Лабораторная работа 9. Функции в C++

1. *ЦЕЛЬ РАБОТЫ*

Изучить способы объявления и использования функций в языке C++.

2. *КРАТКИЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ*

Функция - часть программы, имеющая собственное название. Любая программа на C++ содержит функцию `main()` или `WinMain()`, которая является ее входной точкой. Функция `main()` — это частный случай *функции* вообще. Функции являются основными “строительными блоками” программы.

В программах на C++ можно использовать уже готовые, написанные ранее, «стандартные» библиотечные функции. Например, включив программу строку

```
#include <cmath>
```

можно использовать большое количество математических функций.

В следующей таблице представлены некоторые из математических функций.

Название функции	Краткое описание
`sin(x), cos(x), tan(x)`	Вычисляет соответствующую тригонометрическую функцию
`abs (x)`	Вычисляет абсолютное значение аргумента
`exp(x)`	Функция e^x
`log(x), log10(x)`	Вычисление логарифма (натурального или десятичного)
`pow(x,y)`	Вычисление x в степени y
`sqrt(x)`	Квадратный корень

Определить функцию - значит указать, какие действия и как она должна выполнять. Функцию нельзя вызвать до тех пор, пока она не определена. В общем виде функция определяется таким образом:

```
Тип Имя_функции(список_параметров)
{
  // В фигурных скобках заключено тело функции,
  // составленное из отдельных операторов.
тело_функции
}
```

Оператором возврата из функции в точку ее вызова является оператор `return`. Тип возвращаемого выражения должен совпадать с объявленным типом функции.

Обмен аргументами и результатами между основной программой и функцией производится через *параметры* (*формальные* и *фактические*). Между формальными и фактическими параметрами должны соблюдаться правила соответствия по последовательности и по типам. Передача параметров при вызове функции происходит по значению. Поэтому выполнение функции не может изменить значения переменных, указанных в качестве фактических параметров.

В следующем примере функция `main()` использует функцию `Cub()` для вычисления третьей степени аргумента.

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

//определение вспомогательной функции
float Cub(float f) {return f*f*f;}

int main()
{int x,y;  // объявляем переменные

 cin>>x;   // вводим x
 y=Cub(x); //вызываем функцию для вычисления третьей степени числа x
 cout<<"\nx^3 = "<< y;
 cout<<"\nx^3 = "<< Cub(x);
 return 0;
     }
```

Прототипы функций. Полное определение функции необязательно помещать перед основной частью программы. Можно использовать *прототип* функции – предварительное описание (декларацию) функции, в котором содержится вся необходимая информация для правильного обращения к ней: имя и тип функции, типы формальных параметров. В прототипе имена формальных параметров указывать необязательно. Декларация прототипа функции заканчивается точкой с запятой.

В следующем примере объявим функцию `Sum()` для вычисления суммы трех чисел. Определение функции поместим после функции `main()`.

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

int Sum(int A, char B, char C); // прототип функции, т.е. объявление

int main()
{int x,y,z,S;
 cin>>x>>y>>z; // Вводим три числа
 S=Sum(x,y,z); // Используем функцию
 cout<<"\nSumma "<< x <<"+"<<y<<"+"<<z<<"="<< S;
 S=Sum(4,3,-2); // Можно так использовать функцию
 cout<<"\nSumma 2 = "<< Sum(4,3,-2); // Так тоже можно вызывать функцию
 return 0;
 }

// определение функции, то есть указание того, что она должна делать
int Sum(int A, char B, char C)
{ return A+B+C;
 }
```

Функция, которая не возвращает результата, объявляется как `void`. Например, в следующем примере используется функция `hello()`, которая выводит на экран

приветствие. Так как она не возвращает в основную программу никаких результатов, она объявлена как `void`:

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

void hello(void)
{cout<<"Hello All\n"; return;}


int main()
{cout<<"Primer: ";
 hello();
 return 0;
 }
```

В программировании использование процедур и функций делает процесс разработки программ более простым и быстрым, а создаваемые приложения – более надежными. Эффект при использовании подпрограмм такой же, как если бы тело этих подпрограмм присутствовало бы в вызывающем модуле. Но, в отличие от кода, производимого макроассемблером, подпрограмма может быть скомпилирована где угодно в памяти, после чего на нее можно просто ссылаться и не обязательно компилировать ее внутри реального кода главной программы.

Годами разработчики программного обеспечения совершенствовались в искусстве использования многочисленных маленьких подпрограмм в сильно разветвленных, протяженных программах. Они могут быть написаны и отлажены независимо друг от друга. Это облегчает повторное использование ранее написанных программ, и так легче распределять части работы между различными программистами. Короткие куски проще продумывать и проверять их правильность.

Когда функции компилируются в отдельных частях памяти и вызываются ссылками на них, можно использовать одну и ту же функцию много раз без излишнего расходования места на повторы кода подпрограммы. Так разумное использование подпрограмм может уменьшать размеры кода.

К сожалению, при этом имеется проигрыш в скорости исполнения. Проблему создает необходимость сохранения содержимого регистров перед переходом на подпрограмму и их восстановления при возвращении оттуда. Еще больше времени требуют невидимые, но существенные участки кода, необходимые для передачи параметров в и из подпрограммы.

Существенен также сам способ вызова и передачи параметров подпрограмме. Для автономного тестирования подпрограммы приходится писать специальные тестовые программы для ее вызова.

По этой причине рекомендуется умеренное и разумное использование подпрограмм.

Значения аргументов функции по умолчанию. В прототипах функций можно задавать значения аргументов, передаваемых по умолчанию. Предположим, составлена функция `DrawCircle`, которая рисует на экране окружность заданного

радиуса с центром в заданной точке. Для этой функции можно использовать такой прототип:

```
void DrawCircle (int x=100, int y=100, int R=50);
```

или просто:

```
void DrawCircle (int=100, int=100, int=50);
```

хотя так труднее читать программу.

Вызовы этой функции могут быть такими:

```
DrawCircle ();
```

Рисуется окружность с центром в точке 100;100 и радиусом 50.

```
DrawCircle (200);
```

Центр в точке 200;100, радиус 50.

```
DrawCircle (200,300);
```

Центр в точке 200;300, радиус 50.

```
DrawCircle (200,300,100);
```

Центр в точке 200;300, радиус 100.

```
DrawCircle (,, 100);
```

Это ошибка. Разрешается опускать аргументы только справа.

Следующий прототип был бы ошибочным, так как задавать значения по умолчанию надо “начиная справа”:

```
DrawCircle (int x, int y=100, int R);
```

Перегруженные функции. В программировании то и дело случается писать функции для схожих действий, выполняемых над различными типами и наборами данных. Пусть, например, функция должна возвращать квадрат своего аргумента. В C/C++ возведение в квадрат целого и числа с плавающей точкой — разные операции. Поэтому придется написать две функции — одну, принимающую целый аргумент и возвращающую целое, и вторую, принимающую тип `double` и возвращающую также `double`. Как известно, функции должны иметь уникальные имена. Таким образом, перед программистом встает задача придумывать массу имен для различных функций, выполняющих похожие или даже одинаковые действия. Например, `SquareInt()` и `SquareDbl()` .

В C++ можно создать *перегруженные* (overloaded) имена функций, когда функции с одним именем можно, тем не менее, идентифицировать по их списку параметров, контексту, так сказать, в котором имя употребляется.

Рассмотрите следующий пример с вышеупомянутыми квадратами.

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

//Первый вариант функции Sq
int Sq(int arg)
{return arg*arg;}

//Второй вариант функции Sq
double Sq(double arg)
```

```
{return arg*arg;}

int main(void)
{int x = 11;
 double y = 3.1416;
 printf("%d v kvdrate = %d, %f v kvadrate = %f\n",x,Sq(x),y,Sq(y));
 return 0 ;
 }
```

Компилятор, когда ему встречается вызов перегруженной функции, видит только список фактических параметров, но тип, ею возвращаемый, в вызове никак не подразумевается. Поэтому нельзя перегружать функции, отличающиеся только типом возвращаемого значения.

Перегруженные функции являются, по сути, совершенно различными функциями, идентифицируемыми не только своим именем (оно у них одно и то же), но и списком параметров. Компилятор выполняет *кодирование имен,* дополняя имя функции кодовой последовательностью символов, кодирующей тип ее параметров. Тем самым формируется уникальное внутреннее имя. Тип возвращаемого значения никак не отражается на кодировании имени. Компилятор сам выбирает, какую из нескольких функций вызвать в каждом конкретном случае. Критерием выбора служит количество и тип аргументов, с которыми функция вызывается. Причем, если не удается найти точного совпадения, то компилятор выбирает ту функцию, при вызове которой наиболее легко выполнять для аргументов преобразование типа. Рассмотрим пример программы с перегруженной функцией `print`:

```
#include <stdio.h>
#include <iostream>
using namespace std;

void print(int i) {printf("%d ", i);}
void print(double x) {printf("%f ", x);}
void print(char* s) {printf("%s ", s);}

int main()
{int j=5;
 float e=2.7183;
 double pi=3.1415926;
 print(j);  print(e);  print(pi);
 print("Hello All!");
 return 0;
     }
```

В этом примере определены три варианта функции `print`: для целого и вещественного аргументов, а также для аргумента-строки.

3. ЗАДАНИЕ НА РАБОТУ

1. Ознакомиться с теоретическими положениями, приведенными в данных методических указаниях (МУ), а также с конспектом лекций по данной теме.

2. Проверить работу примеров, приведенных в тексте МУ.

3. Разработать программы, использующие 1) функции; 2) функции с параметрами, заданными по умолчанию; 3) перегруженные функции (см. задания 1-3). Можно разрабатывать как консольные приложения, так и приложения с графическим интерфейсом пользователя.

4. ТРЕБОВАНИЯ К ОТЧЕТУ ПО РАБОТЕ

Отчет по работе должен содержать:

- название и цель работы;
- номер варианта;
- для каждого из заданий – текст задачи по своему варианту, блок-схемы разработанных алгоритмов, текст кода программы, результаты работы программы для разных наборов исходных данных.

5. БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Шилдт Г. C++: базовый курс, 3-е издание. : Пер. с англ. – М.: «Издательский дом «Вильямс», 2005. – 624 с.

2. Пахомов Б.И. C/C++ и MS Visual C++ для начинающих. – СПб.: БХВ-Петербург, 2008. – 624 с.