Exceptions

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

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

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

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

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();

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

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

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

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

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

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

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"
}
```

The Exception Class

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


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

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

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***

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

Handling The Exception (cont'd)

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

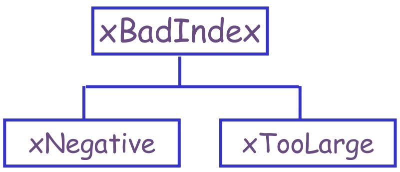
- If no appropriate catch block is found, and the stack is at main(), the program exits

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,dtor,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
```

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;
}
```


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

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

Single Template Exception

```
class xSingleException {};

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

Each Template Exception

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

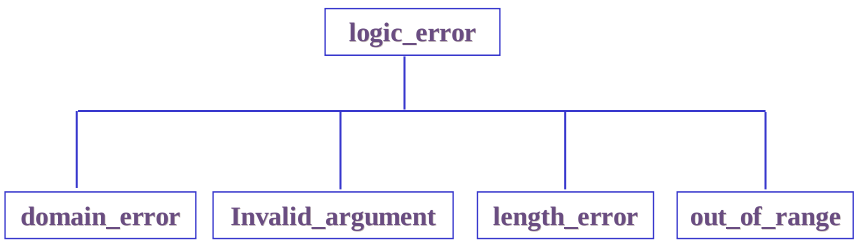

Catching Template Exceptions

- Single Exception (declared outside the template class)
  - catch (xSingleException)
- Each Exception (declared inside the template class)
  - catch (Set<int>::xEachException)

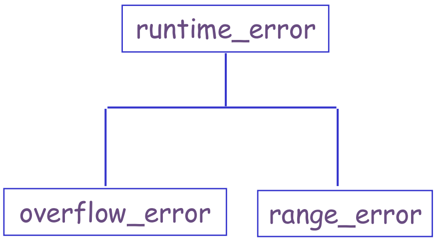
Standard Exceptions

- The C++ standard includes some predefined exceptions, in <stdexcept.h>
- 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

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

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" ;
}
```