Topic 3 (Tmp: Exception Handling)

C++ advanced features review: *when can/should I use them?*

資料結構與程式設計

Data Structure and Programming

Sep, 2011

What is an exception?

1. Runtime error (system exception)
   Segmentation fault, bus error, etc
   e.g. class bad_alloc
2. User-defined exception
   If that happens, I don't want to handle it...
3. Interrupt
   Control-C, etc

What is exception handling?

- When an exception happens, detect it and stop the program "gracefully" (usually by going back to a command prompt), instead of terminating the program abruptly
- Purposes
  - To keep the partial results
  - Continue the program without losing previous efforts

  ➔ (e.g. Windows crashing vs. "a program terminating abnormally")

C++ Exception Handling Mechanism

```
Try {
   // your main program
   // if (exception happens)
   //    throw exception!!
}
catch (ExpectedException1) {
   // exception handling code 1
}
// More exceptions to catch here
catch (...) {
   // The rest of the exceptions
   // note: "..." above is a reserved
   // symbol here
}
```

Predefined Classes for Exception Handling

- int, char*, etc

```
try {
   buf = new char[512];
   if( buf == 0 )
      throw "Memory alloc failure!";
} catch(const char * str ) {
   cout << "Exception raised: "
        << str << '\n';
}
```

- Predefined classes

```
try {
   ...
} catch (bad_alloc) {  // defined in "std"
   // Handle memory out
   ...
}
```


User-defined Classes for Exception Handling

```
class MyException     {
   int  _type;
public:
   MyException(int i) : _type(i) {}
   void print() const { ... }
};
===============
int main()      {
   try {
      ...
      if (...) throw MyException(type);
   }
   catch (MyException& ex) {
      ex.print();
      // handle the exception here...
   }
}
```


Hierarchical exception handling

```
void g() {
   ......
   if (.....)
      throw Ex4f();
}
void f() {
   try {
      .....
      g();
   } catch (Ex4F& e4f) {
      // do something....
      if (.....)
         throw Ex4Main();
      else
         throw e4f;
   }
}
```

```
main() {
   try {
      f();
   } catch (Ex4F& e){
      ......
   } catch (Ex4Main& e) {
      ......
   } catch (...) {
      ...
   }
}
```


Limited Throw

- Limit only certain types of exceptions to pass through
  - void f() throw(OnlyException&);
    ➔ Only exceptions of class "OnlyException" are allowed
  - void f() throw();
    ➔ None of the exception is allowed
- If disallowed exception is thrown

```
> terminate called after throwing an instance
of 'xxxx'
> Abort
```

Uh??? Why do we limit the types of thow?
➔ catch the problematic code earlier!!

The Challenges in Exception Handling

1. Make sure the program is reset to a “clean state”
   - Memory used by unfinished routines needs to be released back to OS
   - All the data fields (e.g. _flags) needs to be made consistent
2. Be able to continue the execution
   - The unaffected data should not be deleted

Handling Interrupt

- Associate an interrupt signal to a handler
  - void signal(int sigNum, void sigHandler(sigNum));
- system-defined sigNum
  - SIGABRT Abnormal termination
  - SIGFPE Floating-point error
  - SIGILL Illegal instruction
  - SIGINT CTRL+C signal
  - SIGSEGV Illegal storage access
  - SIGTERM Termination request
- sigHandler()
  - Predefined: SIG_IGN(), SIG_DFL()
  - User-defined functions
    - void myHandler(int)

Interrupt Handler Implementation

1. Define interrupt handler function
   - Using “signal(int signum, sighandler_t handler)”;

   ➔ “signum” example: SIGINT ➔ Control-C

   ➔ “handler” is: “void myHandler(int)”

   ➔ man signal
2. Define a flag to denote the detection of the interrupt
3. Initialize the flag to “undetected” (e.g. false)
4. Associate the target interrupt to the handler function
5. (In handler function)
   - Re-associate this target interrupt to “SIG_IGN” (why?)
   - Set the flag to “detected”
6. (In upper-level/main function)
   - Check the flag “detected”
   - If (detected) ➔ reset to “undetected”; re-associate this interrupt with your handler;
   - Do something;

An Ctrl-C example: random number generator (1/2)

```
class MyInt {
   static bool   _ctrlCDetected;
public:
   static void reset() { _ctrlCDetected = false; }
   static void intHandler(int intNum);
   static bool isCtrlCDetected() { return _ctrlCDetected;}
   static void setCtrlCDectected(bool b) {
               _ctrlCDetected=b; }
};

void MyInt::intHandler(int intNum)
{
   signal(intNum, SIG_IGN);
   // Ignore another SIGINT in this function
   if (intNum == SIGINT)
      if (!isCtrlCDetected()) setCtrlCDectected(true);
}
```

An Ctrl-C example: random number generator (2/2)

```
bool MyInt::_ctrlCDetected = false;

int main() {
   MyInt::reset();
   // Tie SIGINT to MyInt::intHandler()
   signal(SIGINT, MyInt::intHandler);
   unsigned count = 0, max = 100;
   while (1) {
      if (MyInt::isCtrlCDetected()) {
         MyInt::setCtrlCDectected(false);
         cout << count << endl;
         return 0;
      }
      if (++count == max) count = 0;
   }
   return 1;
}
```