ТЕХНОСФЕРА

Лекция № 5

Углубленное программирование на языке C / 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.

Шаблоны классов

Шаблон класса — элемент языка, позволяющий, среди прочего, использовать типы и значения как параметры, 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**, **параметры-значения** и **параметры-шаблоны**.

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

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 или леводопустимой ссылкой на функцию или объект, указателем на член класса или равный std::nullptr_t.

- Типы-массивы T[] сводятся к указателям T*, а типы-функции — к указателям на функции, соответственно.

Параметры-шаблоны могут иметь необязательный идентификатор, необязательное значение по умолчанию и специфицируются при помощи ключевых слов:

- class — в диалектах C++11, C++14;
- class или typename — в диалекте C++17.

Параметры-типы и параметры-значения: пример

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

Параметры-типы и параметры-шаблоны: пример

```
template <class T>
class Array {
    // …
};

template <class Key, class Value,
          template<typename> class Container = Array>
class Map {
public:
    Container<Key> key;
    Container<Value> value;
    // …
};
```

Конкретизация шаблонов: аргументы-значения (C++03, C++11)

Категория параметра	Ограничения на аргумент
Арифметический тип	Константное выражение типа, соответствующего типу параметра шаблона (возможны неявные преобразования)
Указатель на объект	Адрес объекта со статической продолжительностью хранения и внешним или внутренним связыванием *или* константное выражение, сводимое к соответствующему пустому указателю или значению `std::nullptr_t`.
Указатель на функцию	Адрес функции, обладающей связыванием, *или* константное выражение, сводимое к соответствующему пустому указателю или значению `std::nullptr_t`.
Леводопустимая ссылка	**Не является** временным, безымянным объектом или именованным объектом хранения без связывания

Конкретизация шаблонов: аргументы-значения (C++17)

Параметры-значения шаблонов могут конкретизироваться с использованием аргументов, являющихся произвольными **приведенными константными выражениями**, то есть выражениями, неявно приведенными к чистым праводопустимым выражениям (англ. pure right-hand value, prvalue), в которых приведенное выражение является базовым константным выражением, а цепь неявных преобразований содержит только:

- пользовательские преобразования с использованием constexpr;
- преобразования леводопустимых значений в праводопустимые;
- повышения и не сужающие преобразования целых или символьных типов;
- преобразования массивов и функций в указатели (в C++17);
- преобразования квалификаторов (в C++17);
- преобразования пустых указателей с использованием std::nullptr_t (в C++17).

Аргументы-значения: пример (1 / 2, C++17)

```
template <const int *pci> struct Alpha {};
int   arr[10];          // преобразование массива к указателю
Alpha<arr> alpha;       // преобразование квалификаторов

struct Beta {};
template <const Beta& beta> struct Gamma {};
Beta beta;              // преобразования отсутствуют
Gamma<beta> gamma;      // избыточный квалификатор игнорир.

template <int (&pa) [42]> struct Delta {};
int   container[42];    // преобразования отсутствуют
Delta<container> delta;
```

Аргументы-значения: пример (2 / 2, C++17)

```
// примеры запрещенных к использованию аргументов-значений
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

template <const int& cri> struct Omega {};
Omega<1>       omega;  // временные объекты
```

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 раздельными списками параметров);
- несвязанный шаблон дружественного класса, соответствующий всем возможным конкретизациям шаблона класса (с раздельными списками параметров).

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 {
    // …
private:
    static Sample *_head;
};

template <class T, class U, int size>
Sample<T, U, size> *Sample<T, U, size>::_head = nullptr;
```

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 Sample<int, double, 10>::foo() {
    // …
}
```

Полная и частичная специализация класса: пример

```
// 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(); // …
};
```

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;
```

Идиома SFINAE

Идиома 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

```
// тип T является пустым типом?
template <typename T>
struct is_void : public false_type{};
template <>
struct is_void<void> : public true_type{};

// тип T является указателем?
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.

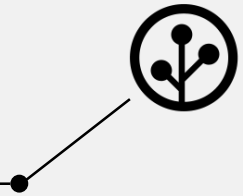
```
template <typename T>
struct remove_extent {
    typedef T type;
};

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

Характеристики типов: оптимизация функций (1 / 2)

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

```
// общий случай
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 (2 / 2)



```
template<typename T>   // оптимизированная реализация
T* copy_imp(const T* first, const T* last, T* out,
            const boost::true_type&) {
    std::memmove(out, first, (last - first) * sizeof(T));
    return out + (last - first);
}
template<typename I1, typename I2>    // общий случай
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 (1 / 2)

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

```
// вызов T::~T() обязателен
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 (2 / 2)

```
// вызов T::~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

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

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

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

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  EPrgStatus {OK, BADINDEX, ILLEGALCAST};
throw ILLEGALCAST;             // enum EPrgStatus
```

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**.

```
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 программной архитектуры (уровень представления, уровень логики, уровень данных).

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

Безопасность классов и методов

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 классов или накладывать **более строгие** ограничения.

Безопасность классов и методов: пример

```
// 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) { /* … */ }
```

Безопасность конструкторов

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)) {
    // …
}
catch(...) {
    // …
}
```

Безопасность деструкторов

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 интерфейса 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**.

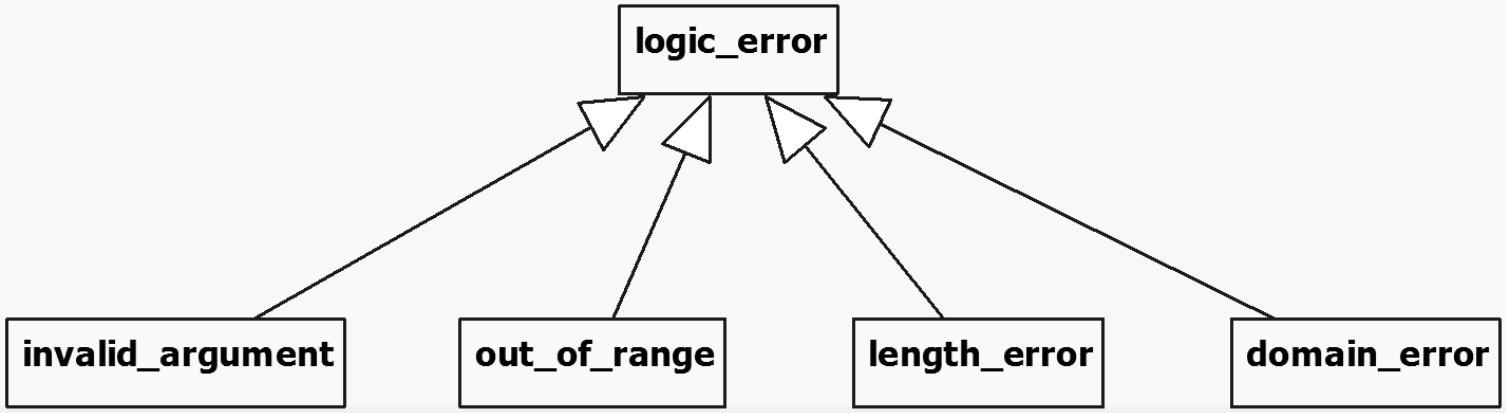
Класс std::exception

```
// 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 std::logic_error, а также производные от него **специализированные** классы:

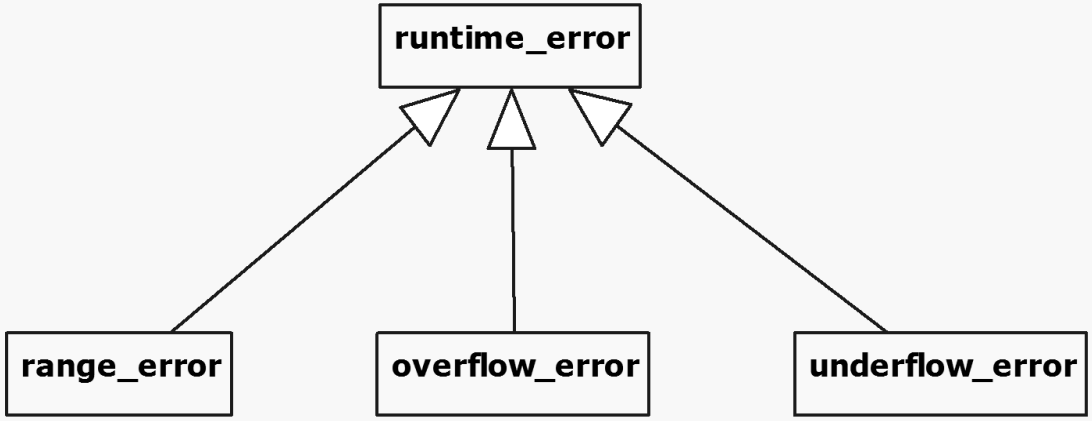
- std::invalid_argument — ошибка «неверный аргумент»;
- std::out_of_range — ошибка «вне диапазона»;
- std::length_error — ошибка «неверная длина»;
- std::domain_error — ошибка «вне допустимой области».



Стандартные классы — ошибки времени исполнения

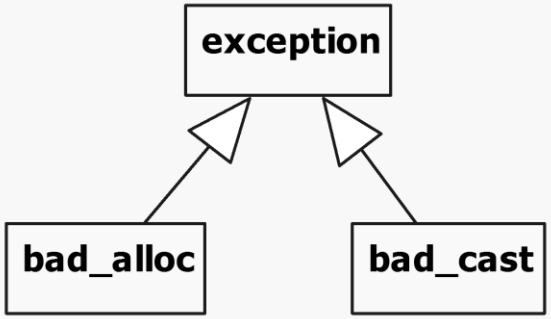
В число классов категории «ошибки времени исполнения» входят **базовый** промежуточный класс std::runtime_error, а также производные от него **специализированные** классы:

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

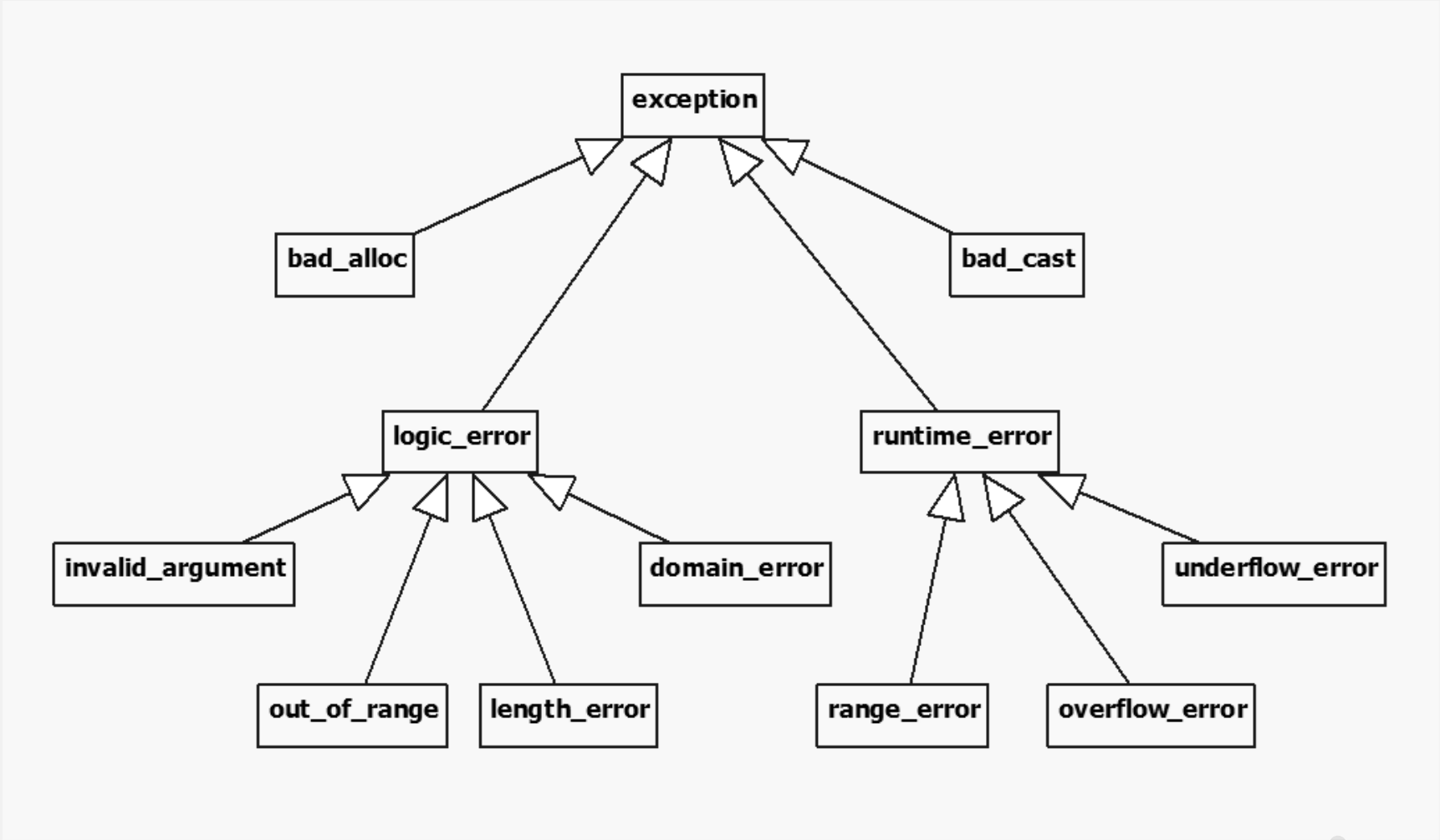


Прочие классы исключений стандартной библиотеки

Также производными от std::exception являются классы std::bad_alloc и std::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»



ТЕХНОСФЕРА

Спасибо за внимание!

Алексей Петров