Лекция 11

Шаблоны с языке C++

Универсальный алгоритм

Универсальным алгоритмом будем называть алгоритм, не привязанный к определенному типу входных и выходных данных, т.е. одинаково хорошо работающий с данными разного типа.

Например, любой из *алгоритмов сортировки* (выбором, обменом и т.д.) может применяться к массивам разного типа (int, float, double, и др.).

Универсальный алгоритм может быть реализован в виде функции C++, однако для каждого типа входных данных необходимо будет определить свою собственную функцию, то есть перегрузить функцию несколько раз.

```
void bubble_sort(int data[], int n)
{
 ...
}
void bubble_sort(float data[], int n)
{
 ...
}
void bubble_sort(my_class data[], int n)
{
 ...
}
```

Такой способ реализации допустим в C++, но его нельзя считать удобным.

Решение 1: псевдоним типа (typedef)

```
typedef T int;
T summa(T array[], int size)
{
  T res = 0;
  for(int i=0; i<size; i++)
    res += array[i];
  return res;
}
```

Такая функция может работать с любыми числовыми данными, но

1) требует переопределения псевдонима T,
2) не может применяться одновременно к двум типам.

Решение 2: шаблон функции C++

Универсальный алгоритм в C++ может быть реализован с помощью **шаблона функции**.

Определение шаблона похоже на определение обычной функции, но при этом используют ключевое слово **`template`**, вслед за которым указывают аргументы в угловых скобках.

Формальные параметры шаблона

1. перечисляются через запятую
2. могут быть именами типов, именами других шаблонов и именами объектов.

Параметр-тип описывается с помощью ключевых слов **`class`** или **`typename`**.

Определение шаблона функции

Синтаксис определения

```
template <параметры_шаблона>
тип имя_функции(список_аргументов)
{
    операторы;
};
```

В теле шаблона функции могут использоваться и формальные аргументы функции, и формальные параметры шаблона.

Пример: шаблон функции для расчета суммы элементов массива

```
template <class T>
T summa(T array[], int size)
{
    T res = 0;
    for (int i=0; i<size; i++)
       res += array[i];
    return res;
}
```

Вызов функции-шаблона

При вызове функции-шаблона, после ее имени указывают *фактические параметры шаблона* в угловых скобках, а затем *фактические аргументы функции* в круглых скобках.

```
имя <факт_параметры>(факт_аргументы);
```

Пример:

```
int iarray[10];
int i_sum;
//...
i_sum = summa<int>(iarray, 10);
```

Объект – параметр шаблона

Шаблоны можно *параметризовать* не только именами типов, но и объектами, поэтому аргумент size можно указать в списке параметров шаблона

```
template <class T, int size>
T summa (T array[])
{
    ...
}
```

объявление

вызов

```
i_sum = summa <int,10>(iarray);
```

Параметрами шаблона могут быть

- параметр-тип (имя типа – int, float, string и т.д.)
- параметр-шаблон (имя другого шаблона)
- параметр-объект или параметр-переменная

Параметром-переменной могут быть переменные следующих типов

✔интегральный (целые, символьные и логические)

✔перечислимый,

✔указатель на объект любого типа или на функцию

✔ссылка на объект любого типа или на функцию

✔указатель на член класса

Параметром-переменной в шаблоне не может быть:

- void
- пользовательский тип
- вещественный тип (float, double)

Примеры

```
template<class d> class X{ ... };     // OK

template<double d> class X{ ... };    // ошибка*

template<double* d> class X{ ... };   // OK

template<double& d> class X{ ... };   // OK
```

Шаблоны и параметрический полиморфизм

Шаблон в языке C++ реализует принципы **параметрического полиморфизма**, то есть возможности применять один и тот же алгоритм для разных типов данных.

Шаблоны широко используются при разработке библиотек. Стандартная библиотека шаблонов (Standard Template Library, STL) – пример такой библиотеки.

Универсальные алгоритмы STL – `find`, `replace`, `shuffle`, `accumulate`, `merge`, и др.

Специализация шаблона

Если к какому-либо конкретному типу аргументов шаблонный алгоритм неприменим, то можно определить обычную функцию, список типов аргументов и возвращаемого значения которой соответствуют объявлению шаблона.

Функция, перегружающая шаблон, называется **специализацией шаблонной функции**.

Пример: алгоритм суммирования в применении к массиву символов (текстовой строке)

Выведение типов

Иногда при вызове шаблона функции после имени можно не указывать фактические параметры в угловых скобках. В этом случае компилятор сам определит параметры шаблона по типу параметров функции.

Эта особенность называется **выведением типов аргументов** шаблона.

```
int iarray[10];
int i_sum;
//...
i_sum = summa(iarray, 10);
```

Преобразование типов

При выведении типов могут применяться только преобразования точного отождествления, то есть:

1) точное совпадение
2) совпадение с точностью до typedef
3) тривиальные преобразования:

- `T[] <-> T*`
- `T  <-> T&`
- `T  -> const T`

Пример преобразования типов

```
template <class T>
T max(T t1, T t2){...}

int main()
{
   max(1,2);          // max<int>(1,2);
   max('a','b');      // max<char>('a','b');
   max(2.7, 4.9);     // max<double>(2.7, 4.9);
   //max('a',1);      // ошибка – неоднозначность,
                      // станд. преобразования не
                      // допускаются
   //max(2.5,4);      // ошибка – неоднозначность,
                      // станд. преобразования не
                      // допускаются
}
```

Устранение неоднозначности

Неоднозначности не возникают при использовании явного квалификатора типа шаблона

```
max <int> ('a',1);
max <double> (2.5,4);
```

Универсальные контейнеры

Универсальным контейнером будем называть программную структуру, которая может использоваться для хранения данных разных типов.

Такие контейнеры также могут быть реализованы с помощью шаблонов C++. Однако в этом случае создается не шаблон функций, а шаблон целого класса.

Пример: контейнер vector для хранения коллекции однотипных данных (аналог массива). В контейнере vector могут храниться данные любого типа (int, float, double, char, и др.). Содержится в библиотеке STL.

Объявление шаблона класса

Синтаксис объявления

```
template <параметры_шаблона>
class имя_класса
{
    поля и методы;
};
```

В объявлении класса-шаблона используются формальные параметры шаблона.

Создание объекта

Создание объекта шаблонного класса называется **инстанцированием шаблона**.

Синтаксис инстанцирования

```
имя_класса <параметры_шаблона> объект;
```

Пример: пара чисел

```
template <typename T1, typename T2>
class pair
{
  private:
   T1 data1;
   T2 data2;
  public:
   pair(T1 d1, T2 d2): data1(d1),data2(d2){};
   void print()
   {
      cout << data1 << ”,” << data2 << endl;
   }
}
```

Примеры инстанцирования шаблона класса pair

```
pair <int, char> c1(5, 'S');
pair <float, bool> c2(0.39, false);
c1.print();
c2.print();
```

Для корректного функционирования шаблона с конкретными типами T1 и T2 необходимо, чтобы для этих типов были определены процедуры инициализации (`data1(d2)`) и вывода объекта в поток (`ostream << data1`).

Пример 2: шаблон массива

Задача: создать шаблон класса array, позволяющего хранить данные любого типа. Предусмотреть следующие возможности:

1) доступ к элементам по индексу
2) средства контроля выхода за пределы
3) массив произвольной длины

В качестве параметров шаблона будем использовать тип элементов T и количество элементов n.

```
#include <iostream>
#include <exception>

template <class T, int n>
class array
{
  public:
    array();
    ~array();
    T& operator[](int k);
  private:
    T *data;
};
```

```
template <class T, int n>
array<T,n>::array()
{
   data = new T[n];
}

template <class T, int n>
array<T,n>::~array()
{
   delete []data;
}
```

```
template <class T, int n>
T& array<T,n>::operator[](int k)
{
   if( k<0 || k>=n )
     throw std::out_of_range("Out!");
   return data[k];
}

template <class T, int n>
std::ostream& operator <<
   (std::ostream& os, array<T,n>& obj)
{
  os << "\nСодержимое массива:";
  for(int i=0; i<n; i++)
    os << endl << obj[i];
  return os;
}
```

```
int main()
{
    setlocale(LC_ALL, "RUS");
    array<float, 20> x;

    for(int i=0; i<20; i++)
      x[i] = 10.1*i+1.15;

    std::cout << x;
    std::cin.get();

    return 0;
}
```