Алфавит, базовые типы и описание данных.

Алфавит языка включает практически все символы, имеющиеся на стандартной клавиатуре:

- латинские буквы A ...Z, a...z;

- цифры 0...9;

- знаки операций и разделители:

{ } [ ] ( ) . , -> & * + - ~ ! / % ? : ; = < > | # ^

Некоторые операции обозначаются комбинациями символов, значения символов операций в ряде случаев зависят от контекста, в котором они употреблены.

Базовые (предопределенные) типы данных объединены в две группы: данные целого типа и данные с плавающей точкой (вещественные).

Данные целого типа могут быть обычными целыми со знаком (signed) и целыми без знака (unsigned). По числу разрядов, используемых для представления данного (диапазону значений) различают обычные целые (int), короткие целые (short int) и длинные целые (long int). Символьные данные (char) также рассматриваются как целые и могут быть со знаком и без знака.

Константы целого типа записываются как последовательности десятичных цифр, тип константы зависит от числа цифр в записи константы и может быть уточнен добавлением в конце константы букв L или l (тип long), U или u (тип unsigned) или их сочетания:

321 - константа типа int,

5326u - константа типа unsigned int,

45637778 - константа типа long int,

2746L - константа типа long int.

Целые константы могут записываться в восьмеричной системе счисления, в этом случае первой цифрой должна быть цифра 0, число может содержать только цифры 0 ... 7:

0777 - константа типа int,

0453377 - константа типа long.

Целые константы можно записывать и в шестнадцатеричной системе счисления, в этом случае запись константы начинается с символов 0x или 0X:

0x45F - константа типа int,

0xFFFFFFFF - константа типа unsigned long.

Константы типа char всегда заключаются в одиночные кавычки, значение константы задается либо знаком из используемого набора символов, либо целой константой, которой предшествует обратная косая черта: 'A', '\33', '\042', '\x1B'. Имеется также ряд специальных символов, которые могут указываться в качестве значений константы типа char:

'\n' - новая строка,

'\t' - горизонтальная табуляция,

'\v' - вертикальная табуляция,

'\r' - перевод каретки,

'\f' - перевод страницы,

'\a' - звуковой сигнал,

'\'' - одиночная кавычка (апостроф),

'\"' - двойная кавычка,

'\\' - обратная косая черта.

Вещественные числа могут быть значениями одного из трех типов: float, double, long double. Диапазон значений каждого из этих типов зависит от используемых ЭВМ и компилятора. Константы вещественных типов могут записываться в естественной или экспоненциальной формах и по умолчанию имеют тип double, например, 15.31, 1.43E-3, 2345.1e4. При необходимости тип константы можно уточнить, записав в конце суффикс f или F для типа float, суффикс l или L для типа long double.

Внешнее определение, объявляющее переменные, заканчивается точкой с запятой:

```
int  i, j, k; // Три переменных типа int без явной инициализации
double  x=1, y=2; //Две переменных типа double с начальными значениями 1 и 2
char     c1='0' ;   // Переменная типа char, ее значение - код литеры 0
```

В качестве спецификаторов класса памяти во внешнем определении может указываться одно из ключевых слов extern, static или typedef, Спецификатор extern означает, что объявляемый объект принадлежит другому программному файлу, а здесь дается информация о его имени и типе и не должно присутствовать инициализирующее выражение. Спецификатор static ограничивает область действия объявляемого имени данным файлом или блоком, если объявление содержится в блоке.

Если объявление данного содержится внутри тела функции (локальное объявление), то можно указывать спецификаторы класса памяти register или auto. Спецификатор register носит рекомендательный характер, компилятор пытается разместить данное этот класса в регистре процессора, если в данный момент имеются свободные регистры. Спецификатор auto принимается по умолчанию и поэтому явно не указывается, он означает, что данное класса auto должно размещаться в программном стеке при вызове функции.

Спецификатор typedef служит для присвоения имени описываемому типу данного и будет рассмотрен подробнее в следующем параграфе.

Наряду с показанными выше константами-литералами, значения которых определяются их представлением в программе, в Си и Си++ предусмотрены константы, которым присваиваются собственные имена - именованные константы. В описании именованной константы присутствует описатель const, например,

```
const double Pi  = 3.141592653;
```

Переменной, идентификатор которой объявлен с описателем const, нельзя присвоить иное значение, чем было установлено при объявлении идентификатора. Инициализирующее значение при объявлении константы является обязательным.

Наряду с базовыми целыми и вещественными типами различных размеров в программе могут объявляться и использоваться данные типов, определяемых программистом: указатели, ссылки, агрегаты данных и данные перечислимого типа.

2 Перечислимый тип

Перечислимый тип вводится ключевым словом **enum** и задает набор значений, определяемый пользователем. Набор значений заключается в фигурные скобки и является набором целых именованных констант, представленных своими идентификаторами. Эти константы называются **перечислимыми константами**. Рассмотрим объявление:

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

С его помощью создается целочисленный тип набором из четырех названий мастей, именующих целочисленные константы. Перечислимые константы - это идентификаторы CLUBS, DIAMONDS, HEARTS и SPADES, имеющие значения - 0, 1, 2 и 3, соответственно. Эти значения присвоены по умолчанию. Первой перечислимой константе присваивают постоянное целое численное значение 0. Каждый последующий член списка на единицу больше, чем его сосед слева. Переменным типа Suit, определенного пользователем, может быть присвоено только одно из четырех значений, объявленных в перечислении.

Другой популярный пример перечислимого типа:

```
enum Months {JAN = 1, FEB, MAR, APR, MAY, YUN, YUL AUG, SEP, OCT, NOV, DEC};
```

Это объявление создает определенный пользователем тип Months с константами перечисления, представляющими месяцы года. Поскольку первое значение приведенного перечисления установлено равным 1, оставшиеся значения увеличиваются на 1 от 1 до 12.

В объявлении перечислимого типа любой константе перечисления можно присвоить целое значение. После того, как константа перечисления определена, попытка присвоить ей другое значение является синтаксической ошибкой.

Использование перечислений вместо целых констант облегчает чтение программы.

Идентификаторы в **enum** должны быть уникальными, но отдельные константы перечисления могут иметь одинаковые целые значения.

Перечислимые константы могут определяться произвольными целочисленными константами, а также константными выражениями:

```
enum ages (milton = 47, ira, harold = 56, philip = harold + 7};
```

Когда нет явного инициализатора, применяется правило по умолчанию, таким образом - ira = 48.

Каждое перечисление является отдельным типом. Типом элемента перечисления является само перечисление. Например, в

```
enum Keyword {ASM, AUTO, BREAK};
```

AUTO имеет тип **Keyword**.

3 Указатели и ссылки

Указатель — предназначен для хранения адреса некоторого объекта (например, переменной) определённого типа.

Общая схема объявления указателя:

```
type* name; // объявили указатель с именем name на объект типа type
type *another; // объявили указатель на объект того же типа, звёздочку можно ставить и перед именем указателя
```

Примеры:

```
int* p;
int *p2;
double* pd;
```

Указателям `p` и `p2` можно будет присвоить адреса переменных типа `int`, но нельзя будет присвоить адреса переменных другого типа или объектов какого-нибудь класса. Аналогично, указателю `pd` можно будет присвоить **только** адреса переменных типа `double`.

Для взятия адреса — используется оператор `&`, размещаемый перед объектом, адрес которого хочется получить.

Примеры:

```
int a = 15;
cout << &a << endl; // вывели адрес переменной a в памяти (не её значение), увидим шестнадцатиричное число
int ar[] = {728, 3, 402, -1};
/*
  Далее выведем адреса элементов массива. Они будут отличаться на размер в байтах
  базового типа массива (в данном случае, int). Ещё раз убедимся, что в памяти
  все элементы массива расположены последовательно друг за другом.
*/
for (int i=0; i<=3; i++) {
  cout << &ar[i] << ' ';
}
cout lt;< endl;
int* p;
p = &a; // скопировали адрес переменной a в указатель p
cout << p << endl; // вывели адрес, хранимый в указателе (совпадёт с ранее виденным адресом)
cout << &p << endl; // вывели адрес самого указателя (он же тоже хранится где-то в памяти, потому имеет адрес)
```

Для перехода по известному адресу — используется оператор `*`, размещаемый перед адресом или указателем хранящем адрес. Под переходом по адресу понимается, что от адреса мы переходим к действиям над значением, хранимом по данному адресу. Это операция называется разыменованием.

```
int a = 15; // переменная a со значением
int* p = &a; // указатель с адресом переменной a
cout << *p << endl; // увидим 15, т.е. значение переменной
```

Получается, что к любой переменной мы можем обратиться как по её имени, так и по её адресу, применив к нему операцию разыменования.

Справедливо тождество: выражение `a == *(&a)`, где a любого типа — всегда истинно.

Над множеством указателей в C++ определён ряд операций:

- p+n, где p — указатель, n — целое положительное число. Результат — некоторый указатель, полученный смещением p на n позиций вправо.
- p-n, где p — указатель, n — целое положительное число. Результат — некоторый указатель, полученный смещением p на n позиций влево.
- p-q, где p и q — указатели на один и тот же тип. Результат — целое число, равное количеству шагов, на которое нужно сместить q вправо, чтобы он достиг указателя p, также этот результат можно называть “расстоянием” между указателями, оно может быть и отрицательным, если элемент, на который направлен указатель q расположен правее (то есть, далее), чем элемент, на который направлен указатель p.
- p++ (инкремент), p-- (декремент), где p — указатель. p=p+1, p=p-1 соответственно.

Кроме того, указатели можно сравнивать с помощью операторов сравнения, которые рассматривались для числовых типов. При этом, большим считается тот указатель, который направлен на элемент, расположенный в памяти далее (т. е. правее). Указатель хранит адрес элемента, то есть некоторое целое положительное число. Соответственно, больше будет тот элемент, которых хранит численно больший адрес. Равными считаются указатели, направленные на один и тот же элемент.

Константный указатель нельзя перемещать (записывать в него другой адрес), но можно его разыменовывать или делать его участников вышеперечисленных операций:

```
int ar[] = {-72, 3, 402, -1, 55, 132};
cout << *ar; // -72
int* p = ar+3; // указатель на 4-ый по счёту элемент массива (со значением -
1)
p--; // переместили указатель влево на 1 элемент
cout << *p; // выведется 402
```

Кроме обычных переменных и указателей в Си++ имеется тип "ссылка на переменную", задающий для переменной дополнительное имя (псевдоним). Внутреннее представление ссылки такое же, как указателя, т.е. в виде адреса переменной, но обращение к переменной по ссылке записывается в той же форме, что и обращение по основному имени. Переменная типа ссылки всегда инициализируется заданием имени переменной, к которой относится ссылка. Объявляется так:

тип &**имя_ссылки** = **имя_переменной**;

тип ссылки и переменной должны быть одинаковыми.

Пример:

```
int main()
{
int a = 5; //объявляем переменную
```

```
int &p = a; /объявляем ссылку. теперь p это псевдоним a
cout << a << ' ' << p << endl; //выведет 5 5
a = 6;
cout << a << ' ' << p << endl; //выведет 6 6
p = 7;
cout << a << ' ' << p << endl; //выведет 7 7
return 0;
}
```

Ссылку можно создать как аргумент в функции, чтобы из функции напрямую работать с переменной, которую в неё передали.

Пример:

```
void func(int &p)
{
p++; //увеличиваем переданную переменную
}

int main()
{
int a = 5; //объявляем переменную
func(a); //передаём переменную.
cout << a << endl; //покажет 6
return 0;
}
```

Также можно создать ссылку на функцию и вызвать её:

```
void func(int s)
{
cout << s << endl;
}

int main()
{
void (&rf)(int) = func;
func(123); //покажет 123
rf(321); //покажет 321
return 0;
}
```

4 Массивы

Из переменных любого типа в Си++ могут образовываться массивы. При объявлении массива за идентификатором массива задается число элементов массива в квадратных скобках:

```
int a [ 5 ] ; // Массив из пяти элементов типа int
```

Индексы элементов массива всегда начинаются с 0, индекс последнего элемента на единицу меньше числа элементов в массиве. Массив может инициализироваться списком значений в фигурных скобках:

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

При наличии списка инициализации, охватывающего все элементы массива, можно не указывать число элементов массива, оно будет определено компилятором:

```
int  c [ ] = { 1, 2, 3 }; //  Массив из трех элементов типа int
```

Пример программы, которая помещает в массив числа от 0 до 9:

```
#include <iostream>
using spacename std;
int main ()
{
 int sample[10]; //эта ин-ия резервирует обл. памяти для 10 эл-в типа int
 int t;
  //помещаем в массив значения.
 for (t=0; t<10; ++t) sample [t]=t;
  //отображаем массив.
 for (t=0; t<10; ++t)cout <<sample[t]<<` `;
 return 0;
}
```

В Си++ все массивы занимают сложные ячейки памяти, другими словами, элементы массива в памяти расположены пос-но друг за другом. Ячейка с наименьшим адресом относится к первому элементу массива, а с наибольшим - к последнему. Например, после выполнения этого фрагмента кода:

```
int i[7];
int j;
for (j=0; j<7; j++) i[j]=j;
массив i будет выглядеть след. Образом:

 i[0] i[1] i[2] i[3] i[4] i[5] i[6]
  0     1    2    3    4    5    6
```

Массивы с размерностью 2 и более рассматриваются как массивы массивов и для каждого измерения указывается число элементов:

```
double  aa [ 2 ] [ 2 ] = { 1, 2, 3, 4 }; // Матрица 2 * 2
```

Массивы типа char могут инициализироваться строковым литералом. Строковый литерал - это последовательность любых символов, кроме кавычек и обратной косой черты, заключенная в кавычки. Если строковый литерал не умещается на одной строке, его можно прервать символом "\" и продолжить с начала следующей строки. В стандарте С++ предусмотрена и другая возможность записи длинных литералов в виде нескольких записанных подряд строковых литералов. Если между строковыми литералами нет символов, отличных от пробелов, такие литералы сливаются компилятором в один.

При размещении в памяти в конце строкового литерала добавляется символ '\0', т.е. нулевой байт. Строковый литерал может применяться и для инициализации указателя на тип char:

```
char   str1 [ 11 ] = "Это строка",
            str2 [ ] = " Размер этого массива определяется"
                       " числом знаков в литерале + 1";
char *pstr = "Указатель с инициализацией строкой";
```

Имя массива в Си/Си++ является указателем-константой, ссылающимся на первый элемент массива, имеющий индекс, равный нулю. Для обращения к элементу массива указывается идентификатор массива и индекс элемента в круглых скобках, например, c[2], aa[0][1].

5 Динамическое выделение и освобождение памяти

В Си++ определены операции размещения данных в динамической памяти и удаления динамических данных из памяти.

Операция new требует в качестве операнда имени типа и предназначена для размещения данного указанного типа в динамической памяти, результатом операции будет указатель на данное. При невозможности выделить память операция new возвращает значение NULL - предопределенную константу, имеющую нулевое значение. Память, выделяемую операцией new, можно инициализировать, указав за именем типа скалярного данного начальное значение в круглых скобках. Примеры применения операции new :

```
int *ip = new int; /* создание объекта типа int и получение указателя на него
*/
int *ip2 = new int(2); // то же с установкой начального значения 2
inr *intArray = new int [ 10 ]; // массив из 10 элементов типа int
double **matr = new double [ m ] [ n ]; // матрица из m строк и n столбцов
```

Данное, размещенное в динамической памяти операцией new, удаляется из памяти операцией delete с операндом-указателем, значение которого получено операцией new, например,

```
delete intArray;  delete ip2;
```

Операция delete только освобождает динамическую память, но не изменяет значение указателя-операнда. После освобождения памяти использовать этот указатель для обращения к данному нельзя.

Размер данного или типа данного в байтах можно получить по операции sizeof. Операнд может быть любого типа, кроме типа функции и битового поля. Если операндом является имя типа, оно должно заключаться в скобки. Возвращаемое значение имеет предопределенный тип size_t, это целый тип, размер которого определяется реализацией компилятора, обычно типу size_t соответствует unsigned int. Размер массива равен числу байт, занимаемых массивом в памяти, размер строкового литерала - это число знаков в литерале +1, т.е. завершающий нулевой байт учитывается при определении длины литерала. Значение, возвращаемое sizeof является константой.

Работа с ссылками:

```
void swap(double &a, double &b)

{
  double tmp = a;
  a = b;
  b = tmp;
}
```

(`double &` - ссылка на тип `double`) - ссылка. При таком объявлении функции при вызове, например, на стек, положатся не значения переменных, а их адреса.
Ссылка - это тот же указатель, который воспринимается и используется как обычная переменная. Отличия в том, что ссылка не может быть равна NULL или не инициализирована.

6 Взаимосвязь массиовов и указателей

Указатели в C++ и их связь с массивами

При создании любого массива в C++, вместе с ним естественным образом создаётся указатель. Имя этого указателя совпадает с именем массива. Тип этого указателя — «указатель на базовый тип массива». В появившемся указателе хранится адрес начального элемента массива. Чтобы начало массива не было потеряно этот указатель является **константным**, т.е. его нельзя направить на какой-то другой элемент массива или записать туда адрес другой переменной даже подходящего типа. Но зато можно скопировать этот адрес в какой-то другой указатель, не являющимся константным.

С учётом того, что в массиве все элементы располагаются в памяти последовательно, начав с указателя, направленного на начальный элемент, мы сможем обойти все элементы массива, смещая указатель на каждом шаге вправо на минимально возможную дистанцию (то есть, на соседний справа элемент подходящего типа). Смещение указателя может производиться с помощью операторов инкремента и декремента.

```
int ar[] = {-72, 3, 402, -1, 55, 132};
int* p = ar;
for (int i=101; i<=106; i++) {
 cout << *p << ' ';
 p++;
```

}- здесь все элементы массива будут выведены на экран без использования индексов (обратите внимание, что при этом параметры цикла могут быть любыми, лишь бы цикл выполнился нужное количество раз)

О взаимозаменяемости указателей и массивов

Во многих случаях они взаимозаменяемы. Например, с помощью указателя, который содержит адрес начала массива, можно получить доступ к элементам этого массива либо посредством арифметических действий над указателем, либо посредством индексирования массива. Однако в общем случае указатели и массивы не являются взаимозаменяемыми.

```
int num[10];
int i;
for(i=0; i<10; i++)
{
*num = i; // Здесь все в порядке.
num++; // ОШИБКА – переменную num модифицировать нельзя.
}
```

Здесь используется массив целочисленных значений с именем num. Как отмечено в комментарии, несмотря на то, что совершенно приемлемо применить к имени num оператор "*" (который обычно применяется к указателям), абсолютно недопустимо изменять значение num, т.к. num — это константа, которая указывает на начало массива. Другими словами, несмотря на то, что имя массива (без индекса) действительно генерирует указатель на начало массива, его значение изменению не подлежит.

Хотя имя массива генерирует константу-указатель, его, тем не менее, (подобно указателям) можно включать в выражения, если, только оно при этом не изменяется. Например следующая инструкция, при выполнении которой элементу num[3] присваивается значение 100, вполне допустима.

```
*(num+3) = 100; // Здесь все в порядке, поскольку num не изменяется.
```

7 Символьные массивы и строки

Чаще всего одномерные массивы используются для создания символьных строк. В C++ строка определяется как символьный массив, который завершается нулевым символом ('\0'). При определении длины символьного массива необходимо учитывать признак ее завершения и задавать его длину на единицу больше длины самой большой строки из тех, которые предполагается хранить в этом массиве.
Строка — это символьный массив, который завершается нулевым символом.
Например, объявляя массив str, предназначенный для хранения
10 -символьной строки, необходимо писать **`char str [11];`**
Заданный здесь размер (11) позволяет зарезервировать место для нулевого символа в конце строки. C++ позволяет определять строковые литералы. Строковый литерал — это список символов, заключенный в двойные кавычки : `"Привет","и т.д.`
Строка, приведенная последней (""), называется нулевой. Она состоит только из одного нулевого символа (признака завершения строки). Нулевые строки используются для представления пустых строк. Следовательно, строка "ПРИВЕТ" в памяти размещается следующим образом: П Р И В Е Т `\0`

Считывание строк с клавиатуры
Проще всего считать строку с клавиатуры, создав массив, который примет эту строку с помощью инструкции cin. Считывание строки, введенной с клавиатуры, отображено в следующей программе.

```
// Использование cin-инструкции для считывания строки с клавиатуры.
#include <iostream>
using namespace std;
int main()
{
char str[80];
cout << "Введите строку: ";
cin >> str; // Считываем строку с клавиатуры.
cout << "Вот ваша строка: ";
cout << str;
return 0;
}
```

Однако при выводе строки, введенной с клавиатуры, программа отобразит только слово "Это", а не всю строку, поскольку оператор ">>" прекращает считывание строки, как только встречает символ пробела, табуляции или новой строки (пробельные символы). Для решения этой проблемы можно использовать библиотечную функцию **gets().** Общий формат ее вызова таков-
`gets(имя_массива);`
Если в программе необходимо считать строку с клавиатуры, можно вызвать функцию **gets()**, а в качестве аргумента передать имя массива, не указывая индекса. После выполнения этой функции заданный массив будет содержать текст, введенный с клавиатуры. Функция gets() считывает вводимые символы до тех пор, пока мы не нажмем клавишу <Enter>. Для вызова функции gets()
в программу необходимо включить заголовок <cstdio>.
Тогда мы сможем ввести в массив строку символов, содержащую пробелы.

```
// Использование функции gets() для считывания строки с клавиатуры.
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
```

```
char str[80];
cout << "Введите строку: ";
gets(str); // Считываем строку с клавиатуры.
cout << "Вот ваша строка: ";
cout << str;
return 0;
}
```

На этот раз после запуска этой версии программы и ввода с клавиатуры текста "Это простой тест" строка считывается полностью, а затем так же полностью и отображается.

```
Введите строку: Это простой тест
Вот ваша строка: Это простой тест
```

`cout << str;` -здесь (вместо привычного литерала) используется имя строкового массива. Имя символьного массива, который содержит строку, можно использовать везде, где допустимо применение строкового литерала. Однако ни оператор ">>", ни функция gets() не выполняют граничной проверки (на отсутствие нарушения границ массива).

.

8 Структуры и объединения

Наряду с массивами в Си++ имеются агрегаты данных типа структур и объединений. Тип структуры представляет собой упорядоченную совокупность данных различных типов, к которой можно обращаться как к единому данному. Описание структурного типа строится по схеме:

struct идентификатор

{ декларатоpы членов } деклараторы_инициализаторы;

Такое объявление выполняет две функции, во-первых объявляется структурный тип, во-вторых объявляются переменные этого типа.

Идентификатор после ключевого слова struct является именем структурного типа. Имя типа может отсутствовать, тогда тип будет безымянный и в других частях программы нельзя будет объявлять данные этого типа. Деклараторы_инициализаторы объявляют конкретные переменные структурного типа, т.е. данные описанного типа, указатели на этот тип и массивы данных. Деклараторы_инициализаторы могут отсутствовать, в этом случае объявление описывает только тип структуры.

Структура, описывающая точку на плоскости, может быть определена так:

```
struct Point_struct    //  Имя структуры
   { int x,  y; }         //  Декларатоpы членов структуры
         point1, *ptr_to_point, arpoint [3];   // Данные структурного типа
```

Члены (компоненты) структуры описываются аналогично данным соответствующего типа и могут быть скалярными данными, указателями, массивами или данными другого структурного типа. Например, для описания структурного типа "прямоугольник со сторонами, параллельными осям координат" можно предложить несколько вариантов:

```
struct Rect1
   { Point  p1; // Координаты левого верхнего угла
     Point  p2 ; // Координаты правого нижнего угла
   } ;

struct Rect2
   { Point p [ 2 ];
   } ;

struct Rect3
   { Point  p;  // Левый верхний угол
     int  width;  // Ширина
     int  high;    // Высота прямоугольника
   } ;
```

Поскольку при описании членов структуры должны использоваться только ранее определенные имена типов, можно предварительно объявить структуру, задающую только имя структурного типа. Например, чтобы описать элемент двоичного дерева, содержащий указатели на левую и правую ветви дерева и указатель на некоторую структуру типа Value, содержащую значение данного в узле, можно поступить так:

```
struct   Value;
struct Tree_element
  { Value  * val;
```

```
    Tree_element *left, *right;
  };
```

Объединение можно определить как структуру, все компоненты которой размещаются в памяти с одного и того же адреса. Таким образом, объединение в каждый момент времени содержит один из возможных вариантов значений. Для размещения объединения в памяти выделяется участок, достаточный для размещения члена объединения самого большого размера. Применение объединения также позволяет обращаться к одному и тому же полю памяти по разным именам и интерпретировать как значения разных типов.

Описание объединения строится по той же схеме, что и описание структуры, но вместо ключевого слова struct используется слово union, например, объединение uword позволяет интерпретировать поле памяти либо как unsigned int, либо как массив из двух элементов типа unsigned char.

```
union uword
  { unsigned int u;
    unsigned char b [ 2 ];
  }
```

Описания объявляемых типов, в том числе структур и объединений могут быть достаточно большими, поэтому в Си++ предусмотрена возможность присваивания типам собственных имен (синонимов. Синоним имени типа вводится с ключевым словом typedef и строится как обычное объявление, но идентификаторы в деклараторах в этом случае интерпретируются как синонимы описанных типов. Ниже приведено несколько примеров объявления синонимов имен типов.

```
typedef  struct { double re, im } COMPLEX;
typedef  int  *PINT;
```

После таких объявлений синоним имени может использоваться как спецификатор типа:

```
COMPLEX  ca, *pca;  // переменная типа COMPLEX и указатель на COMPLEX
PINT  pi;  // указатель на int
```

В Си++ структуры и объединения являются частными случаями объектных типов данных.

9 Арифметические операции, операции отношения, логические операции

Операции и выражения

В языке Си++ определен обширный набор операций над данными. Операции служат для построения выражений. Выражение представляет собой последовательность операндов и знаков операций и служит для вычисления некоторого значения.

В качестве операндов в выражении выступают идентификаторы переменных, константы, и строковые литералы (первичные выражения). Каждая операция предполагает использование определенных типов операндов (целых, вещественных, указателей). Операция присваивания в Си++ также является выражением, в связи с этим различаются операнды, которым можно присвоить новое значение и операнды, значение которых не может меняться. Чтобы операнду можно было присвоить значение, ему должна соответствовать область памяти и компилятору должен быть известен адрес этой памяти. Такие операнды называют L-выражениями (left), так как они могут быть записаны в левой части оператора присваивания.

Двуместные **арифметические операции** умножения (*), деления (/), получения остатка от деления нацело (%), сложения (+) и вычитания (-) имеют обычный смысл и обычный относительный приоритет. Если операнды арифметической операции имеют разные типы, предварительно выполняются стандартные арифметические преобразования и тип результата операции определяется общим типом операндов после стандартных преобразований. Таким образом, выражение 7/2 будет иметь значение 3 типа int, так как оба опернда имеют тип int, а выражение 7.0/2 даст результат 3.5 типа double, поскольку этот тип имеет первый операнд.

Операции "одноместный +" и "одноместный -" имеют обычный математический смысл, знак + не изменяет значения операнда, знак - меняет знак операнда на противоположный.

Операции отношения двух выражений (<, <=, >, >=) требуют операндов арифметического типа или же оба операнда должны быть указателями на одинаковый тип. В случае операндов арифметического типа вычисляются значения операндов, выполняются стандартные арифметические преобразования и возвращается 1 типа int, если отношение выполняется (истинно), или 0, если отношение не выполняется (ложно). Когда сравниваются два указателя, результат зависит от относительного размещения в памяти объектов, на которые ссылаются указатели. Операции сравнения (== и !=) выполняются аналогичным образом, но имеют меньший приоритет.

Выражения отношений могут соединяться логическими связками && (конъюнкция, логическое умножение) и | | (дизъюнкция, логическое сложение). Операндами логических связок могут быть любые скалярные значения и операция && дает реззультат, равный 1 типа int, если оба операнда имеют ненулевые значения, а операция | | дает результат, равный 0, если значения обоих операндов нулевые. Применяется сокращенная форма вычисления значения логических связок, если в операции && первый операнд равен нулю, то второй операнд не вычисляется и возвращается 0, если в операции | | первый операнд не равен нулю, то второй операнд не вычисляется и возвращается значение 1.

Логическое отрицание (унарная операция !) возвращает значение 0 целого типа, если операнд не равен нулю, или значение 1, если операнд равен нулю.

Над целыми операндами допустимы операции поразрядного логического умножения, логического сложения и исключающего или. В этих операциях операнды рассматриваются как последовательности битов и операция выполняется над каждой парой соответствующих разрядов из обоих операндов. Например, результатом выражения (x >> (p - n +1)) & (~(~0 << n)) будет выделение из целого беззнакового x n битов, начиная с бита с номером p, и сдвиг выделенных битов вправо, т.е. выделение n-разрядного целого, хранящегося в машинном слове x начиная с p-го разряда.

10 Автоувеличение-автоуменьшение, составные операции, приоритет

Операции и выражения

В языке Си++ определен обширный набор операций над данными. Операции служат для построения выражений. Выражение представляет собой последовательность операндов и знаков операций и служит для вычисления некоторого значения.

В качестве операндов в выражении выступают идентификаторы переменных, константы, и строковые литералы (первичные выражения). Каждая операция предполагает использование определенных типов операндов (целых, вещественных, указателей). Операция присваивания в Си++ также является выражением, в связи с этим различаются операнды, которым можно присвоить новое значение и операнды, значение которых не может меняться. Чтобы операнду можно было присвоить значение, ему должна соответствовать область памяти и компилятору должен быть известен адрес этой памяти. Такие операнды называют L-выражениями (left), так как они могут быть записаны в левой части оператора присваивания.

Операции автоувеличения и автоуменьшения (++ и --) могут быть префиксными и постфиксными и вызывают увеличение (уменьшение) своего операнда на единицу, т.е. выражение ++x эквивалентно x = x +1, а --x эквивалентно x = x - 1. Префиксная операция выполняется до того, как ее операнд будет использован в вычислении выражения, а постфиксная операция выполняется после того, как ее операнд будет использован в выражении, например, в результате вычисления выражения

++x * 2 + y-- *3

переменная x сначала увеличивается на 1, а затем умножается на 2, переменная y сначала умножается на 3, затем уменьшается на 1. Если перед вычислением этого выражения x и y были равны 1, то результат выражения будет равен 5, кроме того переменная x получит значение 2, а переменная y - значение 0. Операции автоувеличения и автоуменьшения применимы только к целым числам и всегда изменяют значения своих операндов. Операнды этих операций должны быть L-значениями.

Присваивание, обозначаемое знаком = в Си++ рассматривается как операция и возвращает значение, которое было присвоено левому операнду. Операция присваивания вычисляется справа налево, т.е. сначала вычисляется присваиваемое значение, затем выполняется присваивание. Это позволяет записывать выражения вида X = Y = Z = 1 для установки одинаковых значений нескольким переменным. Кроме простого присваивания имеется набор **составных операций присваивания**, в которых присваивание совмещается с указанной двуместной операцией. Они присваивают первому операнду результат применения соответствующей простой операции, указанной перед символом "=", к первому и второму операндам. Например, выражение X += Y эквивалентно выражению X = X + Y, но записывается компактнее и может выполняться быстрее. Аналогично определяются и другие операции присваивания: X %= Y эквивалентно X = X % Y и т.д. При записи составных операций присваивания между символом операции и знаком равенства пробел не допускается. В отличие от операций автоувеличения/автоуменьшения, они применимы и к вещественным числам.

В Си++ имеется конструкция, которая называется условным выражением. Условное выражение строится по схеме:

условие ? выражение1 : выражение2

В качестве условия может выступать любое скалярное выражение. Если результат вычисления условия ненулевой, то значением всего выражения будет выражение1, при нулевом значении условия значение всего выражения определяется выражением2. Второй и третий операнды условного выражения должны быть либо оба арифметического типа, либо однотипными структурами или объединениями, либо указателями одинакового типа, либо один из них - указатель на какой-либо тип, а другой операнд NULL или имеет тип void*. Выражение x > 0 ? 1 : 0 возвращает 1, если x больше 0, и 0 в противном случае.

Операция ": :" (два двоеточия) применяется для уточнения имени объекта программы в случае, когда в этом месте программы известны два одинаковом имени, например, когда одно имя объявлено глобально, а другое в теле функции. Если имени предшествуют два двоеточия, то это глобальное имя. Она имеет самый высокий приоритет, наряду с такими операциями, как обращение к члену структуры по указателю на структуру (указатель -> имя_члена_структуры) или по имени на структуру (имя_структуры.имя_члена_структуры), обращение к элементу массива (указатель [индекс]), преобразование типа данного и вызов функции. Далее следуют операции автоув.(++)/автоум.(--), битовое инвертирование (~ целое_выражение), логическое отрицание (!), одноместные +/- , получение адреса (&) и разыменование указателя (*), выделение динамической памяти (new), освобождение памяти (delete указатель) и размер данного (sizeof выражение). Далее идут *, / ,%, +, -, <<, >>, <, <=, >, >= , ==, !=, поразрядная конъюнкция (&) и дизъюнкция (I), логические И (&&) и ИЛИ (| |), условное выражение (?:), простое присваивание (=) , составное присваивание (@=) и, наконец, операция следования (выражение , выражение).

11 Механизм явного и неявного приведения типов

Поскольку Си++ является типизированным языком, в нем определены явные и неявные преобразования типов данных. **Неявные преобразования** выполняются при двуместных арифметических операциях и операции присваивания и называются стандартными арифметическими преобразованиями. Эти преобразования выполняются в следующей последовательности:

- если один операнд имеет тип long double, другой операнд преобразуется в тип long double;

- иначе, если один операнд имеет тип double, другой операнд преобразуется в тип double;

- иначе, если один операнд имеет тип float, другой операнд преобразуется в тип float;

- иначе, если один операнд имеет тип unsigned long int, другой операнд преобразуется в тип unsigned long int;

- иначе, если один операнд имеет тип long int, другой операнд преобразуется в тип long int;

- иначе, выполняются стандартные преобразования для целых, при этом типы char, short int и битовые поля типа int преобразуются в тип int, затем, если один операнд имеет больший размер (больший диапазон значений), чем другой операнд, то второй операнд преобразуется к типу операнда большего размера;

- в остальных случаях операнды имеют тип int.

Пример:

```
int x = 5;
double y = 15.3;
y = x; //здесь происходит неявное приведение типа к double
x = y; //здесь происходит неявное приведение типа к int
```

Явное преобразование типов может быть задано в двух формах. В первом случае за именем типа в круглых скобках записывается преобразуемое значение, которое может быть первичным выражением или выражением с одноместной операцией. Имя типа в этом случае может быть представлено последовательностью описателей, например, (long int *) pp определеяет преобразование некоторого данного pp в тип указателя на long int.

Вторая форма преобразования типа записывается как вызов функции, при этом имя типа должно задаваться идентификатором, например, int (x).

Пример:

```
int x = 5;
double y = 15.3;
```

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

Провести вещественное деление над целыми числами можно только при помощи явного преобразования типов.

12 Операторы присваивания, условные операторы

Операторы Си++

Операторы - это синтаксические конструкции, определяющие действия, выполняемые программой. Синтаксис некоторых операторов содержит выражения, играющие роль условий, в зависимости от выполнения или невыполнения которых выбирается та или иная последовательность действий. В качестве условий используются любые выражения, дающие скалярные значения, и условие считается выполненным, если это значение отлично от нуля, и невыполненным, если оно равно нулю. Несколько операторов могут быть объединены в составной оператор заключением их в фигурные (операторные) скобки. Признаком конца оператора (кроме составного оператора) служит точка с запятой, являющаяся в этом случае частью оператора.

Перед любым оператором может быть записана метка в виде идентификатора, отделенного от помечаемого оператора двоеточием. Метка служит только для указания ее в операторе перехода.

Наиболее простым является оператор-выражение, представляющий собой полное выражение, заканчивающееся точкой с запятой, например,

x = 3; y = (x +1) * t; i++;

Выражение, оформленное как оператор, вычисляется, но его значение теряется и действие оператора-выражения состоит в побочных эффектах, сопровождающих вычисление, например, при выполнении операций присваивания, автоувеличения и автоуменьшения.

Операторы выбора в Си++ представлены **условным оператором** и переключателем. Условный оператор может использоваться в сокращенной и полной формах, которым соответствуют схемы:

if (выражение-условие) оператор

if (выражение-условие) оператор-1 else оператор-2

В сокращенной форме условного оператора вычисляется выражение-условие и, если его значение отлично от нуля, выполняется следующий за условием оператор, в противном случае не производится никаких действий.

В полной форме условного оператора при ненулевом значении выражения-условия выполняется оператор-1 с последующим переходом к следующему оператору программы, а при нулевом значении выражения условия выполняется оператор-2 с переходом к следующему оператору программы.

Переключатель позволяет выбрать одну из нескольких возможных ветвей вычислений и строится по схеме:

switch (целое выражение) оператор
Оператор в этом случае представляет собой тело переключателя, практически всегда является составным и имеет такой вид:

{ case константа-1 : операторы

case константа-2 : операторы

.....

default : операторы

}

Выполнение переключателя состоит в вычислении управляющего выражения и переходе к группе операторов, помеченных case-меткой, равной управляющему выражению, если такой case-метки нет, выполняются операторы по метке default. Пункт default может отсутствовать и тогда, если управляющему выражению не соответствуют ни одна case-метка, весь переключатель эквивалентен пустому оператору. При выполнении переключателя происходит переход на оператор с выбранной case-меткой и дальше операторы выполняются в естественном порядке. Например, в переключателе

```
switch  (count)
  {  case  1 :   x=1;
     case  2 :   x=2;
     case  3 :   x=3;
     default :   x=4;
  }
```

если значение count равно 1, то после перехода на case 1 будут выполнены все операторы, в результате x станет равным 4. Чтобы разделить ветви переключателя, в конце каждой ветви нужно записать оператор break, не имеющий операндов. По этому оператору происходит выход из переключателя к следующему оператору программы:

```
switch  (count)
   {  case 1  :   x = 1;  break;
      case 2  :   x = 2;  break;
      case 3  :   x = 3;  break;
      default :   x = 4;
}
```

Теперь в зависимости от значения count будет выполняться только одна ветвь переключателя и x будет принимать одно из четырех предусмотренных значений.

13 Операторы цикла

В Си++ имеется три варианта оператора цикла: цикл с предусловием, цикл с постусловием и цикл с параметром.

Цикл с предусловием строится по схеме

while (выражение-условие) оператор
При каждом повторении цикла вычисляется выражение-условие и если значение этого выражения не равно нулю, выполняется оператор - тело цикла.

Цикл с постусловием строится по схеме:

do оператор while (выражение-условие)
Выражение-условие вычисляется и проверяется после каждого повторения оператора - тела цикла, цикл повторяется, пока условие выполняется. Тело цикла в цикле с постусловием выполняется хотя бы один раз.

Цикл с параметром строится по схеме:

for (E1; E2; E3) оператор

где E1, E2 и E3 - выражения скалярного типа. Цикл с параметром реализуется по следующему алгоритму:

1. Вычисляется выражение E1. Обычно это выражение выполняет подготовку к началу цикла.

2. Вычисляется выражение E2 и если оно равно нулю выполняется переход к следующему оператору программы (выход из цикла). Если E2 не равно нулю, выполняется шаг 3.

3. Выполняется оператор - тело цикла.

4. Вычисляется выражение E3 - выполняется подготовка к повторению цикла, после чего снова выполняется шаг 2.

Пусть требуется подсчитать сумму элементов некоторого массива из n элементов.

С использованием цикла с предусловием это можно сделать так:

```
int s=0;
int i=0;
while (i < n) s +=a[ i ++];
```

Эта же задача с применением цикла с постусловием решается следующими операторами:

```
int s = 0;
int i = 0;
do s +=a[ i++]; while ( i < n );
```

Поскольку в данном случае повторениями цикла управляет параметр i, эту задачу можно решить и с помощью цикла третьего типа:

```
int  i, s;
for ( s = 0, i = 0; i < n; i++) s +=a[ i ];
```

Объявления переменных можно внести в выражение E1 оператора for и все записать в виде одного оператора for с пустым телом цикла:

```
for ( int i = 0, s = 0; i < n; s += a [i++] ) ;
```

Для прерывания повторений оператора цикла любого типа в теле цикла может быть использован оператор break. Для перехода к следующему повторению цикла из любого места тела цикла может быть применен оператор continue. Эти операторы по своему назаначению аналогичны соответствующим операторам языка Паскаль.

Несмотря на то, что Си++ содержит полный набор операторов для структурного программирования, в него также включен оператор перехода:

goto метка ;

Метка задает адрес перехода и должна помечать оператор в том же составном операторе, которому принадлежит оператор goto. Вход в составной оператор, содержащий объявления данных, не через его начало, запрещен.

14 Функции. Описание, входные параметры

Любая программа на Си++ содержит хотя бы одну функцию. Алгоритм решения любой задачи реализуется путем вызовов функций. Одна из функций считается главной и имеет фиксированное имя, эта функция вызывается операционной системой при запуске программы, а из нее могут вызываться другие функции.

Описание функции имеет общий синтаксис внешнего определения и отличается от описания переменного синтаксисом декларатора-инициализатора, который содержит функции и список параметров в круглых скобках. Тип функции - это тип значения, возвращаемого функцией. Функция может возвращать значение любого типа, кроме массива и функции, но допускается возвращать указатель на массив или указатель на функцию. Если функция не возвращает никакого значения, тип функции обозначается ключевым словом void.

В списке параметров указываются типы и имена параметров, передаваемых в функцию при ее вызове. Если функция не требует передачи ей аргументов, список параметров может быть пустым.

Полное описание функции содержит также тело функции, представляющее собой составной оператор (последовательность описаний внутренних данных и операторов, заключенная в фигурные скобки). В Си++ описание функций не может быть вложенным, в теле функции нельзя объявить другую функцию.

Функция, возвращающая среднее арифметическое трех вещественных данных, может быть описана так:

```
double sred ( double x, double y, double z)
   { double s;
     s = x + y + z;
     return  s / 3;
   }
```

Для вызова такой функции при условии, что предварительно объявлены переменные p, a, b, и c, можно записать оператор:

```
p = sred (a, b, c );
```

функция должна быть описана до того, как встретится ее первый вызов (две функции не могут вызывать друг друга). Когда программа состоит из нескольких файлов, полное описание функции должно присутствовать только в одном файле, но вызовы функции возможны из разных файлов программы. Есть две формы описания функций, полная форма (определением функции), и сокращенная (описание прототипа функции или просто прототип). Прототип функции содержит только заголовок функции и задает имя функции, тип возвращаемого значения и типы параметров. По этой информации при компиляции можно проверить правильность записи вызова функции и использования возвращаемого значения. Требуется, чтобы вызову любой функции предшествовало в том же файле либо полное определение функции, либо описание ее прототипа.

- При описании прототипа функции можно не указывать имена параметров, достаточно указать их типы.

- Для части параметров функции можно задавать значение по умолчанию, что позволяет вызывать функцию с меньшим числом аргументов, чем предусмотрено описанием

функции. Значение по умолчанию можно указывать только для последних параметров в списке. Например, функцию sred можно описать и так:

```
double sred ( double x, double y, double z = 0)
   { double s;
      s = x + y + z;
      return  s / 3;
   }
```

К такой функции можно обращаться с двумя и с тремя аргументами.

- Функция может иметь переменное число параметров, для части параметров могут быть неизвестны их типы. Неизвестная часть списка параметров обозначается многоточием. Например, функция с прототипом

```
int  varfunc ( int n, ... );
```

имеет один обязательный параметр типа int и неопределенное число параметров неизвестных типов.

При вызове функции аргументы передаются в функцию по значениям. Это значит, что для каждого аргумента в памяти создается его копия, которая и используется при вычислении функции. Поэтому любые изменения значений аргументов в теле функции теряются при выходе из функции. Если аргумент является массивом, в функцию передается указатель на начальный элемент массива и присваивания элементам массива в теле функции изменяют значения элементов массива аргумента.

Помимо передачи аргументов по значению, аргументы передаются еще и по ссылке. В этом случае в функцию передаются не копии переменных, а сами переменные. Чтобы передавать аргументы по ссылке, в описании функции перед названием аргументов указывается инструкция &. Например:

void swap (char &a, char &b).

В C++ с объявлением функции связывается так называемая сигнатура функции, определяемая типом возвращаемого значения и типами параметров. Это позволяет назначать функциям, имеющим аналогичное назначение, но использующим параметры разных типов, одинаковые имена. Например, наряду с приведенной выше функцией sred для вещественных аргументов, в той же программе может присутствовать функция

```
double sred ( int x, int y, int z = 0)
   { int s;
      s = x + y + z;
      return  s / 3;
   }
```

15 Перегрузка функций

В C++ с объявлением функции связывается так называемая сигнатура функции, определяемая типом возвращаемого значения и типами параметров. Это позволяет назначать функциям, имеющим аналогичное назначение, но использующим параметры разных типов, одинаковые имена. Например, наряду с функцией sred для вещественных аргументов

```
double sred ( double x, double y, double z)
   { double s;
     s = x + y + z;
     return  s / 3;
   }
```

в этой же программе может присутствовать функция

```
double sred ( int x, int y, int z = 0)
   { int s;
      s = x + y + z;
      return  s / 3;
   }
```

Таким образом, **перегрузка функций** – это создание нескольких функций с одним именем, но с разными параметрами. Под разными параметрами понимают, что должно быть разным *количество аргументов* функции и/или их *тип*. То есть перегрузка функций позволяет определять несколько функций с одним и тем же именем и типом возвращаемого значения.

Перегрузка функций также называется *полиморфизмом функций*, то есть полиморфическая функция – это функция, отличающаяся многообразием форм.

Под полиморфизмом функции понимают существование в программе нескольких перегруженных версий функции, имеющих разные значения. Изменяя количество или тип параметров, можно присвоить двум или нескольким функциям одно и тоже имя. При этом никакой путаницы не будет, поскольку нужная функция определяется по совпадению используемых параметров. Это позволяет создавать функцию, которая сможет работать с целочисленными, вещественными значениями или значениями других типов без необходимости создавать отдельные имена для каждой функции.

Таким образом, благодаря использованию перегруженных функций, не следует беспокоиться о вызове в программе нужной функции, отвечающей типу передаваемых переменных. При вызове перегруженной функции компилятор автоматически определит, какой именно вариант функции следует использовать.

16 Встраиваемые функции

Подставляемые или встраиваемые (inline) функции – это функции, код которых вставляется компилятором непосредственно на место вызова, вместо передачи управления единственному экземпляру функции.

Если функция является встраиваемой, компилятор не создает данную функцию в памяти, а копирует ее строки непосредственно в код программы по месту вызова. Это равносильно вписыванию в программе соответствующих блоков вместо вызовов функций. Таким образом, спецификатор inline определяет для функции так называемое *внутреннее связывание*, которое заключается в том, что компилятор вместо вызова функции подставляет команды ее кода. Встраиваемые функции используют, если тело функции состоит из нескольких операторов.

Этот подход позволяет увеличить скорость выполнения программы, так как из программы исключаются команды микропроцессора, требующиеся для передачи аргументов и вызова функции.

Например:

```
/*функция возвращает расстояние от точки с координатами(x1,y1) до точки с
координатами (x2,y2)*/
inline float Line(float x1,float y1,float x2, float y2) {
  return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
}
```

Однако использование встраиваемых функций не всегда приводит к положительному эффекту. Если такая функция вызывается в программном коде несколько раз, то во время компиляции в программу будет вставлено столько же копий этой функции, сколько ее вызовов. Произойдет значительное увеличение размера программного кода, в результате чего ожидаемого повышения эффективности выполнения программы по времени может и не произойти.

Причины, по которым функция со спецификатором inline будет трактоваться как обычная не подставляемая функция:

- подставляемая функция является рекурсивной;
- функции, у которых вызов размещается до ее определения;
- функции, которые вызываются более одного раза в выражении;
- функции, содержащие циклы, переключатели и операторы переходов;
- функции, которые имеют слишком большой размер, чтобы сделать подстановку.

Ограничения на выполнение подстановки в основном зависят от реализации. Если же для функции со спецификатором inline компилятор не может выполнить подстановку из-за контекста, в который помещено обращение к ней, то функция считается статической и выдается предупреждающее сообщение.

Еще одной из особенностей подставляемых функций является невозможность их изменения без перекомпиляции всех частей программы, в которых эти функции вызываются.

17 Библиотека времени выполнения

В определении языков Си++ отсутствуют операторы ввода-вывода, операции над строковыми данными и многие другие средства, имеющиеся в других языках программирования. Это компенсируется добавлением в системы программирования Си++ библиотек функций, подключаемых к рабочим программам и называемых библиотеками времени выполнения. Отделение этих библиотек от компилятора позволяет использовать различные варианты этих библиотек.

Часть функций из библиотек времени выполнения стандартизована, в стандарте зафиксированы имя функции, ее назначение, перечень и типы параметров и тип возвращаемого значения.

Чтобы обеспечить возможность контроля правильности вызова функций при компиляции программы, в систему программирования входят файлы заголовков функций времени выполнения. В файлах заголовков определяются прототипы библиотечных функций, а так же константы и типы данных, используемые этими функциями. Например, в Си++ добавлены операции с комплексными числами и десятичными данными:

BCD.H - Данные, представленные в десятичной системе счисления

COMPLEX.H - Функции и операции над комплексными числами.

Имеются также файлы прототипов функций для распределения и освобождения динамической памяти, использования средств DOS и BIOS.

Чтобы использовать какие-либо функции из библиотек времени выполнения в программу должен быть включен файл-заголовок с прототипами требуемых функций. Включение в программу файла-заголовка обеспечивается препроцессорной директивой

```
# include   < имя файла >
```

Любая программа использует какие-либо входные данные и куда-либо выводит полученные результаты, поэтому файл заголовков stdio.h присутствует почто во всех программах. Функции ввода-вывода используют понятие потока, рассматриваемого как последовательность байтов. Поток может быть связан с дисковым файлом или другим устройством ввода-вывода, в том числе с консолью, когда ввод осуществляется с клавиатуры, а вывод - на экран монитора.

В Си++ включены собственные средства потокового ввода-вывода, обеспечивающие контроль типов в операциях ввода-вывода. Для этого определены четыре новых стандартных потока:

cin - для ввода данных,

cout - для вывода данных,

cerr - вывод сообщений об ошибках без буферизации вывода,

clog - вывод сообщений об ошибках с буферизацией вывода.

В качестве знака операции вывода определены знаки << , а знаком операции ввода - знаки >>, те же, что и для операций сдвига. Компилятор по контексту определяет, какую операцию задают эти символы, ввод-вывод или сдвиг.

Чтобы использовать средства ввода-вывода Си++ в программу должен быть включен файл-заголовок iostream.h :

```
# include <iostream.h>
```

В операциях вывода левым операндом должен быть поток вывода, правым операндом - выводимое данное. Результатом операции вывода является поток вывода, что позволяет записывать вывод в виде цепочки операций <<, например,

```
cout << "x1 = " << x1 << " x2 = " << x2 << "\n";
```

Для базовых типов данных определены их преобразования при выводе и точность представления на устройстве вывода. Эти характеристики можно менять, применяя специальные функции, называемые манипуляторами.

В операции ввода левым операндом должен быть поток ввода, а правым операндом - имя вводимого данного для арифметических данных или указатель типа char* для ввода строк, например,

```
cin >> x1 >> x2 >>st ;
```

Операции ввода-вывода выполняются слева направо и последний оператор эквивалентен оператору ((cin>> x1) >> x2) >> st; или трем операторам

```
cin >> x1;  cin >> x2;  cin >> st;
```

Пример программы, запрашивающей у пользователя два целых числа и выводящей на экран их сумму:

```
# include <iostream.h>
  int   x, y ;
  int  main (  )
      { cout << "x = "; cin >> x;   //  Запрос и ввод значения x
        cout << "\n y = ";  cin >> y; // Запрос и ввод значения y
        cout << "\n" << " x + y = " << x + y;
        return 0;
      }
```

18 Основные понятия препроцессорной обработки

Препроцессорная обработка (макрообработка) — это преобразование текста путем замены препроцессорных переменных их значениями и выполнения препроцессорных операторов (директив препроцессора).

В общем случае препроцессорные средства включают:

- определение препроцессорных переменных и присвоенных им значений;

- средства управления просмотром преобразуемого текста;

- правила подстановки значений макропеременных.

Определение препроцессорной переменной часто называют макроопределением или макросом, а подстановку ее значения в обрабатываемый текст — макрорасширением.

Макрообработка состоит в последовательном просмотре исходного текста и выделения в нем лексем. Если выделенная лексема является препроцессорной переменной, она заменяется на свое значение, т.е. строится макрорасширение. Если встречается препроцессорная директива, то она выполняется. Лексемы, не являющиеся препроцессорными переменными или директивами, переносятся в выходной текст без изменения. Результатом такой обработки является текст, не содержащий препроцессорных директив и препроцессорных переменных. Если исходный текст был программой C++, то после макрообработки должен быть получен синтаксически правильный текст C++.

Препроцессор обеспечивает возможность включения в программу исходных текстов из других файлов, в Си++ это выполняется по директиве

include имя файла

При включении файла на место директивы #include вставляется текст из этого файла, а последующие строки исходного файла сдвигаются вниз, после чего вставленный текст сканируется препроцессором.

Директивы препроцессора всегда записывается с новой строки и первым символом директивы должен быть знак #, которому могут предшествовать только пробелы и знаки табуляции. Концом текста директивы служит конец строки. Если директива не помещается в одной строке, в конце строки ставится знак \ и директива продолжается на следующей строке. Количество строк продолжения не ограничивается.

Препроцессорные переменные

Препроцессорная переменная (макроимя) объявляется директивой

```
# define идентификатор  значение
```

Например:

```
# define   DEBUG
```

Переменная DEBUG объявлена. но не имеет значения. В последующем тексте можно проверять, объявлено или нет это имя, и в зависимости от результата проверки включать или не включать в программу некоторые операторы.

Объявленное в define макроимя известно препроцессору от точки его объявления до конца файла или пока не встретится директива

undef имя

Например, #undef DEBUG

Если в последующем тексте встретится имя DEBUG, оно будет рассматриваться как обычное, а не препроцессорное имя.

Имеется ряд предопределенных макроимен, например:

```
_ _LINE_ _        - номер строки в исходном файле,
_ _FILE_ _         - имя обрабатываемого файла,
_ _DATE_ _       - дата начала обработки препроцессором,
_ _TIME_ _       - время начала обработки,
и другие.
```

Предопределенные имена нельзя объявлять в #define или отменять в #undef.

Макроопределения (макросы)

Полный синтаксис директивы #define имеет вид:

define идентификатор(параметры) список_замены

Параметры задаются их именами, список замены - это текст на C++, содержащий имена параметров.

Знак # перед именем параметра означает, что значение аргумента рассматривается как строковый литерал. Если между двумя параметрами в макрорасширении стоят знаки ##, то значения аргументов сцепляются по правилу сцепления строк, например,

```
# define  VAR(a,b)      ( a##b )
x = d [ VAR(i,j)];     то есть  x = d  [ ( ij )];
```

Использование макросов в ряде случаев позволяет сократить исходный текст программы и сделать его более наглядным. Например, если поместить в файл-заголовок макросы

```
#if defined(__cplusplus)
#   define _PTRDEF(name) typedef name  * P##name;
#   define _REFDEF(name) typedef name  & R##name;
#   define _STRUCTDEF(name) struct name; \
           _PTRDEF(name) \
           _REFDEF(name) \
#endif
```

то мы получим возможность одной строкой программы

```
    _STRUCTDEF(MyStruct)
```

объявить имя структурного типа MyStruct, указатель на этот тип PMyStruct и тип ссылки на него RMyStruct, т.е. получить в выходном тексте строчки

```
struct  MyStruct;
typedef  MyStruct  *PMyStruct;
typedef  MyStruct  &RMystruct;
```