КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 21.1 — Стандартные исключения

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Исключения (повторение)

Исключения предоставляют способ реагировать на исключительные обстоятельства (например, ошибки времени выполнения) в программах, передавая управление специальным функциям, называемым *обработчиками*.

Чтобы отловить (*catch*) исключения, часть кода подвергается проверке на исключения, что делается путем помещения этой части кода в блок **try**.

Когда в этом блоке возникает исключительная ситуация, выбрасывается (throw) исключение, в результате чего управление передается обработчику исключения.

Если исключение не сгенерировано, код продолжается нормально выполняться и все обработчики игнорируются.

Исключение выдается с помощью ключевого слова **throw** из блока **try**. Обработчики исключений *объявляются* с ключевым словом **catch**, которое должно быть помещено сразу после блока **try**:

```
#include <iostream>

int main () {
  try {
    throw 20;
  } catch(int e) {
    std::cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
}
An exception occurred. Exception Nr. 20
```

Код, подлежащий проверке на исключения заключен в блок **try**.

В примере выше этот код просто вызывает исключение:

```
throw 20;
```

Выражение **throw** принимает один параметр (в данном случае целочисленное значение 20), который передается в качестве аргумента обработчику исключений.

Обработчик исключений объявляется с ключевым словом **catch** сразу после закрывающей скобки блока **try**.

Синтаксис для **catch** аналогичен обычной функции с одним параметром.

Тип этого параметра очень важен, так как тип аргумента, передаваемого выражением **throw**, проверяется на соответствие типу параметра, и только в том случае, если они совпадают, исключение перехватывается данным обработчиком.

Несколько обработчиков (то есть, выражений **catch**) могут быть сцеплены — каждый с другим типом параметра. Выполняется только тот обработчик, тип аргумента которого соответствует типу исключения, указанному в операторе **throw**.

Ограничения — **goto** и **switch** не могут передавать управление внутрь блока **try** или в обработчик. Тем не менее, **goto**, **break**, **return**, **continue** могут передавать управление наружу из блока **try** или обработчика. Однако, если это присходит, переменные, определенные в блоке **try**, будут разрушены.

Если в качестве параметра **catch** используется многоточие (**...**), данный обработчик будет перехватывать любое исключение, независимо от того, какой тип параметра у этого исключения. Это можно использовать в качестве обработчика по умолчанию, который перехватывает все исключения, не обнаруженные другими обработчиками:

Пример

```
try {
  // code here
}
catch(int param) { std::cout << "int exception"; }
catch(char param) { std::cout << "char exception"; }
catch(...) { std::cout << "default exception"; }
```

В данном случае последний обработчик будет перехватывать любое исключение, тип которого не является ни **int**, ни **char**.

После обработки исключительной ситуации выполнение программы возобновляется после блока **try-catch**, а не после оператора **throw** !.

Также возможно вложить блоки **try-catch** в более внешние блоки **try**. В таких случаях у нас есть возможность перенаправить исключение из внутреннего блока **catch** на его внешний уровень. Это делается с помощью выражения **throw;** без аргументов. Например:

```
try {
    try {
        // code here
    } catch(int n) {
        throw;              // перенаправляем исключение наружу
    }
} catch(...) {
  cout << "Exception occurred";
}
```

Более детальный синтаксис

<try-block> :
    try <compound-statement> <handler-seq>
<function-try-block> :
    try [<ctor-initializer>] <compound-statement> <handler-seq>
<handler-seq> :
    <handler> <handler-seq>
<handler> :
    catch (<exception-declaration> **)** **<**compound-statement>
<exception-declaration> :
    [<attribute-specifier-seq>] <type-specifier-seq> <declarator>
    [<attribute-specifier-seq>] <type-specifier-seq> [abstract-declarator]
    ...
<throw-expression> :
    throw [<assignment-expression>]

<try-block> — это оператор.
Говорят, что код, выполняющий <try-block>, «генерирует исключение».
Код который в результате этого получает управление, называется «обработчиком».
<throw-expression> имеет тип **void**.
<function-try-block> —

A function-try-block associates a handler-seq with the ctor-initializer, if present, and the compound-statement.

Блок <function-try-block> связывает последовательность обработчиков <handler-seq> с инициализатором <ctor-initializer>, если он присутствует, и составным оператором <compound-statement>.

Исключение, генерируемое во время выполнения составного оператора или, при выполнении конструкторов и деструкторов, во время инициализации или уничтожения подобъектов класса, передает управление обработчику в блоке <function-try-block> так же, как и исключение, сгенерированное во время выполнения <try-block> передает управление другим обработчикам.

Пример

```
int f(int);
class C {
    int i;
    double d;
public:
    C(int, double);
};
C::C(int ii, double id) try : i(f(ii)), d(id) {
// constructor statements
} catch (...) {
// handles exceptions thrown from the ctor-initializer
// and from the constructor statements
}
```

Итак, генерация исключения (throwing) передает управление обработчику. Передается объект, и тип этого объекта определяет, какие обработчики могут его перехватить.

Пример

```
throw "Help!";
```

может быть перехвачен обработчиком типа **const char ***:

```
try {
// ...
}
catch(const char* p) {
// тут обработка исключения типа строка символов
}
```

Пример2

```
class Overflow {
public:
    Overflow(char, double, double);
};

void f(double x) {
    throw Overflow('+', x, 3.45e107); // конструктор
}
```

может быт перехвачено обработчиком исключения с типом **Overflow**

```
try {
    f(1.2);
} catch(Overflow& oo) {                          // перехват типа по ссылке
// тут обработка исключения типа Overflow
}
```

Спецификации исключения

В объявлении функции могут быть перечислены исключения, которые функция может прямо или косвенно вызвать. С этой целью в качестве суффикса декларатора этой функции используется спецификация исключений.

Синтаксис

<exception-specification> :

<dynamic-exception-specification>

<noexcept-specification>

<dynamic-exception-specification> :

throw ([<type-id-list>] **)**

<type-id-list> :

<type-id> [**...**]

<type-id-list> **,** <type-id> [**...**]

<noexcept-specification> :

noexcept (<constant-expression> **)**

noexcept

Спецификация исключения может появляться только в декларatore функции для:

- типа функции;
- указателя на тип функции;
- ссылки на тип функции;
- указателя на тип функции-члена, который является типом верхнего уровня объявления или определения, или для них появляется как параметр или тип возвращаемого значения в декларatore функции.

Спецификация исключения не должна появляться в объявлении typedef.

Пример

```
void f() throw(int);              // OK
void (*fp)() throw (int);         // OK
void g(void pfa() throw(int));    // OK
typedef int (*pf)() throw(int); // неправильно
```

Стандартные исключения

Этот заголовок определяет базовый класс для всех исключений, создаваемых элементами стандартной библиотеки — **`exception`**, а также несколько типов и утилит для помощи в обработке исключений.

Типы и классы

`exception` — стандартный класс исключения
`bad_exception` — исключение, созданное обработчиком неожиданных исключений
`nested_exception` — класс вложенных исключений
`exception_ptr` — указатель исключения (тип)
`terminate_handler` — тип функции-обработчика завершения
`unexpected_handler` — тип функции-обработчика неожиданных исключений

Функции

terminate() — функцию-обработчик завершения
get_terminate() — получить функцию-обработчик завершения
set_terminate() — установить функцию-обработчик завершения
unexpected() — функция обработки неожиданных исключений
get_unexpected() — получить функцию-обработчик неожиданных исключений
set_unexpected() — установить функцию-обработчик неожиданных исключений
uncaught_exception() — вернуть статус исключения
current_exception() — получить «умный» указатель на текущее исключение
rethrow_exception() — повторно генерировать исключение
make_exception_ptr() — make exception_ptr
throw_with_nested() — генерировать исключение вместе с вложенными
rethrow_if_nested() — повторно генерировать исключение вместе с вложенными

exception — базовый класс для стандартных исключений

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

Все объекты исключений, генерируемые компонентами стандартной библиотеки, являются производными от этого класса. Следовательно, все стандартные исключения могут быть перехвачены путем перехвата этого типа по ссылке.

Виртуальная функция-член **`exception::what()`** — возвращает строку, идентифицирующую исключение.

Пример

```
// exception example
#include <iostream>       // std::cerr
#include <typeinfo>       // operator typeid
#include <exception>      // std::exception

class Polymorphic {
  virtual void member() {}
};

int main () {
  try {
    Polymorphic *pb = 0;
    typeid(*pb);  // throws a bad_typeid exception
  } catch (std::exception& e) {
    std::cerr << "exception caught: " << e.what() << '\n';
} }
```

Вывод

```
exception caught: std::bad_typeid
```

Без try-блока (LC_MESSAGES=C)

```
terminate called after throwing an instance of 'std::bad_typeid'
  what():  std::bad_typeid
Aborted (core dumped) // Аварийный останов (стек памяти сброшен на диск)
```

Конструкторы exception::exception

(1) По умолчанию

```
exception( ) noexcept;
```

(2) Копирования

```
exception (const exception& e) noexcept;
```

C++98: Действия вызова функции-члена **what( )** после создания копии зависят от конкретной реализации библиотеки.

C++11: Каждое исключение в стандартной библиотеке C++ (включая и это) имеет, по крайней мере, конструктор копирования, который сохраняет строковое представление, возвращаемое членом **what( )**, если совпадают динамические типы.

Пример конструктора

```
#include <iostream>       // std::cout
#include <exception>      // std::exception

struct ooops : std::exception {
  const char* what() const noexcept {return "Ooops!\n";} // переопределяем
};

int main () {

  ooops e;
  std::exception* p = &e;
  try {
      throw e;        // throwing copy-constructs: ooops(e)
  } catch (std::exception& ex) {
      std::cout << ex.what();
  }
  try {
      throw *p;       // throwing copy-constructs: std::exception(*p)
  } catch (std::exception& ex) {
      std::cout << ex.what();
} }
```

Вывод

```
Ooops!
exception
```

exception::operator= — копировать исключение

```
#include <exception>

exception& operator= (const exception& e) noexcept;
```

Копирует объект типа **exception**

Каждое исключение в стандартной библиотеке C++ (включая и это) имеет, по крайней мере, перегруженный оператор присваивания копированием, который сохраняет строковое представление, возвращаемое членом **what()**, если совпадают динамические типы.

e — другой объект типа **exception**.

exception::what() — получить строку, идентифицирующую исключение

```
#include <exception>

virtual const char* what( ) const noexcept;
```

Возвращает последовательность символов с завершающим нулем, которая может использоваться для идентификации исключения. Конкретное представление, на которое указывает возвращаемое значение, определяется реализацией.

Поскольку это виртуальная функция, производные классы могут ее переопределить, чтобы возвращались конкретные значения.

Гарантируется, что возвращаемая строка будет действительной, по крайней мере до тех пор, пока объект, полученный из исключения, не будет уничтожен или пока не будет вызвана неконстантная функция-член объекта исключения.

exception::~exception — разрушает объект-исключение

```
#include <exception>

virtual ~exception;
```

Производный клас может переопределить этот деструктор.

Пример

```
// exception virtual destructor
#include <iostream>       // std::cout
#include <exception>      // std::exception
#include <cstring>        // std::strlen, std::strcpy

// Класс text_exception использует с-строку в динамически выделенной памяти
// для преопределения what()
class text_exception : public std::exception {
  char* text_;
public:
  text_exception(const char* text) {
    text_ = new char[std::strlen(text) + 1];
    std::strcpy(text_, text);
  }
  text_exception(const text_exception& e) {
    text_ = new char[std::strlen(e.text_) + 1];
    std::strcpy (text_, e.text_);
  }
  ~text_exception() throw() { // noexcept
    delete[] text_;
  }
  const char* what() const noexcept {return text_;}
};
```

```
int main () {

  try {
      throw text_exception("custom text");
  } catch (std::exception& ex) {
      std::cout << ex.what();
  }
  return 0;
}
```

Производные типы базового исключения

bad_alloc — исключение, возникающее при сбое выделения памяти
bad_cast — исключение, возникающее при сбое динамического приведения типа
bad_exception — исключение, сгенерированное обработчиком неожиданного исключения
bad_function_call — исключение, возникающее при неправильном вызове
bad_typeid — исключение, возникающее при **typeid()** нулевого указателя
bad_weak_ptr — плохой слабый указатель
ios_base::failure — базовый класс для исключений потока (открытый член класса)
logic_error — исключение логической ошибки
runtime_error — исключение ошибки времени выполнения

bad_alloc — исключение, возникающее при сбое выделения памяти

Пример bad_alloc

```
#include <iostream>     // std::cout
#include <new>          // std::bad_alloc

int main () {

  try {
    int* myarray= new int[10000000000]; //
  } catch (std::bad_alloc& ba) {
    std::cerr << "bad_alloc caught: " << ba.what() << '\n';
  }
  return 0;
}
```

Вывод

```
bad_alloc caught: bad allocation
```

bad_exception — исключение, сгенерированное обработчиком неожиданного исключения

Это особый тип исключения, специально разработанный для того, чтобы быть перечисленным в спецификаторе динамического исключения функции, то есть в ее спецификаторе **throw()**.

Если функция, для которой в ее спецификаторе динамического исключения указан **bad_exception**, генерирует исключение, не указанное в нем, и неожиданно повторно его генерирует, или генерирует любое другое исключение, которое не указанно в спецификаторе динамического исключения, автоматически генерируется это самое **bad_exception**.

Пример bad_exception

```
#include <iostream>       // std::cerr
#include <exception>      // std::bad_exception, std::set_unexpected

void myunexpected () {
  std::cerr << "unexpected handler called\n";
  throw;                                  // перенаправляем исключение наружу
}

void myfunction () throw(int, std::bad_exception) {
  throw 'x'; // генерирует тип char (нет в спецификаторе throw())
}
```

```
int main (void) {

  std::set_unexpected (myunexpected);
  try {
    myfunction();
  } catch (int) {
      std::cerr << "caught int\n";
  } catch (std::bad_exception be) {
      std::cerr << "caught bad_exception\n";
  } catch (...) {
      std::cerr << "caught some other exception\n";
  }
  return 0;
}
```