

9

EXCEPTION

320

Program Errors

- ▶ Kinds of errors with programs
 - Poor logic - bad algorithm
 - Improper syntax - bad implementation
 - Exceptions - Unusual, but predictable problems
- ▶ The earlier you find an error, the less it costs to fix it
- ▶ Modern compilers find errors early

Exception 9

Object Oriented Programming

321

Paradigm Shift from C

- ▶ In C, the default response to an error is to continue, possibly generating a message
- ▶ In C++, the default response to an error is to terminate the program
- ▶ C++ programs are more “brittle”, and you have to strive to get them to work correctly
- ▶ Can catch all errors and continue as C does

Exception 9

Object Oriented Programming

322

assert()

- ▶ a macro (processed by the precompiler)
 - Returns TRUE if its parameter is TRUE
 - Takes an action if it is FALSE
 - abort the program
 - throw an exception
- ▶ If DEBUG is not defined, asserts are collapsed so that they generate no code

Exception 9

Object Oriented Programming

323

assert() (cont'd)

- ▶ When writing your program, if you know something is true, you can use an assert
- ▶ If you have a function which is passed a pointer, you can do
 - `assert(pTruck);`
 - if `pTruck` is 0, the assertion will fail
- ▶ Use of assert can provide the code reader with insight to your train of thought

Exception 9

Object Oriented Programming

324

assert() (cont'd)

- ▶ Assert is only used to find programming errors
- ▶ Runtime errors are handled with exceptions
 - `DEBUG false =>` no code generated for assert
 - `Animal *pCat = new Cat;`
 - `assert(pCat);` // bad use of assert
 - `pCat ->memberFunction();`

Exception 9

Object Oriented Programming

325

Exception 9

assert() (cont'd)

- ▶ assert() can be helpful
- ▶ Don't overuse it
- ▶ Don't forget that it "instruments" your code
 - invalidates unit test when you turn DEBUG off
- ▶ Use the debugger to find errors

Object Oriented Programming
326

Exception 9

Exceptions

- ▶ You can fix poor logic (code reviews, debugger)
- ▶ You can fix improper syntax (asserts, debugger)
- ▶ You have to live with exceptions
 - Run out of resources (memory, disk space)
 - User enters bad data
 - Floppy disk goes bad

Object Oriented Programming
327

Exception 9

Why are Exceptions Needed?

- ▶ The types of problems which cause exceptions (running out of resources, bad disk drive) are found at a low level (say in a device driver)
- ▶ The low level code implementer does not know what your application wants to do when the problem occurs, so s/he "throws" the problem "up" to you

Object Oriented Programming
328

Exception 9

How To Deal With Exceptions

- ▶ Crash the program
- ▶ Display a message and exit
- ▶ Display a message and allow the user to continue
- ▶ Correct the problem and continue without disturbing the user

Steinbach's Corollary to Murphy's Law:
 "Never test for a system error you don't know how to handle."

Object Oriented Programming
329

Exception 9

What is a C++ Exception?

- ▶ An object
 - passed from the area where the problem occurs
 - passed to the area where the problem is handled
- ▶ The type of object determines which exception handler will be used

Object Oriented Programming
330

Exception 9

Syntax

```

try {
    // a block of code which might generate an exception
}
catch(xNoDisk) {
    // the exception handler(tell the user to
    // insert a disk)
}
catch(xNoMemory) {
    // another exception handler for this "try block"
}
          
```

Object Oriented Programming
331

Exception 9

The Exception Class

- ▶ Define like any other class:

```
class Set {
private:
    int *pData;
public:
    ...
    class xBadIndex {}; // just like any other class
};
```

Object Oriented Programming
332

Exception 9

Throwing An Exception

- ▶ In your code where you reach an error node:

```
if (memberIndex < 0)
    throw xBadIndex();
```
- ▶ Exception processing now looks for a catch block which can handle your thrown object
- ▶ If there is no corresponding catch block in the immediate context, the call stack is examined

Object Oriented Programming
333

Exception 9

The Call Stack

- ▶ As your program executes, and functions are called, the return address for each function is stored on a push down stack
- ▶ At runtime, the program uses the stack to return to the calling function
- ▶ Exception handling uses it to find a catch block

Object Oriented Programming
334

Exception 9

Passing The Exception

- ▶ The exception is passed up the call stack until an appropriate catch block is found
- ▶ As the exception is passed up, the destructors for objects on the data stack are called
- ▶ There is no going back once the exception is raised

Object Oriented Programming
335

Exception 9

Handling The Exception

- ▶ Once an appropriate catch block is found, the code in the catch block is executed
- ▶ Control is then given to the statement after the group of catch blocks
- ▶ Only the active handler most recently encountered in the thread of control will be invoked

Object Oriented Programming
336

Exception 9

Handling The Exception (cont'd)

```
catch (Set::xBadIndex) {
    // display an error message
}
catch (Set::xBadData) {
    // handle this other exception
}
//control is given back here
```

- ▶ If no appropriate catch block is found, and the stack is at main(), the program exits

Object Oriented Programming
337

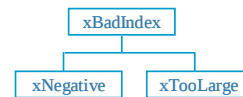
Default catch Specifications

- Similar to the switch statement


```
catch (Set::xBadIndex)
{ // display an error message }
catch (Set::xBadData)
{ // handle this other exception }
catch (...)
{ // handle any other exception }
```

Exception Hierarchies

- Exception classes are just like every other class; you can derive classes from them
- So one try/catch block might catch all bad indices, and another might catch only negative bad indices



Exception Hierarchies (cont'd)

```
class Set {
private:
    int *pData;
public:
    class xBadIndex {};
    class xNegative : public xBadIndex {};
    class xTooLarge: public xBadIndex {};
};
// throwing xNegative will be
// caught by xBadIndex, too
```

Data in Exceptions

- Since Exceptions are just like other classes, they can have data and member functions
- You can pass data along with the exception object
- An example is to pass an error subtype for xBadIndex, you could throw the type of bad index

Data in Exceptions (Continued)

```
// Add member data,ctor,dctor,accessor method
class xBadIndex {
private:
    int badIndex;
public:
    xBadIndex(int iType):badIndex(iType) {}
    int GetBadIndex() { return badIndex; }
    ~xBadIndex() {}
};
```

Passing Data In Exceptions

```
// the place in the code where the index is used
if (index < 0)
    throw xBadIndex(index);
if (index > MAX)
    throw xBadIndex(index);
// index is ok
```

Exception 9

Getting Data From Exceptions

```

catch (Set::xBadIndex &theException)
{
    int badIndex = theException.GetBadIndex();
    if (badIndex < 0 )
        cout << "Set Index " << badIndex << " less than 0";
    else
        cout << "Set Index " << badIndex << " too large";
    cout << endl;
}

```

Object Oriented Programming 344

Exception 9

Caution

- ▶ When you write an exception handler, stay aware of the problem that caused it
- ▶ Example: if the exception handler is for an out of memory condition, you shouldn't have statements in your exception object constructor which allocate memory

Object Oriented Programming 345

Exception 9

Exceptions With Templates

- ▶ You can create a single exception for all instances of a template
 - declare the exception outside of the template
- ▶ You can create an exception for each instance of the template
 - declare the exception inside the template

Object Oriented Programming 346

Exception 9

Single Template Exception

```

class xSingleException {};

template <class T>
class Set {
private:
    T *pType;
public:
    Set();
    T& operator[] (int index) const;
};

```

Object Oriented Programming 347

Exception 9

Each Template Exception

```

template <class T>
class Set {
private:
    T *pType;
public:
    class xEachException {};
    T& operator[] (int index) const;
};
// throw xEachException();

```

Object Oriented Programming 348

Exception 9

Catching Template Exceptions

- ▶ Single Exception (declared outside the template class)


```
catch (xSingleException)
```
- ▶ Each Exception (declared inside the template class)


```
catch (Set<int>::xEachException)
```

Object Oriented Programming 349

Exception 9

Exception Specification

- ▶ A function that might throw an exception can warn its users by specifying a list of the exceptions that it can throw.

```
class Zerodivide{/*...*/};
int divide (int, int) throw (Zerodivide);
```

- ▶ If your function never throws any exceptions

```
bool equals (int, int) throw();
```

- ▶ Note that a function that is declared without an exception specification such as `bool equals (int, int)`; guarantees nothing about its exceptions: It might throw any exception, or it might throw no exceptions.

Object Oriented Programming 350

Exception 9

Exception Specification

- ▶ Exception Specifications Are Enforced At Runtime
- ▶ When a function attempts to throw an exception that it is not allowed to throw according to its exception specification, the exception handling mechanism detects the violation and invokes the standard function **unexpected()**.
- ▶ The default behavior of **unexpected()** is to call **terminate()**, which terminates the program.
- ▶ The default behavior can be altered, nonetheless, by using the function **set_unexpected()**.

Object Oriented Programming 351

Exception 9

Exception Specification

- ▶ Because exception specifications are enforced only at runtime, the compiler might deliberately ignore code that seemingly violates exception specifications.
- ▶ Consider the following:

```
int f(); //no exception specification
```

- ▶ What if f throws an exception

```
void g(int j) throw()
{
    int result = f();
}
```

Object Oriented Programming 352

Exception 9

Concordance of Exception Specification

C++ requires exception specification concordance in derived classes. This means that an overriding virtual function in a derived class has to have an exception specification that is at least as restrictive as the exception specification of the overridden function in the base class.

Object Oriented Programming 353

Exception 9

Concordance of Exception Specification

```
class BaseEx{};
class DerivedEx: public BaseEx{};
class OtherEx {};
```

```
class A {
public:
    virtual void f() throw (BaseEx);
    virtual void g() throw (BaseEx);
    virtual void h() throw (DerivedEx);
    virtual void i() throw (DerivedEx);
    virtual void j() throw(BaseEx);
};
```

```
class D: public A {
public:
    void f() throw (DerivedEx); //OK
    void g() throw (OtherEx); //error
    void h() throw (DerivedEx); //OK
    void i() throw (BaseEx); //error
    void j() throw (BaseEx,OtherEx); //error
};
```

Object Oriented Programming 354

Exception 9

Concordance of Exception Specification

An exception could belong to two groups:

```
class Netfile_err : public Network_err, public File_system_err {
    /* ... */
};
```

Netfile_err can be caught by functions dealing with network exceptions:

```
void f(){
    try {
        // something
    }
    catch (Network_err& e) {
        // ...
    }
}
```

Object Oriented Programming 355

```

void g() {
    try {
        // something else
    }
    catch(File_system_err& e) {
        // ...
    }
}

```

Exception Matching

```

void f() {
    try {
        throw E();
    }
    catch(H) {
        // when do we get here?
    }
}

```

The handler is invoked:

- [1] If **H** is the same type as **E**.
- [2] If **H** is an unambiguous public base of **E**.
- [3] If **H** and **E** are pointer types and [1] or [2] holds for the types to which they refer.
- [4] If **H** is a reference and [1] or [2] holds for the type to which **H** refers.

Resource Management

When a function acquires a resource – that is, it opens a file, allocates some memory from the free store, sets an access control lock, etc., – it is often essential for the future running of the system that the resource be properly released.

```

void use_file(const char* fn)
{
    FILE* f = fopen(fn, "w");
    // use f
    fclose(f);
}

```

Resource Management

Fault-tolerant implementation using try-catch:

```

void use_file(const char* fn)
{
    FILE* f = fopen(fn, "r");
    try {
        // use f
    }
    catch (...) {
        fclose(f);
        throw;
    }
    fclose(f);
}

```

Resource Management

The problem with this solution is that it is verbose, tedious, and potentially expensive.

```

class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a) { p = fopen(n,a); }
    File_ptr(FILE* pp) { p = pp; }
    ~File_ptr() { fclose(p); }
    operator FILE*() { return p; }
};

```

```

void use_file(const char* fn) {
    File_ptr f(fn, "r");
    // use f
}

```

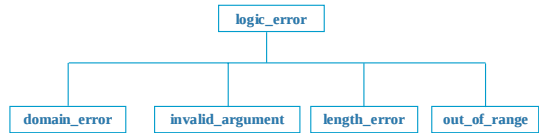
Standard Exceptions

- The C++ standard includes some predefined exceptions, in `<stdexcept>`
- The base class is **exception**
 - Subclass **logic_error** is for errors which could have been avoided by writing the program differently
 - Subclass **runtime_error** is for other errors

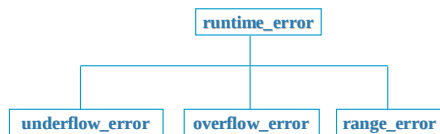
Standard Exceptions

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

Logic Error Hierarchy



Runtime Error Hierarchy



The idea is to use one of the specific classes (e.g. range_error) to generate an exception

Data For Standard Exceptions

```
// standard exceptions allow you to specify
// string information
throw overflow_error("Doing float division in function div");

// the exceptions all have the form:
class overflow_error : public runtime_error
{
public:
    overflow_error(const string& what_arg)
        : runtime_error(what_arg) {};
```

Catching Standard Exceptions

```
catch (overflow_error)
{
    cout << "Overflow error" << endl;
}

catch (exception& e)
{
    cout << typeid(e).name() << ": " << e.what() << endl;
}
```

More Standard Exception Data

- catch (exception& e)
 - Catches all classes derived from *exception*
 - If the argument was of type *exception*, it would be converted from the derived class to the exception class
 - The handler gets a *reference to exception* as an argument, so it can look at the object

RTTI (RunTime Type Information)

Exception 9

- ▶ It's one of the more recent additions to C++ and isn't supported by many older implementations. Other implementations may have compiler settings for turning RTTI on and off.
- ▶ The intent of RTTI is to provide a standard way for a program to determine the type of object during runtime.
- ▶ Many class libraries have already provided ways to do so for their own class objects, but in the absence of built-in support in C++, each vendor's mechanism typically is incompatible with those of other vendors.
- ▶ Creating a language standard for RTTI should allow future libraries to be compatible with each other.

What is RTTI for?

Exception 9

- ▶ Suppose you have a hierarchy of classes descended from a common base. You can set a base class pointer to point to an object of any of the classes in this hierarchy. Next, you call a function that, after processing some information, selects one of these classes, creates an object of that type, and returns its address, which gets assigned to a base class pointer.
- ▶ How can you tell what kind of object it points to?

How does it work?

Exception 9

C++ has three components supporting RTTI:

- ▶ **dynamic_cast** pointer
generates a pointer to a derived type from a pointer to a base type, if possible. Otherwise, the operator returns 0, the null pointer.
- ▶ **typeid** operator
returns a value identifying the exact type of an object.
- ▶ **type_info** structure
holds information about a particular type.



RTTI works only for classes with virtual functions

dynamic_cast<>

Exception 9

- ▶ The `dynamic_cast` operator is intended to be the most heavily used RTTI component.
- ▶ It doesn't answer the question of what type of object a pointer points to.
- ▶ Instead, it answers the question of whether you can safely assign the address of the object to a pointer of a particular type.

```
class Grand { // has virtual methods } ;
class Superb : public Grand { ... } ;
class Magnificent : public Superb { ... } ;
```

```
Grand * pg = new Grand;
Grand * ps = new Superb;
Grand * pm = new Magnificent;
```

```
Magnificent * p1 = (Magnificent *) pm; // #1
Magnificent * p2 = (Magnificent *) pg; // #2
Superb * p3 = (Magnificent *) pm; // #3
```

Which of the previous type casts are safe?

```
Superb pm = dynamic_cast<Superb *>(pg);
```

Exception 9

```
class Grand {
    virtual void speak() ;
};
class Superb : public Grand {
    void speak() ;
    virtual void say() ;
};
class Magnificent : public Superb {
    char ch ;
    void speak() ;
    void say() ;
};
```

```
for (int i = 0; i < 5; i++)
{
    pg = getOne();
    pg->speak();
    ...
}
```

Exception 9

- ▶ However, you can't use this exact approach to invoke the **say()** function; it's not defined for the **Grand** class.
- ▶ However, you can use the **dynamic_cast** operator to see if **pg** can be type cast to a pointer to **Superb**.
- ▶ This will be true if the object is either type **Superb** or **Magnificent**. In either case, you can invoke the **say()** function safely:

```
if (ps = dynamic_cast<Superb *>(pg))
    ps->say();
```

typeid

- ▶ **typeid** is an operator which allows you to access the type of an object at runtime
- ▶ This is useful for pointers to derived classes
- ▶ **typeid** overloads **==**, **!=**, and defines a member function **name**

```
if (typeid(*carType) == typeid(Ford))
    cout << "This is a Ford" << endl;
```

typeid().name

```
cout << typeid(*carType).name() << endl;
// If we had said:
// carType = new Ford();
// The output would be:
// Ford
▶ So:
cout << typeid(e).name()
returns the name of the exception
```

e.what()

- ▶ The class **exception** has a member function **what**

```
virtual char* what();
```
- ▶ This is inherited by the derived classes
- ▶ **what()** returns the character string specified in the throw statement for the exception

```
throw overflow_error("Doing float division in function div");
cout << typeid(e).name() << ": " << e.what() << endl;
```

Deriving New exception Classes

```
class xBadIndex : public runtime_error {
public:
    xBadIndex(const char *what_arg = "Bad Index")
        : runtime_error(what_arg) {}
};
// we inherit the virtual function what
// default supplementary information character string
```

```
template <class T>
class Array{
private:
    T *data ;
    int Size ;
public:
    Array(void);
    Array(int);
    class eNegativeIndex{};
    class eOutOfBounds{};
    class eEmptyArray{};
    T& operator[](int) ;
};
```

```
template <class T>
Array<T>::Array(void){
    data = NULL ;
    Size = 0 ;
}
template <class T>
Array<T>::Array(int size){
    Size = size ;
    data = new T[Size] ;
}
```

```
template <class T>
T& Array<T>::operator[](int index){
    if( data == NULL ) throw eEmptyArray() ;
    if(index < 0) throw eNegativeIndex() ;
    if(index >= Size) throw eOutOfBounds() ;
    return data[index] ;
}
```

```
Array<int> a(10) ;
try{
    int b = a[200] ;
}
catch(Array<int>::eEmptyArray){
    cout << "Empty Array" ;
}
catch(Array<int>::eNegativeIndex){
    cout << "Negative Array" ;
}
catch(Array<int>::eOutOfBounds){
    cout << "Out of bounds" ;
}
```