Section 3.7
Exception Handling

1. Dealing with faults
2. Error handling
3. EH mechanisms
4. Stack unwinding

3.7.1 Dealing with Faults

- What is software robustness?
  - the degree to which a program will keep running despite faults
- What is a fault?
  - it’s a defect, or a bug
  - it’s something that makes your program crash
- No program is perfect!
  - never assume that your program is

Dealing with Faults (cont.)

- Types of faults:
  - bad input
    - wrong data type
    - unexpected format
    - abrupt termination
    - ... and many more ...
  - software bugs
    - array bounds, segmentation violations
    - dereferencing bad pointers, null pointers
    - ... and many, many more ...

Fault Prevention

- What is fault prevention?
  - writing software in a way that minimizes the number of faults
- How do we prevent faults?
  - follow the rules of good SE and good OO design
  - code reviews
  - testing, testing, testing

Fault Detection

- What is fault detection?
  - discovering faults before the user does
- How do we detect faults?
  - debugging
    - this activity is neither organized nor planned
  - testing
    - this activity is organized into test cases
    - it is part of the software development life cycle

Fault Detection (cont.)

- Testing
  - test the success paths
    - try every possible correct scenario
  - test the failure paths
    - try every possible error scenario
    - you don’t want the user to find your bugs

Fault Tolerance

- What is fault tolerance?
  - the ability of a program to keep running in the presence of faults
- How do we tolerate faults?
  - error checking
    - contracts, assertions
  - exception handling
    - an alternate flow of control when an error state is reached

3.7.2 Error Handling

- What are exceptions?
  - exceptional problems that occur during a program’s execution
  - they occur infrequently
    - they are not the same as regular errors
- Two approaches for dealing with errors:
  - inline error handling
  - exception handing

Error Handling (cont.)

- Inline error handling
  - it is the intermixing of program logic and error-handling logic
  - pseudo-code example:

```
do something;
if task did not execute correctly
  process error;
do something else;
if task did not execute correctly
  process error;
```

  - it is a good technique for basic error checking
  - it makes the program difficult to read, maintain, and debug

Error Handling (cont.)

- Exception handling
  - it is used to resolve exceptions, not common errors
  - possible courses of action when an exception occurs:
    - allow the program to continue execution
    - notify the user of the problem
    - terminate the program in a controlled manner
  - it promotes robustness and fault tolerance

Error Handling (cont.)

- Exception handling (EH) separates:
  - error reporting
    - finding a problem
  - error handling
    - handling the problem
- Finding and handling the problem occur in *different places*
- EH provides an alternate return structure
  - the normal function-call-and-return process is bypassed

Error Handling (cont.)

- The C++ standard library has an **`exception`** class
  - it is used as a base class for user-defined exception classes
  - the constructor takes a message string
  - the member function **`what()`** returns that message string

3.7.3 EH Mechanisms

- Example:

```
void func ( )
{
  float x, num, den;
  // initialize num and den
  try
  {
    if (den == 0)
    {
      throw "Divided by zero";
    }
    x = num / den;
  }
  catch (char * error)
  { cout << error; }
  // do more stuff
}
```

error checking

error reporting

error handling

- **`coding example <p1>`**

`try` / `throw` / `catch`

- **`try`**:
  - it’s a block of potentially dangerous code
  - this block may be anywhere on the function call stack

- **`throw`**:
  - it’s a statement that generates an exception
  - this statement may be located:
    - within a **`try`** block
    - within a function called inside a **`try`** block

- **`catch`**:
  - it’s a block that deals with the exception
  - this block must be located immediately following the **`try`** block

Example: `throw` in Called Function

```
void func ( )
{
  float x, num, den;
  // initialize num and den
  try
  { x = divide(num,den); }
  catch (char * error)
  { cout << error; }
  // ...
}
```

error handling

```
float divide (float a, float b)
{
  if (b == 0)
  { throw "divided by zero"; }
  return a/b;
}
```

error checking

error reporting

- **coding example <p2>**

Example: `throw` in Called Function

```
float divide (float a, float b)
{
  if (b == 0)
  { throw "divided by zero"; }
  return a/b;
}
```

error checking

error reporting

```
float middle (float a, float b)
{ return divide(a,b); }
```

```
void func ( )
{
  float x, num, den;
  // initialize num and den
  try
  { x = middle(num,den); }
  catch (char * error)
  { cout << error; }
  // ...
}
```

error handling

- **coding examples <p3> and <p4>**

Multiple `catch` Blocks

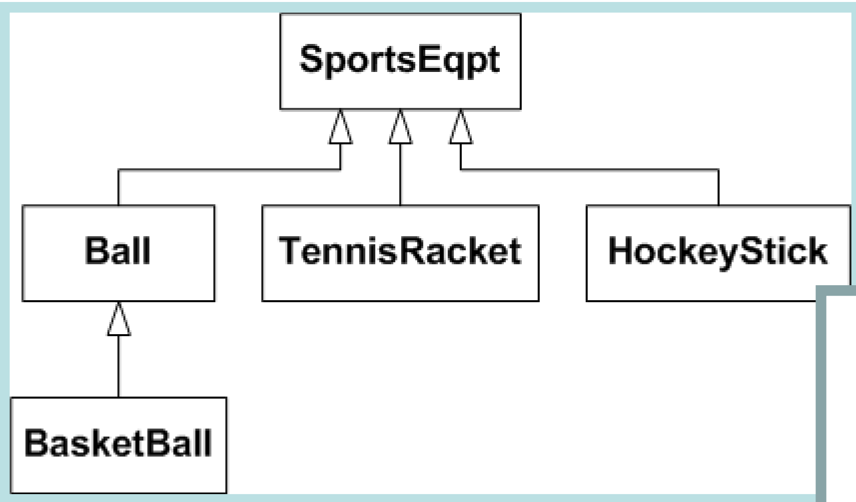
- A **`throw`** statement has one parameter
  - example: **`throw "divided by zero";`**

- **`catch`** block with matching parameter gets executed
  - example: **`catch (char* error) { }`**

- Catch-all feature: **`catch(...)`**
  - throw parameter cannot be used

- **`coding example <p5>`**

Example of Multiple `catch` Blocks




```
void func ( )
{
  try
  {
    if (/*condition*/) { throw Ball(); }
  }

  catch (TennisRacket& t) { }

  catch (HockeyStick& h)  { }

  catch (BasketBall& bb)  { }

  catch (Ball& b)         { }

  catch (SportsEqpt & s)  { }

  catch (…)               { }
}
```

- Order matters!
  - derived class
  - base class
  - catch all

Re-`throw`

- An exception may be caught and thrown again
- Why?
  - error handling in called functions
  - cleanup!
- A **`catch`** block may re-**`throw`**:
  - the same exception
  - a new exception

```
float divide(int a, int b)
{
  if (b == 0)
  { throw "div by zero"; }
  return float(a)/float(b);
}
```

```
float middle(int a, int b)
{
  try { return divide(a,b); }
  catch(char*)
  {
    cout<<"Caught in middle";
    throw;
  }
}
```

```
void func()
{
  float x;
  int num, den; // init num and den
  try { x = middle(num,den); }
  catch(char* error)
  { cout<<error; }
}
```

Throwing a New Exception

- One exception may be thrown and caught, then a different one thrown

- **`coding example <p6>`**

```
float divide(int a, int b)
{
  if (b == 0)
  { throw "div by zero"; }
  return float(a)/float(b);
}
```

```
void func()
{
  float x;
  int num, den; // init num and den
  try { x = middle(num,den); }
  catch(char* error)
  { cout<<error; }
}
```

```
float middle(int a, int b)
{
  try { return divide(a,b); }
  catch(char* error)
  {
    cout<<error<<endl;
    throw "middle error";
  }
}
```

Exception Specifications

- Function declaration may specify types of exceptions

```
int func (int x) throw(int, Error_message) { /* ... */ }
```

  - may **`throw`** **`int`** or **`Error_message`** object

```
int func (int x) { /* ... */ }
```

  - default: may **`throw`** any exception

```
int func (int x) throw() { /* ... */ }
```

  - may **not** **`throw`** any exception at all

- Unexpected **`throw`** calls **`terminate()`**

3.7.4 Stack Unwinding

- What is stack unwinding?
  - the process of transferring control flow due to an exception
  - stack unwinding is initiated at a **`throw`** statement
  - control is handed over to the **`catch`** block matching the **`try`** block
  - what about the called functions on the stack?
    - their cleanup is bypassed

Stack Unwinding (cont.)

- Characteristics of stack unwinding:
  - when a thrown exception is not caught in a particular scope
    - the function terminates
    - the local variables of the function are destroyed
    - control returns to the statement that invoked the function
  - attempt is made to **`catch`** the exception in outer **`catch`** blocks
  - if the exception is never caught, **`terminate()`** function is called

Stack Unwinding Cleanup

- **`throw`** and **`catch`** bypass the normal return structure

- Problems:
  - local variables are lost
  - memory is not deallocated
  - inconsistent state is reached

- Who cleans up?

```
void b()
{ throw "error"; }
```

```
void a()
{
  try { b(); /*...*/ }
  catch(char*)
  {
    /* ... */
    throw;
  }
}
```

```
int main()
{
  try { a(); /*...*/ }
  catch(char* error)
  { cout<<error; }
}
```

Graceful Stack Unwinding

- One idea: put all pointers in the **`catch`** parameter
  - cleanup occurs in the **`catch`** block

  - soooo ugly! violates every rule of OO design

- Better: put a **`catch`** block in every called function
  - each function cleans up after itself
  - good encapsulation

Graceful Stack Unwinding (cont.)

- Best: make everything an object
  - destructors are invoked on scope exit
    - using **`return()`**, **`exit()`**, or **`throw()`**
  - cleanup is automatic
    - as long as the destructors are implemented correctly

- **`coding example <p7>`**

One Idea: Using `catch` Parameter

```
class Error_message
{
public:
  Error_message(char* str, int* p):
     message(str),arrayPtr(p) {}
  char* message;
  int*  arrayPtr;
};
```

```
void g()
{
  /* ... */
  try
  { f(); }
  catch (Error_message& m)
  {
    delete [] m.arrayPtr;
    cout<<m.message;
    exit(1);
  }
}
```

```
void f()
{
  int* a = new int[10];
  /* ... */
  if (/*condition*/)
    throw Error_message("error",a);
  /* ... */
}
```

Better: Unwinding with Re-`throw`

```
void g()
{
  /* ... */
  try { f(); }
  catch (char* error)
  {
    cout<<error;
  }
}
```

```
void f()
{
  int* a = new int[10];
  /* ... */
  try { d(); }
  catch (char*)
  {
    delete [] a;
    throw;
  }
  /* ... */
}
```

```
void d()
{
  /* ... */
  if (/*condition*/)
    throw "error";
  /* ... */
}
```

Best: Unwinding with Destructors

```
class MyArray {
public:
  MyArray(int size) {arr = new int[size];}
  ~MyArray() {delete [] arr;}
private:
  int* arr;
}
```

```
void g()
{
  /* ... */
  try { f(); }
  catch (char* error)
  {
    cout<<error;
  }
}
```

```
void f()
{
  MyArray a(10);
  /* ... */
  d();
  /* ... */
}
```

```
void d()
{
  /* ... */
  if (/*condition*/)
    throw "error";
  /* ... */
}
```