Тема IV. Модульное программирование

Лекция 8

Тема IV. Модульное программирование

§1. Функции

§2. Пользовательские типы данных

§3. Директивы препроцессора

Лекция 8:

§1. Функции
- §1.1. Объявление функций
- §1.2. Определение функций
- §1.3. Вызов функций
- §1.4. Передача параметров в функцию
- §1.5. Передача массивов в функцию
- §1.6. Функции с переменным числом параметров
- §1.7. Рекурсивные функции
- §1.8. Функция **`main()`**
- §1.9. Перегрузка функций
- §1.10. Шаблоны функций

§2. Пользовательские типы данных

§3. Директивы препроцессора

§0. Введение в модульное программирование

Определение

Модуль — отдельный файл, в котором группируются функции и связанные с ними данные.

Достоинства модульного стиля программирования:

- Алгоритмы отделены от данных;
- Высокая степень абстрагирования проекта;
- Модули конструируются и отлаживаются отдельно друг от друга.

Функциональная декомпозиция — это метод разработки программ, при котором задача разбивается на ряд легко решаемых подзадач, решения которых в совокупном виде дают решение исходной задачи в целом.

§1. Функции

☝ **Определение**

Функция — это самостоятельный именованный алгоритм решения некоторой законченной задачи.

§1.1. Объявление функций

Определение

Объявление функции — это создание её прототипа.

- При объявлении функции задается имя функции, тип возвращаемого значения (тип функции) и указывается список передаваемых параметров:

```
[класс] тип имя_функции ([список_параметров]);
```

```
[класс] тип имя_функции ([список_параметров]);
```

[класс] — задает область видимости функции:

- **extern**;
- **static**.

```
[класс] тип имя_функции ([список_параметров]);
```

тип — определяет тип значения, возвращаемого функцией:

```
[класс] тип имя_функции ([список_параметров]);
```

имя_функции — имя функции, т.е. идентификатор.

```
[класс] тип имя_функции ([список_параметров]);
```

список_параметров — определяет величины, которые требуется передать в функцию при вызове.

- элементы списка разделяются запятыми;
- в списке указываются типы и имена.

Примечание:

При объявлении функции имена в списке параметров, передаваемых в функцию, можно опустить.

§1.2. Определение функций

Определение

Определение функции содержит, кроме объявления, ***тело функции***, представляющее собой последовательность операторов.

- Формат определения (описания) функции:

```
[класс] тип имя ([список_параметров])
{
    тело функции;
}
```

```
[класс] тип имя ([список_параметров])[throw(исключения)]
{
    тело функции;
}
```

исключения — список исключений, которые может порождать функция (т.е. ошибки).

§1.3. Вызов функций

- Формат вызова функций:

```
имя_функции ([список_параметров]);
```

Пример:

```
#include <iostream.h>
// объявление функции:
int sum(int x, int y);                    // <-- Вариант 1
//int sum(int, int);                         <-- Вариант 2
main()
{
    int a = 2, b = 3, c, d;
    c = sum(a, b);                         // вызов функции
    cin >> d;
    cout << sum(c, d);                     // вызов функции
    cout << sum(a*10+5, b+3-a/2);          // вызов функции
    return 0;
}
// определение функции:
int sum(int x, int y)
{
    return (x+y); // <-- Здесь x и y – локальные переменные
}
```

§1.4. Передача параметров в функцию

☝ **Определение**

Параметры, перечисленные в заголовке описания функции, называют ***формальными***, а записанные в операторе вызова функции — ***фактическими*** параметрами, или ***аргументами***.

☝ **Определение**

Список аргументов в функции называется ***сигнатурой функций***.

Примечание:

В определении функции и в ее вызове сигнатуры параметры должны строго совпадать. Это означает, что формальные и фактические параметры должны соответствовать друг другу по ***количеству***, ***типу*** и ***порядку следования***.

Пример:

```
float Avg(float a, float b, float c)
{ float S;              // локальная переменная
S=(a+b+c)/3.;
return S;}              // тип совпадает с типом функции
void main(void){
float x1=2.5, x2=7, x3=3.5; float y;
y=Avg(x1, x2, x3);          // фактические параметры – переменные
// обращение в присваивании
printf("x1=%f, x2=%f, x3=%f y=%f\n", x1, x2, x3, y);
y=Avg(2., 4., 7.);//фактические параметры–константы вещественного типа
printf("x1=%f, x2=%f, x3=%f y=%f\n", 2., 4., 7., y);
// фактические параметры – выражения
y=Avg(x1*2., x2+4., sin(PI/2.));
printf("x1=%f,x2=%f,x3=%f,y=%f\n",2*x1,x2+4.,sin(PI/2.),y);
// обращение в функции вывода, фактические параметры
// произвольные, то есть константы, переменные, выражения
printf("x1=%f,x2=%f,x3=%f,y=%f\n",2.,x2,x3+0.7,Avg(2.,x3+0.7));
// оператор-обращение может входить в другие выражения
y=(Avg(0.5, 1.7, 2.9)+Avg(x1,x1+2,x1+2.))*0.5;
printf("y=%f\n",y); }
```

Способы передачи параметров в функцию

По значению

- В стек заносятся копии значений аргументов, и операторы функции работают с этими копиями;
- Доступа к исходным значениям параметров функции нет, т.е. нет возможности изменить исходные значения аргументов.

По адресу

- В стек заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам;
- Исходные значения аргументов могут быть изменены в ходе выполнения операторов функции.

Пример 1:

```
#include <iostream.h>
void increm(int a, int *b, int &c);      // объявление функции
main()
{
    int a = 1, b = 2, c = 3;
    cout << "a b c\n";
    cout << a << ' ' << b << ' ' << c << '\n';
    increm(a, &b, c);                    // вызов функции
    cout << a << ' ' << b << ' ' << c << '\n';
    return 0;
}

void increm(int a, int *b, int &c)       // определение
  функции
{
    a++; (*b)++; c++;
}
```

```
a b c
1 2 3
1 3 4
```

Пример 2:

```
#include <stdio.h>
#include <math.h>
int Triangle(float a, float b, float c, float &p, float &s)
{
// функция имеет два варианта выхода
// параметры a, b, c передаются по значению
// параметры p, s, по ссылке
float pp; // полупериметр
if(a+b<=c||a+c<=b||b+c<=a) // треугольник не существует
return 0;
else
{ // треугольник существует
p=a+b+c;
pp=0.5*p;
s=sqrt(pp*(pp-a)*(pp-b)*(pp-c));
return 1;
}
}
```

```
void main(void)
{
float A, B, C;
// длины сторон – фактические параметры
float Perim, Square;
// периметр и площадь – фактические параметры
// пример обращения
printf("Введите длины сторон треугольника\n");
scanf("%f%f%f", &A, &B, &C);
if(Triangle(A, B, C, Perim, Square)==1)
printf("Периметр = %6.2f, площадь = %6.2f\n", Perim,
Square);
else
printf("Треугольник не существует\n");
}
```

§1.5. Передача массивов в функцию

- ✔ При использовании массива в качестве параметра функции, в функцию передается указатель на тип элемента массива, а точнее — на первый элемент массива.

Пример:

```
/* Передача одномерного массива в функцию */
#include <iostream.h>
int sum(const int *mas, const int n);   //объявление функции
const int n=10;
main()
{
    int marks[n] = {1, 2, 3, 4, 5, 6};
    cout << "Сумма эл-тов массива: " << sum(marks, n);
    return 0;
}
int sum(const int *mas, const int n)    //определение функции
{   // или int sum(int mas[], int n)
    // или int sum(int mas[n], int n)
    int s=0;
    for (int i = 0; i < n; i++) s += mas[i];
    return s;
}
```

Результат выполнения программы:

```
Сумма эл-тов массива: 21
```

Пример:

```
/* Передача двумерного массива в функцию */
#include <iostream.h>
int sum(int **a, const int n_str, const int n_slb);          // 1
// или int sum(int **, const int, const int);
main(){
    int n_str, n_slb;
    int **a, i, j;
    cout << "Кол-во строк: "; cin >> n_str;                   // 2
    cout << "Кол-во столбцов: "; cin >> n_slb;                // 2
    a = new int *[n_str];                                     // 3
    for (i = 0; i < n_str; i++) a[i] = new int [n_slb];       // 3
    for (i = 0; i < n_str; i++)                               // 4
        for (j = 0; j < n_slb; j++){                          // 4
            cout << "a[" << i << ", " << j << "] = ";         // 4
            cin >> a[i][j]; }                                 // 4
    cout << "Сумма эл-тов массива: "<< sum(a, n_str, n_slb); // 5
    ...
    return 0;
}
int sum(int **a, const int n_str, const int n_slb){           // 6
    for (int s = 0, i = 0; i < n_str; i++)                    // 7
        for (int j = 0; j < n_slb; j++) s += a[i][j];         // 7
    return s; }                                               // 8
```

Результат выполнения программы:

```
Кол-во строк: 2
Кол-во столбцов: 3
a[0, 0] = 1
a[0, 1] = 2
a[0, 2] = 3
a[1, 0] = 4
a[1, 1] = 5
a[1, 2] = 6
Сумма эл-тов массива: 21
```

§1.6. Функции с переменным числом параметров

- ✔ В функциях с переменным числом параметров список формальных параметров заканчивается троеточием.
- ✔ Для доступа к необязательным параметрам внутри функции используются следующие макросы библиотеки **`<stdarg.h>`**:
  - **`va_start`** — инициализирует указатель;
  - **`va_list`** — хранит указатель на очередной аргумент;
  - **`va_arg`** — возвращает значение очередного аргумента;
  - **`va_end`** — после перебора всех элементов необходимо обратиться к этому макросу до выхода из функции.

Пример:

```
/* Функция с переменным числом параметров */
#include <stdio.h>
#include <stdarg.h>
main(){
    int n;
    int sred_znach(int,...);       // объявление функции
    n = sred_znach(2,3,4,-1);      // вызов с четырьмя параметрами
    printf("n = %d",n);
    n = sred_znach(5,6,7,8,9,-1); // вызов с шестью параметрами
    printf("\n n = %d",n);
    return 0; }
int sred_znach(int x,...){       // определение функции
    int i = 0, j = 0, sum = 0;
    va_list uk_arg;              // объявление указателя uk_arg на эл-ты
                                 // списка необязательных параметров
    va_start(uk_arg,x);          // установка указателя uk_arg на
                                 // первый необязательный параметр x
    /* выборка очередного параметра и проверка на конец списка: */
    while ((i = va_arg(uk_arg,int)) != -1){
        sum += i;                // сумма эл-тов
        j++; }                   // кол-во эл-тов
    va_end(uk_arg);              // закрытие списка параметров
    return (sum / j); }          // передача сред. знач. в точку вызова
```

Результат выполнения программы:

```
n = 3
n = 7
```

Примечание к примеру:

A. Функция в данном примере имеет один обязательный параметр, который обязательно должен быть равен ‘**–1**’ и является признаком (флагом) окончания списка передаваемых в функцию параметров.

§1.7. Рекурсивные функции

Определение

Рекурсивная функция — функция, вызывающая сама себя.

Хвостовой или концевой рекурсией называется функция, рекурсивный вызов которой производится в конце , непосредственно перед оператором **return** или в самом операторе **return**.

Замечание!

В коде обязательно должна быть предусмотрена проверка условия завершения рекурсии.

Рекурсивная функция должна иметь хоть одну нерекурсивную ветвь, заканчивающуюся оператором возврата.

Пример:

```
/* вычисление факториала (n!): */
long factorial(long n){
    if (n == 0 || n == 1) return 1;
    return (n * factorial(n - 1)); } // n!=n*(n-1)! (5!=5*4!)
```

Примечания:

A. Рекурсия часто может применяться там, где используется цикл.
B. Рекурсивные функции позволяют получить компактный программный код.
C. Расход памяти, времени и опасность переполнения стека требуют внимательности при использовании рекурсивных функций.

§1.8. Функция **main()**

Примечания:

A. Функции **main** передается управление после запуска программы.
B. Функция **main** может передавать значение в вызывавшую ее систему и принимать параметры из внешнего окружения.

- Существует два формата записи функции **main**:

```
тип main () {...}                          //без параметров
```

```
тип main (int argc, char *argv[]) {...} //с двумя пар-ми
```

Примечания:

Функция* main *без параметров

В «**типе**» описывается информационный поток, поступающий из функции обратно к той функции, которая произвела к ней обращение.
В «**списке параметров**» описывается информационный поток, который поступает из функции, производящей обращение, к вызываемой функции.
В заголовке функции **main** описывается интерфейс между функцией и ОС.

Примечания:

Функция main *с двумя параметрами*

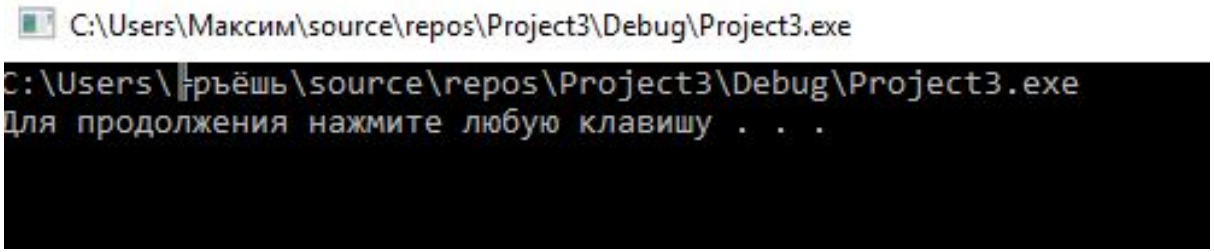
A. Идентификаторы параметров могут быть любыми, но общепринятыми являются указанные ниже:
   - **`argc`** — определяет количество параметров, передаваемых функции;
   - **`argv`** — является указателем на массив указателей типа **`char`**; каждый элемент массива содержит указатель на отдельный параметр командной строки, хранящейся в виде строки.

B. Указатели массива **`argv[]`** ссылаются на полное имя запускаемого на исполнение файла;

Пример:

```
#include <iostream.h>
void main (int argc, char *argv[])
{
    for (int i = 0; i < argc; i++)
    {
        cout << argv[i] << "\n";
    }
}
```

Результат выполнения программы:



§1.9. Перегрузка (полиморфизм) функций

Определение

Использование нескольких функций с одним и тем же именем, но с различными типами параметров называется ***перегрузкой функций***.

Примечания:

A. Компилятор по типу и количеству фактических параметров (аргументов) определяет, какую именно функцию следует вызывать.
B. Если соответствие на одном этапе получается более чем одним способом, то возникает неоднозначность вызова.
C. Избежать неоднозначности можно используя явное преобразование типов аргументов.

Определение

Список аргументов в функции называется ***сигнатурой функций***.

Примечания:

A. Имена переменных не имеют значения.
B. Сигнатуры могут различаться по количеству аргументов или по их типам, либо по тому и другому.

Пример:

```
/* Перегрузка функций */
...
/*Имеется набор функций print( ) со следующими прототипами:*/

void print(const char * str, int width);      // #1
void print(double d, int width);        // #2
void print(long l, int width);          // #3
void print(int i, int width);           // #4
void print(const char * str);           // #5

print("Pancakes", 15);
print("Syrup");
print(1999.0, 10);
print(1999, 12);
print(1999L, 15);
```

Пример:

```
/* Перегрузка функций */
...
/* Имеется 4-е варианта функции, определяющей наибольшее
   значение : */

int max(int, int);          // (1) наибольшее из двух целых
char* max(char*, char*); // (2) наиболее длинная из двух
  строк
int max(int, char*);        // (3) наибольшее из первого
                            //     и длины второго
int max(char*, int);        // (4) наибольшее из длины первого
                            //     и второго
...
void func(int a, int b, char* c, char* d)
{
    cout << max(a,b) << max(d,b) << max(a,c) << max(c,d);
} ...
```

Примечания к примеру:

A. Выбор варианта функции **`max`** осуществляется на этапе компиляции программы в соответствии с типом фактических параметров (аргументов).

B. В данном примере все четыре варианта функции вызываются последовательно.

Пример:

```
/* Перегрузка функций */
/*функция, которая возвращает наибольшее из своих параметров*/

int max(int, int); // функция получает два параметра
int max(int, int, int); // функция получает три параметра

int max(int x,int y)
{return x>y?x:y;
}

int max(int x,int y,int z)
{return x>y?(x>z?x:z):(y>z?y:z);
}

int Angle1, Angle2, Angle3;
// вызов функции с двумя параметрами
int Angle_max=max(Angle1, Angle2);
// вызов функции с тремя параметрами
Angle_max=max(Angle1, Angle2, Angle3);
```

Пример:

```
/* Перегрузка функций */
/* Функцию max() находит наибольшее значение из элементов
  мас-
сива.*/
int max(int *,int); // функция получает два параметра
int max(int x[],int n)
{
  int m=x[0];
  for(int i=0;i<n;i++)
  if(x[i]>m) m=x[i];
  return m;
}
int a[10];
int Angle_max;
Angle_max=max(Angle1,Angle2); // найдет наибольший угол
int Max_value;
Max_value=max(a, 10); // найдет наибольший элемент массива
```

Пример:

```
/* Перегрузка функций (неоднозначность) */
#include <iostream.h>
float  f(float i)  {
    cout << "Function float f(float i)" << endl;
    return i;      }
double f(double i) {
    cout << "Function double f(double i)" << endl;
    return i * 2;  }
int main()
{
    float  x = 10.09;
    double y = 10.09;
    cout << f(x)  << endl;   // вызывается f(float)
    cout << f(y)  << endl;   // вызывается f(double)
    cout << f(10) << endl;   // возникает неоднозначность
                             // преобразовать 10 во float
    return 0;                // или double
}
```

Примечания к примеру:

A. При вызове функции **`f(10)`** возникает неоднозначность и компилятор выдает ошибку:

```
cout << f(y)  << endl; // вызывается f(double)
cout << f(10) << endl; // возникает неоднозначность
                       // преобразовать 10 во float или double
return
}
```

Message

Compiling AMBIGUIT.CPP:

Error AMBIGUIT.CPP 16: Ambiguity between 'f(float)' and 'f(double)'

B. Для устранения неоднозначности в примере требуется ’10’ преобразовать явным образом к требуемому типу: **`f(float(10))`** или

```
cout << f(x)  << endl; // вызывается f(float)
cout << f(y)  << endl; // вызывается f(double)
cout << f(float(10)) << endl;
```

(Inactive E:\WINPRIL\BORLANDC\BIN\AMBIGUIT.EXE)

```
Function float f(float i)
10.09
Function double f(double i)
20.18
Function float f(float i)
10
```

☐ *Замечания! (правила описания перегруженных функций)*

A. Перегруженные функции должны находиться в одной области видимости, иначе произойдет сокрытие функций аналогично одинаковым именам переменных во вложенных блоках (в операторах, функциях и т.п.).

B. Функции не могут быть перегружены, если описание их параметров отличается только модификатором **`const`** или использованием ссылки:

- **`int`** и **`const int`**,

  или

- **`int`** и **`int&`**.

§1.10. Шаблоны функций

✔ Шаблоны функций предназначены:

- для повышения эффективности программы, содержащей однотипную обработку данных различных типов;
- чтобы не возникало необходимости создавать несколько перегружаемых функций с полностью одинаковыми алгоритмами обработки данных.

☝ **Определение**

Шаблон функции — это программно расписанный алгоритм, применяемый к данным различных типов.

Шаблон определяет функцию на основе обобщенного типа, вместо которого может быть подставлен определенный тип данных.

Передавая ***шаблону*** тип в качестве параметра, можно заставить компилятор сгенерировать функцию для этого типа данных.

Определение

Поскольку шаблоны позволяют программировать
на основе обобщенного типа вместо определенного типа данных, этот процесс называют ***обобщенным программированием.***

Определение

Поскольку типы представлены параметрами, шаблоны функций иногда называют ***параметризированными типами.***

Определение

Вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции. Этот процесс называется ***созданием экземпляра*** (***instantiation***) ***шаблона***.

Формат определения шаблона функции:

```
template <class (или typename) список_типов>
имя_функции(список параметров)
{
    /* тело функции */
}
```

список_типов — список обобщенных типов, которые определяются при вызове функции.

список параметров — список параметров, передаваемых в функцию.

Примечания:

A. Конкретный тип данных передается функции в виде параметров на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному в функцию типу, т.е. создается функция, которая автоматически перегружает сама себя.

B. В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только типом, но и просто переменной:

```
template <class TypeA, class TypeB>
    void f(TypeA x, TypeB y, int i) {...}
```

C. При вызове шаблона функции на месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение.

D. Применение шаблонов функций не сокращает размер исполняемого кода.

Пример:

```
/* Сортировка целочисленного и вещественного массивов */
#include <iostream.h>
template <class Type> void SortVybor(Type *b, int n);
main() { const int n=20;
         int i, b[n];
         double a[n];
         for (i = 1; i < n; i++) cin >> b[i];
         SortVybor(b, n); // сорт-ка целочисленного массива
         for (i = 1; i < n; i++) cin >> a[i];
         SortVybor(a, n); // сорт-ка вещественного массива
         return 0; }
/* Шаблон функции сортировки методом прямого выбора : */
template <class Type> void SortVybor(Type *b, int n){
    Type a;              // буферная переменная для обмена эл-тов
    int imin;            // номер наименьш. эл-та в исходн. масс.
    for (int i = 0; i < n-1; i++) {
   for (int j = i+1; j < n; j++)              // выбор
            if (b[j] < b[i])               // наименьшего
        a = b[j]; b[j] = b[i]; b[i] = a; } }   // и обмен
```

Пример:

```
/* Принудительное определение типа параметров функции */

// Шаблон функции определен как
template <class X, class Y, class Z> void Func(Y, Z);

// Вызовы функции:
Func <int, char*, float> (“University”, 3.3);          // 1
Func <int, char*> (“University”, 3.3);                 // 2
Func <int> (“University”, 3.3);                        // 3
Func (“University”, 3.3);                              // 4
```

Примечания к примеру:

1. Все определено программистом.
2. Принудительно определены **`X`** и **`Y`**.
   Параметр **`Z`** определен автоматически как **`double`**.
3. Принудительно определен **`X`**.
   Автоматически определены параметры **`Y`** — **`char*`** и **`Z`** — **`double`**.
4. ОШИБКА: невозможно определить тип **`X`**.