COMP322 - Introduction to C++

Winter 2011

Lecture 10 - Exceptions
& New Features of C++0x

School of Computer Science

McGill University

March 22, 2011

Motivation for exceptions

- Error handling is a difficult problem in general
- Organizing error codes and messages is tricky in C
- Error handling can lead to resource leaks and ugly code

```
bool f() { // true->success, false->failure
  int *pc = malloc(sizeof(int) * 100);
  if (pc == NULL) {
    return false;
  }
  FILE *fp = fopen(outfile, "w");
  if (fp == NULL) {
    free(pc); // release anything allocated
    return false;
  }
  // ...
  free(pc);
  fclose(fp);
  return true;
}
```

Motivation for exceptions, continued

- Using the "goto" statement is tempting:

```
bool f() {
  int *pc = NULL;
  FILE *fp = NULL;
  pc = malloc(sizeof(int)*100);
  if (pc == NULL) {
    goto error;
  }
  fp = fopen(outfile, "w");
  if (fp == NULL) {
    goto error;
  }
  // ...
  free(pc);
  fclose(fp);
  return true;

error:
  if (pc != NULL) free(fp);
  if (fp != NULL) fclose(fp);
  // ...
  return false;
}
```

What is an exception?

- A mechanism for handling *exceptional conditions*, including but not limited to errors.
- Exceptions are a mechanism for passing error information off to the runtime system, which can then select the appropriate handler for the error.
- Stroustrup: "One way of viewing exceptions is as a way of giving control to a caller when no meaningful action can be taken locally".
- Alternative to printing messages or terminating programs within generic libraries.
- For C programmers, an exception is a safer, more flexible replacement for `setjmp()`/`longjmp()`.

Exception syntax in C++

C++ exception syntax is similar to that of Java:

- `try` - a "try" block associates a list of statements with one or more *exception handlers*.
- `catch` - one or more "catch" blocks follow the try block. These define the handler for a given type.
- `throw` - a "throw" statement passes the exception to the runtime system for delivery.
  - Control is immediately transferred to a handler associated with the nearest enclosing `try` block.
  - If no appropriate handler is found, the program exits.
  - The stack is "unwound" and destructors invoked as necessary.

A basic example

```
void g() {
  // etc.
  if (/* something goes wrong */) {
    throw 2;
  }
}

void f() {
  try {
    // ...
    g();
  }
  catch (int code) { // Handle int exceptions
    cerr << "Caught exception " << code << endl;
  }
  catch (...) { // Default handler
    cerr << "Caught unknown exception" << endl;
  }
}
```

Some more details

The `catch` block must specify the type that is to be caught, it need not specify a parameter name.

If a parameter name is not specified, we can't examine the value of the exception or learn anything other than the type:

```
void f() {
  try {
    // ...
  }
  catch (int) { // Handle int exceptions anonymously
    // deal with the exception
  }
  catch (...) { // Always anonymous, even the type is unknown
  }
}
```

Nested exceptions

Try blocks can be nested within one another. The exception will be delivered to the innermost possible block:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Exceptions in C++ vs. Java

- C++ has no `finally` block
- C++ exceptions can throw *any* type
  - basic types (int, char, float, ...)
  - any object derived from the standard class called **exception**
- C++ methods are never required to specify the exceptions they may throw

Functions throwing exceptions

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
void f()
// can throw any type of exception

void f() throw (int)
// throws an integer exception (catch int)

void f() throw()
// cannot throw any type of exception
```

Standard exceptions

- The C++ Standard library provides a base class called `exception` specifically designed to declare objects to be thrown as exceptions.
- It is defined in the <exception> header file under the `namespace std`.
- This class has
  - default and copy constructors
  - operators and destructors
  - a virtual member function called `what` that returns a null-terminated character sequence (`char *`) that can be overwritten in derived classes to contain a description of the exception.

Standard exceptions

```
#include <iostream>
#include <exception>
using namespace std;

class CustomException: public exception
{
  virtual const char* what() const throw()
  {
    return "Custom exception happened";
  }
} custEx;

int main () {
  try
  {
    throw custEx;
  }
  catch (exception& e) // reference to base is OK
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

Standard Library Exceptions

exception	description
`bad_alloc`	thrown by new on allocation failure
`bad_cast`	thrown by dynamic_cast when fails with a referenced type
`bad_exception`	thrown when an exception type doesn't match any catch
`bad_typeid`	thrown by typeid
`ios_base::failure`	thrown by functions in the iostream library

bad_alloc Example

```
try
{
  int * myarray= new int[1000];
}
catch (bad_alloc&)
{
  cout << "Error allocating memory." << endl;
}
```

New Features of C++0x

C++0x

- C++0x is the next standard for ISO C++
- A subset of several C++0x features is currently supported by the GCC version 4.5 compiler: `g++ -std=c++0x`
- High-level aims for the language are to:
  - Make C++ a better language for systems programming and library building
  - Make C++ easier to teach and learn (through increased uniformity, stronger guarantees)

Static Assertions

Issue: Integer sizes are not always the 4 bytes you assume them to be. Code may crash on a different platform.
Solution: The `static_assert` construct helps track these problems, and are useful for when you need to migrate sources to a different platform.

```
static_assert(sizeof(int) == 4, "Integer sizes expected to be 4");

int main()
{
    return 0;
}
```

E.g. On a 64-bit enterprise Linux system, this assertion fails during compilation. Here's the log:

```
g++ 1.cpp --std=c++0x
1.cpp :1:1: error: static assertion failed: " Integer sizes
expected to be 4"
```

Initializer lists and type narrowing

Issue: Type-narrowing is allowed in C++ initializer lists. Compiling with `g++ -Wall` will not warn you about the `double` to `integer` type conversion.

```
int main( )
{
   int nasty[ ] = {8, 99, 2.3, 4.0, 5};
   // ...
   return 0;
}
```

C++0x will not allow it. Log:

```
1.cpp: In function 'int main()':
1.cpp:14:34: error: narrowing conversion of
    '2.29999999999999982236431605997495353221893310547e+0'
    from 'double' to 'int' inside { }
1.cpp:14:34: error: narrowing conversion of '4.0e+0'
    from 'double' to 'int' inside { }
```

Range based `for` loops

- Languages like C# and Java have shortcuts that allow one to write a simple "foreach" statement that automatically walks the list from start to finish.
- C++0x will add a similar feature. The statement `for` will allow for easy iteration over a list of elements:

```
int my_array[5] = {1, 2, 3, 4, 5};
for (int &x: my_array) {
    x *= 2;
}
```

- The "range-based for" will work for C-style arrays, initializer lists, and any type that has a begin() and end() function defined for it that returns iterators.

decltype

Issue: C++has never had an easy mechanism for querying the type of a variable or an expression.
Solution: Enter the `decltype` operator from C++0x, which returns the type of a variable or expression.

Example:

```
T1 x;
T2 y;
typedef T3 decltype(x+y);
T3 z ;
```

Lambda Functions

Lambda functions are anonymous functions: you don't have to define a typical C/C++ function to get the job done. Example with STL sort:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<string> vs = {"This", "is", "a", "C++0x", "exercise"};
    sort(vs.begin(), vs.end(),
        [ ](const string& s1, const string& s2) {
            return s1.size() < s2.size();})

    for (auto ivs = vs.begin(); ivs != vs.end(); ++ivs)
        cout << *ivs << endl;
    return 0;
}
```

Variadic Templates

Issue: How do you define a templated class or a function with a variable number of arguments, each with a potentially different type?

C++0x allows you to define functions and classes with variable numbers of arguments:

```
template<typename... Types>
void f(Types... args) // variable number of function arguments
{
}

template<typename... Types>
class c // class with
{
   // member code
};

// Usages
f('a', ``hello'', 2, 3.1);
class c<int, double, std::vector<string>> c1;
```

Multi-threading

The C++ standard committee plans to standardize support for multithreaded programming.

The new standard will support multithreading, with a new thread library: `std::thread`

With the new standard, all compilers will have to conform to the same memory model and provide the same facilities for multi-threading (though implementors are still free to provide additional extensions).

This means you'll be able to port multi-threaded code between compilers and platforms with much reduced cost. This will also reduce the number of different APIs and syntaxes you'll have to know when writing for multiple platforms.

Concluding Remarks

- This is an exciting time to be a C++ developer.
- Better platform for template programming, increased type safety, systems software, and library development. [1]

[1] C++0x feature support in GCC 4.5
http://www.ibm.com/developerworks/aix/library/au-gcc/index.html