UMassAmherst

CS197c: Programming in C++

Lecture 7

Marc Cartright

http://ciir.cs.umass.edu/~irmarc/cs197c/index.html

Syllabus

- ~~Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs~~
- ~~Lecture 2 : C libraries~~
- ~~Lecture 3 : Classes in C++, C++ I/O~~
- ~~Lecture 4 : Memory & Pointers~~
- ~~Lecture 5 : More Pointers~~
- ~~Lecture 6 : Templates and the STL~~
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

Today's lecture

Reflection,

C++ Exceptions,

and C++11

Reflection in C++: RTTI

- RTTI: Run-Time Type Information
  - rudimentary reflection utilities
- Why RTTI? Sometimes you need to know exactly what you're looking at

- 3 components supported in the language (recently):
  - `dynamic_cast` operator
  - `typeid` operator
  - `type_info` data type

RTTI: *dynamic_cast*

- Attempts a safe typecast
- Say we have:

```
class Vehicle { }
class Car : public Vehicle { }
class Coupe : public Car { }
```

- and later...

```
Vehicle *v;   // ← assume this was given a value earlier
Coupe *c = dynamic_cast<Coupe *>(v);
// at this point c is either a valid pointer or null
```

RTTI: *typeid* and *type_info*

- Used for type equivalence
- typeid returns a std::type_info object
- Similar to instanceof in Java:

```
if (typeid(Coupe) == typeid(*v)) ...
```

C++ Exceptions

- Syntactically similar to Java:

```
float div(int a, int b) {
    if (b == 0) throw “Cannot divide by zero”;
    return ((float) a / (float) b);
}

try {
    float c = div(7, 0);
} catch (const char * s) {
    cerr << s;
}
```

C++ Exceptions using objects

- Libraries for handling exceptions:
  - <exception> : interface to exceptions
  - <stdexcept> : some standard exception types

- First define the "error object":

```
class bad_div {
public:
    int v1;
    int v2;
    bad_div(int a = 0, int b = 0) : v1(a), v2(b){ }
};
```

C++ Exceptions using objects

- Default behavior:

```
try {
    float c = div(7, 0);
} catch (exception &e) {
    cerr << “Uh-oh:” << e.what() << endl;
}
```

- Better than nothing
- Not overly informative

C++ Exceptions using objects

- Modify our try/catch block:

```
float div(int a, int b) throw(bad_div) {
    if (b == 0) throw bad_div(a,b);
    return ((float) a / (float) b);
}

try {
    float c = div(7, 0);
} catch (bad_div & bd) {
    cerr << “Bad arguments to div: “ << bd.v1 << “, “ << bd.v2 <<
    endl;
}
```

C++ Exceptions

- Multiple catches allowed:

```
try {
   // suspicious code
} catch (thrown obj 1) {
} catch (thrown obj 2) {
} catch(...) {
   // Catch whatever wasn't handled above
}
```

<stdexcept>

- logic_error
- domain_error
- invalid_argument
- length_error
- out_of_range
- runtime_error
- range_error
- overflow_error
- underflow_error

Saves us some work, but not as rich as Java...

Uncaught Exceptions

```
try {
    float c = div(7, 0);
} catch (bad_div & bd) {
    // catching code
}
```

If some other exception type is thrown, we're in trouble unless we handle it:

```
void uncaughtHandler() { cout << "I'm dead...\n"; exit(5);}
set_terminate(uncaughtHandler);
```

Problems w/ set_terminate

- Last-ditch
  - last thing to happen
- Ends sloppily
  - terminate by default calls abort

Unexpected Exceptions

```
float div(int a, int b) throw(bad_div) {
    if (b == 0) throw bad_div(a,b);
    if (a == b) throw dumb_div(a,b);
    return ((float) a / (float) b);
}
```

Not specified to be thrown!

This is an *unexpected exception*, and also causes an abort of the program unless we set a handler:

```
void handleUnexpected() {
    throw std::bad_exception(); // ← replace it with a
                                // generic exception
}
set_unexpected(handleUnexpected);
```

C++ Exception problems

- Behavior is harder to predict with dynamic allocation and templates
  - Not like Java
- Default behavior is to drop everything on the floor
  - Java stops mid-sentence and shows you what happened
- No stack tracing built-in

C++11

- Update to the definition of the C++ language
  - Current working draft is 1334 pages!
  - New standard is probably better than Java
- Lots of new features
  - Way too many to actually cover (40 *sections*)
  - Many compiler installations not compliant
- We'll introduce some of them today
- Most, if not all examples, lifted from Wikipedia (highly suggested resource):

http://en.wikipedia.org/wiki/C++11

C++11 Algorithms

- all_of
- any_of
- none_of

```
#include <algorithm>

//are all of the elements positive?
        all_of(first, first+n, ispositive()); //false

//is there at least one positive element?
        any_of(first, first+n, ispositive()); //true

// are none of the elements positive?
        none_of(first, first+n, ispositive()); //false
```

Taken from:
http://www.softwarequalityconnection.com/2011/06/the-biggest-changes-in-c11-and-why-you-should-care/

C++11 Initializer Lists

```
struct Object {
    float first;
    int second;
};

Object scalar = {0.43f, 10};     //One Object, with
                                 //first=0.43f and second=10

Object anArray[] = {{13.4f, 3},
                    {43.28f, 29},
                    {5.934f, 17}}; //An array of three
                                   //Objects
```

C++11 Ranged for loops

- Old and busted:

```
int len = 5;
int my_array[len] = {1,2,3,4,5};
for (int i = 0; i < len; i++) { cout << my_array[i] << endl; }
```

- New hotness

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

C++11 String Literals

- Make it easier to simply write literal stings
- UTF-8, -16, and -32 character literals:
  - u8"This is a Unicode Character: \u2018.”
  - u"This is a bigger Unicode Char: \u2018."
  - U"This is a Unicode Character: \u2018.”
- Raw Strings:
  - R"(The String Data \ Stuff ")"
  - R"delimiter(The String Data \ Stuff ")delimiter"

C++11 Smart Pointers

- Goal: OMG Garbage Collection thank you!
- Not written into the language
  - Added as available library
- Options:
  - auto_ptr (deprecated)
  - unique_ptr
  - shared_ptr
  - weak_ptr

C++11 Smart Pointers - 1

```
std::unique_ptr<int> p1 = new int(5);
std::unique_ptr<int> p2 = p1; //Compile error.
std::unique_ptr<int> p3 = std::move(p1);
    //Transfers ownership. p3 now owns the memory and
    //p1 is rendered invalid.

p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

C++11 Smart Pointers - 2

```
std::shared_ptr<int> p1 = new int(5);
std::shared_ptr<int> p2 = p1; //Both now own the memory.

p1.reset(); //Memory still exists, due to p2.
p2.reset(); //Deletes the memory, since no one else owns
            //the memory.
```

C++11 Smart Pointers - 3

```
std::shared_ptr<int> p1 = new int(5);
std::weak_ptr<int> wp1 = p1; //p1 owns the memory.
{ std::shared_ptr<int> p2 = wp1.lock();  //Now p1 and p2 own the
                                          //memory.
    if(p2) //Always check to see if the memory still exists
    {
        //Do something with p2
    }
} //p2 is destroyed. Memory is owned by p1.
p1.reset(); //Memory is deleted.
std::shared_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an
                                      //empty shared_ptr.

if(p3) { //Will not execute this. }
```

C++11 unions

```
//for placement new
#include <new>
struct Point {
    Point() { }
    Point(int x, int y): x_(x), y_(y) { }
    int x_, y_;
};

union U { int z; double w;
    Point p;                 // Illegal in C++; point has a non-trivial
                             //constructor. However, this is legal in C++11.
    U() { new( &p ) Point(); }          // Can work, but only if
                                        // manually defined
};
```

C++11 Type Inference

- Instead of you digging around and guessing variable types, let the compiler and runtime do it for you

- This can be REALLY helpful when you're not familiar with a new library, or you just wanna save space

C++11 Type Inference - 2

- Explicit initializers can be inferred:

```
auto some_strange_callable_type =
   boost::bind(&some_function, _2, _1, some_object);
auto other_variable = 5;
```

- Can determine other variable types:

```
int some_int;
decltype(some_int) other_integer_variable = 5;
```

C++11 Type Inference - 3

- Which means, for loops, instead of :

```
vector<int> v = {1,2,3,4,5,6,7};
for (std::vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
```

- we can do:

```
vector<int> v = {1,2,3,4,5,6,7};
for (auto itr = v.cbegin(); it != v.cend(); ++it)  //cbegin returns a
                                                    //const_iterator
```

C++11 Regular Expressions

- Regular Expressions can make life much easier when parsing
- Usually hard to use if the language doesn’t have built-in support
- C++11 does though!

C++11 Regular Expressions - 2

```
const char *reg_esp = R"([ ,.\t\n;:])”;  // List of separator characters.
std::regex rgx(reg_esp); // 'regex' is an instance of the template class
                         // 'basic_regex' with argument of type 'char'.
std::cmatch match;       // 'cmatch' is an instance of the template class
                       // 'match_results' with argument of type 'const char *'.
const char *target = "Unseen University - Ankh-Morpork";

// Identifies all words of 'target' separated by characters of 'reg_esp’.
if( std::regex_search( target, match, rgx ) ) {
    // If words separated by specified characters are present.
    const size_t n = match.size();
    for( size_t a = 0; a < n; a++ ) {
        std::string str( match[a].first, match[a].second );
        std::cout << str << "\n";
    }
}
```

C++11 – Wrap Up (for now)

- I can go on and on and ON
- And for some of it, I can’t even explain yet
- Minor change of plan: Lecture 8 will include more components of C++11, +Boost, +OpenGL (a little)

P4 Introduction

- Red-Black Tree
  - balanced binary search tree
- Insert and Delete are complex
  - Usually: 1 lecture each
  - Implemented for you
- Low-level helpers need finishing
  - rotate methods
  - min, max
  - successor, predecessor

Next Lecture

More C+11,

Boost, OpenGL