



# Углубленное программирование на языке C++



Алексей Петров

## Лекция №5. Шаблоны классов и методов. Обработка исключительных ситуаций. Обобщенное и безопасное программирование

1. Обобщенное программирование. Рекурсивное определение шаблонов.
2. Основы метапрограммирования. Идиома SFINAE.
3. Библиотека Boost.Type Traits.
4. Обработка исключительных ситуаций и вопросы производительности.
5. Безопасное программирование. Нейтральность кода. Критерии безопасности и нейтральности классов и методов.
6. Безопасность конструкторов и деструкторов.
7. Исключения в стандартной библиотеке.



**Шаблон класса** — элемент языка, позволяющий параметризовать типы и значения, используемые для автоматического создания (**конкретизации**) классов по обобщенному описанию (**шаблону**).

Использование шаблонов классов — шаг на пути к парадигме **обобщенного программирования**.

Различают **описания** и **определения шаблонов** классов. В отличие от «обычного» класса описания и определения шаблона содержат **списки параметров шаблона**, среди которых выделяются **параметры-типы** и **параметры-константы**.

Параметры-типы шаблона класса представляют некоторый тип данных, параметры-константы — некоторое константное (**вычисляемое при компиляции**) выражение.

# Шаблоны классов: пример



```
template <class T, class U, int size>
// эквивалентно:
// template <typename T, typename U, int size>
class Test
{
public:
    Test() : _size(size) // ...
private:
    T        _prm_1;
    U*       _prm_2;
    int      _size;
};
```

# Дружественные объекты в шаблонах классов



**Дружественными по отношению к шаблонам классов могут быть:**

- дружественная функция или дружественный класс (**не шаблон**);
- связанный шаблон дружественной функции, взаимно однозначно соответствующий шаблону класса (**с общим для обоих шаблонов списком параметров**);
- связанный шаблон дружественного класса, взаимно однозначно соответствующий шаблону класса (**с общим для обоих шаблонов списком параметров**);
- несвязанный шаблон дружественной функции, соответствующий всем возможным конкретизациям шаблона класса (**с отдельными списками параметров**);
- несвязанный шаблон дружественного класса, соответствующий всем возможным конкретизациям шаблона класса (**с отдельными списками параметров**).

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

```
template <class T, class U, int size>
class Test
{
    /* ... */
private:
    static Test *_head;
};
```

```
template <class T, class U, int size>
Test<T, U, size> *Test<T, U, size>::_head = NULL;
```

# Специализация шаблонов классов.

## Специализация члена класса: пример



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

```
// специализация члена класса
template<>
void Test<int, double, 10>::foo()
{
    /* ... */
}
```

# Полная и частичная специализация класса: пример



```
// полная специализация класса
template<> class Test<int, double, 100>
{
public:
    Test<int, double, 100>();
    ~Test<int, double, 100>();
    void foo(); /* ... */
};

// частичная специализация класса
template <class T, class U> class Test<T, U, 100>
{
public:
    Test();
    ~Test (); /* ... */
};
```



# 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;
```

**Идиомы SFINAE** (англ. *Substitution Failure Is Not An Error* — «неудача при подстановке не есть ошибка») является одной из идиом обобщенного программирования и означает ситуацию, при которой **невозможность подстановки параметров шаблона не влечет аварийного завершения компиляции.**

Ситуация, соответствующая идиоме SFINAE, возникает **при разрешении перегруженных вызовов**, в которых среди множества функций-кандидатов найдется хотя бы одна, полученная в результате конкретизации шаблона.

В соответствии с логикой идиомы SFINAE, неудача при подстановке параметров в соответствующий шаблон ведет лишь к удалению данного шаблона из множества кандидатов и не имеет катастрофических последствий для компиляции.

# Идиома SFINAE и интроспекция времени компиляции: пример



```
template <typename T> struct has_typedef_foobar {  
    typedef char yes[1]; // sizeof(yes) == 1  
    typedef char no [2]; // sizeof(no) == 2  
  
    template <typename C>  
    static yes& test(typename C::foobar*);  
  
    template <typename> static no& test(...);  
  
    static const bool value =  
        sizeof(test<T>(0)) == sizeof(yes);  
};  
struct foo { typedef float foobar; };  
// has_typedef_foobar<int>::value == false  
// has_typedef_foobar<foo>::value == true
```

# Интроспекция времени компиляции в библиотеке `boost::type_traits`



**Цель.** Через набор узкоспециализированных, имеющих единый дизайн вспомогательных классов упростить работу с атомарными характеристиками типов ([англ. type traits](#)) в системе типов языка C++.

**Библиотека.**

```
#include <boost/type_traits.hpp>
```

**Реализация.** Библиотека характеристик типов содержит значительное количество внутренне весьма однородных классов (структур), многие из которых открыто наследуют типам [true\\_type](#) или [false\\_type](#) (см. далее).

# Характеристики типов: структуры `is_void`, `is_pointer`



```
// структура is_void
template <typename T>
struct is_void : public false_type{};

template <>
struct is_void<void> : public true_type{};

// структура is_pointer
template <typename T>
struct is_pointer: public false_type{};

template <typename T>
struct is_pointer<T*> : public true_type{};
```

# Характеристики типов: структуры `true_type` и `false_type`

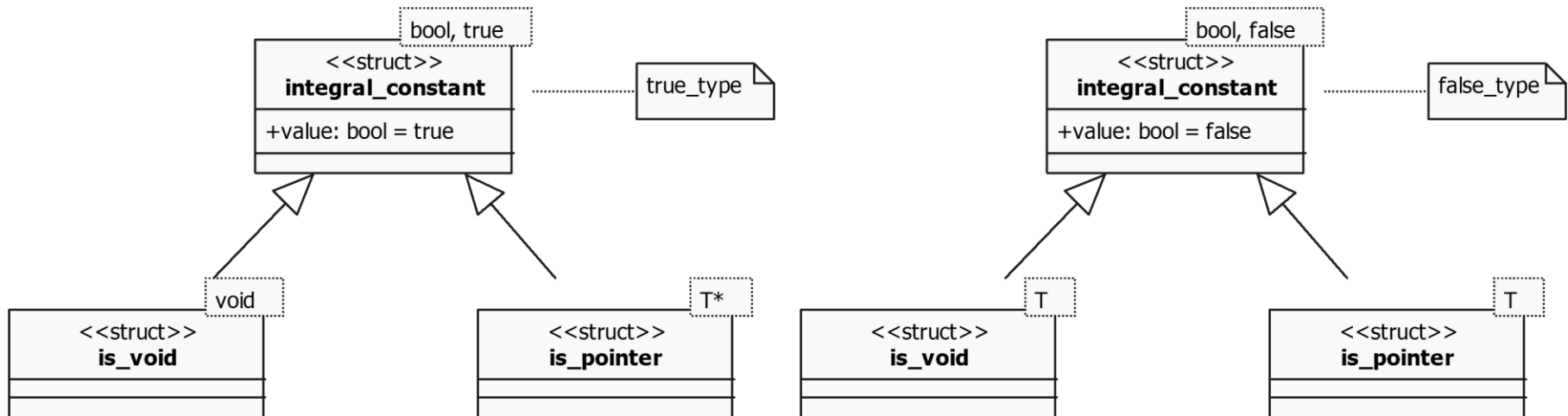


```
template <class T, T val>
struct integral_constant
{
    typedef integral_constant<T, val>    type;
    typedef T                            value_type;
    static const T                       value = val;
};
```

```
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

# Характеристики типов: иерархии классов

Проиллюстрированный подход позволяет строить параллельные иерархии классов, обладающих (потомки **true\_type**) и не обладающих (потомки **false\_type**) искомыми свойствами. Принадлежность к соответствующей иерархии является **различительным признаком**.



# Характеристики типов: структура `remove_extent`



Установить тип элемента массива позволяет структура `remove_extent`.

```
template <typename T>
struct remove_extent
{
    typedef T type;
};
```

```
template <typename T, std::size_t N>
struct remove_extent<T[N]>
{
    typedef T type;
};
```



# Характеристики типов: оптимизация функций (общий случай)



Имея эффективную реализацию функции для типов с конкретными характеристиками, нетрудно должным образом специализировать ее шаблон. Например, для `std::copy` имеем:

```
template<typename I1, typename I2, bool b>
I2 copy_imp(I1 first, I1 last, I2 out,
            const boost::integral_constant<bool, b>&)
{
    while (first != last) {
        *out = *first;
        ++out; ++first;
    }
    return out;
}
```

# Характеристики типов: оптимизация функций(оптимизированная реализация)



```
template<typename T>
T* copy_imp(const T* first, const T* last, T* out,
            const boost::true_type&)
{
    memmove(out, first, (last - first) * sizeof(T));
    return out + (last - first);
}
```

# Характеристики типов: оптимизация функций (вызывающая функция)



```
template<typename I1, typename I2>
inline I2 copy(I1 first, I1 last, I2 out)
{
    typedef typename std::iterator_traits<I1>::
        value_type value_type;
    return copy_imp(first, last, out,
        boost::has_trivial_assign<value_type>());
}
```

# Характеристики типов: исключение деструкторов (вызов `~T()` обязателен)



Столь же нетрудно избежать накладных расходов на вызов деструкторов в случаях, когда этого допускает структура соответствующих классов.

```
template<class T>
void do_destroy_array(T* first, T* last,
    const boost::false_type&)
{
    while(first != last) {
        first->~T();
        ++first;
    }
}
```

# Характеристики типов: исключение деструкторов (вызов `~T()` факультативен)



```
template<class T>
inline void do_destroy_array(T* first, T* last,
    const boost::true_type&) { }

// вызывающая функция
template<class T>
inline void destroy_array(T* p1, T* p2)
{
    do_destroy_array(p1, p2,
        boost::has_trivial_destructor<T>());
}
```

# Понятие исключительной ситуации (1 / 2)



Естественный порядок функционирования программ нарушают возникающие **нештатные ситуации**, в большинстве случаев связанные с ошибками времени выполнения (**иногда — с необходимостью внезапно переключить контекст приложения**).

В языке C++ такие штатные ситуации называются **исключительными** (иначе говоря — **исключениями**).

Примерами исключительных ситуаций являются:

- нехватка оперативной памяти;
- попытка доступа к элементу коллекции по некорректному индексу;
- попытка недопустимого преобразования динамических типов и пр.

# Понятие исключительной ситуации (2 / 2)



Архитектурной особенностью механизма обработки исключительных ситуаций в языке C++ является принципиальная независимость (несвязность) фрагментов программы, где исключение **возбуждается** и где оно **обрабатывается**.

Обработка исключительных ситуаций носит **невозвратный** характер.

# Объекты-исключения.

## Оператор `throw`



Носителями информации об аномальной ситуации (исключении) в C++ являются объекты заранее выбранных на эту роль типов (пользовательских или — *Sic!* — 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   EProgStatus
        {psOK, psBadIndex, psIllegalCast};
throw psIllegalCast;           // enum EProgStatus
```



# Функциональные защищенные блоки



Как защищенный блок может быть оформлена не только часть функции, но и функция целиком (в том числе `main()` и конструкторы классов). В таком случае защищенный блок называют **функциональным**. Например:

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

# Раскрутка стека и уничтожение объектов



Поиск `catch`-блока, пригодного для обработки возбужденного 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 (**строгая гарантия C++**).

Исключение, для обработки которого не найден `catch`-блок, инициирует запуск функции `terminate()`, передающей управление функции `abort()`, которая аварийно завершает программу.

# Повторное возбуждение исключения и универсальный блок-обработчик



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

```
throw;
```

Особая форма блока-обработчика исключений осуществляет перехват любых исключений:

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

# Безопасность ПО

## как показатель качества



**Безопасность** — один из **показателей качества ПО**, оцениваемый путем **статического анализа** состава и взаимосвязей используемых компонентов, **исходного кода** и схемы БД.

Для обеспечения «структурной безопасности» исходного кода необходимо соблюдение стандартов разработки архитектуры и **стандартов кодирования**.

Примечание: См. модели качества ПО, описанные в стандартах:

- ГОСТ Р ИСО/МЭК 9126-93. Информационная технология. Оценка программной продукции. Характеристики качества и руководства по их применению;
- ISO/IEC 9126:2001. Software Engineering — Product Quality;
- ISO/IEC 25010:2011. Systems and Software Engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.

# Стандарты кодирования — за «безопасный код»



Структурная безопасность исходного кода ПО требует соблюдения определенных техник кодирования, одной из которых является систематическая **обработка ошибок и исключительных ситуаций** на всех уровнях архитектуры (**уровень представления, уровень бизнес-логики, уровень (базы) данных**).

Одним из элементов обработки ошибок и исключений является **спецификация (ограничение) типов** исключений, которые могут породить структурные элементы кода.

**Функция C++ безопасна**, если она **не возбуждает** никаких исключений или все возбужденные внутри нее исключения обрабатываются в ее теле.

**Класс C++ безопасен**, если безопасны все его методы.

В свою очередь, небезопасные функции могут специфицировать исключения, возбуждением которых (**и только их!**) способно завершиться исполнение таких функций. Обнаруженное при исполнении нарушение гарантии влечет за собой вызов функции **unexpected()**, по умолчанию вызывающей **terminate()**.

Виртуальные функции в производных классах могут **повторять** спецификации исключений функций в базовых классах или накладывать **более строгие** ограничения.

```
// описания функций  
int foo(int &i) throw();  
bool bar(char *pc = 0) throw(IllegalCast);  
void foobar() throw(IllegalCast, BadIndex);
```

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

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

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

Конструкторы, как и другие методы классов, **могут возбуждать 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))
{
    // тело конструктора
    /* ... */
}
catch(...) { /* ... */ }
```



Деструкторы классов **не должны** возбуждать исключения. Одной из причин этого является необходимость корректного освобождения ресурсов, занятых массивами и коллекциями объектов:

- если вызываемая в деструкторе функция может возбудить исключение, деструктор должен **перехватить и обработать** его (**возможно, прервав программу**);
- если возможность реакции на исключение необходима клиентам класса во время некоторой операции, в его открытом интерфейсе должна быть функция (**не деструктор!**), которая такую операцию выполняет.

В общем случае деструктор класса может специфицироваться как **throw()**:

```
~Alpha() throw();
```

# Безопасность или нейтральность кода?



От безопасности программного кода важно отличать **нейтральность**, под которой, согласно терминологии Г. Саттера (Herb Sutter), следует понимать способность методов класса прозрачно «пропускать сквозь себя» объекты-исключения, полученные ими на обработку, но не предназначенные для них.

## Нейтральный метод:

- **может** обрабатывать исключения;
- **должен** ретранслировать полученные им исключения методу-обработчику в неизменном виде и сохранять свою работоспособность при любых обстоятельствах.

# Пользовательские и стандартные классы исключительных ситуаций



Для передачи из точки возбуждения исключения в точку его обработки сведений об условиях возникновения аномалии программист может определять и использовать **собственные классы исключительных ситуаций**.

Такие классы могут быть сколь угодно **простыми** (включая **пустые**) или **сложными** (содержащими члены данных, конструкторы, деструкторы и интерфейсные методы).

Стандартная библиотека языка C++ содержит собственную иерархию классов исключений, являющихся прямыми или косвенными потомками базового класса **exception**. Потомки класса **exception** условно представляют две категории ошибок: **логические** ошибки и ошибки **времени исполнения**.

# Класс `std::exception`



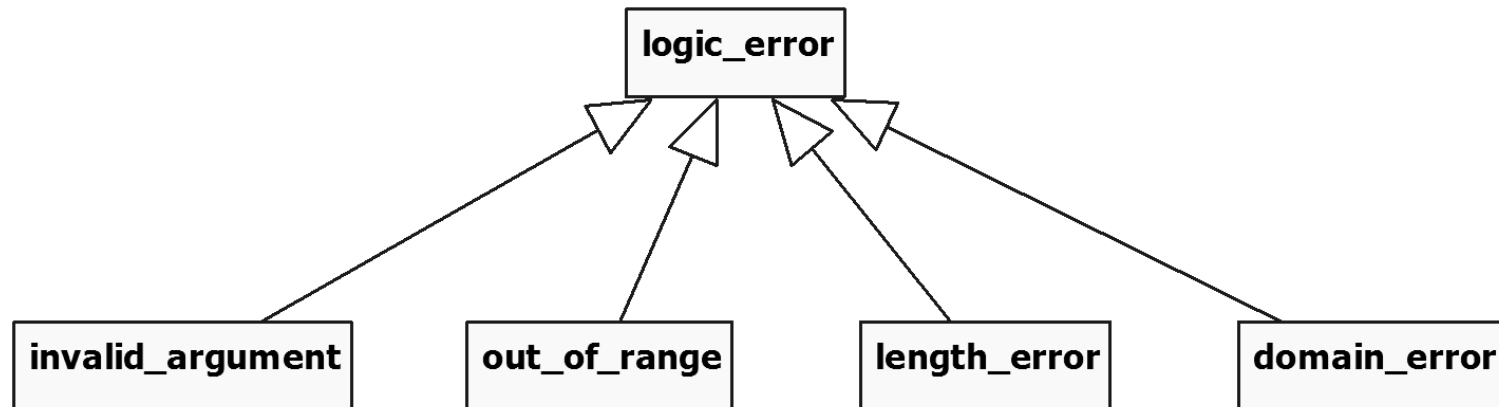
```
namespace std {  
    class exception {  
    public:  
        exception() throw();  
        exception( const exception & ) throw();  
  
        exception& operator=(  
            const exception & ) throw();  
  
        virtual ~exception() throw();  
  
        virtual const char* what() const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

# Стандартные классы — логические ошибки



В число классов категории «логические ошибки» входят **базовый** промежуточный класс `logic_error`, а также производные от него **специализированные** классы:

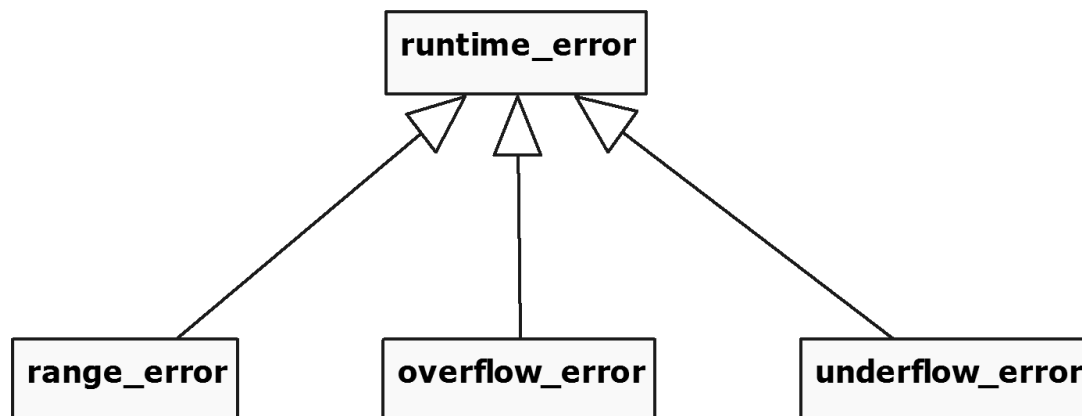
- `invalid_argument` — ошибка «неверный аргумент»;
- `out_of_range` — ошибка «вне диапазона»;
- `length_error` — ошибка «неверная длина»;
- `domain_error` — ошибка «вне допустимой области».



# Стандартные классы — ошибки времени исполнения

В число классов категории «ошибки времени исполнения» входят базовый промежуточный класс `runtime_error`, а также производные от него **специализированные** классы:

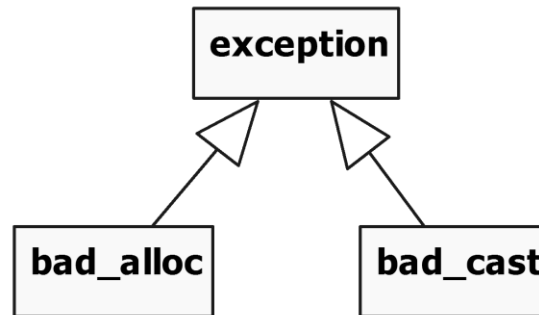
- `range_error` — ошибка диапазона;
- `overflow_error` — переполнение;
- `underflow_error` — потеря значимости.



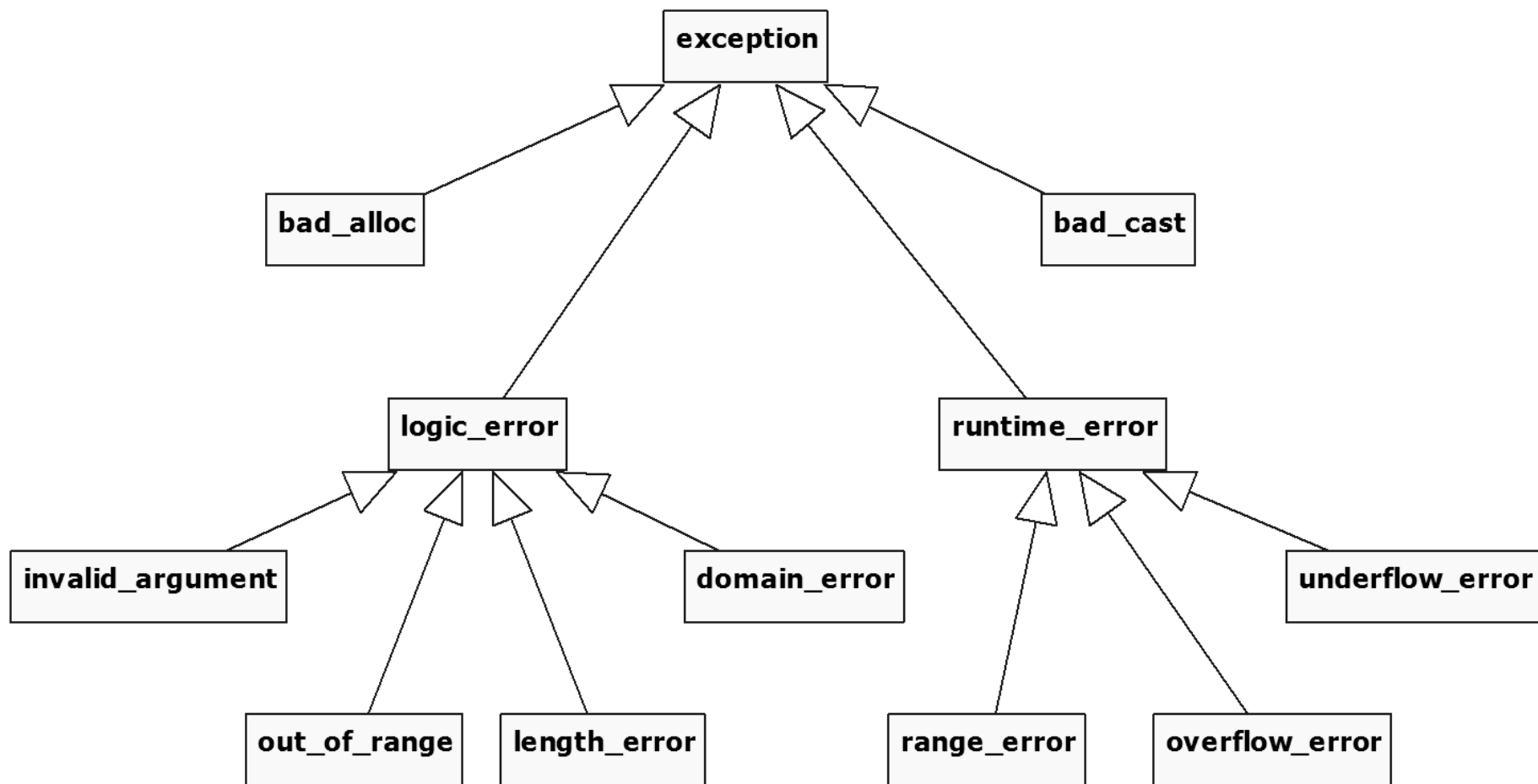
# Прочие классы исключений стандартной библиотеки



Также производными от `exception` являются классы `bad_alloc` и `bad_cast`, сигнализирующие об ошибках при выделении динамической памяти и неудаче при выполнении «ссылочного» варианта операции `dynamic_cast`, соответственно.



# Классы исключений стандартной библиотеки: «вид сверху»







**Спасибо за внимание**

**Алексей Петров**