Лабораторная работа №5.

Функции. Перегрузка функций. Шаблоны функций.

Функция – это именованная последовательность описаний и операторов, выполняющая законченное действие. Функция в C++ понимается как простейший способ модульности программы. Для эффективной работы функции достаточно знать ее интерфейс, т.е. имя функции, количество аргументов и их тип, а так же тип возвращаемого результата. Реализация функции может быть скрыта от пользователя, это свойство в ООП называется *инкапсуляцией*.

Функция, перед ее вызовом, должна быть объявлена и определена. Объявление функции (прототип, заголовок) содержит имя функции, тип возвращаемого результата и список параметров. Определение помимо объявления содержит еще тело функции, представляющее собой блок, заключенный в фигурные скобки.

Общий формат объявления функции следующий:

```
[класс] тип_результата имя_функции ([список параметров])[throw]
{
        // тело функции – последовательность
        // описаний и операторов
}
```

Необязательный модификатор класс задает область видимости функции, используя ключевые слова extern и static:

- extern – глобальная видимость во всех модулях проекта (по умолчанию);
- static – видимость только в пределах данного модуля.

Тип возвращаемого результата может быть любым, кроме массива и функции. Если функция не должна возвращать результата (аналог процедуры), указывается тип void.

Список параметров определяет тип и количество передаваемых параметров. Элементы разделяются запятыми, перед каждым из них должен присутствовать тип параметра. Если функция не должна принимать параметров указывается тип void или пустые скобки – ().

Если функция участвует в обработке исключительных ситуаций, указывается ключевое слово throw, о чем речь пойдет в одной из следующих лабораторных работ.

Пример функции суммирования двух целых чисел.

```
#include<iostream>
using namespace std;
// прототип функции
int summa(int,int);        // имена параметров указывать не обязательно

int main()
{
        int var_1, var_2;
        cout << " введите первое слагаемое: "; cin >> var_1;
        cout << " введите второе слагаемое: "; cin >> var_2;
        // вызов функции
        cout << " сумма = " << summa(var_1, var_2) << endl;
        return 0;
}

// определение функции
int summa(int arg_1, int arg_2)
{ return arg_1+arg_2; }
```

Функция может быть определена как встроенная с помощью модификатора inline. Отличие состоит в том, что при вызове inline-функции не происходит передача управления на код, представляющий эту функцию, а осуществляется подстановка самого кода в точку вызова. Это позволяет снизить накладные расходы, связанные с вызовом функции, но может увеличить объем исполняемого модуля. Использование inline-функций носит рекомендательный характер.

Все величины, объявленные внутри функции, а также ее формальные параметры считаются локальными. Областью их действия является функция, время жизни – время работы функции. Ссылаться на них вне функции нельзя.

Если необходимо сохранить значение некоторой локальной переменной от вызова к вызову, ее снабжают модификатором static, например:

```
void function()
{
        static int count = 0;
        while(exp)
        {
                // …
                cout << “  статическая переменная = “ << count++ << endl
        }
// …
}
```

Значение переменной count будет храниться до следующего вызова в сегменте данных. Это позволяет организовать подсчет числа вызовов той или иной функции.

В теле функции видны все объекты, являющиеся глобальными, а, следовательно, их можно изменять. Подобный прием используется для расширения интерфейса между отдельными функциями. Тем не менее, делать этого не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению подобных функций в общую библиотеку.

Функция не может вернуть в качестве результата указатель на локальную переменную, например,

```
int *function()
{
        int var = 38;
        return &var;
}
```

Это происходит по той причине, что после выхода из функции, память, занятая ее локальными параметрами освобождается.

Механизм передачи параметров функции является основным способом обмена информацией между функциями. Различают два способа передачи параметров в функцию: *по значению и по адресу*. При передаче по значению в стек заносятся копии фактических параметров и функция работает именно с копиями. Любые изменения параметров не приведут к изменению исходных значений. При передаче по адресу в стеке формируются копии адресов параметров, а функция осуществляет доступ к параметрам по полученным копиям. Как результат функция может изменить значения исходных параметров. Передача параметров по адресу может быть реализована через указатели или через ссылки. Следующий пример показывает использование различных способов передачи параметров.

```
// прототип функции
void function(int, int *, int &);


int main()
{
        int i = 10, j = 20, k = 30;
        cout << “ До:  i= “ << i << “  “  << “ j= “ << j <<  “ “ << “ k= “ << k << endl;
        function(i, &j, k);
        cout << “ После:  i= “ << i << “  “  << “ j= “ << j <<  “ “ << “ k= “ << k << endl;
        return 0;
}
// определение функции
void function(int a, int *b, int &c)
{
        a++; (*b)++; c++;
```

```
}
```

Первый параметр передается по значению; его изменение в теле функции (a++) никак не отразится на исходном (передаваемом) значении переменной i. Второй и третий параметры передаются по адресу, в частности, второй с помощью указателя, а третий – с помощью ссылки. Различие между указателем и ссылкой очевидно, указатель необходимо разыменовывать. Изменение фактических переданных параметров j и k будет. Выполните эту программу и убедитесь в этом.

Передача массивов в функцию как параметров, а также получение массива как результата работы функции, разрешается только через указатели. Рассмотрим передачу массива в качестве параметра на простом примере – суммирования элементов целочисленной матрицы.

```
int summa(const int *arr, const int size)
{
        // или int summa(int arr[], const int size)
        // или int summa(int arr[size]) этот вариант менее предпочтителен
        int sum =0;
        for(int i = 0; i<size; i++)
        sum += arr[i];
        return sum;
}
```

При передаче многомерных массивов все размерности должны передаваться в качестве параметров, например,

```
int summa(const int *arr, const int nstr, const int nstb);, как варианты
int summa(const int **arr, const int nstr, const int nstb);
int summa(const int arr[][], const int nstr, const int nstb);
```

В первом случае объявляется как одномерный массив указателей на массив, во втором – как указатель на указатель, в третьем – явное объявление двумерного массива.

В языке C++ нельзя передать функцию в качества параметра другой функции, но *указатель на функцию* передать можно. Для этого необходимо объявить указатель на функцию. Рассмотрим на примере.

```
//  прототип функции
int summa(int, int);

// указатель на функцию
typedef int ( *ptr_fun)(int,int);

// функция, получающая в качестве параметра указатель на другую функцию
void function(ptr_fun pf)
{
        cout << " сумма = " << pf(38,64) << endl;
}
```

Функция называется *рекурсивной*, если она вызывает саму себя. Классический пример рекурсивной функции – вычисление факториала числа.

```
long factorial(long n)
{
        If(n==0||n++1) return 1;
        return (n + factorial (n -1));
}
```

Любая рекурсивная функция может быть преобразована в не рекурсивную функцию с помощью операторов цикла. Использование рекурсивных функций предпочтительно из-за их более лаконичного описания, однако накладные расходы возрастают.

Перегружаемые функции, то есть, функции с одним именем – одна из особенностей языка C++, позволяющая выполнить один и тот же алгоритм для параметров разного типа. Перегружаемые

функции позволяют реализовать слабую форму *полиморфизма* – одну из основ ООП. Компилятор самостоятельно определяет, какую именно функцию вызвать по типу фактических параметров. Рассмотрим пример перегруженных функций.

```
int summa(int arg_1, int arg_2)
{
	return arg_1+arg_2;
}
double summa(double arg_1, double arg_2)
{
	return arg_1+arg_2;
}
```

Первое, что выполняет компилятор, осуществляет выбор, соответствующий типу фактических параметров вариант функции. Если точного соответствия типов не найдено, выполняется продвижение типов в соответствии с общими правилами преобразования типов, например, bool и char в int, float, double, указателей в тип void * и т. д. Следующим шагом является преобразование типов, заданных пользователем. Если соответствия не нашлось, вызов считается неоднозначным и выдается сообщение об ошибке.

При определении перегружаемых функций следует руководствоваться следующими правилами:

- функции должны находиться в одной области видимости;
- функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать;
- функции не считаются перегруженными, если описание их параметров отличается модификатором const или использованием ссылки;
- функции не считаются перегруженными, если они возвращают результаты различных типов.

Более универсальным средством параметризации алгоритма являются *шаблоны функций*. С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции на этапе компиляции. Компилятор автоматически генерирует код, соответствующий переданному типу. Общий формат объявления шаблона функции следующий:

```
template <class Type> заголовок _функции_шаблона
{
	// тело функции
}
```

Вместо слова Type можно использовать произвольный идентификатор пользователя. Конструкция <class Type>, именуемая списком шаблона может содержание произвольного числа типов, например, <class T1, class T2, class T3>. Слово class здесь можно заменить на слово typename. Рассмотрим пример:

```
template<class Type>  Type summa(Type arg_1, Type arg_2)
{
	return arg_1+arg_2;
}
```

Вызов шаблона функции ничем не отличается от вызова обычной функции, для данного случая, cout << " Сумма = " << summa(20,30). Всю работу по генерации последовательности кода, соответствующую типу фактических параметров, берет на себя компилятор. Программист может "облегчить участь" компилятора, указав явно передаваемый тип в угловых скобках между именем функции и списком фактических параметров , например, cout << " Сумма целых чисел = " << summa<int>(20,30); или cout << " Сумма вещественных чисел = " << summa<float>(20.5,30.6);. Подобный прием называют специализацией шаблона.

Шаблоны функций можно перегружать как обычные функции.

Для каждого варианта выполнить следующие задания.

Задание №1. *Передача в функцию параметров стандартных типов.* Написать функцию вывода таблицы значения функции из лабораторной работы №2 для аргументов, изменяющихся в заданных пределах с заданным шагом, с точностью ε. Значение аргумента и точность передать в качестве параметров функции.

Задание №2. *Передача в функцию указателя на функцию.* Пользуясь функцией из задания №1, объявить указатель на нее и передать его как параметр некоторой другой функции.

Задание №3. *Передача одномерных массивов в функцию.* Пользуясь массивом, определенным в пункте А лабораторной работы №3, определить функции, реализующие подпункты данного пункта.

Задание №4. *Передача строк в функцию*. Определить функцию, считывающую строку символов (длина строки не более 100 символов), подсчитать, сколько в каждой строке числовых символов.

Задание №5. *Передача многомерных массивов в функцию.* Пользуясь массивом, определенным в пункте В лабораторной работы №3, определить функции, реализующие подпункты данного пункта.

Задание №6. *Передача структур в функцию*. Определить функцию, получающую в качестве аргумента структуру и выводящую поля данной структуры.

Задание №7. *Рекурсивные функции*. Написать функцию упорядочивания массива по возрастанию, используя рекурсию.

Задание №8. *Перегружаемые функции*. Пользуясь заданием №3 данной работы, перегрузить функцию для массивов типов int и double.

Задание №9. *Шаблоны функции*. Определить шаблон функции, реализующий подпункт 1 пункта В (или пункт В) лабораторной работы №3 для произвольных арифметических типов. Вызвать шаблон как обычную функцию и со спецификатором шаблона.