

Лекция 8

17/11/2018

Темы

- Шаблонные методы
- Шаблонные классы
- Исключения

Шаблоны классов

- **Шаблон класса** – элемент языка, позволяющий использовать типы и значения как параметры, используемые для автоматического создания классов по общему описанию
- Использование шаблонов классов – шаг на пути к парадигме **обобщенного программирования**
- Определение шаблона содержат списки параметров шаблона, среди которых выделяются:
 - **параметры-типы**
 - **параметры-значения**
 - **параметры-шаблоны**

Параметры-значения и параметры-шаблоны

- Параметры-значения могут иметь необязательный **идентификатор**, необязательное **значение по умолчанию** и должны иметь **тип**, являющийся логическим, символьным или целым, перечислением, указателем, ссылкой на объект или `std::nullptr_t`
 - Типы-массивы `T[]` сводятся к указателям `T*`
- Параметры-шаблоны могут иметь необязательный идентификатор, необязательное значение по умолчанию и специфицируются при помощи:
 - **class** – C++11, C++14
 - **class** или **typename** - C++17.

Параметры-типы и параметры-значения

```
template<class T, class U, int size>
```

```
// эквивалентно
```

```
// template<typename T, typename U, int size>
```

```
class Sample {
```

```
    T _prm1;
```

```
    U* _prm2;
```

```
    int _size;
```

```
public:
```

```
    Sample(): _size(size) { } //...
```

```
};
```

Параметры-типы и параметры-шаблоны

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

```
template<class TKey, class TValue,  
          template<typename> class Container = Array>  
class Map {  
  public:  
    Container<TKey> key;  
    Container<TValue> value;  
    // ...  
};
```

Аргументы-значения (1/3)

```
template<const int *pci> struct Alpha { };
```

```
int arr[10];
```

```
Alpha<arr> alpha;           // преобразование массива к указателю
```

```
struct Beta { };
```

```
template<const Beta& beta> struct Gamma { };
```

```
Beta beta;
```

```
Gamma<beta> gamma; // преобразование отсутствует
```

Аргументы-значения (2/3)

```
template<int (&pa)[42]> struct Delta { };
```

```
int container[42];
```

```
Delta<container> delta;
```


Запрещенные аргументы-значения (3/3)

```
template<class T, const char* str> class Phi { };  
Phi<int, "ERROR"> phi;    // строковые литералы
```

```
template<int* p> class Chi{ };  
int a[42];  
struct Psi { int ns; static int s; } psi;  
Chi<&a[10]> chi1;  // адреса элементов массивов  
Chi<&psi.ns>chi2;  // адреса нестатических подобъектов  
Chi<&psi.s>  chi3; // Допустимо  
Chi<&Psi::s> chi4; // Допустимо
```

Статические члены шаблонов классов

- Шаблоны классов могут содержать **статические члены данных**, собственный набор которых имеет каждый конкретизированный согласно шаблону класс.

```
template<class T, class U, int size>
```

```
class Sample {
```

```
    static Sample* _instance;
```

```
};
```

```
template<class T, class U, int size>
```

```
Sample<T, U, size>* Sample<T, U, size>::_instance = nullptr;
```

Специализация шаблонов классов.

Специализация члена класса.

- Шаблоны классов допускают **частичную (полную) специализацию**, при которой отдельные (все) параметры шаблона заменяются конкретными именами типов или значениями константных выражений.

template<>

void Sample<int, double, 10>::foo() {

//...

}

Полная специализация класса

```
template<> class Sample<int, double, 100> {  
  public:  
    Sample<int, double, 100>();  
    ~Sample<int, double, 100>();  
    void foo(); // ...  
};
```

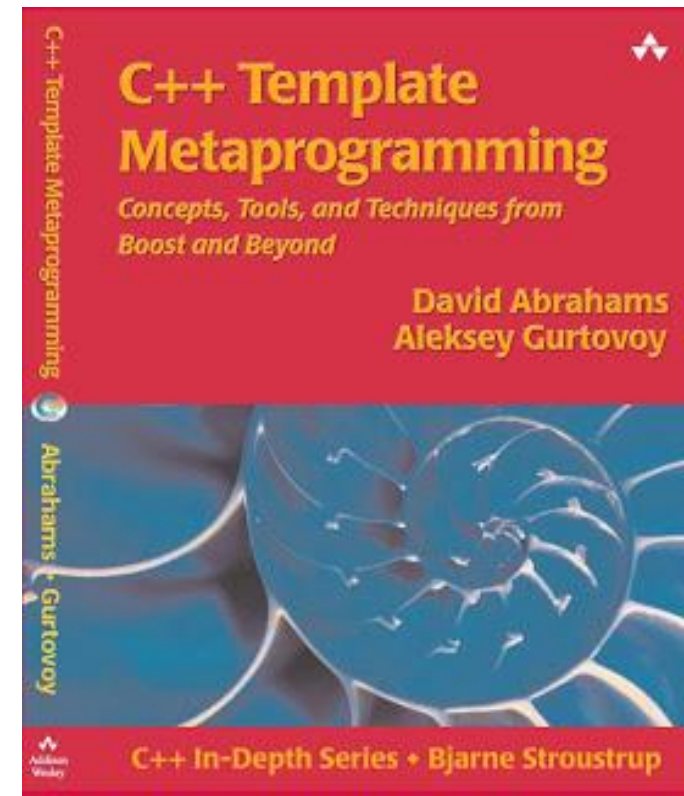
Частичная специализация класса

```
template<class T, class U> class Sample<T, U, 100> {  
  public:  
    Sample ();  
    ~Sample ();  
    void foo(); // ...  
};
```

Рекурсивное определение шаблонов как пример метапрограммирования

```
template <unsigned long N>
struct binary {
    static unsigned const value = binary<N / 10>::value << 1 | N % 10;
};
```

```
template<> struct binary<0> {
    static unsigned const value = 0;
};
unsigned const one = binary<1>::value;
unsigned const three = binary<11>::value;
unsigned const five = binary<101>::value;
```



Понятие исключительной ситуации

- Естественный порядок функционирования программ нарушают возникающие нештатные ситуации, в большинстве случаев связанные с ошибками времени выполнения (но не всегда).
 - нехватка оперативной памяти
 - попытка доступа к элементу коллекции по некорректному индексу
 - попытка недопустимого преобразования динамических типов и пр.
- Обработка исключительных ситуаций носит **невозвратный** характер.

Объекты-исключения. Оператор throw

- Носителями информации об аномальной ситуации (исключении) в С++ являются объекты заранее выбранные на эту роль типов (пользовательских или... базовых, например char*). Такие объекты называются **объектами-исключениями**.
- Жизненный цикл объектов-исключений начинается с возбуждения исключительной ситуации посредством оператора **throw**.

throw "Illegal cast";	// char*
throw IllegalCast();	// class IllegalCast
enum Status { Ok, BadIndex, IllegalCast};	
throw IllegalCast;	// enum Status
throw NULL;	// int
throw nullptr;	// pointer

Пример

```
int main() {  
    try {  
        cout << "Throwing an integer exception...\n";  
        throw 42;  
    } catch (int i) {  
        cout << " the integer exception was caught, with value: " << i  
<< endl;  
    }  
    return 0;  
}
```

Try-блок

- Именованный формальный параметр

```
try { /* */ } catch (const std::exception& e) { /* */ }
```

- Неименованный формальный параметр

```
try { /* */ } catch (const std::exception&) { /* */ }
```

- Особая форма блока-обработчика исключений осуществляет перехват любых исключений

```
try { /* */ } catch (...) { /* */ }
```

Try-блок

```
try {  
    string("abc").substr(10); // генерирует std::length_error  
} catch (const std::exception& e) {  
    cout << e.what(); // "invalid string position"  
}
```

Try-блок

```
try {  
    f();  
} catch (const std::exception& e) {  
    // будет выполнен если f() сгенерирует исключение  
} catch (const std::runtime_error& e) {  
    // недостижимый код  
}
```

Функциональные защищенные блоки

- Защищенный блок может быть оформлен не только как часть функции, но и функции целиком (в том числе `main()` и конструкторы классов). В таком случае защищенный блок называют функциональным.

```
void foobar() try {  
    // ...  
}  
catch(/* ... */) { /*... */ }  
catch(/* ... */) { /*... */ }  
catch(/* ... */) { /*... */ }
```

Раскрутка стека и уничтожение объектов

- Поиск **catch**-блока, пригодного для обработки исключения, приводит к **раскрутке стека** – последовательному выходу из составных операторов и определений функций.
- В ходе раскрутки стека происходит **уничтожение локальных объектов**, определенных в покидаемых областях видимости. При этом деструкторы локальных объектов вызываются штатным образом.
- Исключение, для обработки которого не найден **catch**-блок, инициирует запуск функции **terminate()**, передающей управление функции **abort()**, которая аварийно завершает программу.

Повторное возбуждение исключения. Универсальный блок-обработчик

- Оператор **throw** без параметров помещается (только) в catch-блок и повторно генерирует исключение. При этом его копия не создается.

throw;

- Особая форма блока-обработчика исключений осуществляет перехват любых исключений

catch(...) { /* ... */ }

Описание контракта исключений. Безопасные функции

// описание функций

int foo(**int** &i) **throw**();

bool bar(**char*** pc = 0) **throw**(IllegalCast);

void foobar() **throw**(IllegalCast, BadIndex);

// определение функций

int foo(**int** &i) **throw**() { /* ... */ }

bool bar(**char*** pc = 0) **throw**(IllegalCast) { /* ... */ }

void foobar() **throw**(IllegalCast, BadIndex) { /* ... */ }

Описание контракта исключений.

Безопасные конструкторы

- Конструкторы **могут генерировать исключения**
- Для обработки **всех** исключений, возникших при исполнении конструктора, его тело и список инициализации должны быть совместно помещены в функциональный защищенный блок.

```
Beta::Beta(int value)
try : Alpha(foo(value)) {
    // ...
} catch(...) {
    // ...
}
```

Функциональные блоки try-catch

```
struct S {  
    string m;  
    S(const string& arg) try : m(arg, 100) {  
        cout << "constructed, m = " << m << endl;  
    } catch(const std::exception& e) {  
        cerr << "arg=" << arg << " failed: " << e.what() << endl;  
    } // throw;  
};
```

Функциональные блоки try-catch

```
int f(int n = 2) try {  
    ++n;  
    throw n;  
} catch(...) {  
    ++n;  
    return n;  
}
```

Деструкторы

- Деструкторы классов не должны возбуждать исключений.

`~Alpha() throw();`

Нейтральный код

- От безопасности программного кода важно отличать **нейтральность**, под которой, согласно терминологии Г. Саттера (Herb Sutter), следует понимать способность в методах “пропускать сквозь себя” исключения, полученные ими на обработку.
- **Нейтральный** метод:
 - **может** обрабатывать исключения
 - **должен** ретранслировать исключения (в неизменном виде или дополненном виде).

Класс std::exception

```
class exception {  
public:  
    exception() throw();  
    exception(const exception&) throw();  
    exception& operator=(const exception&) throw();  
    virtual ~exception() throw();  
    virtual const char* what() const throw();  
};
```

Стандартные классы – логические ошибки

- Базовый промежуточный класс **std::logic_error**
 - **std::invalid_argument** - неверный аргумент
 - **std::out_of_range** - вне диапазона
 - **std::length_error** - неверная длина

Стандартные классы – ошибки времени исполнения

- Базовый промежуточный класс **std::runtime_error**
 - **std::range_error** - ошибка диапазона
 - **std::overflow_error** - переполнение (в т. ч. структуры данных)
 - **std::underflow_error** - потеря порядка (в т. ч. попытка получить элемент из пустой коллекции)

Стандартные классы

- Базовый класс **std::exception**
 - **std::bad_alloc** - ошибка выделения динамической памяти
 - **std::bad_cast** - ошибка приведения типа (dynamic_cast)

Стандартные классы

