Введение в язык C++

Углич П. С.[1]

[1]Южный федеральный университет

Лекция 3

Outline

Шаблоны функций

Шаблоны функций похожи на шаблоны классов, но определяют семейство функций.С помощью шаблонов функций можно задавать наборы функций, основанных на одном коде, но действующих в разных типах или классах. Следующий шаблон функции меняет местами два элемента.

```
template <typename T>
void Swap(T a, T b)
{
T r = a;
a = b;
b = r;
} // конец шаблона функции
```

Шаблоны функций

Все шаблоны функций начинаются со слова template, после которого идут угловые скобки, в которых перечисляется список параметров. Каждому параметру должно предшествовать зарезервированное слово class или typename.

```
template <class T>
```

или

```
template <typename T>
```

или

```
template <typename T1, typename T2>
```

Шаблоны функций

Шаблон функции не позволяет разработчику менять местами объекты разных типов, поскольку компилятор во время компиляции знает типы параметров a и b.

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
Swap( j, k );           //OK
Swap( j, Hello );       //error
```

Второй вызов Swap приводит к ошибке времени компиляции

Шаблоны функций

Для шаблона функции можно явно задавать аргументы.
Например:

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
   f<char>(j);   // Generate the specialization f(char).
   // If not explicitly specified, f(int) would be deduced.
}
```

Шаблоны функций

```
template <typename T>
void printArray(const T * array, int count)
{
    for (int ix = 0; ix < count; ix++)
        cout << array[ix] << "   ";
    cout << endl;
} // конец шаблона функции printArray
```

Аргументы по умолчанию

Во многих случаях функции имеют аргументы, которые используются настолько редко, что достаточно значения по умолчанию. В таких случаях возможность задания аргументов по умолчанию позволяет указывать только те аргументы функции, которые важны в конкретном вызове.
Пример:

```
int print( char *s );                    // Print a string.
int print( double dvalue );              // Print a double.
int print( double dvalue, int prec );
```

Аргументы по умолчанию

Во многих приложениях для аргумента prec можно задать разумное значение по умолчанию, что исключает необходимость наличия двух функций:

```
int print( char *s );
int print( double dvalue, int prec=2 );
```

При использовании аргументов по умолчанию обратите внимание на следующие моменты:

- Они должны быть последними аргументами.Поэтому следующий код недопустим:

  ```
  int print( double dvalue = 0.0, int prec );
  ```

- Переопределение аргумента по умолчанию в последующих объявлениях не допускается, даже если оно совпадает с оригиналом.

Аргументы по умолчанию

Следующий код недопустим

```
// Prototype for print function.
int print( double dvalue, int prec = 2 );

...

// Definition for print function.
int print( double dvalue, int prec = 2 )
{
...
}
```

В последующих определениях допускается добавлять дополнительные аргументы по умолчанию.

Локальные и глобальные переменные

Обычно глобальные переменные объявляются перед главной функцией, но можно объявлять и после функции main(), но тогда данная переменная не будет доступна в функции main().

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void example();
int variable = 48; // инициализация глобальной переменной
int main(int argc, char* argv[])
{       int variable = 12; // инициализация локальной перемен
        cout << "local  variable = " << variable << endl;
        example(); // запуск функции
        return 0; }
void example()
{        cout << "global variable = " << variable << endl; }
```

Локальные и глобальные переменные

В C++ существует такая операция, как разрешение области действия ::. Эта операция позволяет обращаться к глобальной переменной из любого места программы.

```
cout << "local  variable = " << ::variable << endl; /
```

Операция разрешения области действия ставится перед именем глобальной переменной, и даже, если есть локальная переменная с таким же именем, программа будет работать со значением, содержащимся в глобальной переменной.

Заголовочные файлы

Программа на языке C++ состоит из одной или нескольких функций, объявлений и определений глобальных объектов, подключения пространства имен и заголовочных файлов.
Среди функций должна быть одна и только одна с именем main().
Все функции, составляющие программу, могут быть расположены в одном файле — исходном модуле.
Реальные программы на C++, как правило, состоят из нескольких исходных модулей.
В случае вынесения функций в отдельный cpp-файл следует создавать соответствующий ему h-файл, содержащий объявления соответствующих функций.
Рекомендуется, чтобы имена cpp-файла и соответствующего ему h-файла совпадали.
Основное правило рекомендует ограничиться одними объявлениями.

Заголовочные файлы

```
typedef int(*IFnI)(int);
void my_map(int *a, int * b, int size, IFnI f);
int F1(int x);
int F2(int x);

template <typename T>
void Swap(T a, T b)
{
T r = a;
a = b;
b = r;
} // конец шаблона функции
```

Заголовочные файлы

Может оказаться, что заголовочный файл многократно включается в сложную программу.
В результате возникают ошибки, связанные с многократным определением. (Объявлений может быть сколько угодно, определение может быть только одно). В каждом заголовочном файле следует проверить, не был ли этот файл уже включен в проект. Это делается при помощи препроцессорного флага.

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
//
необходимые
объявления
#endif    // HEADER_FLAG
```

Заголовочные файлы

В качестве имени препроцессорного флага рекомендуется записать имя заголовочного файла символами верхнего регистра и заменить точку символом подчеркивания

Пространства имен

Пространство имен — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, когда база кода включает несколько библиотек.

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Пространства имен

Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут получить доступ к членам, используя полное имя идентификатора, например

```
 std::vector<std::string> vec;

ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

Пространства имен

using для отдельного идентификатора

```
using std::string
```

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

using (C++) для всех идентификаторов в пространстве имен

```
using namespace std;
```

Код в файлах заголовков всегда должен содержать полное имя в пространстве имен.

Как правило, пространство имен объявляется в файле заголовка. Если реализации функций находятся в отдельном файле, определяйте имена функций полностью, как показано в следующем примере.

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

Реализации функций в файле contosodata.cpp должны использовать полные имена, даже если вы укажете директиву using в начале файла:

```
#include "contosodata.h"
using namespace ContosoDataServer;

void ContosoDataServer::Foo()
{
   //no qualification because using directive above
   Bar();
}
int ContosoDataServer::Bar(){return 0;}
```

Пространство имен может быть объявлено в нескольких блоках в одном файле и в нескольких файлах.

```
// defining_namespace_members.cpp
// C2039 expected
namespace V {
        void f();
    }
    void V::f() { }          // ok
    void V::g() { }          // C2039, g() is not yet
                             // a member of V
    namespace V {
        void g();
    }
}
```

Глобальное пространство имен

Если идентификатор не объявлен явно в пространстве имен, он неявно считается входящим в глобальное пространство имен. Без особой необходимости старайтесь не делать объявлений в глобальной области, если это возможно. Исключением является точка входа функция main, которая должна быть в глобальном пространстве имен. Чтобы явно указать глобальный идентификатор, используйте оператор разрешения области видимости без имени, как сделано в

```
::SomeFunction(x);
```

Вложенные пространства имен

```
namespace ContosoDataServer
{
    void Foo();
    namespace Details
    {
        int CountImpl;
        void Ban() { return Foo(); }
    }
    int Bar(){...};
    int Baz(int i) { return Details::CountImpl; }
}
```

Указатели

Указатель хранит адрес элемента данных и ему известен тип хранимых данных, т. е. способ интерпретации данных при доступе к памяти.
Объявление указателя

```
<тип> * <имя>;
```

Инициализация указателя возможна адресом существующей переменной соответствующего типа с помощью операции взятия адреса (&), значением другого указателя или адресом новой переменной, размещаемой в памяти с помощью операции new:

```
int *p, *q, *t;
int a=0;
p=&a;
t=p;
q=new int;
```

Доступ к данным через указатель осуществляется с помощью операции разыменования (*):

```
cout<<*p;
```

Оператор объявления массива должен определять тип элементов в массиве, имя массива и количество элементов в массиве:

```
<тип> <имя>[<размерность>];
```

Доступ к элементам массива производится с помощью операции индексации [ ]. В функции передавать массив можно следующим образом:

```
int check_null(int a[], int n)
```

В C++ при передаче массивов в качестве параметров функции передается адрес первого элемента массива. Функции могут изменять значения элементов массива.
Для предотвращения модификации исходного массива внутри тела функции можно в определении функции применять к параметру-массиву спецификатор типа const.

Функции не должны модифицировать массивы без крайней необхо димости (принцип наименьших привилегий):

```
int check_null(const int a[], int n)
```

Другой способ передачи массивов в качестве параметров в функции связан с использованием указателей:

```
int check_null(const int *a, int n)
```

Динамические массивы

Если массив создается путем объявления, то память под него будет отведена перед началом выполнения программы или функции и будет закреплена за массивом до завершения времени жизни массива. Оператор new позволяет выделять память под массив во время выполнения программы, причем указывая его размер не «по максимуму», а необходимый при конкретном выполнении программы.

```
int *a, n;
cout<<"Input the array size: "<<endl;
cin>>n;
a=new int[n];
for (int i = 0; i<n;i++)
cin>>a[i];
```

Для освобождения памяти применяется оператор delete;

```
delete[] a;
```

Динамические массивы

Неважно, как создан массив, имя массива – это адрес (указатель) первого по счету элемента массива с индексом [0].
Над указателями определены: адресная арифметика, операции присваивания и сравнения. Если к указателю прибавляется число, то это означает, что указатель сдвигается вправона такое количество байт, какое занимает указанное число элементов заданного типа.
Указатели можно сравнивать на равенство/неравенство. Если имеется два указателя на один массив, то разность этих указателей будет равна количеству элементов массива, расположенных между этими указателями.
Выражение a++ ссылается на элемент массива с индексом 1, так же как и выражение &a[1]. Но a++ изменяет значение указателя a в отличие от &a[1].

Динамические массивы

```
bool simm( int *a, int n ) {
    int *b=a+n-1;
    while (a<=b)
        if (*a++!=*b--)
            return false;
    return true;
}
```

Традиционный способ передачи параметров-массивов в функции заключается в передаче указателя на начало массива и размера массива. Существует еще один метод предоставления необходимой информации функции — указание диапазона элементов массива. Это осуществляется путем передачи двух указателей: один определяет начало диапазона элементов массива, другой — его конец.

Динамические массивы

Примеры:

```
bool simm(int *a, int *b) {
while (a <= b)
if (*a++ != *b--)
return false;
return true;
}
```

Динамические массивы

```
int SUM(int *begin, int *end) {
int sum = 0;
while (begin <= end)
{
sum += *begin;
begin++;
}
return sum;
}
```

Двумерные массивы в динамической памяти

Если массив будет содержать целые значения, то описание выглядит следующим образом:

```
int **a;
```

Сначала память выделяется для указателей на соответствующие одномерные массивы (строки матрицы):

```
a=new int *[n];
```

После этого выделяется память для каждой строки:

```
for (int i=0; i<n; i++)
     a[i]=new int[m];
```