



C++ Craft: #5

Шаблоны

Язык Программирования С++

- Бьерн Страуструп - глава «Шаблоны».

*Здесь – Ваша цитата
- Б. Страуструп*

перегрузка

```
void swap( int& x, int& y )
{
    const int temp = x;
    x = y;
    y = temp;
}

void swap( double& x, double& y )
{
    const double temp = x;
    x = y;
    y = temp;
}

swap( 1, 2 ); // int
swap( 11.0 ,12.0 ); //double
```

обобщенное программирование

```
template< class T >
void swap( T& x, T& y )
{
    const T temp = x;
    x = y;
    y = temp;
}

swap< int >( 1, 2 ); // int
swap< double >( 11.0 ,12.0 ); //double
swap< int >( 11.0, 12.0 ); // !!! implicit cast double to int
```

template< class T > и
template< typename T >
одно и то же



шаблонный класс

```
template< class T >
class template_example
{
    T* t_ptr_;
public:
    explicit template_example ( T* t = NULL );
};

template< class T >
template_example < T >::template_example ( T* t )
    : t_ptr_( t )
{
}
```

определение и конкретизация

```
template< class T > // определение шаблона
class template_example
{
    T* t_ptr_;
public:
    explicit template_example ( T* t = NULL )
        : t_ptr_( t )
    {}
    void reset( T* t ) { //... }
    T* get() { //... }
    //...
};

// конкретизация шаблона
template_example < double > double_ptr( new double( 9 ) );
double_ptr.get(); // сгенерированы только конструктор и get()
template_example < string > string_ptr( new string( "Hello!" ) );
```

параметры шаблона

- параметры-типы
- параметры обычных типов
- параметры шаблоны

параметры-типы

```
template< class T >  
class buffer  
{  
    T v[ 100 ];  
    public:  
    explicit template_example () {}  
};
```

```
buffer< char > c_buff;  
buffer< int> i_buff;  
buffer< my_class > my_buff;
```


параметры обычных типов

```
template< class T, int max >  
class buffer  
{  
    T v[ max ];  
    public:  
    explicit template_example () {}  
};
```

```
buffer< char, 128 > c_buff;  
buffer< int, 256 > i_buff;  
buffer< my_class, 15 > my_buff;
```

параметры-шаблоны

```
template< template< class > class H, class S >  
void f( const H< S >& value)  
{ //... }
```

```
template< template< class, class > class V, class T >  
void f( V< T, std::allocator< T > > &v )  
{ //... }
```

параметры шаблонов по умолчанию

```
template< class T, int max = 10 >
class buffer
{
    T v[ max ];
    // ...
};
buffer< char > c_buff; // тоже что и buffer< char, 10 > c_buff;
buffer< int, 20 - 10 > i_buff; // тоже что и buffer< int, 10 > i_buff ;
buffer< my_class > my_buff; // тоже что и buffer< class, 10 > my_buff ;
```

```
template < class Key,                      // map::key_type
           class T,                       // map::mapped_type
           class Compare = less<Key>,     // map::key_compare
           class Alloc = allocator<pair<const Key,T> > // map::allocator_type >
class map;

map< int, double > my_map;
```

аргументы шаблона

- Константные выражения
- Адрес объекта или функции с внешней компоновкой
- Не перегруженный указатель на член

Выведение типа

```
void f( std::vector< int >& v )  
{  
    std::sort ( v.begin(), v.end() );  
}
```

```
template< class T, class U > T implicit_cast( U u ) { return u; }  
void f( int i )  
{  
    implicit_cast( i ); // error: невозможно вывести тип T  
    implicit_cast < double >( i );  
}
```

```
template< class T> max( const T&, const T& );  
void k()  
{  
    max( 7, 'a' ); // неоднозначность: нет преобразования  
}
```

перезгрузка

```
template< class T > T sqrt( T );  
template< class T > complex< T > sqrt( complex< T > );  
double sqrt( double );  
  
void f( complex< double > z )  
{  
    sqrt( 2 );           // sqrt< int >( int )  
    sqrt( 2.0 );         // sqrt( double )  
    sqrt( z );           // sqrt< double >( complex< double > )  
}
```

специализация

// copy.h

```
template< class C > bool copy( const C& from, C& to )
```

```
{
```

```
    to = from;
```

```
    return true;
```

```
}
```

```
template<> bool copy < string>( const string& from, string& to );
```

// copy.cpp

```
template<>
```

```
bool copy < string>( const string& from, string& to ) {    // ... }
```

```
template<>
```

```
bool copy <>( const vector& from, vector& to ) // ! тип определен через  
аргументы
```

```
{ // ...}
```

специализация

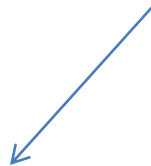
```
// parser.h
enum message_type { trade = 1, market = 2, limit = 3 };
template< message_type type > bool parse_message( const std::string&
    message )
{
    throw std::logic_error( "undefined message type" );
}
template<> bool parse_message< trade >( const std::string& message );
template<> bool parse_message< limit >( const std::string& message );

// parser.cpp
template<>
bool parse_message< trade >( const std::string& message ) { // ... }
template<>
bool parse_message< limit >( const std::string& message ) { // ... }
```

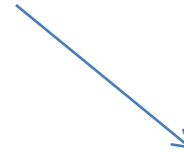


```
template< class T > class vector { //... };
```

специализация



Полная



Частичная

```
template<> class vector< void* > { //... };
```

```
template< class T > class vector< T* > { //... };
```

порядок специализаций

```
template< class T > class vector; // общий шаблон
```

```
template< class T > class vector< T* >; // специализация для  
// любого указателя
```

```
template< > class vector< void* >; // специализация для void*
```

наследование и шаблоны

- Поллиморфизм времени выполнения
- Поллиморфизм времени компиляции

наследование

```
template< class T > base_class { //... };
```

```
template< class T >  
class nested : public base_class< T >  
{  
    //...  
};
```

```
template< class T >  
class nested_vector : public std::vector< int >  
{  
    //...  
};
```

наследование

```
class base {...};  
class derived : public base {...};  
  
void f(set<base*>& s)  
{  
    // ...  
    s.insert( new derived() ); // ok  
}  
  
void g(set<derived*>& s)  
{  
    f( s ); // error  
}
```

члены-шаблоны

```
template< class T >
class template_member
{
    //...
    template< class U >
    const U implicit_cast( T t ) { return t; }
};
```

виртуальные члены-шаблоны

```
class template_member_error
{
    //...
    template< class T >
    virtual void bad_function( T t ) const = 0; // error
};
```

```
template< class T >
class template_member
{
    //...
    virtual void good_function() const { //... } // OK
};
```

SFINAE

Substitution failure is not an error

- Когда речь заходит о SFINAE, это обязательно связано с перегрузкой функций.
- Это работает при автоматическом выводе типов шаблона (type deduction) по аргументам функции.
- Некоторые перегрузки могут отбрасываться в том случае, когда их невозможно инстанциировать из-за возникающей синтаксической ошибки; компиляция при этом продолжается как ни в чём не бывало, без ошибок.
- Отбросить могут только шаблон.
- SFINAE рассматривает только заголовок функции, ошибки в теле функции не будут пропущены.

СОВЕТЫ

- Используйте шаблоны для представления алгоритмов, применяемых ко многим типам аргументов;
- Объявляйте и определяйте специализации;
- Используйте `typedef` для шаблонов;
- Отлаживайте конкретные примеры до их обобщения в шаблоны;
- Используйте шаблоны вместо наследования, когда время выполнения имеет исключительное значение;
- Используйте шаблоны, когда нельзя определить базовый класс.