# C++ второй семестр

Казначеев Дмитрий, Купоросов Василий, Недиков Константин

18 февраля 2017 г.

## Содержание

# 1. Шаблоны

## 1.1. Шаблонные классы

Пусть мы написали свой класс, хранящий внутри себя объекты какого-то типа.

```
class MyArray {
private:
  int *array;
};

class Scoped_ptr {
  GaussNumber *ptr;
};
```

Но иногда нам хочется завести вектор или умный указатель другого типа. Приходится вручную переписывать весь класс. Как такую проблему решать в языке си?
Используем препроцессор. Обратите внимание на слэши в конце строк. Они нужны для многострочного дефайна, ## – для склеивания MyArray_ и типа, подставленного на место TYPE.

MyArray.h:

```
#define MyArray(TYPE) class MyArray_ ## TYPE { \
private:                                       \
  size_t size;                                 \
  TYPE *array;                                 \
public:                                        \
  TYPE get(size_t index) {                     \
    return array[index];                       \
  }                                            \
};
```

main.c:

```
#include "MyArray.h"
MyArray(int);      // подставляем определение класса с интами
MyArray(double);   // подставляем определение класса с даблами
int main() {
  MyArray_int a;
  MyArray_double b;
}
```

Если хочется посмотреть, как изменился код после препроцессора, можно запустить gcc с ключем -E.

Рассмотрим особенности данного подхода.

1) Все методы класса должны быть inline, потому что, если класс объявлен в разных файлах, то на стадии линковки будет double definition. Если реализация метода находится внутри класса, то она автоматически станет inline.

2) Препроцессор человек простой. Видит строку – заменяет её. Поэтому возникают спецэффекты. В приведенном ниже коде название переменной TYPE заменится на int.

```
#define MyArray(TYPE) . . .
MyArray(int);
MyArray_int arr;
int TYPE;
```

3) Нельзя делать сложные объявления. Следующая строчка у препроцессора получится не лучшим образом.

```
MyArray(MyArray(int)) arr;
```

Шаблоны позволяют делать всё то же самое, но на уровне компилятора. Этот стиль называется обобщенное (generic) программирование. Рассмотрим синтаксис.

MyArray.h:

```
template <typename T>
class MyArray {
private:
  size_t size;
  T *array;
public:
  T& get(size_t index) {
    return array[index];
  }
  T& MyArray<T>::operator[](size_t index);
  MyArray<T>& MyArray<T>::operator=(const MyArray<T>& obj);
};

template <typename T> // <class T> эквивалентно <typename T>
T& MyArray<T>::operator[](size_t index) {
  return array[index];
}
```

main.cpp:

```
#include "MyArray.h"
int main() {
  MyArray<int> a;
  MyArray<double> b;
  MyArray<MyArray<int> > arr; // работает всегда
  MyArray<MyArray<int>> arr;  // только в 11 стандарте
}
```

Важно, что здесь не применима идея раздельной компиялации, так как неизвестно заранее, какой тип захочет пользователь. Поэтому реализацию не выносят в отдельный .cpp файл. Таким образом время компилции и объём файлов увеличиваются.

Если взять стандартную библиотеку языка с++, то не будет .so (.dll) файлов (в отличие от библиотек языка си). Это потому что в STL всё написано с шаблонами и не кмпилируетяс заранее.

Класс может иметь несколько шаблонных параметров.

```
template <typename K, typename V>
class TreeItem {
  K key;
  V value;
}
```

## 1.2. Шаблонные функции

По традиции реализацию функций и методов класса всё же выносят в отдельный .h файл. Но еще раз напомню, что раздельная компиляция здесь неприменима.

swap.h:

```
template <typename T>
void swap(T& a, T& b);
```

swap_ind.h:

```
template <typename T>
void swap(T& a, T& b) {
  T t(a);
  a = b;
  b = t;
}
```

Если можно однозначно вывести типы аргументов, то тип для функции можно не указывать.

```
int main() {
  int a = 2;
  int b = 3;
  swap<int> (a, b);
  swap(a, b) // swap<int> выведется автоматически;
}
```

Но это не всегда возможно.

```
template <typename T>
class MyArray {
private:
  size_t size;
  T *array;
public:
  MyArray(size_t s) {
    array = new T[size];
  }
};

MyArray arr(10); // Ошибка. Неизвестен тип хранимых объектов.
MyArray<MyArray<int> > arr(10); // Ошибка. Для внутреннего объекта отсутствует
                                // конструктор по умолчанию.
```

**TODO: Написать несколько вариантов, когда компилятор не может вывести шаблонный параметр функции.**

Рассмотрим пример функции с несколькими шаблонными типами.

```
template <typename T, typename V>
void copy(MyArray<T>& a, MyArray<V>& b) { }

MyArray<int> a;
MyArray<double> b;
copy<int, int>(a, b); // int приведется к double.
copy(a, b); // Типы однозначно выводятся.
copy<int, string>(a, b); // Ошибка. int не приведется к string.
```

Характерный пример. функция сортировки. Рассмотрим три реализации.

1) Процедурное программирование (язык си).

```
typedef int (*compare)(const void *, const void *) func_ptr;
void sort(void *array, size_t n, size_t elem_size, func_ptr cmp);
```

2) ООП.

```
class comparable { //базовый класс для всех сравниваемых типов
  virtual int compare(const comparable *o) const = 0;
};
void sort(comparable **array, size_t n);
```

Чтобы объекты можно было отсортировать, они должны наследоваться от comparable и у них должна быть реализована виртуальная функция compare().

По-сути мы сортируем массив указателей, чтобы внутри сортировки независимо от типа можно было всегда смещаться на одно и то же число байтов. Отсюда берется двойной указатель (**array).

Недостатки:

- Виртуальные функции. Динамическое связывание работает дольше.
- Нельзя на халяву отсортировать инты. Приходится городить новый класс.

3) generic programming (шаблоны):

```
template <typename T>
void sort(T *array, size_t n);

GaussNumber a[100];
Sort<GaussNumber>(a);
```

Здесь нет проблем со смещением, потому что код компилируется с уже известными типами. Каждый класс должен иметь оператор '<'.

Недостатки:

- увеличение время компиляции и работы.
- увеличение размера исполняемого файла (функции для каждого типа).
- нельзя заранее скомпилировать.

# 2. Исключения

**TODO:**

# 3. Стандартная библиотека шаблонов

**TODO:**

# 4. Стандарт C++11

**TODO:**