Механизм исключений в C++. RTTI

Лекция 9

Средства обработки ошибок. Исключения в C++

Обработка **исключительных ситуаций** в C++ 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 оператора **throw**, а перехватываться и обрабатываться в теле одного из обработчиков **catch,** которые располагаются непосредственно за **try**-блоком.

Исключение - объект некоторого типа, в частности, встроенного.

Или:

Исключение – это аномальное поведение во время выполнения, которое программа может обнаружить.

Средства обработки ошибок. Исключения в C++

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;*/}
```

Правила обработки исключений

- Обработчики (ловушки) просматриваются в порядке следования, выбирается соответствующий типу исключительной ситуации.
- Обработчик с параметром базового типа перехватывает исключительную ситуацию типа-наследника.
- Обработчик catch(…) перехватывает все исключительные ситуации, не перехваченные до него.
- Если обработчик так и не найден, выполняется стандартная функция terminate(), которая регистрирует аварийное завершение работы и вызывает abort()

Перехват исключений.

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 операторе **catch** можно указывать только его тип.

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

```
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 (void) {
      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 на предыдущем слайде.

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, получим:

Catch of Error_of_A

Если закомментировать строку

// **catch** (Error_of_A) { cerr << “Catch of Error_of_A \n”; },

получим

Catch of Error

Что будет напечатано в результате работы программы?

```
int f(int k) {
 try { k++;
    switch (k) {
    case '1': throw 2;
    case 1  : throw 1;
    case 0  : throw "Exception";      };
   return 100;
        }
 catch (int k){ cout << k << " catch1\n"; throw;}
 catch (const char*){ cout << " catch2\n"; return 50;}
}
class Box {int d;
public:  Box(int j) { d = f(j); cout << d <<endl;}   };
int main(){
  try {   Box b(-1),c(0),a(12); cout<<"Finish"<<endl;    }
  catch (int) { cout << "catch3\n";}
  catch (const char *) { cout << "catch4\n";}
  return 0; }
```

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.

Спецификация исключений в функциях

*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 недекларированное 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.

Жизненный цикл исключения

1. Создание временного объекта – копии исключительной ситуации.
2. Уничтожение объектов, созданных в try-блоке, с запуском для них не-обходимых деструкторов, освобождающих динамическую память (свертка стека)
3. Выход из try-блока.
4. Подбор и выполнение обработчика для данного try-блока в соответствии с типом исключительной ситуации (статическая ловушка).
5. Если необходимый обработчик для данного try-блока не найден или в обработчике имеется инструкция throw без параметров, сигнализирующая о незавершенности обработки исключительной ситуации, то происходит выход в объемлющий try-блок с повторением пунктов 2-4 (динамическая ловушка).
6. Если исключительная ситуация осталась необработанной после выхода из всех объемлющих try-блоков, то вызывается функция terminate().

```
class animal{ int weight;
   public:  ~animal(){cout<<“animal destructor"<<endl;}
  };
class elephant: public animal{
     int trunk_length;   char * color;
    public:  ~elephant(){cout<<"elephant"<<endl;}
   };
int main(){
  elephant jumbo;  animal* a = new elephant();
  try{
   try{elephant kuzia; throw "error"; cout<<"message 1"<<endl;}
   catch(int ) {cout<<"catch 1"<<endl;}
   catch(const void* ) {cout<<"catch 2"<<endl;throw;}
   catch(const char* s) {cout<<s<<"catch 3"<<endl;}
}
catch(const char* s){cout<<s<<" catch 4"<<endl;}
catch(...){cout<<"All exceptions”<<endl;}
delete a;
return 0;}
```

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: пусть Base - полиморфный класс, а - Derived - класс, производный от Base.

```
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

Оператор ***typeid*** наиболее полезен, 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 ( void ) {
        int i; Base *p, b_ob;
        Derived1 ob1; Derived2 ob2;
        cout << «Тип i - » << typeid(i).name( ) << endl;
        p = & b_ob;
        cout<<“p ук-ет на объект типа ”<<typeid(*p).name()<<endl;
        p = & ob1;
        cout<<“p ук-ет на объект типа ”<<typeid(*p).name()<<endl;
        p = & ob2;
        cout<<“p ук-ет 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;
}
```

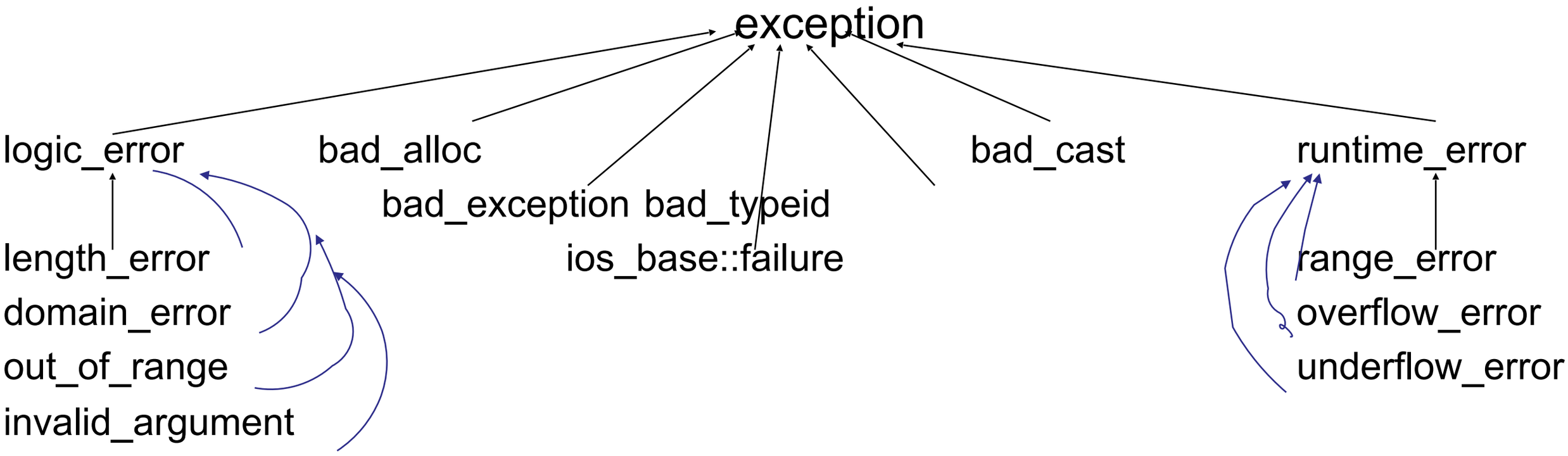
Стандартные исключения.

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

bad_cast и **bad_typeid**, генерируемые соответственно при неверной работе операций dynamic_cast и typeid, и расположенные в файле <typeinfo>,

out_of_range, генерируемое методом at () контейнеров STL, и расположенное в файле <stdexcept> ,

bad_alloc, генерируемое операцией **new** при невозможности выделения динамической памяти и расположенное в файле <new>.

Чтобы операция new при ошибке выделения динамической памяти возвращала 0, надо использовать следующую ее форму:

T * p = **new** (nothrow) T;

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.