

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Исключения

ВОПРОСЫ

- ⊙ Что такое специализация шаблона класса? Какие ограничения накладываются на специализацию шаблона класса?
- ⊙ Возможно ли унаследовать один шаблон класса от другого? Какие существуют ограничения при наследовании шаблонов и чем они вызываются?
- ⊙ Каким образом можно обратиться к методам базового шаблона при наследовании?

ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ В C++

ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

- ⊙ Инструкции, которые могут возбуждать исключения, должны быть заключены в **try-блок**.
- ⊙ Такой блок начинается с ключевого слова **try**, за которым идет последовательность инструкций, заключенная в фигурные скобки, а после этого – список обработчиков, называемых **catch-предложениями**.
- ⊙ **try**-блок группирует инструкции программы и ассоциирует с ними обработчики исключений.

БЛОК CATCH

- ⊙ В языке C++ исключения обрабатываются в предложениях `catch`. Когда какая-то инструкция внутри `try`-блока возбуждает исключение, то просматривается список последующих предложений `catch` в поисках такого, который может его обработать
- ⊙ Catch-обработчик состоит из трех частей:
 - ⊙ ключевого слова `catch`,
 - ⊙ объявления одного типа или одного объекта, заключенного в круглые скобки (оно называется *объявлением исключения*),
 - ⊙ и составной инструкции.

ПРИМЕР БЛОКА CATCH

```
catch ( pushOnFull ) {
    cerr << "trying to push value on a full stack\n";
    return;
}
catch ( popOnEmpty ) {
    cerr << "trying to pop a value on an empty stack\n";
    return;
}
catch ( const char * ) {
    cerr << "some const char* exception\n";
    return;
}
catch ( ... ) {
    cerr << "some const char* exception\n";
    return;
}
```

Блок catch

- ⊙ Для обработки исключения выбирается тот обработчик, для которого типы в объявлении исключения и в возбужденном исключении совпадают (или могут быть приведены друг к другу)
- ⊙ После завершения обработчика выполнение приложения возобновляется с инструкции, идущей за последним `catch`-обработчиком в списке.

ОБРАБОТЧИК CATCH

- ◎ Обработчик исключительной ситуации считается найденным, если тип в блоке `catch`
 - ◎ Тот же, что и в `throw` (T, const T, T&, const T&)
 - ◎ Является открытым базовым классом для класса исключения, указанного в `throw`
 - ◎ Является указателем, который может быть преобразован к типу указателя в `throw`
 - ◎ В качестве параметра `catch` вместо класса указано «...», что данный блок перехватывает и обрабатывает любой тип исключений

```
catch ( ... ) {  
    cerr << "I'm your last chance!!!\n";  
}
```


ОБЪЯВЛЕНИЕ ИСКЛЮЧЕНИЯ

- ⊙ Объявлением исключения в catch-обработчике могут быть объявления типа или объекта.
- ⊙ Объявление типа исключения позволяет идентифицировать исключительную ситуацию, но не позволяет получить дополнительную информацию о возникшем исключении.

```
catch ( pushOnFull ) {  
    cerr << "trying to push value on a full stack\n";  
    return;  
}
```

ОБЪЕКТ-ИСКЛЮЧЕНИЕ

- ⊙ Если необходимо получить значение или как-то манипулировать объектом, созданным в выражении `throw` создается объект-исключение.
- ⊙ При перехвате объектов-исключений (или ссылок на объекты исключения) инструкции внутри `catch`-обработчика могут обращаться к информации, сохраненной в объекте выражением `throw`.

ГЕНЕРАЦИЯ ОБЪЕКТА-ИСКЛЮЧЕНИЯ

11

```
// НОВЫЙ класс исключения:  
// он сохраняет значение, которое не удалось поместить в стек  
class pushOnFull {  
public:  
    pushOnFull( int i ) : _value( i ) { }  
    int value { return _value; }  
private:  
    int _value;  
};  
  
void iStack::push( int value )  
{  
    if ( full() )  
        throw pushOnFull( value );  
    // ...  
}
```

ПЕРЕХВАТ И ОБРАБОТКА ОБЪЕКТА-ИСКЛЮЧЕНИЯ

- ⊙ В объявлении исключения в `catch`-обработчике появляется определение ссылки на объект `eObj`, с помощью которого вызывается функция-член `value()` класса `pushOnFull`

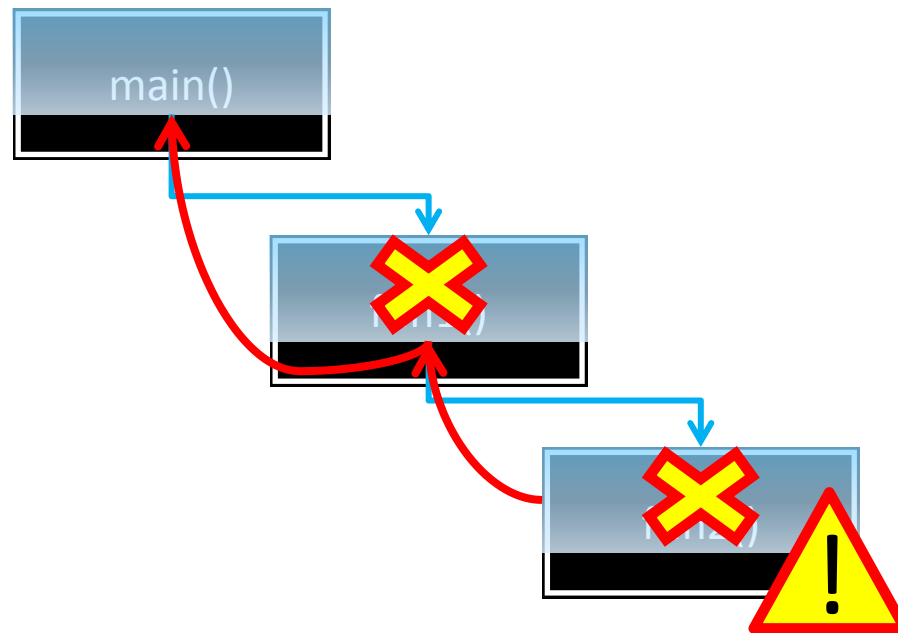
```
catch ( pushOnFull &eObj ) {  
    cerr << "trying to push value << "eObj.value()  
        << "on a full stack\n";  
}
```

РАСКРУТКА СТЕКА ИСКЛЮЧЕНИЯ

ОБРАБОТКА ВОЗНИКАЮЩЕГО ИСКЛЮЧЕНИЯ

- ⊙ Когда выражение `throw` находится в `try`-блоке, все ассоциированные с ним предложения `catch` исследуются с точки зрения того, могут ли они обработать исключение.
- ⊙ Если подходящее предложение `catch` найдено, то исключение обрабатывается. В противном случае поиск продолжается в вызывающей функции.
- ⊙ Этот поиск последовательно проводится во всей цепочке вложенных вызовов.

РАСКРУТКА СТЕКА



РАСКРУТКА СТЕКА

- ⊙ Процесс, при котором приложение, при возникновении исключительной ситуации, последовательно покидает составные блоки (вложенные блоки инструкций и вызовы функций) в поисках блока `catch`, способного обработать возникшее исключение, называется *раскруткой стека*.
- ⊙ По мере раскрутки прекращают существование локальные объекты, объявленные в блоках, из которых произошел выход.
- ⊙ **Вызываются деструкторы локальных объектов.**

ПОВТОРНОЕ ВОЗБУЖДЕНИЕ ИСКЛЮЧЕНИЙ

- ① Передать исключение другому `catch`-обработчику можно с помощью *повторного возбуждения исключения*.
- ① Для этой цели предусмотрена конструкция `throw`, которая может быть вызвана только `catch`-обработчика.
- ① Если объект-исключение передается по ссылке, то можно изменить состояние объекта-исключения и передать обновленный объект при инициализации `throw`.

```
void calculate( int op ) {  
try {  
    // исключение, возбужденное mathFunc(), имеет значение zeroOp  
    mathFunc( op );  
}  
catch ( EHstate &eObj ) {  
    // что-то исправить  
    // модифицируем объект-исключение  
    eObj = severeErr;  
    throw;  
}  
}
```

ОБРАБОТКА БЕЗ CATCH

- ⊙ Если обработчик исключения не находится, вызывается функция `terminate()` из стандартной библиотеки C++.
- ⊙ По умолчанию `terminate()` активизирует функцию `abort()`, которая аномально завершает программу.
- ⊙ Можно переопределить функцию `terminate()` если требуется отдать последние почести погибающему приложению.

RAII - RESOURCE ACQUISITION IS INITIALIZATION

RESOURCE ACQUISITION IS INITIALIZATION

- ⊙ Получение ресурса есть инициализация (англ. Resource Acquisition Is Initialization (RAII)) — шаблон проектирования объектно-ориентированного программирования, смысл которого заключается в том, что получение некоторого ресурса совмещается с инициализацией, а освобождение — с уничтожением объекта.

ПРИМЕР РАБОТЫ БЕЗ RIIA

- ◎ В Java значениями переменных являются не сами объекты, а ссылки на них.
- ◎ Поэтому для всех объектов память выделяется динамически, а сами объекты имеют неопределённое время жизни.
- ◎ В связи с этим, освободить ресурс в деструкторе невозможно, так как неизвестно, когда объект будет удалён (и будет ли удалён вообще). Поэтому в Java нет деструкторов, которые бы гарантированно вызывались.

ПРИМЕР РАБОТЫ БЕЗ RIIA

```
try {  
    File file = new File("/path/to/file");  
    // Do stuff with file  
} finally {  
    file.close();  
}
```

ПРИМЕР РАБОТЫ БЕЗ RIIA

- ⊙ Та же проблема в C++ решается по-другому: посредством закрытия файла в деструкторе объекта. Так как при раскручивании стека всегда вызываются деструкторы локальных объектов, файл закроется в любом случае.

```
try {  
    File file = new File("/path/to/file");  
    // Do stuff with file  
}  
//whatever
```


ИСКЛЮЧЕНИЯ В КОНСТРУКТОРАХ И ДЕСТРУКТОРАХ

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ В КОНСТРУКТОРАХ

- ⊙ Конструкторы не могут возвращать коды ошибок, соответственно исключения – это единственный метод, чтобы понять, что в конструкторе что-то пошло не так.
- ⊙ Но необходимо правильно обрабатывать исключительные ситуации в таких случаях.
- ⊙ Особенно если в конструкторе формируются объекты в динамической памяти.

ПРИМЕР ИСКЛЮЧИТЕЛЬНОЙ СИТУАЦИИ В КОНСТРУКТОРЕ

```
class Test {
public:
    Test() {
        std::cout << "Test::Test()" << std::endl;
        // Здесь, в соответствии с RAII, захватили ресурсы
        if ( 1 ) { throw std::runtime_error( "AAAAAAAA" );
        } else {}
    }
    ~Test() {std::cout << "Test::~~Test()" << std::endl;
        // А здесь мы освобождаем те самые важные ресурсы...}
};

int main() {
    Test* t = 0;
    try { t = new Test();    // Вроде бы создали...
    } catch ( const std::exception& exc ) {
        std::cout << exc.what() << std::endl;
    }
    delete t;    // Удалили? Ну-ну...
    return 0;
}
```

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ В КОНСТРУКТОРАХ

- ⊙ Не стоит пытаться выдавать исключения за тело конструктора, т.к. иначе, даже если мы не забываем вызвать `delete`, он не будет вызывать деструктор неправильно созданного объекта.
- ⊙ В этом случае, никакие ресурсы, которые были инициализированы в конструкторе до вызова исключения (связи с БД, открытые файлы, и т.п.) уже никогда не будут освобождены (если они создавались в динамической памяти).

РЕШЕНИЕ ПРОБЛЕМЫ?

- ⊙ Вместо указателей на объекты в динамической памяти использовать один из множества типов «умных указателей», которые удаляются автоматически при выходе из области видимости (как локальные объекты)

```
class Cnt {
private:
    X *xa;
    X *xb;
public:
    Cnt(int a, int b) {
        cout << "Cnt::Cnt" << endl;
        xa = new X(a);
        xb = new X(b);
    }
    ~Cnt() {
        cout << "Cnt::~~Cnt" << endl;
        delete xa;
        delete xb;
    }
};
```

```
class Cnt {
private:
    auto_ptr<X> ia;
    auto_ptr<X> ib;
public:
    Cnt(int a, int b) : ia(new X(a)),
        ib(new X(b)) {
        cout << "Cnt::Cnt" << endl;
    }
    ~Cnt() {
        cout << "Cnt::~~Cnt" << endl;
    }
};
```

УКАЗАТЕЛЬ AUTO_PTR

- ◎ `std::auto_ptr<>` - реализует семантику владения.
- ◎ Реализует так называемое *разрушающее копирование* - при присваивании, объект передается от одного указателя другому, удаляясь у первого, чтобы не удалять объект дважды при выходе из области видимости.

ИСКЛЮЧИТЕЛЬНАЯ СИТУАЦИЯ В ДЕКТРУКТОРЕ

```
class test
{
public:
    test() { }
    ~test(){
        throw std::runtime_error("Game over!");
    }
};

int main() {
    try {
        test t;
        throw std::runtime_error("Error!");
    }
    catch(std::exception const&)
    {}
    return 0;
}
```

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ В ДЕСТРУКТОРАХ

- ⊙ Когда исключение покидает блок, все локальные объекты, созданные в этом блоке, уничтожаются.
- ⊙ Если деструктор объекта, уничтожаемого во время развертки стека, генерирует исключение, то программа будет завершена досрочно, и ее уже ничего не спасет – вызывается функция `terminate`.

ЧТО ДЕЛАТЬ С ОШИБКАМИ В ДЕСТРУКТОРЕ?

- ⊙ Запишите данные об ошибке в лог.
- ⊙ Выведете сообщение на принтер, отправьте по SMS, позвоните и сообщите об этом бабушке...
- ⊙ **НО НИ В КОЕМ СЛУЧАЕ НЕ ИНИЦИАЛИЗИРУЙТЕ ИСКЛЮЧЕНИЕ**
- ⊙ Исключение при раскрытке стека всегда будет инициализировать функцию `terminate()`.
- ⊙ **Обрабатывайте все ошибки деструктора внутри деструктора.**

- ⊙ Исключительные ситуации обрабатываются в блоке `catch`, который идет после `try`.
- ⊙ После `try` может идти несколько блоков `catch`, каждый из которых настроен на перехват определенного типа исключений
- ⊙ При возникновении исключения инициализируется раскрутка стека, заключающаяся в вызове деструкторов локальных объектов и передаче исключения вызывающей функции.
- ⊙ Если обработчик не найден, вызывается функция `terminate()` которая вызывает функцию `abort()`
- ⊙ Получение ресурса есть инициализация (англ. Resource Acquisition Is Initialization (RAII) – метод работы с ресурсами в C++
- ⊙ С исключительными ситуациями в конструкторах надо разбираться очень тщательно, иначе можете потерять память или выделенные в конструкторе ресурсы (используйте умные указатели)
- ⊙ НЕ ДАВАЙТЕ ИСКЛЮЧЕНИЯМ ВЫЙТИ ИЗ ДЕКТРУКТОРА