

**Подготовительная
программа по
программированию на
C/c++**

Занятие №8

Валентина Глазкова

Обработка исключительных ситуаций

- Понятие исключительной ситуации
- Технология обработки исключительных ситуаций
- Классы исключений в стандартной библиотеке
- Реализация журналирования

Понятие исключительной ситуации (1 / 2)

- Естественный порядок функционирования программ нарушают возникающие **нештатные ситуации**, в большинстве случаев связанные с ошибками времени выполнения.
- В языке C++ такие штатные ситуации называются **исключительными (исключениями, exception)**.
- Примерами исключительных ситуаций являются:
 - нехватка оперативной памяти;
 - попытка доступа к элементу коллекции по некорректному индексу;
 - попытка недопустимого преобразования динамических типов и пр.

Понятие исключительной ситуации (2 / 2)

Архитектурной особенностью механизма обработки исключительных ситуаций в языке C++ является принципиальная независимость (несвязность) фрагментов программы, где исключение **возбуждается** (генерируется) и где оно **обрабатывается** (перехватывается).

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 базовых, например, `char*`). Такие объекты называются **объектами-исключениями**.

Жизненный цикл объектов-исключений начинается с возбуждения исключительной ситуации посредством оператора `throw`:

```
throw "Illegal cast";           // const char *
/* ... */
throw IllegalCast(); // class IllegalCast
/* ... */
enum      EPrgStatus
        {psOK, psBadIndex, psIllegalCast};
throw psIllegalCast; // enum EPrgStatus
```

Защищенные блоки.

Операторы `try`, `catch`

Небезопасные с точки зрения исключений фрагменты кода могут инкапсулироваться в **защищенный** (**контролируемый**) **блок**, за которым следует один или несколько связанных с ним **блоков-обработчиков исключений**. Защищенные блоки кода начинаются ключевым словом `try`, блоки-обработчики — словом `catch`.

При возбуждении исключения в одном из операторов защищенного блока все следующие за ним операторы блока автоматически пропускаются, а управление передается ближайшему соответствующему блоку-обработчику исключения (т.е. **первому подходящему** для этого блоку в порядке их указания в исходном коде).

Если такой блок не найден, управление берет на себя стандартная функция `terminate()`.



Защищенные блоки.

Операторы try, catch: пример 1

```
void foo()
{
    /* ... */
    throw "Illegal id";
    /* ... */           // не выполняется
}

void bar()
{
    try {
        foo();
        /* ... */       // не выполняется
    }
    catch(const char *s) {
        std::cout << s << std::endl;
    }
}
```



Операторы try, catch: пример 2

```
void main () {
```

```
try {
```

```
    int i; cin >> i;
```

```
    if (i > 0) {
```

```
        throw i;
```

```
        i = 0;
```

```
    } else {
```

```
        throw "Need positive number!";
```

```
    }
```

```
}
```

```
catch (int a) {
```

```
    cout << "Error #" << a << endl;
```

```
}
```

```
catch (const char* s) {
```

```
    cout << s << endl;
```

```
}
```

```
}
```

Генерация исключения типа `int`,
представленного переменной `i`

Не будет вызвано

Генерация исключения
типа `const char*`,
представленного
строковым литералом

Блок обработки исключений типа
`int`, сохраняемых в переменную `a`

Блок обработки исключений типа
`const char*`, сохраняемых в
переменную `s`



Операторы try, catch: пример 3

```
void main () {  
    try {  
        throw 'a'; // Is not handled by catch(int)  
    }  
    catch (int a) {  
        cout << "Error #" << a << endl;  
    }  
    catch (...) {  
        cout << "Unknown error!" << endl;  
    }  
}
```

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**. Например:

```
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, созданных в текущем стековом кадре, 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 может помещаться (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(...) { /* ... */ };
```



Универсальный блок-обработчик: пример

```
void main () {  
    try {  
        int i; cin >> i;  
        if (i == 0) {  
            throw 1.5;  
        }  
    }  
    catch (int a) {  
        cout << "Error #" << a << endl;  
    }  
    catch (const char* s) {  
        cout << s << endl;  
    }  
    catch (...) {  
        cout << "Unknown error!" << endl;  
    }  
}
```

Для перехвата всех возможных исключений (не обработанных предыдущими директивами catch) используется троеточие. Если не перехватить исключение в программе, оно будет обработано ОС и пользователь получит сообщение о падении программы



Повторное возбуждение исключений: пример

```
void g() { throw 1; }

void f() {
    try { g(); }
    catch (int) {
        throw;
    }
}

void main () {
    try {
        f();
    }
    catch (int a) {
        cout << "Error #" << a << endl;
    }
}
```

Запись `throw` без параметров означает, что обработанный объект исключения передается дальше по стеку обработки (на более объемлющий уровень). Новый объект в этом случае не создается, а используется тот же самый.

Варианты описания исключений

Описание исключения в блоке-обработчике может содержать тип объекта, ссылку или указатель на объект.

```
catch(IllegalCast)      { /* ... */ }
```

```
catch(IllegalCast ic)   { /* ... */ }
```

```
catch(IllegalCast &ric) { /* ... */ }
```

```
catch(IllegalCast *pic) { /* ... */ }
```



Исключения при создании глобальных объектов

```
class A {  
public:  
    A() { throw 1; }  
};  
A theA;  
void main () {  
    try {  
        // Не может обработать исключения,  
        // возникающие при создании  
        // глобальных объектов  
    }  
    catch (A& a) {  
        cout << "Never called";  
    }  
}
```


Безопасность ПО

как показатель качества

В «промышленном программировании» **безопасность ПО** рассматривается как один из **структурных показателей качества** продукта, оцениваемых путем **статического анализа** архитектуры, состава используемых компонентов, **исходного кода** и схемы БД.

Для обеспечения «структурной безопасности» исходного кода необходимо соблюдение стандартов разработки архитектуры и **стандартов кодирования**.

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 архитектуры (*уровень представления, уровень бизнес-логики, уровень (базы) данных*).

Примечание: Ряд функций стандартной библиотеки языка C++ имеет статус потенциально небезопасных, то есть способных привести к переполнению буфера либо иным дефектам и уязвимостям.

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

Безопасность классов и методов (1 / 2)

Функция C++ безопасна, если не возбуждает никаких исключений (или, что то же самое, 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 гарантий влечет за собой вызов функции `unexpected()`, по умолчанию вызывающей `terminate()`.

Виртуальные функции в производных классах могут **повторять** спецификации исключений функций в базовых классах или накладывать **более строгие** ограничения.

Безопасность классов и методов

(2 / 2)

Например:

```
// о б ъ я в л е н и я  ф у н к ц и й
int  foo(int &i) throw();
bool bar(char *pc = 0) throw(IllegalCast);
void foobar() throw(IllegalCast, BadIndex);
```

```
// о п р е д е л е н и я  ф у н к ц и и
int  foo(int &i) throw()
{ /* ... */ }
bool bar(char *pc = 0) throw(IllegalCast)
{ /* ... */ }
void foobar() throw(IllegalCast, BadIndex)
{ /* ... */ }
```

Безопасность конструкторов

Конструкторы **могут возбуждать исключения**, как и другие методы классов.

Для обработки **всех** исключений, возникших при исполнении конструктора, его тело и список инициализации должны быть совместно помещены в функциональный защищенный блок.

Например:

```
Beta::Beta(int value)
try
    : Alpha(foo(value))
{ // тело конструктора
    /* ... */
}
catch(...) { /* ... */ }
```



Исключения при создании класса

```
class B { public:
    B() { cout << "B::B()" << endl; }
    ~B() { cout << "B::~~B()" << endl; }
};

class D : public B { public:
    D() { cout << "D::D()" << endl; throw 1; }
    ~D() { cout << "D::~~D()" << endl; }
};

void main () {
    D* pd = 0;
    try { pd = new D; }
    catch (int) {
        cout << (pd == 0) << endl;
    }
}
```

```
B::B()
D::D()
B::~~B()
1
```

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*), иначе программа завершит работу аварийно.

Если возможность реакции на исключение необходима клиентам класса во время некоторой операции, в его открытом интерфейсе должна быть функция (не деструктор), которая эту 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 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).



Пользовательские классы исключительных ситуаций: пример 1

```
class A {};  
class B : public A {};  
void f() {  
    A a; A* pa = new A;  
    throw B();  
    delete pa;  
}  
void main () {  
    try {  
        f();  
    }  
    catch (A& a) {  
        cout << "Error handled" << endl;  
    }  
}
```



Пользовательские классы исключительных ситуаций: пример 2

```
class A {  
public:  
    class Error {};  
    void f() { throw Error(); }  
};  
  
void main () {  
    try {  
        A a;  
        a.f();  
    }  
    catch (A::Error) {  
        cout << "Handled" << endl;  
    }  
}
```

Исключения в стандартной библиотеке. Класс *exception*

Стандартная библиотека языка C++ содержит собственную иерархию классов исключений, являющихся прямыми или косвенными потомками базового класса `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();  
    };  
}
```

Потомки класса `exception` условно представляют две категории ошибок: **логические** ошибки и ошибки **времени исполнения**.

Классы – логические ошибки

В число классов категории «логические ошибки» входят **базовый** промежуточный класс `logic_error`, а также производные от его **специализированные** классы:

`invalid_argument` — ошибка «неверный аргумент»;

`out_of_range` — ошибка «вне диапазона»;

`length_error` — ошибка «неверная длина»;

`domain_error` — ошибка «вне допустимой области».

Классы – ошибки времени исполнения

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

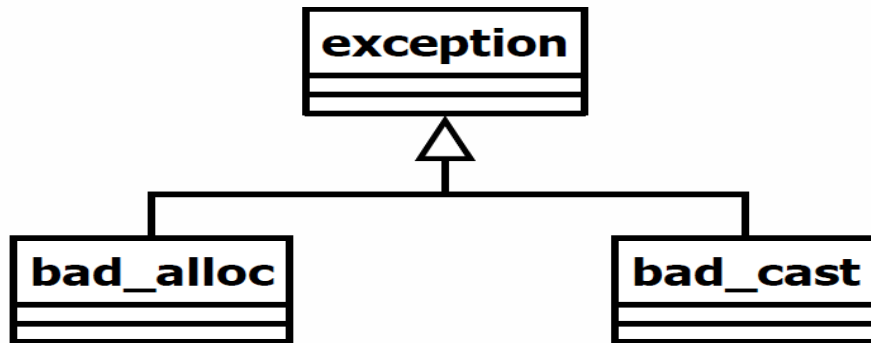
`range_error` — ошибка диапазона;

`overflow_error` — переполнение;

`underflow_error` — потеря значимости.

Прочие классы исключений стандартной библиотеки

Также производными от `exception` являются классы `bad_alloc` и `bad_cast`, сигнализирующие об ошибках при выделении динамической памяти и неуспешном выполнении «ссылочного» варианта операции `dynamic_cast`, соответственно.





Исключения при выделении динамической памяти

```
void f() throw(std::bad_alloc) {  
    // Code generates std::bad_alloc exception  
    while (1) {  
        int* p = new int [1000000];  
    }  
}  
  
void g() throw(const char*) {  
    // Code generates const char* exception  
    while (1) {  
        int* p = new (nothrow) int [1000000];  
        if (!p) throw "Out of memory";  
    }  
}
```

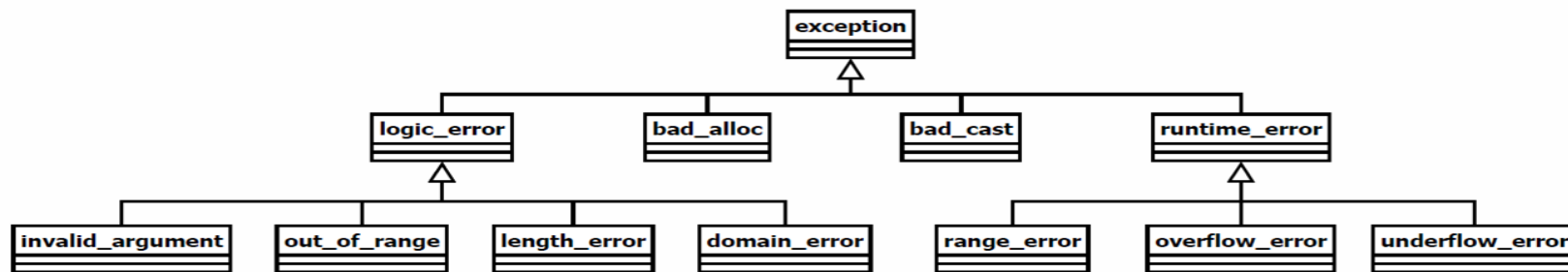



Исключения неуспешном выполнении «ссылочного» варианта операции `dynamic_cast`

```
class Base { virtual void foo() {} };  
class Derived : Base {};  
void main () {  
    try {  
        Base b;  
        // OK: dynamic_cast returns 0  
        Derived* pd = dynamic_cast<Derived*>(&b);  
        // Error: cannot initialize reference  
        Derived& rd = dynamic_cast<Derived&>(b);  
    }  
    catch (bad_cast& e) {  
        cout << "bad_cast caught: " << e.what();  
    }  
}
```

Объект `e` класса `bad_cast`, объявленного
в стандартной библиотеке из
пространства имен `std`

Классы исключений стандартной библиотеки: «вид сверху»





Исключения: пример 1

```
int f() {  
    try {  
        throw 'a';  
    }  
    catch(int) { cout << "err4"; }  
    catch(char) { cout << "err5"; throw; }  
}  
  
void main () {  
    try {  
        f();  
        throw 10;  
    }  
    catch (int) { cout << "err1"; }  
    catch (char) { cout << "err2"; }  
}
```

err5 err2



Исключения: пример 2

```
void f(char c) {  
    try {  
        if (c >= '1' && c <= '9') c++;  
        else throw "0";  
        try { if (c == '9') throw c; c++; }  
        catch (const char* c)  
        { cout << c << " 1 "; }  
        catch (char c)  
        { cout << c << " 2 "; throw; }  
        cout << c << " ";  
    }  
    catch (int k)  
    { cout << k << " 3 "; }  
    catch (const char* c)  
    { cout << c << " 4 "; }  
}
```

```
void main () {  
    try {  
        f('a'); // 0 4  
        f('2'); // 4  
        f('8'); // 9 2 9 5  
        f('5'); // not executed  
    }  
    catch (char c)  
    { cout << c << " 5 "; }  
    catch (...)  
    { cout << "all"; }  
}
```



Исключения: пример 3

```
struct A {  
  
    A () { cout << "A_Constr &" << endl ; }  
    A (const A & a) { cout << "A_Copy &" << endl ; }  
    ~A () { cout << "A_Destr\n"; }  
  
};  
  
struct B : A {  
  
    B () { cout << "B_Constr &" << endl; }  
    ~B () { cout << "B_Destr &" << endl ; }  
  
};  
  
void g() { B bg; throw bg;}  
  
int main () {  
  
    try { A a; g(); }  
    catch ( B & ) { cout << "B&_Catch &" << endl ; }  
    catch ( A & ) { cout << "A&_Catch &" << endl; }  
    return 0;  
  
}
```

```
A_Constr  
A_Constr  
B_Constr  
A_Copy  
B_Destr  
A_Destr  
A_Destr  
B&_Catch  
B_Destr
```



Исключения: пример 4

```
void f(X & x, int n);
struct X {
    X () {
        try { f(*this, -1); cout << "a"; }
        catch (X){ cout << "b"; }
        catch (int){ cout << "c"; }
    }
    X (X &) { cout << "d"; }
    virtual ~X () { cout << "e"; }
};
struct Y: X {
    Y () {
        try { f (*this, 0); cout << "f"; }
        catch (Y) { cout << "g"; }
        catch (int){ cout << "h"; }
        cout << "i";
    }
    Y (Y &){ cout << "j"; }
    ~Y () { cout << "k"; }
};
```

l a d e m e

```
void f(X & x, int n) {
    try {
        if (n < 0) throw -n;
        else if (n == 0) throw x;
        else throw n;
    }
    catch (int){ cout << "l"; }
}

int main() {
    try { Y a; }
    catch (...){ cout << "m"; return 1; }

    cout << "n"; return 0;
}
```



Исключения: пример 5

```
struct S {  
    S (int a) {  
        try { if (a > 0) throw *this; else if (a < 0) throw 0; }  
        catch (S & ) { cout << "SCatch_S&" << endl; }  
        catch (int) { throw; }  
        cout << "SConstr &" << endl;  
    }  
    S (const S & a) { cout << "Copy &" << endl; }  
    ~S ( ) { cout << "Destr &" << endl; }  
};  
  
int main ( ) {  
    try { S s1(1), s2(-2); cout << "Main &" << endl; }  
    catch (S &) { cout << "MainCatch_S& &" << endl; }  
    catch ( ... ) { cout << "MainCatch_... &" << endl; }  
    return 0;  
}
```

Copy
SCatchS&
Destr
SConstr
Destr
MainCatch...



Исключения: пример 6 (1/2)

```
struct X; void f(X & x, int n);
struct X {
    X() {
        try {f(*this, -1); cout << 1 << endl;}
        catch (X) {cout << 2 << endl;}
        catch (int) {cout << 3 << endl;}
    }
    X (X &) {cout << 4 << endl; }
    ~X () {cout << 5 << endl; }
};
struct Y: X {
    Y () {f(*this, -1); cout << 6 << endl; }
    Y (Y &) {cout << 7 << endl; }
    ~Y () {cout << 8 << endl; }
};
```




Исключения: пример 6 (2/2)

```
void f(X & x, int n) {  
    try{  
        if (n < 0) throw x; if (n > 0) throw 1; cout << 9 << endl;  
    }  
    catch (int) { cout << 10 << endl; }  
    catch (X& a) { cout <<11<<endl; f(a, 1); cout <<12<<endl; throw;}  
}  
  
int main() {  
    try { Y a; }  
    catch (...) { cout << 13 << endl; return 0; }  
    cout << 14 << endl;  
    return 0;  
}
```

4 11 10 12 2 5 4 11 10 12 5 13 5

Реализация журналирования

- Одним из простейших и вместе с тем эффективных инструментов отладки и фиксации действий программы является **журналирование**
- **Журналирование** предполагает вывод в файл (или на экран) информации о событиях, возникающих в программе, и о промежуточном состоянии программы (например: факт вызова функции и данные о ее аргументах, содержимое переменных, тип исключительной ситуации и сообщение об ошибке и т.п.)
- Отладочные сообщения должны находиться в ключевых узлах программы и позволять отследить ход ее выполнения
- Вывод на экран буферизируется построчно ('\n')
- В случае фатальной ошибки, приводящей к аварийному завершению программы (наиболее распространена ошибка сегментации - Segmentation Fault) отладочные сообщения, находящиеся в буферах, не будут выведены
- Часто бывает удобно иметь возможность оперативно отключать и включать отладочные выводы



Реализация журналирования: пример (1/2)

Ф а й л debug.h

```
#ifndef DEBUG_H
#define DEBUG_H
#include <stdio.h>
#define PDEBUG(level, fmt, args,...)
#ifdef DEBUG
#undef PDEBUG
#define PDEBUG(level, fmt, args,...)
    if(level <= DEBUG)
        printf("%s: %d: " fmt, __FUNCTION__, __LINE__, ## args)
#endif
#endif
```



Реализация журналирования: пример (2/2)

```
#include <stdio.h>
```

```
#define DEBUG 10
```

```
#include "debug.h"
```

```
int main()
```

```
{
```

```
    int i = 0;
```

```
    while(i < 6)
```

```
    {
```

```
        PDEBUG(1, "i = %d", i);
```

```
        i++;
```

```
    }
```

```
}
```

```
$ ./debug
```

```
main: 10: i = 0
```

```
main: 10: i = 1
```

```
main: 10: i = 2
```

```
main: 10: i = 3
```

```
main: 10: i = 4
```

```
main: 10: i = 5
```

Валентина Глазкова

Спасибо за внимание!