Тема: «Функции»

Вопросы:

1. Понятие функции. Прототипы функций. Передача параметров.
2. Встраиваемые функции. Перегрузка функций. Указатель на функцию.
3. Примеры использования функций. Область видимости и классы памяти.

Описание функции

Самый распространенный способ задания в C++ каких-то действий –это вызов функции, которая выполняет такие действия.

Функция – это именованная часть программы (блок кода, не входящий в основную программу), к которой можно обращаться из других частей программы столько раз, сколько потребуется. Основная форма описания функции имеет вид:

```
Тип < Имя функции> (Список параметров)
{
Операторы тела функции
}
```

Тип определяет тип значения, которое возвращает функция с помощью оператора return. Если тип не указан, то по умолчанию предполагается, что функция возвращает целое значение (типа int).

Список параметров состоит из перечня типов и имен параметров, разделенных запятыми. Функция может не иметь параметров, но круглые скобки необходимы в любом случае.

Первая строка описания функции, содержащая тип возвращаемого значения, имя функции и список параметров, называется заголовком функции. Параметры, перечисленные в заголовке описания функции, называются формальными, а записанные в операторе вызова функции – фактическими. Тип возвращаемого значения может быть любым, кроме массива и функции. В приведенном ниже фрагменте программы функция перемножает два числа и возвращает результат в основную программу с помощью оператора return через переменную z.

```
int multiply(int x, int y) // заголовок функции
{
int z = (x * y ); // тело функции return z ;
}
```

До использования функции ее необходимо объявить. Объявление функции осуществляется с помощью прототипа, который сообщает

компилятору, сколько аргументов принимает функция, тип каждого аргумента и тип возвращаемого значения.

Пример использования функции для вычисления произведения двух чисел.

```
// Вычисление произведения двух чисел
#include<iostream.h> #include <conio.h> #include <stdlib.h>
int multiply(int x, int y); // объявление прототипа функции main( )
{
clrscr();
int x, y, result;
cout <<”\nВведите первое число:”; cin >> x;
cout <<”\nВведите второе число:”; cin>>y;
result = multiply(x, y); // вызов функции cout << “\nРезультат =” << result;
cout<<"\nНажмите любую клавишу ..."; getch();
return 0 ;
}
int multiply(int x, int y) //заголовок функции
{
return x * y ; // тело функции
}
```

Результаты работы программы:

```
Введите первое число: 3
Введите второе число: 2
Результат = 6
```

Эта программа запрашивает у пользователя два числа, а затем вызывает функцию multiply() для вычисления произведения. Прототип функции объявляется в голове программы и представляет собой тот же заголовок функции, но с точкой запятой <<;>> в конце. При желании имена переменных в прототипах можно не указывать.

```
// Нахождение максимального числа из трех заданных чисел
#include<iostream.h> #include <conio.h>
int maxs(int, int, int); // прототип функции main()
{
clrscr(); //очистка экрана int x, y, z, s;
cout<<"\nВведите три числа:\n"; cin>>x>>y>>z;
s=maxs(x, y, z) ; cout<<"\nРезультат ="<< s;
```

```
cout<<"\nНажмите любую клавишу ..."; getch();
return 0;
}
int maxs(int x, int y, int z)
{
int max=x; if(y>max) max=y; if(z>max) max=z; return max;
}
```

Результаты работы программы:

Введите три числа: 5 15 1

Результат =15.

Прототип функции maxs имеет вид int maxs(int, int, int);. Этот прототип указывает, что maxs имеет три аргумента типа int и возвращает результат типа int. Отметим, что прототип функции имеет такой же вид, что и заголовок описания функции maxs, за исключением того, что в него не включены имена параметров(x,y,z). Значение возвращается из функции с помощью оператора return. Тип возвращаемого значения может быть любым, кроме массива и функции. Могут быть также функции, не возвращающие никакого значения. В заголовке таких функций тип возвращаемого значения объявляется void. Если тип возвращаемого значения не указан, он по умолчанию считается равным int.

```
//Возведение произвольного числа m в степень i
#include <iostream.h>
#include <conio.h>
void step(float, int); // прототип функции main( )
{
int i; float m, s;
clrscr(); // очистка экрана cout<<"\nВведите m = "; cin>> m;
cout<<"\nВведите i = "; cin>> i; step(m, i);
cout<<"\nНажмите любую клавишу ..."; getch();
}
void step(float base, int k) // заголовок функции
{
int i; float p = 1; for(i=1;i<=k;++i) p*=base;
cout<<"\n"<< base << " в степени "<< k << " равно "<<p<<"\n";
}
```

Результаты работы программы:

Введите m = 2

Введите i = 3

2 в степени 3 равно 8

В этой программе реализован алгоритм возведения в целую степень любого числа. Функция step принимает по значению два числа типов float и integer, а результат ее вычисления не возвращается в основную программу. При вызове тип каждого параметра функции сопоставляется с ожидаемым типом точно так же, как если бы инициализировалась переменная описанного типа. Это гарантирует надлежащую проверку и преобразование типов. Например, обращение к функции step(float x, int n) с помощью оператора step(12.3,"abcd") вызовет недовольство компилятора, поскольку "abcd" является строкой, а не int. При вызове step(2, i) компилятор преобразует 2 к типу float, как того требует функция. Функция step может быть определена, например, так:

```
float step(float x, int n)
{
if (n < 0) error("извините, отрицательный показатель для step( )"); switch (n)
{
case 0: return 1; case 1: return x;
default: return x*step(x, n -1);
}
}
```

Первая часть определения функции задает имя функции(step), тип возвращаемого ею значения (float) и типы и имена ее параметров (float x, int n). Значение возвращается из функции с помощью трех операторов return.

Правила работы с функциями:

- Функция может принимать любое количество аргументов или не иметь их вообще.

- Функция может возвращать значение, но это не является обязательным.

- Если для возвращаемого значения указан тип void, функция не возвращает никакого значения. При этом функция не должна содержать оператора return, однако при желании его можно оставить.

- Если в объявлении функции указано, что она возвращает значение, в теле функции должен содержаться оператор return, возвращающий это значение. В противном случае компилятор выдаст предупреждение.

- Функции могут иметь любое количество аргументов, но возвращаемое значение всегда одно.

- Аргументы могут передаваться функции по значению, через указатели или по ссылке.

Основные преимущества построения программ на основе функций сводятся к следующему:

- Программирование с использованием функций делает разработку программ более управляемой.

- Повторное использование программных кодов, т.е. использование существующих функций как стандартных блоков для создания новых программ.

- Возможность избежать в программе повторения каких-либо фрагментов.

Передача параметров

При вызове функции выделяется память для ее формальных параметров, и каждый формальный параметр инициализируется значением соответствующего фактического параметра. Имеются два способа обращения к функциям – ***вызов по значению*** и ***вызов по ссылке***.

Когда аргумент передается вызовом по значению, создается копия аргумента, и она передается вызываемой функции. Основным недостатком вызова по значению является то, что при передаче большого количества данных создание копии может привести к большим потерям времени выполнения. В случае вызова по ссылке оператор вызова дает вызываемой функции возможность прямого доступа к передаваемым данным, а также возможность изменения этих данных.

Ссылочный параметр – это псевдоним соответствующего аргумента. Чтобы показать, что параметр функции передан по ссылке, после типа параметра в прототипе и заголовке функции ставится символ амперсанта. Рассмотрим функцию

```
int multiply (int x, int &y) //заголовок функции
{
return x * y ;
}
```

В функцию multiply параметр x передается по значению, а параметр y по ссылке. Вызов по ссылке хорош в смысле производительности, так как исключает процедуру копирования данных, однако при этом появляется вероятность того, что вызываемая функция может испортить передаваемые в нее данные.

```
#include<iostream.h> #include <conio.h>
int multiply(int, int &); main( )
{
clrscr(); //очистка экрана int x1,y1,z;
cout<<"\nВведите первый сомножитель="; cin>>x1;
cout<<"\nВведите второй сомножитель="; cin>>y1;
cout<<"\nРезультат ="<<myltiply(x1, y1) ;
cout<<"\nx1="<<x1<<"\ny1="<<y1;
cout<<"\n----------\n"; cout<<"\nНажмите любую клавишу
...";
getch();
}
int multiply(int x, int &y)
{
return ++x * ++y;
}
```

Результаты работы программы при введенных значениях x1=1 и y1=2:
Результат= 6 x1=1, y1=3

Приведенные данные показывают, что при передаче параметров по ссылке их исходные значения портятся. Так, исходное значение переменной y1 было равно 2, а после вызова функции оно стало равным 3, тогда как значение переменной x1, передаваемой по значению, осталось неизменным. Это объясняется тем, что при вызове функции multiply в выражении ++x увеличивается локальная копия первого фактического параметра, тогда как в ++y фактический параметр увеличивается сам. Поэтому функции, которые изменяют свой передаваемый по ссылке параметр, трудно интерпретируются, и их по возможности лучше избегать.

Однако при наличии громоздких аргументов типа больших массивов целесообразно использовать передачу параметров по ссылке, запрещая функции изменять значения аргументов. Это может быть осуществлено передачей в функцию аргументов, как констант, что достигается установкой ключевого слова const перед соответствующими переменными в списке аргументов.

Для этого заголовок функции myltiply( ) должен иметь следующий вид:
int multiply(int x, const int &y).

Указание ключевого слова const в прототипе функции не обязательно. При таком описании функции аргумент y не будет копироваться при вызове функции, но внутри функции изменить значение y будет невозможно. На

любую попытку изменить аргумент у компилятор выдаст сообщение об ошибке.

Альтернативной формой передачи параметра по ссылке является использование указателей. При этом адрес переменной передается в функцию не операцией адресации (&), а операцией косвенной адресации (*). В списке параметров подобной функции перед именем переменной указывается символ *, свидетельствуя о том, что передается не сама переменная, а указатель на нее. В теле функции тоже перед именем параметра ставится символ разыменовывания *, чтобы получить доступ через указатель к самой переменной. А при вызове функции в нее в качестве аргумента должна передаваться не сама переменная, а ее адрес, полученный с помощью операции адресации &.

Приведем пример рассмотренной ранее функции myltiply( ), но с передачей параметра по ссылке с помощью указателя:

```
#include<iostream.h> #include <conio.h>
int multiply(int ,int *); // прототип функции main()
{
clrscr(); //очистка экрана int x1,y1,z;
cout<<"\nВведите первый сомножитель="; cin>>x1;
cout<<"\nВведите второй сомножитель="; cin>>y1;
z=myltip(x1, &y1); // вызов функции
cout<<"\nРезультат ="<< z; cout<<"\n----------\n";
cout<<"\nНажмите любую клавишу ...";
getch();
}
int multip(int x, int *y) // заголовок функции
{
return *y*x; // изменение значения параметра
}
```

Перегрузка функций

Язык С++ позволяет определять функции с одним и тем же именем, но разным набором параметров. Подобная возможность называется перегрузкой функций (function overloading). Компилятор же на этапе компиляции на основании параметров выберет нужный тип функции.

Чтобы определить несколько различных версий функции с одним и тем же именем, все эти версии должны отличаться как минимум по одному из следующих признаков:

1) имеют разное количество параметров;

2) соответствующие параметры имеют разный тип.

При этом различные версии функции могут также отличаться по возвращаемому типу. Однако компилятор, когда выбирает, какую версию функции использовать, ориентируется именно на количество параметров и их тип.

Рассмотрим простейший пример:

```
#include <iostream>
 int sum(int, int);
double sum(double, double);
 int main()
{
   int result1 {sum(3, 6)}; // выбирается версия int sum(int, int)
   std::cout << result1 << std::endl; // 9

   double result2 {sum(3.3, 6.6)}; // выбирается версия double sum(double, double)
   std::cout << result2 << std::endl; // 9.9
}
int sum(int a, int b)
{
   return a + b;
}
double sum(double a, double b)
{
   return a + b;
}
```

Здесь определены две версии функция sum, которая складывает два числа. В одном случае она складывает два числа типа int, в другом - числа типа double. При вызове функций компилятор на основании переданных аргументов определяет, какую версию использовать. Например, при первом вызове передаются числа int:

```
int result1 {sum(3, 6)};
```

Соответственно для этого вызова выбирается версия

```
int sum(int, int);
```

Во втором вызове в функцию передаются числа с плавающей точкой:

```
double result2 {sum(3.3, 6.6)};
```

Поэтому выбирается версия, которая принимает числа double:

```
double sum(double, double);
```

Аналогично перегруженные версии функции могут отличаться по количеству параметров:

```
#include <iostream>
 int sum(int, int);
int sum(int, int, int);
 int main()
{
   int result1 {sum(3, 6)}; // выбирается версия int sum(int, int)
   std::cout << result1 << std::endl; // 9

   int result2 {sum(3, 6, 2) }; // выбирается версия int sum(int, int, int)
   std::cout << result2 << std::endl; // 11
}
int sum(int a, int b)
{
   return a + b;
}
int sum(int a, int b, int c)
{
   return a + b + c;
}
```

Перегрузка функций и параметрами-ссылки

При перегрузке функций с параметрами-ссылками следует учитывать, что параметры типов data_type и data_type& не различаются при перегрузке. Например, два следующих прототипа:

```
void print(int);
void print(int&);
```

Не считаются разными версиями функции print.

Перегрузка и параметры-константы

При перегрузке функций константный параметр отличается от неконстантного параметра только для ссылок и указателей. В остальных случаях константый параметр будет идентичен неконстантному параметру. Например, следующие два прототипа при перегрузке различаться НЕ будут:

```
void print(int);
void print(const int);
```

Во втором прототипе компилятор игнорирует оператор const.

Пример перегрузки функции с константными параметрами

```
#include <iostream>
 int square(const int*);
int square(int*);
 int main()
{
   const int n1{2};
   int n2{3};
   int square_n1 {square(&n1)};
   int square_n2 {square(&n2)};
   std::cout << "square(n1): " << square_n1 << "\tn1: " << n1 << std::endl;
   std::cout << "square(n2): " << square_n2 << "\tn2: " << n2 << std::endl;
}

int square(const int* num)
{
   return *num * *num ;
}
int square(int* num)
{
   *num = *num * *num; // изменяем значение по адресу в указателе
   return *num;
}
```

Здесь функция square принимает указатель на число и возводит его в квадрат. Но первом случае параметр представляет указатель на константу, а во втором - обычный указатель.

В первом вызове

```
int square_n1 {square(&n1)};
```

Компилятор будет использовать версию

```
int square(const int*);
```

так как передаваемое число n1 представляет константу. Поэтому переданное в эту функцию число n1 не изменится.

В втором вызове

```
int square_n2 {square(&n2)};
```

Компилятор будет использовать версию

int square(int*);

в которой для примера также меняется передаваемое значение. Поэтому переданное в эту функцию число n2 изменит свое значение. Консольный вывод программы

square(n1): 4 n1: 2
square(n2): 9 n2: 9

С передачей константной ссылки все будет аналогично.

Область видимости переменных

Областью видимости (областью действия) переменной называется область программы, в которой на данную переменную можно сослаться. На некоторые переменные можно сослаться в любом месте программы, тогда как на другие – только в определенных ее частях.

Класс памяти определяется, в частности, местом объявления переменной.

Локальные переменные объявляются внутри некоторого блока или функции. Эти переменные видны только в пределах того блока, в котором они объявлены.

Блоком называется фрагмент кода, ограниченный фигурными скобками "{ }".

Глобальные переменные объявляются вне какого-либо блока или функции. Спецификации класса памяти могут быть разбиты на два класса: автоматический класс памяти с локальным временем жизни и статический класс памяти с глобальным временем жизни. Ключевые слова auto и register используются для объявления переменных с локальным временем жизни. Эти спецификации применимы только к локальным переменным. Локальные переменные создаются при входе в блок, в котором они объявлены, существуют лишь во время активности блока и исчезают при выходе из блока. Спецификация auto, как и другие спецификации, может указываться перед типом в объявлении переменных. Например: auto float x, y;. Локальные переменные являются переменными с локальным временем жизни по умолчанию, так что ключевое слово auto используется редко. Далее будем ссылаться на переменные автоматического класса памяти просто как на автоматические переменные.

Пусть, например, имеется следующий фрагмент кода:

{

```
int i = 1;

...

i++;

...

},
```

к которому в ходе работы программы происходит неоднократное обращение. При каждом таком обращении переменная i будет создаваться заново (под нее будет выделяться память) и будет инициализироваться единицей. Затем в ходе работы программы ее значение будет увеличиваться на 1 операцией инкремента. В конце выполнения этого блока переменная исчезнет и выделенная под нее память освободится. Следовательно, в такой локальной переменной невозможно хранить какую-то информацию между двумя обращениями к блоку. Спецификация класса памяти register может быть помещена перед объявлением автоматической переменной, чтобы компилятор сохранял переменную не в памяти, а в одном из высокоскоростных аппаратных регистров компьютера.

Например: register int i = 1;.

Если интенсивно используемые переменные, такие как счетчики или суммы могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память могут быть исключены. Это сокращает время вычислений. Компилятор может проигнорировать объявления register. Например, может оказаться недостаточным количество регистров, доступных компилятору для использования. К тому же оптимизирующий компилятор способен распознавать часто используемые переменные и решать, помещать их в регистры или нет. Так что явное объявление спецификации register применяется редко.

Ключевые слова extern и static используются, чтобы объявить идентификаторы переменных как идентификаторы статического класса памяти с глобальным временем жизни. Такие переменные существуют с момента начала выполнения программы. Для таких переменных память выделяется и инициализируется сразу после начала выполнения программы. Существует два типа переменных статического класса памяти: глобальные

переменные и локальные переменные, объявленные спецификацией класса памяти static.

Глобальные переменные по умолчанию относятся к классу памяти extern. Глобальные переменные создаются путем размещения их объявлений вне описания какой-либо функции и сохраняют свои значения в течение всего времени выполнения программы. На глобальные переменные может ссылаться любая функция, которая расположена после их объявления или описания в файле.

Локальные переменные, объявленные с ключевым словом static, известны только в том блоке, в котором они определены. Но, в отличие от автоматических переменных, локальные переменные static сохраняют свои значения в течение всего времени выполнения программы. При каждом следующем обращении к этому блоку локальные переменные содержат те значения, которые они имели при предыдущем обращении.

Вернемся к уже рассмотренному выше примеру, но укажем для переменной i статический класс:

```
{
static int i = 1;
...
i++;
...
}
```

Инициализация переменной i произойдет только один раз за время выполнения программы. При первом обращении к этому блоку значение переменной i будет равно 1. К концу выполнения блока ее значение станет равно 2. При следующем обращении к блоку это значение сохранится и при окончании повторного выполнения блока i будет равно 3.

Таким образом, статическая переменная способна хранить информацию между обращениями к блоку и, следовательно, может использоваться, например, как счетчик числа обращений. Все числовые переменные статического класса памяти принимают нулевые начальные значения, если программист явно не указал другие начальные значения. Статические переменные – указатели тоже имеют нулевые начальные значения.

Спецификации класса памяти extern используются в программах с несколькими файлами. Пусть, например, в модуле Unit1 или в файле Unit1.cpp объявлена глобальная переменная int a = 5; .

Тогда, если в другом модуле Unit2 или в файле Unit2.cpp объявлена глобальная переменная extern int a;, то компилятор понимает, что речь идет об одной и той же переменной. И оба модуля могут с ней работать. Для этого

даже нет необходимости связывать эти модули директивой #include, включающей в модуль Unit1 заголовочный файл второго модуля.

Таким образом, область действия идентификатора – это часть программы, в которой на идентификатор можно ссылаться. Например, на локальную переменную, объявленную в блоке, можно ссылаться только в этом блоке или в блоке, вложенном в этот блок. Существует четыре области действия идентификатора – функция, файл, блок и прототип функции.

Идентификатор, объявленный вне любой функции (на внешнем уровне), имеет область действия файл. Такой идентификатор известен всем функциям.