Шаблоны классов и компиляция программы

2019

Обычные шаблоны

```
// обычный шаблон
template <class X, class Y>
class pair {};

// значение по умолчанию
template <class T, class Allocator = std::allocator<T>>
class vector;

// два значения по умолчанию
template<class Ch, class Tr = std::char_traits<Ch>,
    class Allocator = std::allocator<Ch>> class basic_string;
```

Шаблоны с константами

```
// параметр-константа
template <class T, std::size_t>
class array;

// параметр-константа с произвольным типом
template <class T, T v>
struct integral_constant;

// конкретизация шаблона (instantiation)
typedef integral_constant<bool,true> true_type;
typedef integral_constant<bool,false> false_type;

// шаблон на основе другого шаблона
template <bool B>
using bool_constant = integral_constant<bool, B>;
```

Шаблоны шаблонов

```
// шаблон из шаблона
template <template <class X, class Y> class Container>
struct FloatContainer:
public Container<float,std::allocator<float>> {
    typedef Container<float,std::allocator<float>> base_type;
    // использование конструкторов базового класса
    using base_type::base_type;
};

FloatContainer<std::vector> x(5, 1.f);
FloatContainer<std::list> y{1.f, 2.f, 3.f, 4.f, 5.f};
std::ostream_iterator<float> it(std::cout, " ");
std::copy(x.begin(), x.end(), it); // 1 1 1 1 1
std::copy(y.begin(), y.end(), it); // 1 2 3 4 5
```

Полная специализация

```
template <class T> struct atomic;

template <> struct atomic<int>;  // полная специализация для int
template <> struct atomic<char>; // полная специализация для char

// полная (на самом деле нет) специализация для shared_ptr
template <class T>
struct atomic<std::shared_ptr<T>>;
```

Частичная специализация

```
template <class T, class Deleter = std::default_delete<T>>
class unique_ptr;

// частичная специализация для массивов
template <class T, class Deleter>
class unique_ptr<T[], Deleter>;

template <class T>
class default_delete {
    void operator()(T* ptr) { delete ptr; }
};

// частичная специализация для массивов
template <class T>
class default_delete<T[]> {
    void operator()(T* ptr) { delete[] ptr; }
};
```

- Полная специализация превращает шаблон в класс.
- Частичная специализация превращает шаблон в другой шаблон.

Компиляция без шаблонов

myclass.hh:

```
struct MyClass {
    void mymethod();
};
```

myclass.cc:

```
#include "myclass.hh"
#include <iostream>
void MyClass::mymethod() {
    std::cout << "Hello world\n";
}
```

Команда компиляции:

```
g++ -c myclass.cc -o myclass.o
```

Препроцессор:

```
$ g++ -E -c myclass.cc    # без #include <iostream>
# 1 "myclass.cc"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "myclass.cc"
# 1 "myclass.hh" 1
struct MyClass {
 void mymethod();
};
# 2 "myclass.cc" 2
void MyClass::mymethod() {
 std::cout << "Hello world\n";
}
```

Содержимое объектного файла:

```
$ g++ -c myclass.cc -o myclass.o   # c #include <iostream>
$ nm -C myclass.o
                 U __cxa_atexit
                 U __dso_handle
000000000000005c t _GLOBAL__sub_I__ZN7MyClass8mymethodEv
000000000000001e t __static_initialization_and_destruction_0(...)
0000000000000000 T MyClass::mymethod()
                 U std::ios_base::Init::Init()
                 U std::ios_base::Init::~Init()
                 U std::cout
0000000000000000 r std::piecewise_construct
0000000000000000 b std::__ioinit
                 U std::basic_ostream& std::operator<<(...)
```

Компиляция с шаблоном

myclass.hh:

```
template <class T>
struct MyClass {
    void mymethod();
};
```

myclass.cc:

```
#include "myclass.hh"
#include <iostream>
template <class T>
void MyClass<T>::mymethod() {
    std::cout << "Hello world\n";
}
```

Препроцессор:

```
$ g++ -E -c myclass.cc    # без #include <iostream>
# 1 "myclass.cc"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "myclass.cc"
# 1 "myclass.hh" 1
template <class T>
struct MyClass {
 void mymethod();
};
# 2 "myclass.cc" 2
template <class T>
void MyClass<T>::mymethod() {
 std::cout << "Hello world\n";
}
```

Содержимое объектного файла:

```
$ g++ -c myclass.cc -o myclass.o
$ nm -C myclass.o
                 U __cxa_atexit
                 U __dso_handle
000000000000003e t _GLOBAL__sub_I_myclass.cc
0000000000000000 t __static_initialization_and_destruction_0(...)
                 U std::ios_base::Init::Init()
                 U std::ios_base::Init::~Init()
0000000000000000 r std::piecewise_construct
0000000000000000 b std::__ioinit
```

Добавим конкретизацию шаблона в myclass.cc:

```
template struct MyClass<float>;
```

Содержимое объектного файла:

```
$ g++ -c myclass.cc -o myclass.o
$ nm -C myclass.o
                 U __cxa_atexit
                 U __dso_handle
000000000000003e t _GLOBAL__sub_I_myclass.cc
0000000000000000 t __static_initialization_and_destruction_0(...)
0000000000000000 W MyClass<float>::mymethod()
                 U std::ios_base::Init::Init()
                 U std::ios_base::Init::~Init()
                 U std::cout
0000000000000000 r std::piecewise_construct
0000000000000000 b std::__ioinit
                 U std::basic_ostream& std::operator<<(...)
```

Добавим файл main.cc:

```
#include "myclass.hh"
int main() {
    MyClass<float> x; // конкретизация шаблона
    x.mymethod();
    return 0;
}
```

Содержимое объектного файла:

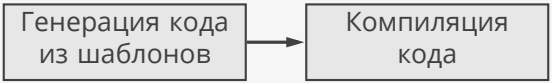
```
$ g++ main.cc -o main.o
$ nm -C main.o
0000000000000000 T main
                 U MyClass<float>::mymethod()
```

Соберем программу целиком:

```
$ g++ main.o myclass.o -o myprog
$ nm -C myprog
                 ...
00000000004006b6 T main
0000000000400724 W MyClass<float>::mymethod()
                 U std::ios_base::Init::Init()@@GLIBCXX_3.4
                 U std::ios_base::Init::~Init()@@GLIBCXX_3.4
0000000000601040 B std::cout@@GLIBCXX_3.4
00000000004007d0 r std::piecewise_construct
                 U std::basic_ostream& std::operator<<(...)
```

- Только конкретизации шаблона появляются в объектном файле.
- Специализация шаблона конкретизацией не является.
- Каждый уникальный набор аргументов шаблона создает новую конкретизацию.

Два этапа компиляции




```
template <class T>
void print(std::vector<T> x) {
    // iterator — тип или шаблон?
    std::vector<T>::iterator first = x.begin(); // ошибка
    typename std::vector<T>::iterator first = x.begin(); // ок
}

std::vector<int> x{1,2,3,4};
std::vector<int>::iterator first = x.begin(); // ок
print(x);
```

Пример: реестр классов

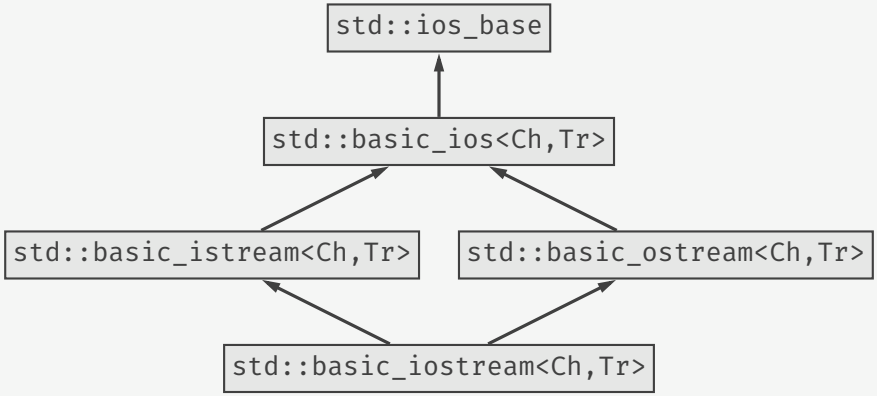
```
template <class BaseType>
struct Registry {
    template <class T> void register_class();
};

class Object {};
class MyObject: public Object {};

template <class BaseType> void
register_my_classes(Registry<BaseType>& reg) {
    reg.register_class<MyObject>(); // ошибка
    reg.template register_class<MyObject>(); // ок
}

Registry<Object> reg; register_my_classes(reg);
```

Шаблоны «раздувают» код!

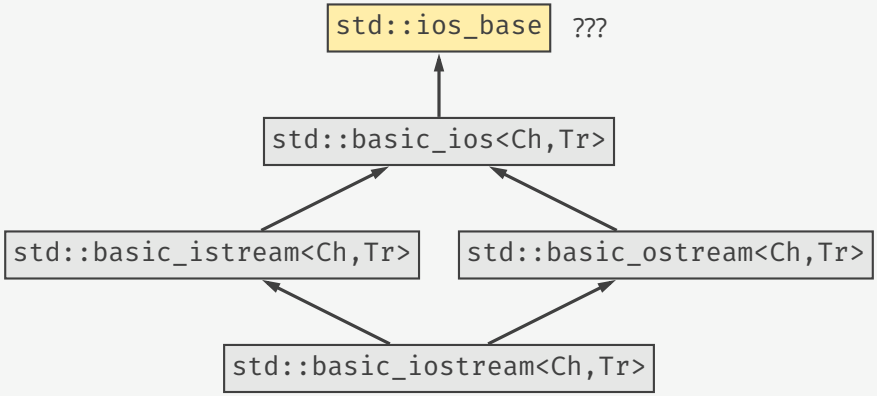
```
template <class T> void myprint() {
    std::vector<T> x(10);
    std::ostream_iterator<T> it(std::cout, "\n");
    std::copy(x.begin(), x.end(), it);
}

myprint<int>();
myprint<float>();
myprint<std::string>();
```

Шаблоны	Команда	Размер, Кб	nm -C	myprint	std::vector
3	g++ -O0	42	239	3	42
3	g++ -O3	14	46	3	
3	g++ -O3 -flto	14	45		
1	g++ -O3 -flto	9	42		

Пример: базовый класс для потоков



Пример: базовый класс для потоков



Вы. Как избавиться от «раздувания» кода?

Ведущий программист. Вынести не зависящие от аргументов шаблона методы за пределы класса-шаблона.

Руководитель команды. Вынести все шаблоны в отдельную библиотеку и сделать их конкретизацию в cc-файлах.

Пример: std::iterator_traits

```
namespace std {
    // объявлеие шаблона
    template <class Iterator>
    struct iterator_traits;

    // частичная специализация для указателей
    template <class T>
    struct iterator_traits<T*> {
        typedef random_access_iterator_tag iterator_category;
        typedef T                          value_type;
        typedef ptrdiff_t                  difference_type;
        typedef T*                         pointer;
        typedef T&                         reference;
    };
}
```

```
namespace My {
    template <class T>
    struct MyIterator { ... };
}
// частичная специализация для MyIterator
namespace std {
    template <class T>
    struct iterator_traits<My::MyIterator<T>> {
        typedef input_iterator_tag iterator_category;
        typedef T                  value_type;
        typedef ptrdiff_t          difference_type;
        typedef T*                 pointer;
        typedef T&                 reference;
    };
}
```

Пример: std::hash

```
namespace std {
    // объявление шаблона
    template <class Iterator>
    struct hash;

    // частичная специализация для примитивных типов
    template <>
    struct hash<int> {
        typedef size_t result_type;
        typedef int argument_type;
        result_type operator()(argument_type x) const {
            return static_cast<result_type>(x);
        }
    };
}
```

```cpp
namespace My {
    struct Person { int id; std::string first_name, last_name; };
}
namespace std {
    // полная специализация для Person
    template <>
    struct hash<My::Person> {
        typedef size_t result_type;
        typedef My::Person argument_type;
        result_type operator()(const argument_type& x) const {
            return static_cast<result_type>(x.id);
        }
    };
}
```

Пример: флаги

Попытка №1:

```
enum class OpenFlag: int {
    Append = 1,
    Truncate = 2,
    ReadOnly = 4,
    WriteOnly = 8
};
OpenFlag flags = OpenFlag::Append | OpenFlag::WriteOnly; // ошибка
```

Попытка №2:

```
OpenFlag operator|(OpenFlag a, OpenFlag b) {
    return OpenFlag(static_cast<int>(a) | static_cast<int>(b));
}
OpenFlag flags = OpenFlag::Append | OpenFlag::WriteOnly; // ок
```

Попытка №3:

```
template <class T>                          // SFINAE
struct flag_traits {};                      // substitution failure
                                            // is not an error
template <class Flag>
auto operator|(Flag a, Flag b) ->
typename flag_traits<Flag>::flag_type {
    typedef typename flag_traits<Flag>::int_type tp;
    return Flag(static_cast<tp>(a) | static_cast<tp>(b));
}

template <>
struct flag_traits<OpenFlag> {
    typedef OpenFlag flag_type;
    typedef int int_type;
};

OpenFlag flags = OpenFlag::Append | OpenFlag::WriteOnly; // ок
```

Вариант с std::enable_if:

```
template <class Flag>
auto operator|(Flag a, Flag b) ->
typename std::enable_if<is_flag<Flag>::value,Flag>::type {
    using int_type = ...
    return Flag(static_cast<int_type>(a) | static_cast<int_type>(b));
}

template <>
struct is_flag<OpenFlag>: public std::true_type {};

OpenFlag flags = OpenFlag::Append | OpenFlag::WriteOnly; // ок
```

Шаблоны в других языках