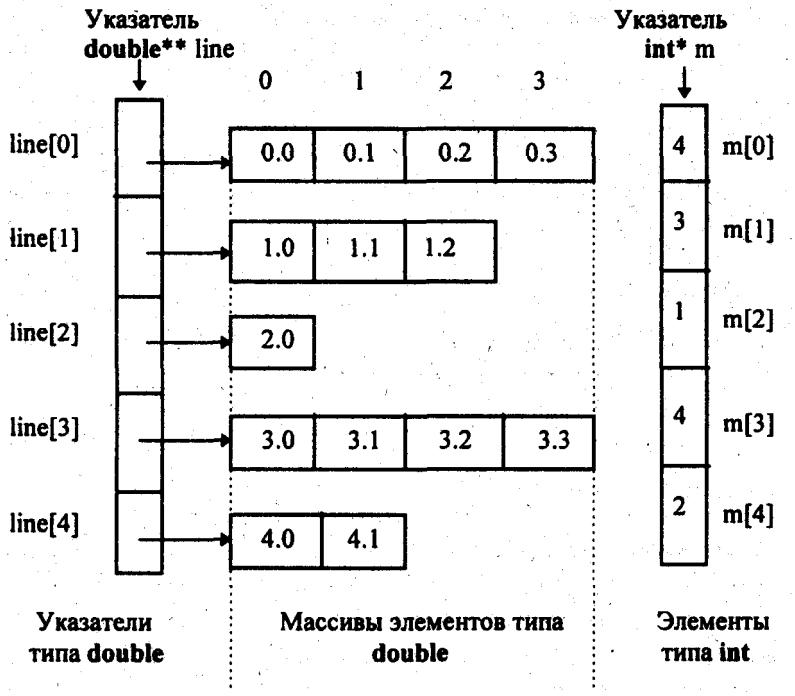


Рис. 6. Схема массива со строками разной длины для случая: число строк n = 5, количество элементов в строках: 4,3,1,4,2

После окончания работы с данными все блоки динамически выделенной памяти рекомендуется явным образом освободить. Для этих целей в приведенном коде используется несколько вызовов функции *free.* После вывода очередной строки массива (цикл по *i*) функция *free (line[i])* освобождает участок памяти, адресуемой указателем *line[i]*. После окончания цикла освобождаются блоки, выделенные для массива указателей (вызов функции *free(line);*) и массива целых (вызов функции *free(m);*).

Обратите внимание! Из-за ошибок работы с динамической памятью может возникнуть ситуация, когда программа при выполнении пытается получить доступ к памяти, которая не была выделена или уже была освобождена. После освобождения памяти значение адреса, присвоенное указателю, не следует использовать, так как эта область может быть выделена под другую переменную. Рекомендуется в такой указатель записать значение пустого указателя NULL.

13. Структуры

Из основных типов пользователь может конструировать производные типы, такие, как структуры и объединения. Структуры (типы, определенные через ключевое слово *struct*) и объединения (определенные через ключевое слово *union*) рассматриваются в C++ как разновидности классов, т.к. в них допустимо использование функций, аналогично их использованию в определении классов.

13.1. Объявление структур

В отличие от массивов, т.е. последовательности элементов, имеющих одинаковый тип, структуры – это множество поименованных элементов в общем случае разных типов. Структуры объединяют в единое целое набор некоторой связанной информации и объявляются с ключевым словом struct.

Тип *struct* в C++, как вид класса, обладает всеми свойствами классов, однако все члены класса *struct* по умолчанию открыты (*public*). Для ограничения доступа к элементам структуры их можно переопределить как закрытые, используя модификатор доступа *private*.

Объявление типа *struct* имеет вид:

```
struct имя типа
{
описание элементов
};
```

Элементы структуры называются полями, могут иметь любой тип, в том числе быть указателями на тип самой структуры.

Объявление структуры сообщает компилятору о том, какие объекты войдут в структуру:

```
struct book
 {  char title[81];
        int year;
        int page;
        float price;
 };
```

Такое объявление часто называют структурным шаблоном.

Имена полей в структуре должны различаться. Имена элементов разных структур могут совпадать. Элементом структуры может быть другая структура.

При описании структуры после закрывающейся фигурной скобки обязательно ставится (;). Область действия шаблона структуры зависит от того, в каком месте он размещен: если он установлен вне функции, то он доступен всем функциям программы, следующим за его определением, если в теле функции – то он может использоваться только внутри функции.

В описании структуры может отсутствовать имя типа структуры, но при этом необходимо сразу указать переменные типа описанной структуры:

```
struct
{
        int a;
}  object,&pob=object;
```

Описание структурного шаблона не создает структурной переменной. Объявление переменной *libry*

struct book libry;

создает объект типа *struct book*, под который выделяется *sizeof(struct book)* байт памяти в соответствии со структурным шаблоном.

Можно объединить описание структурного шаблона и фактическое определение структурной переменной:

```
struct book
 {   char title[81];
        int year;
        int page;
        float price;
 } libry;
```

Используя тип *struct book*, можно описать несколько объектов, в том числе массив структур и указатель на структуру:

*struct book libry, catalog[10], *plibry;*

Элементы структуры в памяти запоминаются последовательно, в том порядке, в котором они объявляются: первому элементу соответствует меньший адрес памяти, последнему - больший.

Структурную переменную можно инициализировать в операторе объявления подобно массиву:

```
struct book libry = {     "C++",
                          "Straustrup",
                          1990,
                           500
                                    };
```

Доступ к элементам структуры осуществляется с помощью операции точка. В следующих операторах поле *libry.author* получает значение в результате копирования или в результате ввода значения с клавиатуры:

```
strcpy(libry.author,"Bjorn Straustrup");
gets (libry.author);
```

13.2. Массивы структур

Описание массива структур аналогично описанию любого другого массива:

```
struct book catalog[10];
```

Каждый элемент массива *catalog* представляет собой структуру типа book.

Для доступа к элементу массива используется индекс, через операцию точка указывается имя поля i-й структуры:

```
catalog[2].title
catalog[4].price
catalog[2].title[5]       //6 элемент символьного массива
                          //в 3-й структуре массива
```

Элемент структуры может быть объявлен как указатель на тип структуры, в которую он входит. Это позволяет создавать связанные списки структур:

```
struct sample
{
        char h;
        float *pf;
        struct sample *next;
} x;
```

Присвоим переменной *x.next* адрес динамически выделенной области памяти для хранения значения структуры того же типа:

```
x.next = (struct sample*)malloc(sizeof(x));
```

или

```
x.next = new struct sample;
```

Функция *malloc* выделяет блок памяти размером *sizeof(x)* байт и возвращает указатель на него. Операция *new* выделяет блок памяти под объект типа *struct sample*, его адрес сохраняется в *x.next*.

При описании переменных типа структура после определения шаблона ключевое слово *struct* можно опускать. Исключение составляет случай, когда имя типа структуры совпадает с именем переменной:

```
struct str
{
        char x[80];
        int   n;
};
...

{  char   *str;
   struct  str  s;
}
```

Во внутреннем блоке локальная переменная *str* перекрыла имя типа структуры *str*, объявленной на глобальном уровне. В этом случае ключевое слово *struct* указывает компилятору, что *str* – это имя типа.

Выделим память под массив из 10-ти динамических структур:

```
struct st
{
        char name[20];
        int n;
```

```
};

struct st *pst = new st[10];
```

Доступ к полям структуры через указатель в С++ производится с помощью операции косвенной адресации –>:

```
for(i=0; i<10; i++)
{
        gets(pst[i]–>name);
        pst[i]–>n=strlen (pst[i]–>name);
}
```

13.3. Вложенные структуры и указатели на структуры

Элементом структуры может быть другая структура.

```
struct myfile              //1-й шаблон – имя, тип, версия файла
{  char  name[10];
   char  typ[4];
   int  ver;
};

struct dat                 //2-й шаблон - дата
{  int  day;
   char  month[4];
   int  year;
};

struct dir                 //3-й шаблон - директорий
{  struct  myfile  file;
   int size;
   struct dat  f_dat;
};

struct dir my_f[100] ,*pst;    //описание переменных
```

Шаблон для вложенной структуры должен располагаться перед определением фактической структурной переменной в рамках другой структуры.

Для получения элемента вложенной структуры необходимо дважды применить операцию точка:

my_f[0].size; ***//элемент size 1-ой структуры***
(my_f[2].file).ver; ***//элемент ver вложенной структуры file***
//в 3-й структуре my_f

Круглые скобки можно опустить, поскольку операция точка (**.**) – выделение элемента структуры выполняется слева – направо.

Указатель **pst* можно использовать для ссылок на любые структуры типа *dir*. Чтобы связать указатель со структурой, можно использовать операцию получения адреса, например:

pst = &my_f[0];

Указатель содержит адрес структуры, которая занимает *sizeof(my_f[0])* байт в оперативной памяти (26, если целый тип занимает 2 байта), поэтому добавление единицы к *pst* прибавляет 26 к значению адреса, т.е. если *pst* указывает на *my_f[0]*, то *pst+1* будет указывать на *my_f[1]*.

После определения значения указателя на структуру доступ к элементам структуры через указатель выполняется с помощью операции косвенного доступа (–>): *pst–>size*, что эквивалентно записи *(*pst).size*

Если *pst == &my_f[0],* то выполняются следующие тождественные равенства:

pst -> size == my_f[0].size
(pst+i)->size==my_f[i].size

С другой стороны, если *pst == &my_f[0],* то

**pst == my_f[0],*
*(*pst).size == my_f[0].size*

Таким образом, к полю *size* можно обратиться следующими способами:

*pst->size == my_f[0].size == (*pst).size*

Для вложенных структур:

my_f[0].fil.fver == pst->(fil.fver)
my_f[0].fil.fname[0] == pst->(fil.fname[0])

Напомним, что самый высокий приоритет имеют операции идентификации функции и массива (скобки – () и []) и доступа к элементу структуры – точка и косвенная адресация(. и –>), которые выполняются слева направо.

13.4. Структуры и функции

Для передачи информации о структуре в функцию используются следующие способы:

- Использование в качестве фактического аргумента указателя на структуру, т.е. передача по адресу, когда функция может изменить значение структуры. Соответствующий формальный аргумент объявляется как указатель на структуру с тем же шаблоном. При этом структурный шаблон должен быть объявлен перед вызывающей функцией.
- Использование в качестве фактического аргумента самой структуры -передача по значению. При этом функция работает с копией значения структуры.
- Использование в качестве фактического аргумента элемента структуры - значения некоторого поля структуры, которое может быть передано как по адресу, так и по значению. При этом соответствующие формальные аргументы вызываемой функции объявляются как переменные, тип которых соответствует типу передаваемых элементов структуры.

14. Объединения, битовые поля и перечислимые типы

14.1 Объединения (union)

Объявление объединения синтаксически совпадает с объявлением структуры, но начинается с ключевого слова *union*. В отличие от структур, где элементы в памяти размещаются последовательно в соответствии описанию, в объединениях под все элементы отводится одно место в памяти, и все элементы имеют один и тот же начальный адрес. Размер объединения определяется максимальным размером его элементов (полей).

Объединения обеспечивают возможность доступа к одному и тому же участку памяти с помощью переменных разных типов. Объединения экономят память и используются, если в каждый момент времени в программе используется только одно поле.

Определим переменную *val* типа объединение:

```
union small
{       int x;
        char s[10];
        float y;
} val, *p;
```

При работе с объединениями используются теже операции, которые применимы к структурам (присваивание объединения и копирование его как единого целого, взятие адреса объединения и доступ к отдельным его элементам).

Под объект *val* память выделяется в соответствии с размером максимального поля *sizeof(val.s)*==10 байт.

В каждый момент времени в переменной типа *union small* может храниться только одно значение: *val.x*, или *val.s*, или *val.y*.

Доступ к элементам объединения, как и к элементам структуры, возможен через указатель *(p->x, p->s[i], p->y*), если перед этим *p=&val;*.

Фактически объединение - это структура, все элементы которой имеют нулевое смещение относительно ее базового адреса и размер которой позволяет поместиться в ней самому большому ее элементу.

14.2. Битовые поля

До сих пор мы имели дело с наименьшей единицей памяти для хранения данных – одним байтом. С помощью одного байта памяти

можно закодировать 256 значений. Если диапазон изменения данных гораздо меньше, то использовать для них байт памяти не рационально.

В языке С++ есть возможность работать с битами памяти. Для этого их надо объявить элементами структуры. Элементом структуры может быть один бит или их последовательность, не превышающая машинного слова. Битовые поля в структуре описываются как целые (*unsigned* или *unsigned char),* с добавлением после имени поля символа *':'* и его длины.

Представим некоторый контрольный байт *cntrl_byte* структурой битовых полей:

```
struct bit_area
{  unsigned char er:1; //бит ошибки
   unsigned char rd:1; //бит готовности
   unsigned char dat:6;//поле данных
} cntrl_byte;
```

Как мы знаем, максимальное целое число, размещаемое в поле длиной *n* бит, равно 2^{n-1}. Память под битовые поля выделяется в байтах справа налево.

Пусть программа работает с датами: число меняется в интервале 1÷31, месяц соответственно *1÷12*, день недели *1÷7*. Можно объявить их как обычные числа, но при большом количестве дат это потребует большого объема памяти. Рациональнее разместить эти числа в структуре, выделив каждому из них требуемое количество бит:

```
struct MyDate
{
   unsigned day       :6;
   unsigned month     :5;
   unsigned day_of_week:4;
};
```

Таким образом, вся информация о дате разместилась в *2*-х байтах, т.е. в одном слове памяти. Крайний левый бит при этом остается свободным.

Битовые поля применяются для экономного хранения данных, т.к. позволяют упаковать значения не по границам байтов, а также для работы с данными, в которых отдельные биты имеют самостоятельное функциональное значение (например, для организации удобного доступа к регистрам внешних устройств – доступе к элементам файловых таблиц FAT).

У битового поля может отсутствовать идентификатор. Неименованное битовое поле означает пропуск указанного числа бит перед размещением следующего элемента структуры. Неименованное поле, для которого указан нулевой размер, гарантирует размещение следующего элемента структуры на границе *int* (машинного слова).

14.3. Перечислимые типы данных

Перечислимый тип (перечисление) – определение целочисленных констант, для каждой из которых вводятся имя и значение. Общий вид объявления перечислимых переменных:

enum имя типа {список перечисления} идентификатор;

где *enum* – служебное слово, означающее перечисление; *список перечисления* – разделенная запятыми последовательность идентификаторов или именованных констант следующего вида: имя константы = значение константы. Значение константы может отсутствовать.

Объявим тип *enum progr*:

enum progr{C,Pascal,Foxpro,Basic,Fortran} lang;

Значением переменной *lang* может быть одна из констант - *C, Pascal* и т.д. Можно записать следующие операторы:

```
lang=C;
if (lang==C)  printf("I know C++\n"); //Я знаю C++
```

Элементы списка перечисления имеют тип *int*, т.е. перечислимый тип – это подмножество целого типа. Над перечислимым типом определены те же операции, что и над целым типом. Константы могут быть явно проинициализированы. Иначе по умолчанию первому элементу списка присваивается значениие *0,* второму – *1* и т.д.

```
enum {elem1,elem2,elem3} m;        //внешнее объявление
printf("%d  %d  %d  %d\n", elem1, elem2, elem3, m);
                  //результат   0      1       2     0
```

Пример явной инициализации констант (не проинициализированный элемент *elem2* по умолчанию получает значение 0):

enum mas{elem1=-1, elem2, elem3=5};

Элементам перечисления могут быть присвоены одинаковые значения:

enum mas{elem1=2,elem2,elem3=2,elem4};

Если переменная перечислимого типа объявлена как внешний объект, то она инициализируется нулем, иначе ее значение не определено.

Как покажем в п. 17.2, для определения констант можно написать ряд директив *#define*, но перечислимый тип *enum* позволяет это сделать быстрее и изящнее.

Приведем программу изменения координат *x*, *y* на единицу нажатием соответственно клавиш *0* (влево), *1* (вправо), *2* (вверх), *3* (вниз):

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define  PRIN  printf("x,y :%d,%d",x,y);break;

int main()
{
  enum direct {left, right, up, down}dir;
  int x=0,y=0,step=1;
  printf("x,y :%d,%d",x,y);
  do
  {  printf("  dir= ");
     scanf("%d",&dir);

        switch (dir)
     { case left: x-=step; PRIN
       case right: x+=step; PRIN
       case up: y+=step; PRIN
       case down: y-=step; PRIN
       default: puts("incorrectly direction");
      }
  } while ((dir==0)||(dir==1)||(dir==2)||(dir==3));
  getch();
  return 0;
}
```

Экран

```
x,y :0,0  dir= 1
x,y :1,0  dir= 1
x,y :2,0  dir= 1
x,y :3,0  dir= 2
x,y :3,1  dir= 2
x,y :3,2  dir= 3
incorrectly direction
```

15. Объявление имени типа и абстрактный описатель типа

15.1. Объявление имени типа typedef

Помимо явного объявления типа в C++ предусмотрены дополнительные средства описания имён типов. Оператор *typedef* используется для объявления новых имен типов данных программы, которые затем используются для обозначения производных и основных типов.

В табл. 6 в первом столбце содержится объявление имени типа, которое задается заглавными буквами, во втором столбце – объявление переменных с использованием нового имени типа, в третьем – эквивалентное явное объявление типа без создания его нового имени.

Таблица 6

Объявление типа	Применение	Значение
*typedef char *STR;*	*STR s1, s2;*	*char *s1, *s2;*
*typedef int *FUN();*	*FUN f1, f2;*	*int *f1(), *f2();*
typedef struct *{ int n;* *char str[20];* *} ST;*	*ST rec1, rec2;*	*struct* *{ int n;* *char str[20];* *} rec1, rec2;*
*typedef int(*PFUN)();*	*PFUN p;*	*int (*p)();*

Таким образом, *typedef* определенному типу данных назначает новое символьное имя.

Кроме операторов описания, объявленные имена типов могут использоваться в списке формальных параметров в объявлении функций, в операциях приведения типов и в операции *sizeof*.

С помощью *typedef* можно улучшить читаемость программы, например:

```
typedef struct
{
        char fam[20];
    int ball[4];
} STUD;

STUD spisok[10];
```

Обратите внимание! Объявление типа *typedef* не создает новые типы, а создает удобные имена типов.

15.2. Абстрактный описатель типа

Имена типов мы используем при объявлении программных объектов (идентификаторов констант, переменных, функций), а так же в ряде следующих случаев, когда идентификатор отсутствует:

- в списке формальных параметров в объявлении функций,
- в операциях приведения типов,
- в операции *sizeof* (определение размера типа).

Абстрактный описатель – это описатель без идентификатора, состоящий из одного или нескольких модификаторов типа – указателя, массива, функции и круглых скобок, выделяющих приоритет модификаторов типа.

Чтобы правильно интерпретировать абстрактный описатель типа, нужно определить в нём место подразумеваемого идентификатора. Для этого следует знать правило:

- модификатор указателя (*) всегда задается перед идентификатором в описателе,

- модификаторы массива [] и функции () – после него.

Приведем примеры абстрактных описаний типов:

```
int             //целое
int *           //указатель на целое
int (*)[10]     //указатель на массив из 10 элементов
int *()             //функция, возвращающая указатель на целое
int (*)(void)   //указатель на функцию без аргументов,
                // возвращающую значение типа int
int f(int**,char); // функция, получающая в качестве
                              //аргументов указатель на адрес целого
                              //и символ, возвращающая целое значение
```

В прототипе функций следует использовать абстрактный описатель типа (имя переменной не несет никакой смысловой нагрузки):

```
//поиск максимумов в одномерном и двухмерном массивах

int max_array_1(int [], int);
int max_array_2(int [][100], int, int);
```

16. Работа с файлами и потоками

В языке С++ нет встроенных средств ввода/вывода данных. Эти операции выполняются с помощью:

- функций библиотеки *stdio.h* языка Си,а именно *printf*, *scanf, gets, puts, getchar, getche, getc, putchar*, организующих ввод-вывод для терминала;
- функций библиотеки *iostream* языка *C++*, которая использует концепции ООП. Стандартный входной поток *stdin* и стандартный выходной поток *stdout* языка Си в С++ связываются с объектами типа *istream* и *ostream* с именами соответственно *cin* и *cout*. Для выполнения ввода/вывода перегружены две операции - операция потока для ввода >> и операция потока для вывода <<.

Функции и операторы ввода/вывода позволяют читать данные из файлов и с устройств и писать данные в файл и на устройства. При этом передача данных осуществляется побайтно или группами байтов. Данные рассматриваются как неструктурированный набор данных – поток (напомним, что в Паскале есть файлы разных типов - структурированные файлы).

При открытии потока с ним связывается определенная область оперативной памяти – буфер, и обмен информацией между программой и файлом осуществляется через буфер, что позволяет увеличить скорость передачи данных. Ввод/вывод данных осуществляется блоками, размер которых равен размеру буфера. В файле *stdio.h* размер буфера установлен константой *BUFSIZ:*

#define BUFSIZ 512 или
#define BUFSIZ 4096

Можно управлять размером и наличием буфера с помощью функций

*void setbuf(FILE *f, char *p);*
*void setvbuf(FILE *f, char *p, int mode, size_t size);*

где *f* – указатель на структуру *FILE*, *p* - указатель на устанавливаемый буфер, *size* – размер буфера, *mode* – параметр (целое), устанавливающий режим, принимающий одно из значений - *_IOFBF* (полная буферизация), *_IONBF* (запрет буферизации), *_IOLBF* (для вывода – построчная буферизация, т.е. опустошение буфера при записи в буфер символа новой строки).

Если производится операция вывода в файл, то данные помещаются в буфер и информация из буфера переписывается в файл при полном заполнении буфера или при закрытии потока.

Если информация считывается из файла, то сначала информация копируется из файла блоками в буфер (при открытии файла и далее при опустошении буфера), а затем из буфера вводится в переменные программы.

Буферизация ввода/вывода осуществляется автоматически и позволяет ускорить выполнение программы за счёт уменьшения количества обращений к медленно работающим внешним устройствам. Существует возможность не буферизированного ввода/вывода данных.

Общая схема файлового ввода/вывода при использовании стандартных функций, объявленных в библиотеке *stdio.h*:

- Включение с помощью директивы *include* файла заголовков *stdio.h* и объявление указателя на структуру (*FILE *f);*.
- Открытие файла вызовом функции *fopen*, присваивание возвращаемого ею значения указателю на структуру *FILE*. Проверка правильности открытия файла, т.е. указатель не должен равняться *NULL*.
- Чтение и запись в файл с помощью функций *fprintf, fscanf, fputs, fgets, fgetc, fputc, getc, putc, freed, fwrite* и других, объявленных в библиотеке <*stdio.h*>.
- Закрытие файла вызовом функции *fclose*.

16.1. Открытие, закрытие, перенаправление потока

При работе с файлами при открытии потока для ввода/вывода физический файл связывается со структурой типа *FILE*. Имя типа *FILE* определяется с помощью конструкции *typedef* в файле *stdio.h*.

Структура *FILE* содержит разнообразную информацию о файле – указатель текущей позиции в потоке, указатель на буфер обмена, размер файла, текущий счётчик байтов и т.д.

При открытии потока (файла) возвращается указатель на структуру *FILE*, который используется для последующих ссылок на поток.

Автоматически при работе программы открываются пять предопределенных потоков - *stdin*, *stdout*, *stderr*, *stdoux*, *stdprn* (стандартный ввод, стандартный вывод, стандартный вывод сообщений об ошибках, стандартный дополнительный поток - порт, стандартное устройство печати).

Остановимся на основных функциях работы с файлами.

1. Открытие потока

Для соединения потока с файлом используется функция

FILE fopen(const char* name,const char* mode);*

где *name* – имя открываемого файла (путь к файлу), *mode* – режим открытия файла. При успешном открытии потока функция возвращает указатель на поток (полученный поток связывается с файлом), в противном случае *NULL*.

Режимы открытия файла функцией *fopen*:

”r” – чтение в текстовом режиме;
”w” – запись в текстовом режиме;
”a” –добавление в текстовом режиме;
”rb” – чтение в бинарном режиме;
”wb” – запись в бинарном режиме;
”ab” –добавление в бинарном режиме;
”r+” или *”w+”*, или *”a+”* – чтение и запись в текстовом режиме;
”r+b” или *”w+b”,* или *”a+b”* – чтение и запись в бинарном режиме;
”rb+” или *”wb+”,* или *”ab+”* – чтение и запись в бинарном режиме.

При открытии файла в режимах *”r”, ”rb”, ”r+”, ”r+b”* индикатор позиции устанавливается на начало файла. В случае открытия несуществующего файла функция *fopen* возвращает значение *NULL*.

При открытии файла в режимах *”w”, ”wb”, ”w+”, ”w+b”* создается новый файл. *Обратите внимание!* Если используется ключ *”w”* для существующего файла, то файл стирается и создаётся новый файл с тем же именем (индикатор позиции устанавливается на начало файла).

При открытии файла в режимах ”a”, ”ab”, ”a+”, ”a+b” создается новый файл. *Обратите внимание!* Если файл с заданным именем существует, то он открывается, и индикатор позиции устанавливается на конец файла (дозапись информации в файл).

Следует учитывать, что если текстовый файл открывается в режиме чтения и записи, то базовая операционная система может открыть его в бинарном режиме. Максимальное количество файлов, которые можно открыть одновременно, задается переменной *FOPEN_MAX*. Максимальная длина имени файла задается переменной *FILENAME_MAX*.

Как в текстовом, так и в бинарном режиме можно использовать все функции для доступа к файлу. При работе в бинарном режиме поток записывает на диск и считывает с диска точные копии данных, переданные функциями записи данных на диск и требуемые функциями чтения данных с диска соответственно. Разница между текстовым и бинарным режимами работы потока включает три момента:

- символ <*CTRL+Z*> интерпретируется как конец файла;

• символ конца строки обрабатывается следующим образом: при записи в текстовый поток из комбинации символов *'\r'* (возврат каретки, код *0x0D*) и *'\n'* (новая строка, код *0x0A*) в файл записывается только символ *'\n'*, а при чтении из текстового потока символ *'\n'* преобразуется в комбинацию символов *''\r\n''*;

• так как при записи в текстовый поток может происходить преобразование количества и представления символов, то для получения требуемой позиции в файле нужно использовать функции *fgetpos* и *ftell*.

2. Закрытие потока ввода-вывода

Для отсоединения потока от файла используется функция

int fclose(FILE f);*

которая закрывает файл, освобождая при этом все буферы потока. При успешном завершении функция возвращает 0, а в случае неудачи – EOF (-1).

3. Перенаправление потока

Для перенаправления потока используется функция

FILE freopen(const char* name, const char* mode,*
FILE f);*

которая закрывает файл, соединенный с потоком *f*, и соединяет с этим потоком файл *name* в режиме *mode.* В случае успеха функция возвращает указатель на поток, а в случае неудачи – *NULL*. Параметр *mode* принимает те же значения, что и в функции *fopen*.

Приведем примеры проверки выполнения операций с файлами.

• Проверка успешного открытия файла на чтение:

```
#include<stdio.h>
#include<stdlib.h>  //содержит прототип функции exit()
int main()
{FILE* myfile;
 char file_name[40];
 gets(file_name);
 if((myfile=fopen(file_name,"r"))==NULL)
 {      printf("Error:File %s not found\n",file_name);
        exit(1); //exit(0) -нормальное завершение процесса
 }      //exit(1) - сигнал об ошибке
```

```
//работа с файлом с использованием указателя myfile
. . .
 fclose(myfile);
 return 0;
}
```

- Проверка удачного закрытия потока

```
if(fclose(myfile)==-1)
{   printf("File not Close");
    exit(1);
}
```

Функция *fclose* закрывает один заданный поток, функция *fcloseall* закрывает все потоки, кроме стандартных.

16.2. Работа с индикаторами ошибки, позиции и конца файла

1. С каждым потоком связан индикатор ошибки, который находится в установленном положении, если в потоке, связанном с файлом, произошла ошибка. В противном случае индикатор ошибки находится в сброшенном состоянии. Для работы с индикатором ошибки используются функции *ferror* и *clearerr*.

Функция

```
int ferror(FILE* f);
```

возвращает ненулевое значение, если индикатор ошибки установлен, в противном случае функция возвращает 0.

После возникновения ошибки работа с файлами возможна только после сброса ошибки с помощью функций *clearerr* и *rewind*.

Функция

```
void clearerr(FILE* f);
```

сбрасывает индикаторы ошибки и конца файла для потока *f*.

Продемонстрируем работу этих функций:

```
FILE *f;
f=fopen(name, "r");
int c = getc(f);          //определяет ошибку при попытке чтения
if (ferror(f))
    printf("Error reading from  %s \n", name);
clearer(f);               //очищает признак ошибки
                          //и признак конца файла для потока f
```

2. Для каждого файла после его открытия определяется индикатор позиции, который указывает на смещение от начала файла в байтах.

Функция

void rewind(FILE f);*

устанавливает индикатор позиции на начало файла, связанного с потоком *f*. При этом сбрасывается индикатор ошибки и конца файла.

Функция

int fseek(FILE f, long offset, int mode);*

сдвигает индикатор позиции файла на *offset* байтов. В случае успешного завершения функция возвращает 0, в противном случае – ненулевое значение. Параметр *mode* указывает на режим сдвига и может принимать следующие значения:

- *SEEK_SET* – смещение от начала файла;
- *SEEK_CUR* – смещение от текущей позиции;
- *SEEK_END* – смещение от конца файла.

Данные константы определены в файле *stdio.h* :

```
#define  SEEK_SET   0
#define  SEEK_CUR   1
#define  SEEK_END   2
```

Поэтому в функции *fseek* параметр *mode* может принимать значения *0, 1, 2.*

Например, работа следующих двух функций эквивалентна:

```
rewind(f);
fseek(f, 0L, SEEK_SET);
```

а функция с указанными ниже параметрами оставляет индикатор в текущей позиции:

```
fseek(f, ftell(f), 0);
```

Функция

int fsetpos(FILE f, const fops_t *pos);*

устанавливает индикатор позиции файла *f* в позицию, на которую указывает параметр *pos*, предварительно полученный с помощью функции *fgetpos*. Индикатор конца файла сбрасывается.

Функция

int fgetpos(FILE f, fops_t* pos);*

сохраняет текущее значение индикатора позиции файла *f* по адресу *pos*. Сохраняемое значение может быть полезно только для последующего обращения к функции *fsetpos.*

Функция

long ftell(FILE);*

в случае успешного завершения возвращает текущую позицию файла *f*, а в случае неудачи – возвращает значение 1L. Для бинарного потока позиция равна смещению в байтах от начала файла, а в случае текстового потока – значению, которое может использоваться функцией *fseek.*

3. Структура *FILE* содержит индикатор конца файла, который устанавливается в ненулевое значение функцией чтения из файла при достижении этой функцией конца файла. Состояние конца файла читается функцией

int feof(FILE f);*

которая возвращает ненулевое значение, если индикатор конца файла установлен, в противном случае функция возвращает 0.

16.3. Ввод/вывод данных

В C++ для работы с текстовыми файлами используется символьный ввод/вывод данных, для бинарных файлов ввод/вывод реализуется блоками.

16.3.1. Символьный ввод/вывод

1. Ввод/вывод символов

Для записи и чтения символов из текстового файла используются функции *fputc, putc, fgetc, getc.*

Функция

*int fgetc(FILE *f);*

читает символ из потока *f* в форме *int* и продвигает индикатор позиции на следующий символ. В случае успешного завершения функция возвращает прочитанный символ. В случае достижения конца файла (ошибки)

функция возвращает EOF и устанавливает индикатор конца файла (индикатор ошибки). Для проверки состояния индикатора используется функция *ferror*. Индикатор ошибок (*error indicator*) является элементом объекта *FILE*.

Вывод символов в файл реализует функция

*int fputc(int ch, FILE *f);*

которая помещает символ *ch* в поток (файл) *f*, и возвращает *EOF* в случае ошибки, иначе - записанный в файл символ *ch*.

Ввод-вывод может быть осуществлён также с помощью функций

*int getc(FILE *f);*
*int putc(char c, FILE *f);*

работа которых эквивалентна функциям *fgetc* и *fputc*, но для которых существуют одноимённые макроопределения (содержатся в *stdio.h*) :

#define putc(c,f) строка замещения
#define getc(f) строка замещения

В *stdio.h* содержатся также макроопределения функций ввода/вывода *getchar* и *putchar*:

#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)

и объявлены прототипы функций

int getchar(void);
int putchar(const int_c);

2. Ввод-вывод символьных строк

Для записи и чтения строк из символьного потока используются функции *fputs* и *fgets*.

Функция

*char *fgets(char *str , int n, FILE *f);*

читает строковое значение из потока *f* по адресу *str* до тех пор, пока не встретит символ перехода на новую строку *'\n'* или число введённых

символов не достигнет значения (*n-1*)- объявленного размера *str*. В конец строки добавляется нулевой символ *'\0'*. Чтение прекращается также в случае достижения конца файла. Функция возвращает *NULL* при обнаружении ошибки или достижении конца файла, в случае успешного ввода - указатель *str*.

Функция

*int fputs (char *str , FILE *f);*

записывает строку символов, хранимую по адресу *str,* в поток *f*, заменяя при этом символ конца строки *'\0'* на символ '\n'. Возвращает *EOF* в случае ошибки, иначе - последний записанный символ (неотрицательное число).

Приведем фрагмент кода записи текста, вводимого с клавиатуры, в файл, и его вывод на экран. Признак окончания ввода - ввод пустой строки:

```
FILE *fi;
char str[80];
if ((fi=fopen("fil.dat","w"))==NULL)
   exit(1);   //прерывает  выполнение программы
else
        while(strcmp(gets(str), ""))
      fputs(str, fi);
fclose(fi);

if ((fi=fopen("fil.dat","r"))!=NULL)
   while(fgets(str,80,fi)!=NULL)
      puts(str);
fclose(fi);
```

Если окончание ввода текста с клавиатуры – признак конца файла, формируемый нажатием клавиш <*ctrl*>/<*z*>, то блок операторов ввода текста будет иметь вид:

```
fi = fopen ("f.dat", "");
while(gets(str)!=NULL)
   fputs(str, fi);
```

Пример кода функции копирования текстовых файлов:

```
int CopyFile( char *old_name, char *new_name)
{   FILE *f_old, *f_new;
    char str[80];
    f_old=fopen(old_name, "r");
    f_new=fopen(new_name, "w");
    if (f_old==NUL||f_new==NULL)
    {  puts("Error opening file"); //Ошибка открытия файла
       return 1;
    }
    fgets(str, 80, f_old);
    while (!feof(f_old))
    {   fputs (str, f_new);
        fgets (str, 80, f_old);
    }
    return 0;
}
```

3. Форматированный ввод-вывод

Для форматированного ввода/вывода в текстовые файлы используются функции fscanf и fprintf.

Функция

*int fprintf(FILE *f, const char *format, <список вывода>);*

выводит в поток *f* переменные списка вывода, в соответствии с форматной строкой *format*. Возвращает число записанных символов.

Функция

*int fscanf(FILE *f, const char *format, <список ввода>);*

вводит данные из потока *f* по адресам, указанным в списке ввода, в соответствии с форматной строкой *format*. Возвращает число введенных переменных, а в случае достижения конца файла значение *EOF*.

В следующем коде показана работа функций форматного, символьного и строкового ввода/вывода:

```
#include<stdio.h>
int main()
{   int n;
    char str[50],str1[50],ch;
```

```
   FILE *fp;
   if (fp=fopen("name.txt","w"))!=NULL)
   {  puts("\nEnter digit: ");          //введите цифру
      scanf("\n%d",&n); fprintf(fp,"%d\n",n);
      puts("\nEnter symbol: ");       //введите символ
      ch=getchar(); putc(ch, fp);
      puts("\nEnter string:");         //введите строку
      gets(str); fputs(str, fp);
      fclose(fp);
   }
       else printf("\n error");

       if ((fp=fopen("name.txt","r"))!=NULL)
   {  fscanf(fp,"%d",&n);  printf("n=%d\n",n);
      ch=getc(fp);  putchar(ch);
      fgets(str1,50,fp);   puts(str1);
      fclose(fp);
   }
   else printf("\n error");
   return 0;
}
```

16.3.2. Ввод-вывод блоками

Ввод/вывод блоками используется при работе с бинарными потоками. Блок – это область памяти, содержимое которой вводится (выводится) в поток (из потока). Для чтения и записи блоками используются функции:

```
size_t  fread(void *buf, size_t size, size_t  count,                FILE
*f);
size_t  fwrite(const void *buf, size_t size, size_t  n,
                    FILE *f);
```

Функция *freed* читает из потока *f* блок данных длиной *size*count*, и записывает его по адресу *buf*. Возвращает количество прочитанных элементов, равное *count* в случае успешного завершения операции чтения, и меньшее *count* в случае ошибки или конца файла. Здесь *size_t* – беззнаковый целочисленный тип, используется для представления типа операции *sizeof*. Тип *size_t* определяется в заголовочных файлах.

Функция *fwrite* записывает в поток *f* содержимое блока памяти по адресу *buf (count* элементов длиной *size* байт). Возвращает число реально записанных в файл элементов.

Если структуре типа *struct book* присвоено значение в соответствие описанию ее полей, то значение структуры *libry* можно поместить в файл, а затем вывести из файла:

```
struct book
{   char title[81];
    char author[21];
    int year;
    int page;
    float price;
};
struct book libry;
FILL * fi = fopen("libry.dat", "r");
//присваивание полям структуры значений
. . .
fwrite(&libry, sizeof(libry), 1, fi);
fread(&libry, sizeof(libry), 1, fi);
//вывод значений полей структуры на экран
. . .
```

Если требуется считать две записи по указанному адресу, следует описать массив структур:

*struct book libry**[2];***

Сформируем массив структур и запишем структуры в файл:

```
File *f1, *f2;
typedef struct
{   char fam[20];
    int gr;
    int ball[4];
} STUD;             //ввели новое имя типа STUD для шаблона структуры
STUD  p[10];      //p – массив из 10-ти структур
const char name[5]="name";
system("cls");
if ((f1=fopen(name,”w”))==NULL) exit(0);
        //Ввод исходной информации в массив
```

```
printf("\n Enter the number of students:");
scanf("%d", &kol);
        //Ввод фамилии, группы, оценок экзаменов
printf("\n Name, group number, the exams cores |n");
for (int i=0;  i<kol;  i++)
{   scanf("%s  %d", p[i].fam, &p[i].gr);
    for (k=0; k<4; k++)
       scanf("%d", &p[i].ball[k]);
}
        //запись массива в файл
fwrite(p,sizeof(p[i]),kol,f1);
fclose(f1);
system("cls");
f2=fopen(name,"r");
printf("\n Name \t\t group number\t\t\t exams cores ");
i=0;
//вывод структуры из файла в массив
while(fread(&p[i],sizeof(p[i]),1,f2))
    i+=1;
//вывод значения массива
for(int c=0; c<i; c++)
{   printf("\n%s\t", p[c].fam);
    printf("%d\t\t\t", p[c].gr);
    for(k=0; k<4; k++)
        printf("%d\t", p[c].ball[k]);
}
fclose(f2);
```

Приведем код вывода файла на экран чтением его содержимого

- в виде текста функциями ввода/вывода строк;
- в виде текста функциями форматного ввода/вывода;
- в виде бинарного файла в шестнадцатеричном и символьном формате.

Дамп файла (*dump*) – копирование содержимого файла на устройство вывода или во внешнюю память. При выводе файла в бинарном виде символ *'\n'* преобразуется в символы возврат каретки и перевод строки с кодами *0xd* и *0xa*, при этом в символьном дампе выводятся два символа «точка».

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define BYTES  7
```

```
int main (void)
{  FILE *fb;
   char file_name [80], buf [BYTES], str[80];
   int i, n, x;

   while(puts("Enter the File's Name:"),
        (fb=fopen(gets(file_name),"rb"))==NULL)
   {  printf("Error of opening File %s \n", file_name);
      putchar('\n');
   }

   puts("   Text:");
   while(fgets(str,80,fb)) puts(str);  //вывод текстового файла
   puts("");

   puts("    Text and Numbers:");
   fseek(fb,0,0);          //текущая позиция в начало файла
   fgets(str,80,fb); puts(str);  //вывод строк
   for(i=0; i<5; i++)
   {  fscanf(fb,"%d",&x);              //вывод чисел
      printf("%2d",x);
   }
   puts("");
   puts(" BINARY FILE:");
   fseek(fb,0,0);      //текущая позиция в начало файла
   do
   {  n=fread( buf, 1, BYTES, fb); puts("");
      for (i = 0; i < n; i++)  //шестнадцатеричный дамп
         printf("%#5x", buf[i]);
      puts("");
      for (i = 0; i < n; i++)    //символьный дамп
      {  if (iscntrl(buf[i]))  //если символ управляющий,
            printf("%5c", '.');    //вывод точки
         else printf("%5c", buf[i]); //иначе – символа
      }
      puts("");
   }
   while (n==BYTES);
   {
      getch();
      return 0;
```

```
    }
    return 0;
}
```

16.4. Работа с буфером

Буфером называется область оперативной памяти, используемая потоком для временного хранения данных из файла. Для работы с буферами используются функции *setvbuf, setbuf, fflush*.

Функция

```
int setvbuf(FILE* f, char* buffer, int mode,
            size_t size);
```

определяет буфер ввода/вывода и режим работы с ним для потока *f*. Вызывается эта функция после открытия файла, но перед доступом к нему. При успешном завершении функция возвращает значение 0, а в случае неудачи – ненулевое значение.

Параметр *buffer* указывает на блок памяти для буфера. Если этот параметр равен *NULL,* то функция *setvbuf* использует функцию *malloc* для захвата памяти под буфер.

Параметр *mode* определяет режим работы с буфером и может принимать следующие значения:

- *_IOFBF* – вывод данных из буфера во внешнюю память выполняется только при полной загрузке буфера или при закрытии файла;
- *_IOLBF* – вывод данных из буфера во внешнюю память выполняется при записи в буфер символа ’\n’;
- *_IONBF* – нет буферизации, в этом случае параметры *size* и *buffer* игнорируются.

Параметр *size* определяет длину буфера в байтах.

Функция

```
int setbuf(FILE* f, char* buffer);
```

вызывает функцию *setvbuf*. Причем если значение параметра *buffer* не равно *NULL*, то функция *setvbuf* вызывается следующим образом:

```
setvbuf(f, buffer, _IOFBF, BUFSIZE);
```

где константа *BUFSIZE* задает длину буфера по умолчанию. Эта константа описана в заголовочном файле *stdio.h*. В противном случае буферизация не используется, т.е. функция *setvbuf* вызывается следующим образом:

```
setvbuf(f, 0, _IOFBF, BUFSIZE);
```

Функция

int fflush(FILE f);*

очищает буфер вывода, записывая данные из буфера потока *f* в соединенный с этим потоком файл. В случае успешного завершения функция возвращает значение 0, а в случае неудачи – EOF. Если значение параметра *f* равно *NULL*, то освобождаются буферы всех потоков, которые работают в режиме вывода.

16.5. Функции удаления, переименования и создания временного файла

Функция удаления файла с указанным именем (*filename*):

int remove (const char filename);*

Пример удаления файла:

```
if(remove(filename))
    printf("There is no such a file.\n");
else
    printf("The file was deleted.\n");
```

Функция переименования файла именем с именем *old_name* в файл с именем *new_name:*

*int rename(const char *old_name, const char *new_name);*

Если файл с именем new_name уже существует, то работа функции зависит от реализации. В случае успешного завершения функция возвращает 0, а в случае неудачи – ненулевое значение.

Функция создания временного файла в режиме ”w+b”:

FILE tmpfile(void);*

После закрытия потока файл удаляется. В случае успешного завершения функция возвращает указатель на файл, а в случае неудачи – *NULL*.

Функция, возвращающая имя для временного файла:

char tmpnam(char* str);*

Максимальное количество имен равно *TMP_MAX*, максимальная длина имени равна *L_tmpnam*. Если значение параметра *str* равно *NULL*, то

функция возвращает указатель на свою строку, в противном случае возвращается указатель *str*.

Пример использования временного файла

```
int code;
char name[80];
double salary = 0.0;
FILE* temp;                 //временный файл
//открытие временного файла
if(!(temp = tmpfile()))
    {  printf("Create temp file failed.\n");
       return 1;
    }
printf("Input code, name and salary.\n");
printf("Press Ctrl+z to exit.\n>");
//ввод данных и  запись их во временный файл
scanf("%d%s%lf", &code, &name, &salary);
while (!feof(stdin))
    { fprintf(temp, "%d %s %f ", code, name, salary);
        printf(">");
        scanf("%d%s%lf", &code, &name, &salary);
    }
//установка индикатора позиции на начало файла
rewind(temp);
//вывод данных из временного файла
printf("\nRead from the temporary file.\n");
fscanf(temp, "%d%s%lf", &code, name, &salary);
while (!feof(temp))
    { printf("code = %d name = %s sal = %f\n",
        code, name, salary);
        fscanf(temp, "%d%s%lf", &code, name, &salary);
    }
//закрытие временного файла
   fclose(temp);
```

17. Препроцессор языка C++

Директива препроцессора – это специальная команда компилятора, которая выполняется непосредственно перед компиляцией программы. Во время препроцессорной обработки компилятор читает текст программы и обрабатывает его в соответствии с указанными директивами препроцессора. Синтаксис директив препроцессора:

- директивы начинаются символом # в первой позиции строки;
- директива заканчивается в конце строки, при этом знак (;) не ставится, а символ '\' продолжает определение директивы на следующую строку;
- пробел в директиве является разделителем.

Директивы могут быть указаны в любом месте исходного файла, их действие остается в силе до конца файла.

Существует несколько типов директив препроцессора, часть которых мы уже использовали в наших программах.

17.1. Директива включения заголовочных файлов

С директивой препроцессора *include* мы познакомились в п. 3.1 и использовали ее в программах при подключении заголовочных файлов.

Заголовочные файлы содержат объявления и определения функций, шаблонов, имен типов, констант, перечислений, а также директивы препроцессора, в том числе могут содержать директивы *include*, включающие в него другие файлы заголовков и т.д.

При компиляции строка директивы *include* будет заменена содержимым файла, имя которого указано в директиве в угловых скобках или двойных кавычках.

При этом размер создаваемой программы не увеличивается, т.е. прототипы нужны только на этапе компиляции и не переносятся в объектный модуль, не увеличивая объектный код. Прототипы функций, не вызываемых в модуле, не используются компилятором.

При наличии прототипов вызываемые функции не обязаны размещаться с вызывающей функцией в одном файле, а могут оформляться в виде отдельных модулей или находиться в уже оттранслированном виде в библиотеке объектных модулей.

Это относится к функциям, которые готовит программист для включения в свою программу, и к функциям из стандартных библиотек компилятора. В последнем случае определения библиотечных функций,

уже оттранслированные и оформленные в виде объектных модулей, находятся в библиотеке компилятора.

17.2. Директива define

Директива *define* используется для задания констант, ключевых слов, операторов и выражений, используемых в программе.

#define идентификатор строка замещения

При компиляции каждое вхождение идентификатора замещается строкой замещения. Этот процесс часто называют макрорасширением, а идентификатор – макроопределением или макрокомандой.

Обратите внимание! Если идентификатор появляется в тексте программы внутри двойных кавычек, то он не заменяется строкой замещения.

Директива *define* применяется в трех случаях:

1. Для создания символических констант – во всем файле вместо идентификатора при препроцессорной обработке исходного файла подставляется строка замещения, которая может быть константой, выражением или инструкцией. Символические константы, как правило, обозначаются заглавными буквами:

```
#define  MAX   100
#define  MASK   0377
#define  TXT   "Input array:\n"
...
int  arr[MAX], i;
printf(TXT);
for(i=0; i<MAX; i++) scanf ("%d", &arr[i]);
for(i=0;i<MAX;i++) arr[i]&=MASK;//побитовое ”И” над целым
```

Если в приведенный фрагмент добавить директиву

```
#define  FORR  for(1=0;i<MAX;i++)
```

где строка замещения является инструкцией, то код фрагмента сократится и будет иметь следующий вид:

```
FORR  scanf("%d", &arr[i]);
FORR  arr[i]&= MASK;
```

Директиву *define* удобно использовать для определения целых констант:

```
#define  TRUE   1
#define  FALSE  0
#define  SIZE_ARRAY   100
```

Чтобы отменить последнее определение для некоторого идентификатора, используется директива

#undef идентификатор

Обратите внимание! Директива *#define* и инструкция объявления *typedef* – совершенно разные по действиям конструкции языка С++:

- *typedef* обрабатывается компилятором, а *#define* - препроцессором;
- *typedef* дает символические имена только типам данных, а в *#define* – строка замещения может быть константой, выражением, инструкцией или макроопределением;
- *typedef* более гибкий инструмент, т.к. компилятор может выполнить проверку состоятельности объявления.

Приведем пример, поясняющий отличия *typedef* и *define*. Следующие две инструкции эквивалентны по описанию переменных:

```
int *p1, *p2, *p3;
```
и
```
typedef  int* PTI;
PTI p1, p2, p3;
```

Если использовать директиву *define* следующим образом:

```
#define  PTI  int*
PTI  p1,  p2,  p3;
```

то получим не эквивалентную инструкцию описания переменных:

```
int  *p1, p2, p3; //p2 и p3 в этом случае int, а не *int
```

2. Директива *define* используется для записи макроопределений с аргументами (макросы):

#define идентификатор(арг1,арг2, ...) строка замещения

Например, файл *ctype.h* содержит уже знакомое нам (п. 2.1.2) макроопределение *isdigit(c)* для распознавания символа-цифры, позволяющее повысить наглядность программы при его использовании вместо соответствующего выражения с операциями отношения:

```
#define  isdigit(c)  ((c)>='0'&&(c)<='9')
```

Правила написания макроопределений:

- макроопределение не должно содержать пробелов (1-й пробел заканчивает макроопределение);
- строку замещения, равно как и каждый аргумент в этой строке, необходимо заключать в круглые скобки:

```
#define  ABS(X)     ((X)<0?-(X):(X))

#define  MAX(X,Y)  ((X)>(Y)?(X):(Y))
 printf("%d", MAX(i,i+1));
```

Каждое вхождение выражения *MAX(выражение1, выражение2)* в тексте программы заменяется строкой замещения ***((X)>(Y)?(X):(Y))***, причем аргументы макроопределения *X* и *Y* заменяются на *выражение1* и *выражение2* соответственно.

Подобное использование директивы *define* похоже на использование функции, но различия между использованием функции и макроопределения весьма существенны:

- макроопределение создает «строчной» код, т.е. компилятор всякий раз при обработке макроопределения формирует и компилирует инструкцию кода в программе. Функция компилируется только один раз;
- при вызове функции управление передается функции, а после завершения ее работы реализуется возврат в вызывающую программу. Код с макроопределениями выполняется последовательно.
- вызов функции передает значение аргумента в функцию во время выполнения программы. Макровызов передает строку замещения в программу до ее компиляции.

Если макроопределение применить 10 раз, то в программу вставится 10 строк кода, т.е. увеличится объем кода. При вызове функции 10 раз увеличится время работы программы.

Таким образом, использование макроопределений экономит время, а функций - память.

Обратите внимание! Часто встречается ошибка, когда аргументы макроса не заключаются в скобки. Покажем, что получится в этом случае.

Пусть программа содержит директиву:

```
#define   CUB(X) (X*X*X)
```

Тогда макрокоманда *CUB(a+3)* заменится на *(a+3*a+3*a+3)*, что, как мы видим, не является значением $(a+3)^3$.

Препроцессор не выполняет вычислений, он только *точно* делает предложенные подстановки.

В следующем примере в комментариях приведены результаты выполненных подстановок директивы *define*. Предлагаем самостоятельно разобраться, почему получается результат, приведенный в комментариях:

```
#define  SQ(x)  x*x
#define  PR(x)  printf("x равен % d \n",x)
int main()
{
      int x=4;
   PR(SQ(x));      //SQ(x)              равно 16
   PR(SQ(x+2));        //SQ(x+2           равно 14
   PR(SQ(5));          //SQ(5)            равно 25
   PR(100/SQ(5));      //100/SQ(2)        равно 100
   PR(SQ(++x));        //SQ(++x)          равно 36
   return 0;
}
```

Приведем эквивалентные выполненным подстановкам операторы**:**

```
printf("SQ(x) равно  %d\n", x*x);
printf("SQ(x+2) равно  %d\n", x+2*x+2);
printf("SQ(5) равно  %d\n", 5*5);
printf("100/SQ(2) равно %d \ n",100/2*2);
printf("SQ(++x) равно  %d\n", ++x*++x);
```

В макроопределении *x* замещается символом, использованным в макровызове программы. При этом даже внутри двойных кавычек в определении *PR* переменная замещается соответствующим аргументом.

3. Директива *define* используется для условной компиляции кода. Используется вместе с директивами *ifdef* и *ifndef.*

В зависимости от некоторых условий можно компилировать те или иные части исходного текста. Управление этим процессом осуществляется с помощью следующих директив препроцессора:

```
#if константное выражение
{...}
[#elif конст. выражение {...}]   //альтернативные условия
[#elif конст. выражение {...}]
# else
{...}
#endif
```

Пусть программа использует различные типы данных в зависимости от некоторых условий. Введем следующие обозначения констант:

```
#define     SHORT     0
#define     LONG          1
#define     REAL          2
#define     COORDSYM  REAL
...
#if  COORDSYM==SHORT        //не компилируется
   short x,y;
#if  COORDSYM==LONG         //не компилируется
   long x,y;
#if  COORDSYM==REAL         //компилируется
   double x,y;
#endif
```

Временно приостановить выполнение кода можно с помощью директивы

```
#if  0  //аналог временного комментирования фрагментов кода
...
#endif
```

Наиболее часто используются директивы условной компиляции *ifdef* и *ifndef*, применяемые вместе с директивой *define*:

```
//1 фрагмент
#ifdef  идентификатор  //истина, если идентификатор определен
{Код  программы}            //ранее директивой define
```

```
#else
{Код  программы}
#endif
```

```
//2 фрагмент
#ifndef идентификатор    //истина, если идентификатор
{Код  программы}          //в настоящий момент не определен
#else
{Код программы}
#endif                    //конец условной компиляции
```

Если проверки дают значение истины, то строки от *#else* до *#endif* при компиляции игнорируются, в противном случае строки от проверки до *#else* (а при отсутствии *#else* до *#endif*) игнорируются.

Условную компиляцию имеет смысл использовать при отладке программы вместо заключения отдельных частей программы в комментарии.

Если заключаемый в комментарий /*...*/ участок программы содержит еще один такой же комментарий, нарушается согласование открывающих и закрывающих комментарии символов и программа работать не будет.

Условная компиляция устраняет эту проблему:

```
#if 1
  { исполняемая  часть программы }
#else
  { не исполняемая часть  программы }
#endif
```

Условную компиляцию делает программу мобильной, переносимой с одной платформы на другую, определяя те части программы, которые будут реализованы в том или ином случае.

18. Динамические списки

При разработке приложений, оперирующих с большим количеством входных данных, возникает вопрос об их хранении во время выполнения программы. Для этого используется динамическая память, с которой мы уже работали, в том числе создавали динамические массивы (п.7). Динамические массивы при их обработке часто не удобны из-за фиксированного размера и вычислительной сложности вставки и удаления элементов.

Данная проблема решается с помощью связанных списков динамических переменных. Компоненты в них добавляются и удаляются во время выполнения программы, и их количество зависит исключительно от размера доступной памяти. Кроме того, с использованием динамических списков данных эффективно реализуются алгоритмы поиска, вставки информации в нужное место списка и удаление информации из списка.

Недостатком динамических списков является то, что в каждый момент времени нам доступны только определенные компоненты - начало, конец списка и его текущее обрабатываемое значение. Здесь преимущество имеют динамические массивы - нам в любой момент доступен любой его элемент по индексу или через указатель.

Для создания динамических списков используется тип *struct*, который содержит одно или несколько полей-указателей на этот же тип. Если такое поле одно, то мы имеем линейный динамический связный список, если несколько – то нелинейный.

18.1. Линейные односвязные списки

Линейный связный список - это структура данных, в которой компоненты расположены в линейном порядке. Каждая компонента списка содержит информационную часть, а также ссылку на следующую компоненту.

Обратите внимание! Динамическое представление списков позволяет расположить компоненты в памяти произвольно, снабдив каждую компоненту указателем места расположения в памяти следующей компоненты. Т.е. для адресов компонент списка не выполняется свойство упорядоченности.

Связанные явными ссылками компоненты образуют цепочку элементов. Линейная структура данных, хранящая символьные значения, приведена на рис. 7:

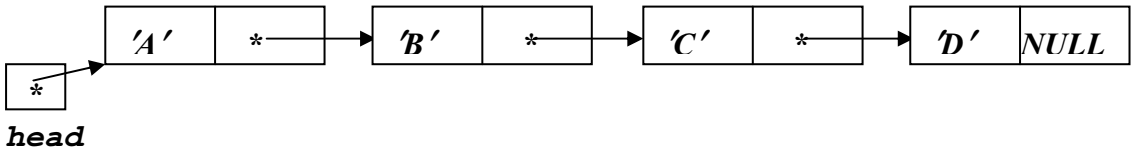


Рис. 7. Динамический список, хранящий символы

В переменной с именем *head* (голова списка) хранится указатель на начало списка, а поле-указатель последней компоненты списка хранит значение *NULL*. Компонента такого списка описывается с помощью структуры:

```
struct        list
{ char ch;
  list* next; //ссылка на следующий элемент
} *head;
```

Память под компоненту списка выделяется динамически в нужный момент работы программы.

Можно ввести еще одну переменную для эффективного добавления компонент в конец списка - указатель на его конец *tail* (хвост). Компоненты списков, как правило, имеют несколько информационных полей, которые обозначим *info*. Тогда список можно представить следующим образом (рис.8):

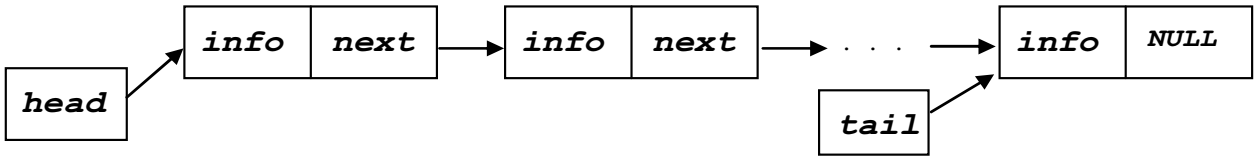


Рис. 8. Линейный односвязный список

где *head* - указатель на начало списка, - *tail* на его конец.

Для создания пустого списка указателю *head* нужно присвоить значение *NULL*.

1. Добавление компонент в список

Для создания новой компоненты нужно динамически выделить под нее память и заполнить ее информационную часть. Если компонента

помещается в конец списка, то ее ссылочной части следует присвоить значение *NULL*.

Рассмотрим два случая – 1) список пуст; 2) в нем есть хотя бы одна компонента. Если список пуст, то *head* и *tail* будут указывать на вновь созданный компонент. Если компоненты в списке присутствуют, то последнюю компоненту списка нужно связать с добавляемой, т.е. ссылочной части компоненты с адресом *tail*, присвоить адрес *cur* добавляемой компоненты, а затем "передвинуть" *tail* , присвоив ему адрес *cur.*

Создадим список из *n* чисел, расположенных в порядке убывания от *n* до 1, и выведем его значения на экран. Алгоритмы создания и вывода списка реализуем в виде функций *creat* и *print:*

```
struct        list
        { int x;
          list* next;        //ссылка на следующий элемент
        };
list* creat(int);            //создание списка из n компонент
void print(list *);          //вывод списка

int main()
{struct list *head;
 int n;
 cin>> n;
 head = Creat(n);            //вызов creat
 Print(head);                //вызов print
 getch();
}

list* Creat(int n)   //определение функции creat
{list *cur,*first;
 if (n) first = new list;
 first->x = n; cur = first; n--;
 while (n>0)
        { cur->next = new list;
          cur = cur->next;              //"передвинули" указатель
          cur->x = n;
          cur->next = NULL;
          n--;
        }
```

```
return first;
}

void Print(list *cur)  //определение функции print
{while (cur->next)
   {cout<<cur->x<<'\t';
    cur = cur->next;
   }
 cout<<cur->x;
}
```

В общем случае добавлять компоненту в список можно в любое место, а не только в его конец, в зависимости от поставленной задачи, например:

- вставка элемента в начало списка;
- вставка элемента, не нарушая упорядоченности списка;
- вставка элемента *x* за заданным значением *y*;
- вставка элемента *x* перед заданным значением *y*;
- вставка элемента *x* *k*-м по порядку;

Функция вставки компоненты *x* за значением *y* имеет вид:

```
void vst_el(list *head,int x,int y)
 { link *v, *cur=head;
  while((cur->next)!= NULL) //эквивалент while(cur->next)
   {if (cur->x == y)
      {v = new struct list;
       v->x = x ;
       v->next = cur->next;
       cur->next = v; return;
       }
    else cur = cur->next;
  }
```

2. Удаление компонент из списка

Если нужно удалить компоненту из списка, необходимо "перенаправить" указатель (рис. 9), чтобы не потерять часть списка, расположенного за удаляемым элементом.

Рис. 9. Удаление элемента в односвязном списке

Необходимо рассмотреть все возможные ситуации обработки информации:

- список может быть пуст к данному моменту,
- если удаляется первая компонента списка, то вторая компонента становится первой.

Данные ситуации предусмотрены в коде:

```
struct list* Delete(struct list *head,
                    int del_elem)
{
   struct list *cur,*del;

   if (head == NULL)return NULL;              //список пуст
   if (head->info == del_elem)
   {   cur = head;                            //удаление первой
       head = head->next;                     //компоненты списка
       delete cur;
       return head;
   }
   cur = head;                                //удаление
   while (cur->next!= NULL)
   {   if (cur->next->info == del_elem)
       {   del = cur->next;
           cur->next = del->next;
           delete del;
           return head;
       }
       else cur = cur->next;
   }
   return head;
}
```

Функция полного удаления списка (освобождения памяти) имеет вид:

```
Clean_spisok(struct list *head)
{
        struct list *cur,*del;
         del = head;
         while(head)
```

```
        {       head = head->next;
                delete del;
                del = head;
        }
    return head;
}
```

18.2. Стеки

В рассмотренных выше примерах при обработке списков использовалась стратегия First In First Out (FIFO), т.е. первый пришел – первый вышел. Стек функционирует по принципу LIFO (Last - In - First-Out), т.е. последний пришел – первый вышел.

Стек – это динамическая структура линейно связанных компонент, для которых разрешено добавлять или удалять компоненты только с одного конца списка, называемого *вершиной (головой)* стека, при этом изменяется адрес вершины стека. Каждый стек имеет базовый адрес, от которого производятся все операции со стеком.

Важным применением стека в программах является использование стековой памяти для хранения адресов возврата из вызванной в вызывающую функцию, обработки прерываний, передаче через стек параметров в функции, размещение в стеке локальных параметров функций, реализация алгоритмов, имеющих рекурсивный характер. С понятием стек мы познакомились в п. 2.1.1, когда рассматривали механизм работы функции форматного вывода.

В результате добавления или исключения компоненты изменяется значение указателя *head* на вершину стека.

Определим структуру данных для создания стека:

```
struct stack
        {  int  x;
           struct  stack *next;
        }* head,*cur;
```

Механизм пошагового создания стека можно представить следующим образом:

```
        head = NULL;            //дно стека
        cur = new list;    //создание 1-й компоненты
        cur->next=NULL;
```

```
cur->x=1;
head = cur;              //вершина стека
cur = new list;  //добавление 2-й компоненты
cur->next=head;
cur->x=2;
head = cur;              //изменение адреса вершины стека
```
и т.д.

Вызовем и определим функцию создания стека из 10-ти компонент:

```
struct stack *head = Creat_st(10); //вызов функции

stack* Creat_st(int n)              //определение функции
{       stack *cur,*head;
        if (n) head = new struct stack;
        head->x = n; head->next = NULL;
        n--;
        while (n>0)
    {   cur = new struct stack;
        cur->x = n; cur->next =head;
                head = cur; n--;
    }
        return head;
}
```

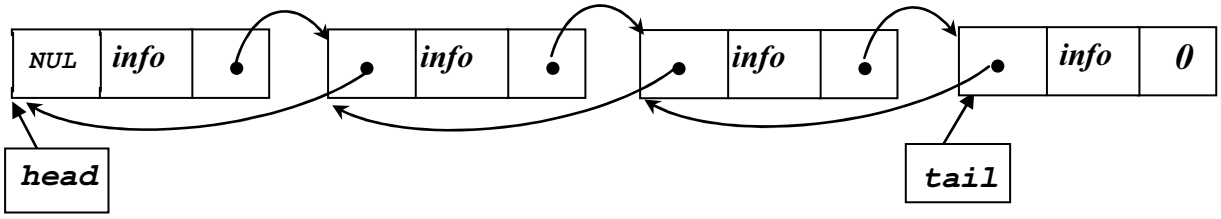
Функция исключения компоненты стека имеет вид:

```
struct stack* Del_elem(struct list *head)
{   struct list *del;
    del = head;
    head = head->next; //новый адрес стека
    delete del;                 //удаление (освобождение памяти)
    return head;
 }
}
```

18.3. Линейные двусвязные списки

Часто вычислительные процессы требуют при обработке данных списков возврата к предыдущим компонентам. Использование односвязных списков сопряжено с проблемой возврата в этих случаях в начало списка.

В *линейном двусвязном списке* продвижение возможно в любом из двух направлений. Каждый элемент двусвязного списка содержит два указателя: указатель на следующую компоненту и указатель на предыдущую компоненту (рис. 10). При этом первая и последняя компоненты списка содержат нулевые указатели соответственно на предыдущую и последующую компоненты. Линейные двусвязные списки позволяют достаточно просто осуществлять вставки и удаления элементов слева и справа от текущего элемента.



Р

Рис. 10. Линейный двусвязный список

Структура для создания двунаправленного списка имеет вид:

```
struct list
{   struct list *prev;  //указатель на предыдущую компоненту
        int   info;
    struct list *next;  // указатель на следующую компоненту
} *p, *head, *tail;
```

При формировании двунаправленного списка алгоритм остается тем же, что и при формировании однонаправленного списка, только добавляется еще один оператор присваивания полю *prev* значения. Приведем алгоритм добавления элемента в конец двунаправленного списка:

```
tail->next = new struct list;   //добавляем в конец списка
                                        // новый элемент
tail->next->prev = tail;              //его полю prev присваиваем
                                        //адрес предыдущего элемента
tail = tail->next;                 //новое значение адреса
                                        //конца очереди
//ввод информации в поле tail->info
        . . .
tail->next = NULL;                    //признак конца очереди
```

Обратите внимание! В алгоритмах удаления и вставки меняются значения полей-указателей у двух элементов списка, между которыми вставляем или удаляем элемент.

Так как список двунаправленный, то понятие «начала» и «конца» – чисто условное. Поэтому добавлять (исключать) элементы можно и в начало, и в конец (из начала и из конца) очереди.

18.4. Циклические списки

Часто для повышения эффективности алгоритмов обработки данных удобно использовать циклические списки, в структуре которых отсутствуют первый и последний элементы, а есть текущий элемент *cur*, который указывает на одну из компонент списка. Для получения из линейного двусвязного списка циклического двусвязного списка необходимо полям пустых указателей присвоить значения указателей противоположных концов списка. Для входа в такую циклическую структуру или для передачи ее в качестве аргумента в функцию следует иметь ее адрес. Это может быть значение адреса *cur* текущей обрабатываемой компоненты или фиксированное значение *head* (рис. 11).

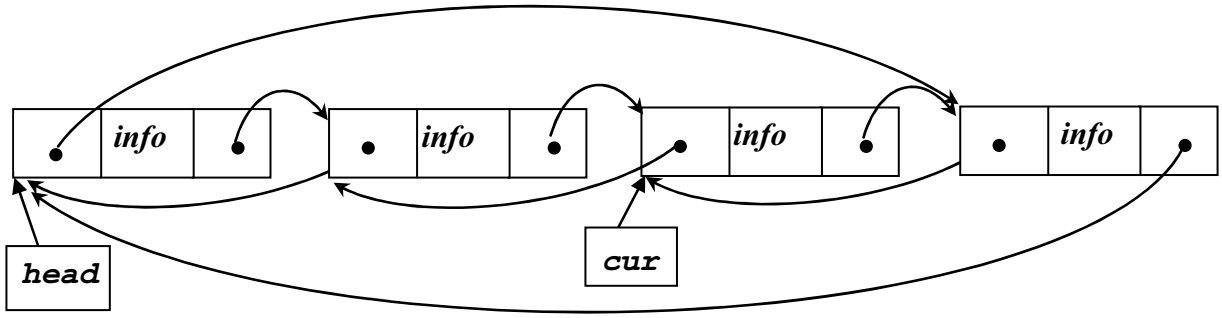


Рис. 11. Циклический двунаправленный список

Обратите внимание! При решении задачи поиска заданного элемента в циклическом списке для предотвращения зацикливания при его обходе значение указателя, с которого начинается просмотр списка, сохраняется и служит ограничителем просмотра.

18.5. Бинарные деревья

Структура компоненты бинарного дерева содержит два указателя на такие же компоненты следующего уровня. Одно из информационных полей компоненты дерева называется ключом. По его значению

идентифицируют данную компоненту, т.е. находят нужную информацию. Обычно в качестве ключей используют целые числа или строки, т.е. порядковые типы.

Двоичные деревья удобны для организации быстрого поиска данных, особенно в тех случаях, когда необходимо иметь возможность выдать данные в отсортированном виде.

Бинарное дерево поиска – это упорядоченное дерево, причем для каждого узла выполняется правило: все ключи левого поддерева меньше значения ключа этого узла, а все ключи правого поддерева больше значения ключа этого узла.

Обратите внимание! Бинарное дерево представляет собой рекурсивную структуру, т.к. каждое его поддерево само является бинарным деревом, и, следовательно, каждый его узел в свою очередь является корнем дерева. При этом можно построить как рекурсивные, так и не рекурсивные алгоритмы работы с деревьями.

Алгоритм создания дерева

Первая компонента становится корнем дерева. Просмотр дерева для добавления в него вершины начинается с корня. Если ключ добавляемой компоненты меньше ключа очередной вершины дерева, то идем по адресу *left* (по левой ветке дерева), иначе по адресу *right* (по правой ветке дерева). Процесс повторяем до тех пор, пока не дойдем до вершины дерева с нулевым полем *left* или *right*, в котором сохраняем адрес добавленной в дерево компоненты.

На рис.12 представлено упорядоченное дерево, построенное по данному алгоритму, если компоненты добавляются в дерево в следующем порядке - 15, 25, 10, 12, 6, 9, 3, 30, 20, 18, 40:

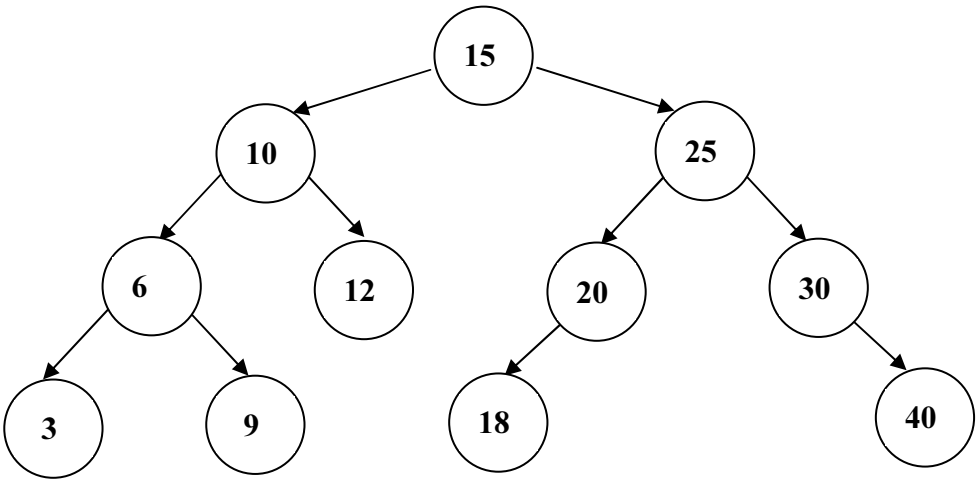


Рис.12. Бинарное дерево поиска

Приведем рекурсивный алгоритм создания бинарного дерева:

```
typedef struct                  //структура для работы с деревом
        { char info[80];        //данные и одновременно ключ
          struct tr *left;
          struct tr *right;
        } tree;
//прототип функции создания дерева
tree *Cr_tree(tree*,tree*,char*);

//фрагмент кода
char s[80];
tree *root=NULL;       // root – корень, построено пустое дерево
do
 {
   printf("Enter information:");
   gets(s);
   root=Cr_tree(root,root,*s); //вызов функции Cr_tree
 } while(strcmp(s,"\n");

//определение функции создания дерева
tree *Cr_tree(tree *root, tree *r, char s[80])
{
  if(!r)
        {  r=new tree;     //выделение памяти под компоненту
           if(!r) {printf("Out of memory\n");
                   exit(0); }
           r->left=NULL;  r->right=NULL; //присваивание
           strcpy(r->info, s);              //полям значений
           if(!root)   return r;            //корень создан
           if(strcmp(s,root->info)<0)
                        root->left=r;
           else         root->right=r;
           return r;
        }
  if(strcmp(s,r->info)<0)
                Cr_tree(r,r->left,s);     //рекурсивные вызовы
  else   Cr_tree(r,r->right,s);
```

```
  return root;
}
```

Использование рекурсивных функций расходует системные ресурсы, но в данном случае использование рекурсии является оправданным, поскольку нерекурсивные функции для работы с деревьями гораздо сложнее и для написания, и для восприятия кода программы.

Алгоритм обхода дерева можно реализовать рекурсивно, а также с использованием списков – стека или очереди. Обход можно выполнять в как глубину, так и в ширину. Обход дерева всегда начинается с его корня.

Рекурсивная процедура обхода дерева будет иметь вид:

```
void detour_in_depth(tree *r)
{
   if (!r)
      return;
   printf("%d ", r->data);
   prefix(r->left);
   prefix(r->right);
}
```

Алгоритм обхода дерева в глубину с помощью стека

Обрабатывается корень дерева, и, далее, если поле *left* отлично от нуля, то обрабатывается вершина по адресу *left*, а вершина по адресу *right* заносится в стек (в стеке сохраняется адрес правого поддерева).

Продолжаем процесс обработки вершин дерева по адресу *left*, занося в стек вершины по адресу *right*. Если адрес *left* очередной вершины равен нулю, то извлекаем для обработки вершину из стека.

Порядок обхода вершин дерева, представленного на рис.12, в глубину будет выполнен в следующем порядке: 15, 10, 6, 3, 9, 12, 25, 20, 18, 30, 40.

Алгоритм обхода дерева в ширину с помощью очереди

В очередь помещаем корень дерева. Далее выполняются действия: из очереди извлекается компонента и обрабатывается, а в очередь добавляются вершины дерева по адресам обрабатываемой компоненты *left* и *right*.

Порядок обхода вершин дерева, представленного на рис.12, в ширину будет выполнен в следующем порядке: 15, 10, 25, 6, 12, 20, 30, 3, 9, 18, 40.

Функции, реализующие вывод на экран вершин дерева по данному алгоритму:

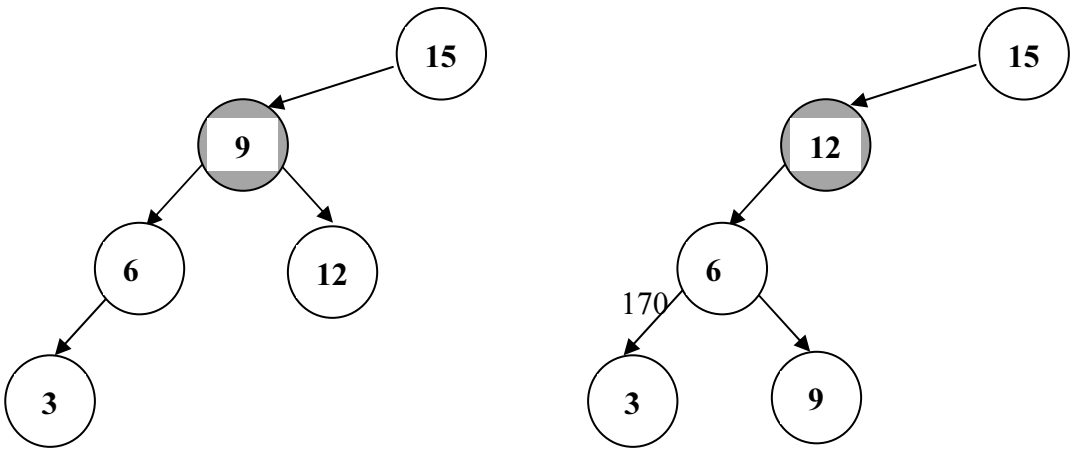
```
void add(tree *elem); //добавление в очередь элемента elem
tree *del(); //удаление из очереди элемента (его обработка)
int empty(); //возвращает 1, если очередь пуста,
                    //0 в противном случае

void width(tree *root)  //обход дерева
{
   if (!root)
      return;
   add(root);
   while (!empty())
        {
      tree *curr = del();
      printf("%d ", curr->data);
      if (curr->left)
         add(curr->left);
      if (curr->right)
         add(curr->right);
   }
}
```

Алгоритм удаления вершины дерева

На место удаляемой вершины дерева помещаем вершину, найденную по следующему правилу: идем по адресу *left* (*right)* удаляемой вершины, и, далее, по адресам *right* до тех пор, пока не дойдем до вершины с нулевым значением поля *right* (*left)*. Ее ставим на место удаляемой.

На рис. 13 приведена только левая часть нашего дерева с корнем 15 в случае реализации алгоритмов, когда на первом шаге выполнен переход от удаляемой вершины по адресу *left*, затем по адресам *right* и наоборот. На место удаляемой вершины может быть помещена вершина с ключом 9 или с ключом 12.



Обратите внимание! Упорядоченность дерева после удаления его вершин не нарушается.

Список рекомендуемой литературы

1. Страуструп Б. Язык программирования C++: специальное издание / Б.Страуструп. – Бином, Невский Диалект, 2008. – 1104 с.
2. Уэйт М. Язык Си, руководство для начинающих / М. Уэйт, С.Прата, Д.Мартин. – М.: Мир, 1988. – 512 с.
3. Эккель Б. Философия C++. Введение в стандартный C++/ Б. Эккель. Питер, 2004. – 572 с.
4. Эккель Б. Философия C++. Практическое программирование / Б. Эккель, Ч. Эллисон. – Питер, 2004. – 608 с.
5. Шилдт Г. Полный справочник по C++ / Г. Шилдт. – Вильямс, 2007. – 800 с.
6. Дейтел Х. Как программировать на C++ / Х. Дейтел, П. Дейтел. – Бином-Пресс, 2010. – 1456 с.
7. Лафоре Р. Объектно-ориентированное программирование в C++ / Лафоре Р. – Питер, 2004. – 928 с.
8. Павловская Т. C/C++. Программирование на языке высокого уровня / Т. Павловская. – Питер, 2009. – 464 с.
9. Павловская Т., Щупак Ю. C++. Объектно-ориентированное программирование. Практикум / Т. А. Павловская, Ю. А. Щупак. –Питер, 2008. – 272 с.
10. Пышкин Е. Основные концепции и механизмы объектно-ориентированного программирования / Е.Пышкин. – БХВ-Петербург, 2005. – 640 с.

СОКОЛОВА ЭЛЕОНОРА СТАНИСЛАВОВНА
ДМИТРИЕВ ДМИТРИЙ ВАЛЕРЬЕВИЧ
ЛЯХМАНОВ ДМИТРИЙ АЛЕКСАНДРОВИЧ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

Редактор **О.В. Пугина**

Компьютерный набор и верстка **Э.С. Соколова**

Подписано в печать XX.XX.2015. Формат 60x86 $^{1}/_{16}$. Бумага офсетная.
Печать офсетная. Усл. печ. л. 10. Уч.-изд. л. XX,XX– Кол-во стр. : 160.
Тираж 100 экз. Заказ XXX.

Нижегородский государственный технический университет им. Р.Е. Алексеева
Типография НГТУ.
Адрес университета и полиграфического предприятия:
603950, г. Нижний Новгород, ул. Минина, 24