Object Oriented Programming

Exceptions

July 30, 2009 Dr. Mordo Shalom

Program Failures

- A program (also a good one) may fail for endless reasons:
  - A non-existent library, or file
  - A badly formatted input
  - An inaccessible resource:
    - Network?
    - Permissions?
  - Not enough memory.
  - ...

Run Time Failures (Errors)

- The common part of these conditions is that they can detected only at run-time.
- In C when we detect such a condition, we have the following options:
  - To stop the program (too drastic).
  - To return some error value.

Error Reporting in C

- Stopping the program:

```
inline int& Vect::operator[](int i) //obtain range checked element
{
   if(i<0 || i>=size)
   {
      cerr<<"illegal Vect index: "<<i<<"\n";
      exit(1);
   }
   return p[i];
}
```

Error Reporting in C

- Returning an error value:

```
int DoThisAndThat(int x, int y, File *fp)
{
   if( fp == NULL ) // exceptional situation
      return -1;
   if ( x/y > 100000 )// another exceptional situation
      return 0;
   // otherwise proceed
   //...
}
```

- Problems:
  - We might not have "free" return values.
  - The calling function might not know how to handle the situation.

Error Reporting/Handling in modern languages

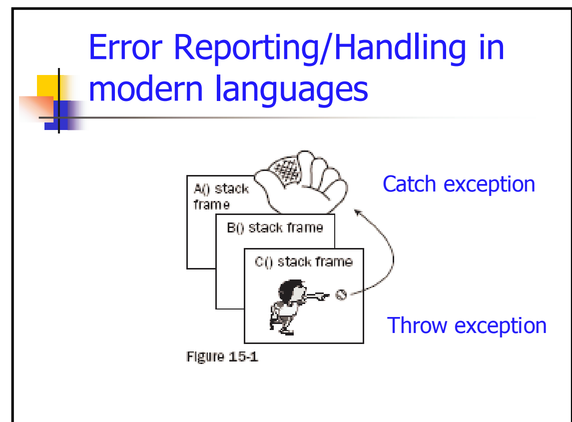


Figure 15-1

The exception mechanism

- An exception is an object containing information about the error condition.
- The detecting function reports the error by throwing an exception.
- The handling is done by an ancestor function by catching the exception.
- A stack rewinding occurs and control passes to the catching function.

Advantages over return values

- No “free” values needed.
- Impossible to forget checking for exceptions.
- It propagates upwards automatically.

throw vs. return

- A function to release control in two ways:
  - return <value> // normal
  - throw <exception> // error

```
int ff(...)
{
   //...
   if (...)
      throw X;
   //...
   return Y;
}
```

Exception Handling (catch)

- The invocation has to be within a try block.

```
try {
   // ...
   ff(x);
   // ...
}
```

- The handling is done in a catch block.

```
try {
   // ...
}
catch (int err){
   cout << “caught: “<< err << endl;
}
catch (const char* err_msg){
   cout << “Error: “ << err_msg << endl;
}
```

Exception Handling-cont’d

- One or more catch blocks.
- When exception is thrown, control passes to one of the catch blocks.
- The choice of the appropriate catch block is by the type of thrown object.
- When catch block ends control passes to the block after the catch blocks.

```
try {
   // ...
   ff(x);
   // ...
}
catch (int err){
   cout << “caught: “<< err << endl;
}
catch (const char* err_msg){
   cout << “Error: “ << err_msg << endl;
}
```

Common Exception situations

- Wrong input
- Unsuccessful memory allocation
- Unsuccessful I/O operation
- Unsuccessful dynamic_cast
- Invalid memory accesses.

Common Exception Types

- Exception can be of any type (as opposed to Java)
- However, an exception should be of a special type that is dedicated to this purpose (as in Java).
- A set of classes throwing exceptions, are accompanied by a set of exception classes.
- Exception classes are sometimes empty.

The standard exception type

- There is a standard exception type that we can inherit from it:

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

```
using std::exception;
```

The standard exception type

- Usually an exception type inherits from the class “exception”. In this way:
- we inherit some functionality (the most important being what()).
- We get some standardization
- STL behaves similarly.
- Any block catching an “exception” will catch our exception too (rarely unwanted).

An int-throwing example

```
class Vect
{
public:
   //constructors and destructor
   Vect(int n = DEFAULTSIZE); //create a size n array
   Vect(int a[], unsigned int n); //initialization by array
   Vect(const Vect& v); //copy constructor
   ~Vect() { delete []p; }
   int Size() const {return size;}
   Vect& operator=(const Vect& v);
   int& operator[](int i);
   const int& operator[](int i) const;
   const Vect& operator()(int *a, unsigned int n);
   operator int*()const; //casting operator
   friend ostream& operator<<(ostream&, Vect&);
   friend istream& operator>>(istream&, Vect&);
   friend ofstream& operator<<(ofstream&, Vect&);
   friend ifstream& operator>>(ifstream&, Vect&);
protected:
   int *p; //base pointer
private:
   enum{DEFAULTSIZE = 10}; //DEFAULTSIZE must be positive!!!
   int size; //number of elements
};
```

An int-throwing example

```
Vect::Vect(int n)
: size(n)
{
    if (n < 0)
        throw -1;
    if (n == 0)
        throw 0;
    if (n < 5)
        throw 1;
    if (n > 10000)
        throw 10;
    p = new int[n];
    for (int i = 0; i<n; i++)
        p[i] = 0;
}
```

```
int& Vect::operator[](int i)
{
    int n = Size();
    if (i >= 0 && i < Size())
        return p[i];
    throw 7;
    return p[0];   // to avoid warning
}
```

A client program

```
try
{
    Vect intArray(arraySize);
    for (int j = 0; j< arraySize; j++)
        intArray[j] = rand()%1000;
    while(1)
    {
        cout << "Enter Array index: ";
        cin >> index;
        cout << "intArray[" << index << "] = " <<intArray[index]<<"\n";
    }
} ....
```

```
catch (int ex){
    switch(ex)
    {
    case 7:
        cout << "Out of array boundary\n";
        break;
    case -1:
        cout << "You asked for an array";
        cout << " of negative number of objects!\n";
        break;
    case 0:
        cout << "You asked for an array";
        cout << " of negative number of objects!\n";
        break;
    case 1:
        cout << "This array is too small...\n";
        break;
    case 10:
        cout << "This array is too big...\n";
        break;
    default:;
    }
}
catch (...){
    cout << "Something went wrong!\n";
}
```

default catch

Using empty classes

```
class Vect
{
public:
    //constructors and destructor
    Vect(int n = DEFAULTSIZE); //create a size n array
    Vect(int a[], unsigned int n); //initialization by array
    Vect(const Vect& v); //copy constructor
    ~Vect() { delete []p; }
    int Size() const {return size;}
    Vect& operator=(const Vect& v);
    int& operator[](int i);
    const int& operator[](int i) const;
    const Vect& operator()(int *a, unsigned int n);
    operator int*()const; //casting operator
    friend ostream& operator<<(ostream&, Vect&);
    friend istream& operator>>(istream&, Vect&);
    friend ofstream& operator<<(ofstream&, Vect&);
    friend ifstream& operator>>(ifstream&, Vect&);
    // exception classes
    class xBoundary {};
    class xTooBig {};
    class xTooSmall{};
    class xZero {};
    class xNegative {};
protected:
    int *p; //base pointer
private:
    enum{DEFAULTSIZE = 10}; //DEFAULTSIZE must be positive!!!
    int size; //number of elements
};
```

```
Vect::Vect(int n)
: size(n)
{
    if (n < 0)
        throw xNegative();
    if (n == 0)
        throw xZero();
    if (n < 5)
        throw xTooSmall();
    if (n > 10000)
        throw xTooBig();
    p = new int[n];
    for (int i = 0; i<n; i++)
        p[i] = 0;
}
```

```
int& Vect::operator[](int i)
{
    int n = Size();
    if (i >= 0 && i < Size())
        return p[i];
    throw xBoundary();
    return p[0];   // to avoid warning
}
```

No change in try block

```
int main()
{
    size_t arraySize;
    int index;
    bool badSize;
    do{
        badSize = false;
        cout<<"Enter array size: ";
        cin>>arraySize;
        srand(time(0));
        try
        {
            Vect intArray(arraySize);
            for (int j = 0; j< arraySize; j++)
                intArray[j] = rand()%1000;
            while(1)
            {
                cout << "Enter Array index: ";
                cin >> index;
                cout << "intArray[" << index << "] = " <<intArray[index]<<"\n";
            }
        } ...
```

```
        catch (Vect::xBoundary){
            cout << "Out of array boundary\n";
        }
        catch (Vect::xTooBig) {
            cout << "This array is too big...\n";
            badSize = true;
        }
        catch (Vect::xTooSmall) {
            cout << "This array is too small...\n";
            badSize = true;
        }
        catch (Vect::xZero) {
            cout << "You asked for an array";
            cout << " of zero objects!\n";
            badSize = true;
        }
        catch (...){
            cout << "Something went wrong!\n";
        }
    } while(badSize);
    cout << "Done.\n";
    return 0;
}
```

default catch

Using non-empty classes

Using non-empty classes

- Sometimes we want to pass additional information with the error, besides of its type.
- We can also define a hierarchy of classes.

```
class Vect
{
public:
    // exception classes
    class xBoundary {
    public:
        xBoundary(int index):itsIndex(index){}
        virtual void PrintError(){
            cout << "boundary error. Received: " << itsIndex << endl;
        }
    private:
        int itsIndex;
    }; //end of class xBoundary

    class xSize{
    public:
        xSize(int size):itsSize(size) {}
        virtual int Size() { return itsSize; }
        virtual void PrintError(){
            cout << "Size error. Received: " << itsSize << endl;
        }
    protected:
        int itsSize;
    }; //end of class xSize
```

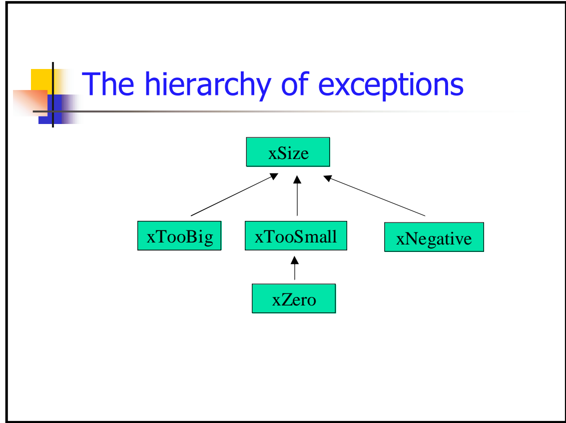
C'tor

method

```
    class xTooBig : public xSize{
    public:
        xTooBig(int size):xSize(size){}
        virtual void PrintError(){
            cout << "Too big! Received: " << itsSize << endl;
        }
    }; //end of class xTooBig

    class xTooSmall : public xSize{
    public:
        xTooSmall(int size):xSize(size){}
        virtual void PrintError(){
            cout << "Too small! Received: " << itsSize << endl;
        }
    }; //end of class xTooSmall

    class xZero  : public xTooSmall{
    public:
        xZero(int size):xTooSmall(size){}
        virtual void PrintError(){
            cout << "Zero!!. Received: " << itsSize << endl;
        }
    }; //end of class xZero

    class xNegative : public xSize{
    public:
        xNegative(int size):xSize(size){}
        virtual void PrintError(){
            cout << "Negative! Received: "<< itsSize << endl;
        }
    }; //end of class xNegative
```

```
    //constructors and destructor
    Vect(int n = DEFAULTSIZE)throw(xSize); //create a size n array
    Vect(int a[], unsigned int n); //initialization by array
    Vect(const Vect& v); //copy constructor
    ~Vect() { delete []p; }

    int Size() const {return size;}
    Vect& operator=(const Vect& v);
    int& operator[](int i);
    const int& operator[](int i) const;
    const Vect& operator()(int *a, unsigned int n);
    operator int*()const; //casting operator

    friend ostream& operator<<(ostream&, Vect&);
    friend istream& operator>>(istream&, Vect&);
    friend ofstream& operator<<(ofstream&, Vect&);
    friend ifstream& operator>>(ifstream&, Vect&);
    // ...
protected:
    int *p; //base pointer
private:
    enum{DEFAULTSIZE = 10}; //DEFAULTSIZE must be positive!!!
    int size; //number of elements
};
```

The hierarchy of exceptions



Catching the exception

```
catch (Vect::xBoundary& boundary)
{
    boundary.PrintError();
}

catch (Vect::xSize& sizeError)
{
    sizeError.PrintError();
}
```

catch parameter

exception class member function

Catching the exception

- Every catch block has an associated type
- No automatic conversions in catching exceptions. E.g. a catch block catching a double, will not catch a thrown int exception.
- However polymorphic catching is possible.
- If an exception can be caught by more than one catch block, the first one in the list is used.

Catching the exception-cont'd

- Corollary: A catch of a subtype can not appear after a catch a a super-type.
- If a const is thrown, than the catch should catch a const too.
- If no matching catch block is found, a stack unwinding occurs.
- If a stack unwindin[...]ogram aborts.

Microsoft Visual C++ Debug Library

Debug Error!

Program: ...

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

(Press Retry to debug the application)

Abort Retry Ignore

Catching all exceptions

- First way: The default catch block. Will catch everything that is not caught before.

```
catch (...)
{
    cout << "Something went wrong!\n";
}
```

- Second way: Catching the exception type. Will catch all objects that inherit from exception.

```
catch (exception&)
{
    cout << "Something went wrong!\n";
}
```

Cleaning up in exceptions

- When a stack unwinding occurs, all the automatic variables are (automatically) destroyed.
- However, cleaning up dynamically allocated memory and other resources is the programmer's responsibility.

Stack unwinding - a bogus example

```
int main()
{
  try {
    funcOne();
  } catch (exception& e) {
    cerr << "Exception caught!\n";
    exit(1);
  }
  return (0);
}
void funcOne() throw(exception)
{
  string str1;
  string* str2 = new string();
  funcTwo();
  delete str2;
}
void funcTwo() throw(exception)
{
  ifstream istr;
  istr.open("filename");
  throw exception();
  istr.close();
}
```

Memory leak if funcTwo() throws exception

Possible solution

```
void funcOne() throw(exception)
{
  string str1;
  string* str2 = new string();
  try {
    funcTwo();
  } catch (...) {
    delete str2;
    throw; // rethrow the exception
  }
  delete str2;
}
```

Re-throw exception

Possible solution 2: smart pointers

- We can wrap the pointers in a smart pointer object.
- The wrapped object will be an automatic variable.

```
template <class Type>
class auto_ptr{
  //...
};
```

C'tors throwing exceptions

- A c'tor does not return a value.
- However it can throw an exception like any other function.
- If and when this happens, all the objects constructed, are destroyed effectively using their respective d'tors.
  - Parent class' objects,
  - Contained objects.
- All the rest is the programmer's responsibility.

Throwing exceptions from d'tors

- D'tors can, but should not throw exceptions.
- Why?
  - Consider an exception thrown by a non d'tor function.
  - As a result, a stack unwinding usually occurs.
  - Consequently some d'tors may be invokes.
  - If one of them throws an exception, then the run-time system has two exceptions to deal with.

Throwing exceptions from d’tors

- It is possible to check if a d’tor is invoked from within a stack unwinding process.
- By using the uncaught_exception() function,
- Whose usage is complicated.
- Do not throw exceptions in d’tors, or use uncaught_exception()

Limiting throwables

- By default any type can be thrown by a function.
- However we can declare a function (in its prototype) to throw only certain types.
- In this case the run-time system checks that the thrown object is of one of these types.

```
void f(...) throw ( XBoundary, Xfoo)
{...}
```

throw list

throw list

- A function w/o a throw list, can throw anything.
- A function with an empty throw list can not throw.
- The throw list is not part of the signature. We can not overload a function by changing only its throw list.

throw list – cont’d

- The throw list is checked at run-time.
- Although, checking in compile time is perfectly doable.
- Java checks this at compile time.

```
void f(...) throw ( XBoundary, Xfoo)
{
   //...
   throw (5); // will compile
   //...
}
```

int

- This will cause an unexpected exception run-time error.

set_unexpected

- The default behavior can be changed:

```
void MyUnexpected()
{
   cout<<"Unexpected exception"<<endl;
   throw runtime_error("Runtime Error");
}

int main()
{
   // ...
   unexpected_handler old_unexpected; // in <eh.h>
   old_unexpected = set_unexpected(MyUnexpected);// save the old unexpected
   // ...
   set_unexpected(old_unexpected);
```

A C-style function ptr

*

set_terminate

- Similarly, the function to be called when an exception is not caught can be changed

```
void MyTerminate()
{
   cout<<"terminate called"<<endl;
   exit(1);
}

int main()
{
   // ...
   terminate_handler old_terminate;
   old_terminate = set_terminate(MyTerminate);// save the old terminate
   // ...
   set_terminate(old_terminate);
```

A C-style function ptr

*

bad_alloc

- The <new> library contains functions to handle memory allocation problems.
- The bad_alloc exception is thrown when memory can not be allocated.

```
try
{
    ptr = new int[numInts];
}
catch (bad_alloc& e) {
    cerr << "Unable to allocate memory!\n";
    // Handle memory allocation failure
}
```

*

bad_alloc

- This feature can be disabled in a per-allocation basis, using nothrow.

```
ptr = new(nothrow) int[numInts];
 if (ptr == NULL) {
     cerr << "Unable to allocate memory!\n";
     // Handle memory allocation failure
}
```

*

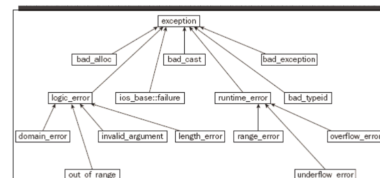
new_handler

bad_alloc can also be disabled by setting the new_handler.

```
#include <new>

void myNewHandler()
{
  cerr << "Unable to allocate memory. Terminating program!\n";
  abort();
}

int main(int argc, char** argv)
{
  // ...
  new_handler oldHandler = set_new_handler (myNewHandler);
  // ...
  set_new_handler(oldHandler);
  // ...
```

*

A hierarchy of standard exceptions



All the c'tors receive a string argument, to be returned by what().

invalid_argument and runtime_error

```
void readIntegerFile(const string& fileName, vector<int>& dest)
throw (invalid_argument, runtime_error)
{
    ifstream istr;
    int temp;
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception
        string error = "Unable to open file " + fileName;
        throw invalid_argument(error);
    }
    // Read the integers one-by-one and add them to the vector
    while (istr >> temp) {
        dest.push_back(temp);
    }
    if (istr.eof()) {
        // We reached the end-of-file.
        istr.close();
    } else {
        // Some other error. Throw an exception
        istr.close();
        string error = "Unable to read file " + fileName;
        throw runtime_error(error);
    }
}
```

```
int main(int argc, char** argv)
{
   // ...
   try {
      readIntegerFile(fileName, myInts);
   }

   catch (const invalid_argument& e) {
      cerr << e.what() << endl;
      exit (1);
   }

   catch (const runtime_error& e) {
      cerr << e.what() << endl;
      exit (1);
   }
   // ...
}
```