

Средства обработки ошибок. Исключения в С++

Обработка **исключительных ситуаций** в С++ организуется с помощью ключевых слов **try, catch** и **throw**.

Операторы программы, при выполнении которых необходимо обеспечить обработку исключений, выделяются в **try-catch**-блок.

Если ошибка произошла внутри **try**-блока (в частности, в вызываемых из **try**-блока функциях), то соответствующее **исключение** должно генерироваться с помощью операции **throw** (ее приоритет ниже, чем у «=», но выше, чем у «,»), а перехватываться и обрабатываться в теле одного из обработчиков **catch**, которые располагаются непосредственно за **try**-блоком.

Исключение — объект некоторого типа, в частности, встроенного.

Операторы, находящиеся после места генерации ошибки в **try**-блоке, игнорируются, а после обработки исключения управление передается первому оператору, находящемуся за обработчиками исключений.

try-catch-блоки могут быть вложенными.

Общий синтаксис **try-catch** блока:

```
try {  
..... throw исключение; .....  
}  
catch (type) {---/*throw;/*}  
catch (type arg) {---/*throw;/*}  
...  
catch (...) {---/*throw;/*}
```

Перехват исключений

С каждым **try**-блоком может быть связано несколько операторов **catch**. Они просматриваются по очереди сверху вниз.

Какой именно обработчик **catch** будет использоваться, зависит от типа сгенерированного исключения.

Выбирается первый обработчик с типом параметра, **совпадающим** с типом исключения. **Ловушки с базовым типом** (или с указателем или ссылкой на базовый тип) **перехватывают все исключения с производным типом** (или его адресом), т.е. производные типы должны стоять раньше базовых типов.

Если исключение перехвачено каким-либо обработчиком **catch**, аргумент `arg` получает его значение, которое затем можно использовать в теле обработчика. Если доступ к самому исключению не нужен, то в обработчике **catch** можно указывать только его тип.

Существует специальный вид обработчика, перехватывающего любые исключения — **catch (...){---**. Естественно, он должен находиться в конце последовательности обработчиков **catch**.

Если для сгенерированного исключения в текущем **try**-блоке нет подходящего обработчика, оно перехватывается объемлющим **try**-блоком (`main()`→`f()`→`g()`→`h()`).

Если же подходящего обработчика так и не удалось найти, может произойти **ненормальное** завершение программы. При этом вызывается стандартная библиотечная функция **terminate ()**, которая в свою очередь вызывает функцию **abort ()**, чего лучше избегать.

Пример

```
class A {  
  public:  
    A () {cout << "Constructor of A\n";}   
    ~A () {cout << "Destructor of A\n";}   
};  
class Error {};  
class Error_of_A : public Error {};  
void f () {  
  A a;  
  throw 1;  
  cout << "This message is never printed" << endl;  
}  
int main () {  
  try {  
    f ();  
    throw Error_of_A();  
  }  
  catch (int) { cerr << "Catch of int\n"; }  
  catch (Error_of_A) { cerr << "Catch of Error_of_A\n"; }  
  catch (Error) { cerr << "Catch of Error\n"; }  
  return 0;  
}
```

Результат работы программы с предыдущего слайда:

Constructor of A

Destructor of A // т.к. в f обработчика нет, поиск идет дальше,
// но при выходе из f вызывается деструктор
// локальных объектов.

Catch of int

Если поменять строки внутри **try**-блока, получим:

Catch of Error_of_A

Если закомментировать строку

```
// catch (Error_of_A) { cerr << "Catch of Error_of_A\n"; },
```

получим

Catch of Error

Пример использования классов исключений

```
class MathEr {...virtual void ErrProcess();...};
```

```
class Overflow : public Math Er {... void ErrProcess();...};
```

```
class ZeroDivide : public Math Er {... void ErrProcess();...};
```

...

Через параметры конструктора исключения можно передавать любую нужную информацию.

Если использовать виртуальные функции, можно после **try**-блока задать единственный обработчик **catch**, имеющий параметр типа базового класса, но перехватывающий и обрабатывающий любые исключения:

```
try { ...  
}  
catch (MathEr & m) {... m. ErrProcess(); ...}
```

Организованная таким образом обработка исключений позволяет легко модифицировать программы.

Исключения, генерируемые в функциях

В заголовке функции можно указать типы исключений (через запятую), которые может генерировать функция (эту возможность удобно использовать при описании библиотечных функций):

```
тип_рез имя_функции (список_арг) [const] throw (список_типов) { ... }
```

Если список типов **пустой**, то функция не может генерировать **никаких** исключений.

Если же функция все-таки сгенерировала недеklarированное исключение, вызывается библиотечная функция ***unexpected*** () работающая аналогично функции *terminate*()).

Использование аппарата исключений – единственный безопасный способ нейтрализовать ошибки в конструкторах и деструкторах, поскольку они не возвращают никакого значения, и нет другой возможности отследить результат их работы.

Если деструктор, вызванный во время свертки стека, попытается завершить свою работу при помощи исключения, то система вызовет функцию *terminate*()), что крайне нежелательно. Отсюда важное требование к деструктору: ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы.

Действия, выполняемые с момента генерации исключения до завершения его обработки

- При генерации исключения (*throw* *x*) создается объект-исключение – копия *x* (работает конструктор копирования). С этой копией будет работать выбранный далее обработчик; она существует до тех пор, пока обработка исключения не будет завершена. В случае *throw X(1)* происходит оптимизация, в результате которой сразу создается объект-исключение.
- Для всех других объектов *try*-блока, созданных к этому моменту, перед выходом из *try*-блока освобождается память; при этом для объектов – экземпляров классов вызывается деструктор. То же делается и для уже созданных подобъектов: членов класса – объектов другого класса и баз. Этот процесс называют «раскруткой» («сверткой») стека.
- Если в списке обработчиков *catch* этого *try*-блока найден подходящий, то выполняются его операторы; затем выполнение программы продолжается с оператора, расположенного за последним обработчиком этого *try*-блока.
- Если в списке данного *try*-блока не нашлось подходящего обработчика, то поиск продолжается в динамически объемлющих *try*-блоках (при этом процесс свертки стека продолжается).
- Если подходящего обработчика так и не нашлось, то вызывается функция *terminate()* и выполнение программы прекращается.

Механизм RTTI (Run-Time Type Identification)

Механизм RTTI состоит из трех частей:

1. операция **dynamic_cast**
в основном предназначена для получения указателя на объект производного класса при наличии указателя на объект полиморфного базового класса
2. операция **typeid**
служит для идентификации точного типа объекта при наличии указателя на полиморфный базовый класс
3. структура **type_info**
позволяет получить дополнительную информацию, ассоциированную с типом

Для использования RTTI в программу следует включить заголовок `<typeinfo>`

(1). Операция ***dynamic_cast*** реализует приведение типов (указателей или ссылок) полиморфных классов в динамическом режиме.

Синтаксис использования операции *dynamic_cast*:

dynamic_cast < целевой тип > (выражение)

Если даны два полиморфных класса **B** и **D** (причем **D** – производный от **B**), то *dynamic_cast* всегда может привести D^* к B^* .

Также *dynamic_cast* может привести B^* к D^* , но только в том случае, если объект, на который указывает указатель, действительно является объектом типа **D** (либо производным от него)!

При неудачной попытке приведения типов результатом выполнения *dynamic_cast* является `0`, если в операции использовались указатели.

Если же в операции использовались ссылки, генерируется исключение типа ***bad_cast***.

Пример:

```
Base * bp, b_ob;
```

```
Derived * dp, d_ob;
```

```
bp = & d_ob;
```

```
dp = dynamic_cast <Derived *> (bp);
```

```
if (dp)
```

```
    cout << "Приведение типов прошло успешно";
```

```
bp = &b_ob;
```

```
dp = dynamic_cast <Derived *> (bp);
```

```
if (!dp)
```

```
    cout << "Приведения типов не произошло";
```

(2)-(3) Информацию о типе объекта можно получить с помощью операции *typeid*.

Синтаксис использования операции *typeid*:

***typeid* (выражение)** или
***typeid* (имя_типа)**

Операция *typeid* возвращает ссылку на **объект класса *type_info***, представляющий либо тип объекта, обозначенного заданным выражением, либо непосредственно заданный тип.

В классе *type_info* определены следующие открытые члены:

bool operator == (**const** *type_info* & объект); // для сравнения типов

bool operator != (**const** *type_info* & объект); // для сравнения типов

bool before (**const** *type_info* & объект); // для внутреннего использования

const char * name (); // возвращает указатель на имя типа

Операция *typeid* наиболее полезна, если в качестве аргумента задать указатель полиморфного базового класса, т.к. с его помощью во время выполнения программы можно определить тип реального объекта, на который он указывает. То же относится и к ссылкам.

typeid часто применяется к разыменованным указателям (*typeid (* p)*). Если указатель на полиморфный класс *p* == NULL, то будет сгенерировано исключение типа ***bad_typeid***.

Пример

```
class Base {  
    virtual void f ( ) {...};  
};  
class Derived1: public Base {...  
};  
class Derived2: public Base {...  
};  
int main ( ) {  
    int i;  
    Base *p, b_ob;  
    Derived1 ob1;  
    Derived2 ob2;  
    cout << "Тип i - " << typeid ( i ) . name ( ) << endl;  
    p = & b_ob;  
    cout << "p указывает на объект типа " << typeid ( * p ) . name ( ) << endl;  
    p = & ob1;  
    cout << "p указывает на объект типа " << typeid ( * p ) . name ( ) << endl;  
    p = & ob2;  
    cout << "p указывает на объект типа " << typeid ( * p ) . name ( ) << endl;  
    if ( typeid (ob1) == typeid (ob2) )  
        cout << "Тип объектов ob1 и ob2 одинаков\n";  
    else  
        cout << "Тип объектов ob1 и ob2 не одинаков\n";  
    return 0;  
}
```

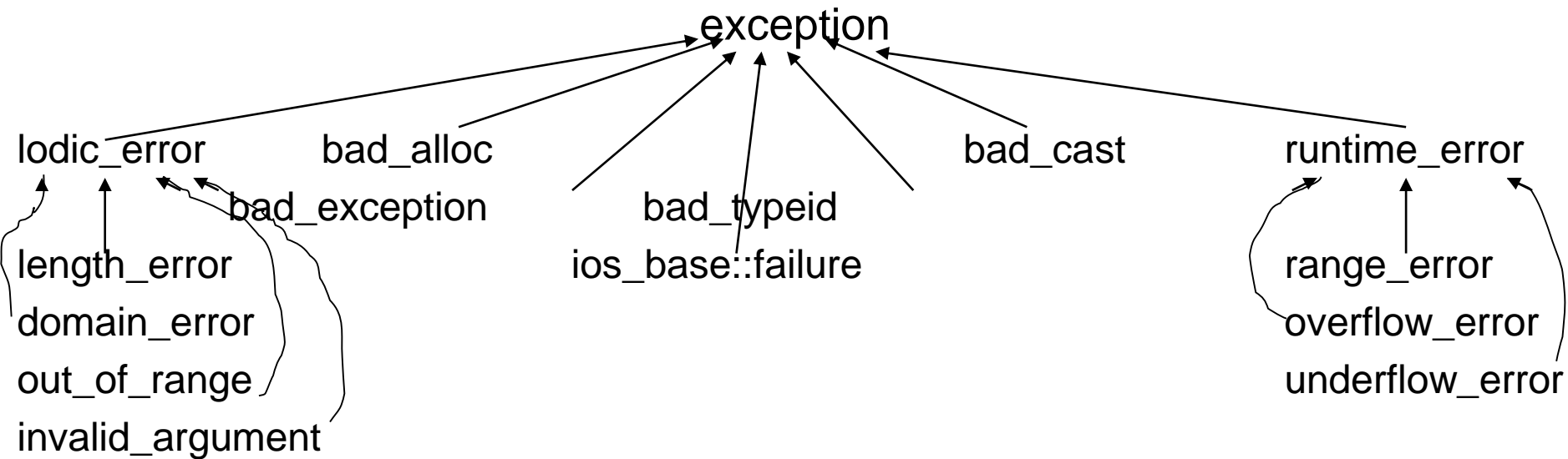
Стандартные исключения

Текст по генерации стандартных исключений вставляется компилятором.

Стандартные исключения объединены в иерархию классов, в вершине которой находится стандартный абстрактный библиотечный класс ***exception***, описанный в `<stdexcept>` :

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

Иерархия классов стандартных исключений



Из этих классов исключений мы рассматриваем только исключения типов:

bad_alloc – генерируется, если **new** не может выделить динамическую память, но если использовать при обращении к операции **new** параметр **nothrow**:

new (nothrow) T;

при невозможности выделения памяти **new** возвращает 0 (NULL).

bad_cast и **bad_typeid** – генерируются соответственно при неверной работе операций **dynamic_cast** и **typeid**, и расположенные в файле стандартной библиотеки `<typeinfo>`, а также

out_of_range – генерируется методом **at ()** контейнеров STL, и расположено в файле `<stdexcept>`.

Пример использования стандартных исключений

```
void f () {  
    try { ...  
        // использование стандартной библиотеки  
    }  
    catch (exception & e) {  
        cout << "Стандартное исключение" << e.what() << '\n';  
    }  
    catch (...) {  
        cout << "Другое исключение" << '\n';  
        ...  
    }  
}
```

Иерархию классов стандартной библиотеки можно брать за основу для своих исключений.