C++

January 8, 2013

Mike Spertus

mike_spertus@symantec.com

YIM: spertus

This week’s lecture

- Today is a survey of C++
  - Want to give some of the big picture without worrying too much about the technical details
  - Don’t worry if some things are unclear or not covered in enough depth
  - We’ll present language features systematically in the coming weeks

COURSE INFO

Texts

- Required:
  - C++ Standard
  - Use near-final draft at
    http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
- Recommended:
  - http://isocpp.org/
  - http://www.open-std.org/jtc1/sc22/wg21/
  - Koenig and Moo, Accelerated C++
  - Scott Meyers, Effective C++ (C++ best practices)
  - Stroustrup: Programming: Principles and Practice Using C++
    - Text for Texas A&M's Programming 101 by the inventor of C++
  - Josuttis and Vandervoorde, C++ Templates: The Complete Guide
    - As you will see, much of the course will be about templates in one guise or other.
  - Anthony Williams, C++ Concurrency in Action
    - C++11 multithreading. Our main topic for the last few weeks.
    - If you don't want to buy/read the whole book, Anthony's blog includes what you need in his multithreading in C++ series
      - http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.htm
- Whatever books/sites work for you

The most important rule

- If you are ever stuck or have questions or comments
- Be sure to contact me
  - mike@spertus.com
  - Yahoo! IM: spertus
- Or your TA
  - Paul Bossi
    - Email and google chat: paullbossi@gmail.com

Homework and Lecture Notes

- Homework and lecture notes posted on chalk.uchicago.edu
  - Choose CSPP 51044 and then go to Lab/Lectures in the navigation pane
- Homework submission instructions are at http://cspp51044-wiki.cs.uchicago.edu/index.php/Main_Page#Use_HWSubmit
  - If you are on Windows, you can ftp to a CS machine and run hwsubmit from there
  - Contact me or graders if you need help submitting
  - If necessary, you can email HW to me, but only send text files and images or my mailer may block
- Due on the following Monday before class
- If you submit by Friday, you will receive a grade and comments back before Monday, so you can try submitting again
- In general, graded HW will be returned by the following Monday

Grading

- 2/3 HW
  - Many extra credit opportunities
  - Extra credit can get your HW total for the quarter to 100% (but no higher) to cancel out any problems you miss
- 1/3 Final
  - The biggest part of the final is to do a code review of a willfully bad (but unfortunately not worse than some code you'll see in real-life)

WHAT IS C++

C++ for programmers?

- Prior knowledge of C++ helpful but not required
- Expect you know how to program in some language
  - Not an introductory course on programming
- Gloss over features that are similar to those in Java, C, etc.: `if`, `for`, `?:`,…
  - OK if these are unfamiliar to you
    - Ask questions in class
    - Email or IM me or the Tas
    - Look up in the many recommended texts

Why not to learn C++ on the street

- It certainly didn’t work for me
  - After over a decade of picking up C++ through general exposure and writing commercial C++ libraries, I thought I was an expert C++ programmer
  - Then I joined the C++ standards committee
  - Turns out I was a rank amateur
  - The real inventors of C++ routinely used many powerful C++ techniques and idioms that I had never heard of
    - If you don’t know what RAII is, look it up immediately after this talk
- Their libraries were much more powerful, flexible, performant, and easy to use than any I had produced are even imagined

A brief history of C++

- 1979 “C with classes” invented by Bjarne Stroustrup
- 1983 Renamed C++
- 1998 First standard.
- Boost libraries released
- 2003 A minor standard revision
- 2011 C++11 standard.
  - Called C++0X under development so “X” ended up being “b”! (We were overly ambitious)
  - This course will spend a lot of time on the new C++11 features

C++11

- Many new features
  - Threads
  - Memory model
  - Lambdas
  - Rvalue references
  - Initializer lists
  - “auto” variables
  - Many more
- http://www2.research.att.com/~bs/C++0xFAQ.html
- Many of these are already supported by compilers
- A big part of the course
- See http://www.open-std.org/jtc1/sc22/wg21/ to understand how the designers of C++ think

What C++ isn’t

- C++ isn’t a better C or a worse Java
- Don't be misled by superficial similarities to C and Java.
- Good C or Java code is not necessarily bad C++ code
  - >90% of C++ programmers make this mistake
  - We will heavily emphasize best practices specific to C++
  - One of my goals today is to convince you that this is correct
    - Knowing how to program in Java, C, or other languages does not mean you know how to program in C++

What C++ is

- Bjarne Stroustrup, C++' inventor, has been moderating a discussion to come up with a "sound bite" description for C++
  - "C++ is a programmer's language"
  - "C++ is a flexible and expressive language"
  - "C++ is a multi-paradigm language"
  - "C++ is an industrial-strength toolset for the programming professional. "
  - "C++ is a high-performance general-purpose programming language"

C++ is a lightweight abstraction language

- Large, professional computer programs need to be written using abstraction and patterns to be maintainable and evolvable but at the same time they need to run efficiently
- Unlike almost any other language, C++ gives you the best of both world, allowing you to create powerful abstractions that are lightweight
- In other words, there is no performance penalty associated with using/creating abstractions

C++ is a standard

- The standard is your toolbox
- C++ is a large language. When questions arise (and they will), the standard is the authoritative answer
  - Standardized in 1998
  - Minor revision in 2003
- An “almost complete C++98 standard” is publicly available at `ftp://ftp.research.att.com/pub/c++std/WP/CD2/`
- Accurate in all significant ways but a “draft” to meet ISO requirements limiting free public access to the actual standard
- The current standard is C++11
  - Most current compilers support some C++11 features (e.g., lambdas) but no current compiler is close to supporting the full C++11 specification.
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf is a freely available draft that is virtually identical to the official standard.
  - All modern compilers support C++98 reasonably well

C++ is not a standard

- Just as important as knowing what the C++ standard says, you need to know what it doesn't say
- No compiler yet fully implements C++11
- Even simple code relying on "obvious" non-standardized behavior may be very fragile
- Relying on non-standard C++ behavior is necessary. E.g.,
  - Bits in an integer
  - DLLs
  - "There are no interesting standards-compliant program"
- However, it reduces portability and is fragile
- If you need to rely on non-standardized behavior (and you will), try to rely on "implementation-defined" rather than undefined behavior, so at least it is defined somewhere

C++ is a multi-paradigm language

- While C++ can be used for object-oriented programming, object-orientation is only one programming paradigm supported by C++
  - And not the dominant one
  - The C++ container library has no runtime-polymorphism at all
  - Even the entire standard library has no more than about a dozen “virtual” functions

SOME CODE

Goldilocks code samples

- This lecture will include introductory, intermediate, and advanced code samples
- Don’t worry, you don’t need to understand them all
  - Some may be too easy
  - Some may be too hard
  - Hopefully, some will be just right
- Want to give an idea of where we’re heading, but if you just understand “Hello, World” this lecture, you’ll be fine

Hello, World

```
#include <iostream>

int
main()
{
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

Frame and Frame2

- From Koenig and Moo, *Accelerated C++*
- See files in chalk

Vectors

- You will want to know at least a little about vectors for the HW (ask me if this isn't enough)
- Note: The program below uses some C++11 (auto)

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

int main()
{
  vector<int> v; // a vector of ints
  v.push_back(1);
  v.push_back(2);
  for(auto it = v.begin(); it != v.end(); it++) {
    cout << *it << ", ";
  } // Prints "1, 2,"
  return 0;
}
```

Defining functions

```
int square(int n)
{
  return n*n;
}

int main()
{
  return square(2);
}
```

Defining functions

```
int square(int n)
{
  return n*n;
}

double square(double n) // OK to have two functions
{                       // with same name (overloading)
  return n*n;           // as long as compiler can tell
}                       // which you mean by context
                        // We’ll make this more precise
int main()              // in a few weeks
{
  return square(2) + square(3.1416);
}
```

Generics

- The key to implementing lightweight abstractions is C++' powerful generic mechanism (also known as templates).
- Templates let you give a name to a not yet specified type
- We will spend more time on generics than any other topic
  - They are that important to modern C++
  - They are the most distinguishing feature of C++

Generic functions

```
template<typename T>
T square(T n)
{
  return n*n;
}

int main()
{
  return square(2) + square(3.1416);
}
```

Defining functions

```
template<typename T>
T square(T n)
{
  return n*n;
}

Circle square(Circle c)
{
  cout << "Cannot square the circle" << endl;
}

int main()
{
  return square(2) + square(3.1416);
}
```

Example: Copy

- How do we copy?
- In C

```
memcpy(cp, dp, n);
```

- This works in C++, but what if we are copying to/from an
  - array of objects?
  - list?
  - stream?
  - …

std::copy

- C++ standard library provides a standard copy function
```
copy(sp, sp+16, destp);
```
- From vector to array
```
vector<char> v;
...
copy(v.begin(), v.end(), destp);
```

std::copy—(Cont)

- From array to vector
- If we just copy to the vector, we will write off the end of the vector.

```
copy(cp, cp+8, v.end());// Wrong!
copy(cp, cp+8, back_inserter(v));
```

- The point is that copy takes "iterators," and std::back_inserter creates an iterator that appends to the end of a vector

Iterators

- Roughly speaking, iterators in C++ generalize pointers to array elements in C
- Much more general
  - Can iterate an array
  - Can iterate a container
  - Can iterate a stream
  - Can be input or output
  - Can be sequential or random access
  - Can append
  - etc.

Can we copy to a stream?

- Sure, we just need to turn it into an iterator
- Printing a comma-delimited vector of doubles

```
vector<double> v;
...
copy(v.begin(), v.end(),
 ostream_iterator<double>(cout,", "));
```

std::copy implementation

- Logically

```
template<class InItr, class OutItr>
OutItr
copy(InItr beg, InItr end, OutItr out)
{
  while(beg != end) {
    *out++ = *beg++;
  }
  return out;
}
```

Object copying vs. Binary copying

- All types know how to copy themselves
- For some types, a binary copy is possible
- Other types cannot be binary copied
  - E.g., they contain reference counted pointers or virtual base classes (whatever those are), etc.
- When possible, binary copies are much faster

Example: Optimized copy

- Programmers traditionally use their knowledge of underlying types to either write a memcpy or a hand-coded object-based copy loop. Breaking encapsulation like this makes the code less robust and extensible:
  - What happens when a programmer working on one part of the code adds a smart pointer to a struct definition without being aware that some other part of the program memcpy's the struct
  - What happens when the array gets replaced by a vector?
  - What happens in generic code?

Active code

-
  ```
  char *cp1, *cp2, *cp3;
  T *tp1, *tp2, *tp3;
  vector<T> v;
  ...
  copy(cp1, cp2, cp3); // Should memcpy
  // The following should all either memcpy or
  // object copying as appropriate
  copy(tp1, tp2, tp3);
  copy(tp1, tp2, v.begin());
  copy(v.begin(), v.end(), tp3);
  ```
- The above code should morph appropriately if the definition of T changes. This would simultaneously optimize ease-of-use, maintainability, and performance

Templates to the rescue

- C++, Java, and .NET have generics, but they are very different, so don't be confused.
- In Java and .NET, templates are designed to create typesafe algorithms that are the same for all types
- C++ can do this too:

```
template<class InputIter, class OutputIter>
void copy(InputIter scan, InputIter end,
OutputIter out) {
    while(scan != end) {
          *out++ = *scan++;
    }
}
```

Important Warning

- Read the following slide for culture. Don't worry if it seems like a foreign language (you’re not expected to understand it yet!). I promise it will be second nature by the end of the course.

Overloading

- Let’s create another function named copy that just works in arrays of characters (so its safe to memcpy)

-
  ```
  void
  copy(const char *scan, const char *end, char *out) {
        memcpy(out, scan, end – scan);
  }
  ```
- Now `memcpy` is automatically called for character arrays because the copy on this slide is a more precise fit than the template function on slide 37
  - Client code still runs fine if types change to something else
  - Can’t do this with C# and Java because no overloading
- What about `long`, `const unsigned int`, `bool`, `double`, `char **`…?
  - Could address this with a lot of tedious code and ugly macros.
  - Easy to make a mistake
  - What if you have structs that are safe to `memcpy`?

The Mental Model

- Templates are the compile-time equivalent of object-oriented dispatch
- Rough mental model:

Run-time	Polymorphism	Inheritance	Virtual Overrride
Compile-time	Templates	(Unspecialized) Template	Template specialization

- Template specializations as being as important as virtual functions

Boost

- The Boost organization is the testbed for trying out future extensions to the standard library
- Many Boost libraries are virtually considered part of the standard themselves and many (including type traits) were adopted in C++11
- We will use the Boost type traits library here

Type traits

- Type traits are template types that inherit from the special types `true_type` or `false_type`.
- For example, `is_pointer` tells if a type is a pointer. (Note: the code below is included in TR1, you can just use it):

```
template <typename T> struct is_pointer : public
false_type{};
template <typename T> struct is_pointer<T*> : public

true_type{};
```

Testing for "binary-copyability"

- We are going to use the type trait `has_trivial_assign`
- `has_trivial_assign<T>` inherits from `false_type` if nothing is known about T.
- `has_trivial_assign<T>` has specializations inheriting from `true_type` for all built-in type expressions that can be assigned with a binary copy.
- That's just what we need!

The code

- **Remember! Just listing for culture, completeness, and to start to familiarize you with the concepts. It is expected that there will be much that is not comprehensible in this listing at this stage of the course, although it will become second nature over the next ten weeks.**

```
template<typename I1, typename I2>
inline I2 optimized_copy(I1 first, I1 last, I2 out)
{
  typedef typename std::iterator_traits<I1>::value_type value_type;
   return copy_imp(first, last, out, has_trivial_assign<value_type>());
}
```

The Code—Continued

```
template<typename I1, typename I2, bool b>
I2
copy_imp(I1 first, I1 last, I2 out,
         const boost::integral_constant<bool, b>&)
{
   while(first != last) {
      *out++ = *first++;
   }
   return out;
}

template<typename T>
T* copy_imp(const T* first, const T* last, T* out, const boost::true_type&)
{
   memcpy(out, first, (last-first)*sizeof(T));
   return out+(last-first);
}
```

Is it worth it?

- The user interface is as simple and uniform as possible
- Type implementations are completely encapsulated
  - Changes to types don’t cause non-local problems
- Performance?

Version	Type	Time
Conventional	char	8s
Optimized	char	1s
Conventional	int	8s
Optimized	int	2.5s

The downside

- The code for `copy` is more complex
- However,…
  - All the complexity is in one place
  - The client code is less complex
  - Client programs will be more flexible and maintainable
- My opinion
  - If performance doesn’t matter, then not justified
    - But don’t put memcpy in your code either. All the advantages of preferring copy remain
  - If it does matter, then the performance gains may be worth it
  - If performance is critical, it may be worth providing the `has_trivial_assign` specialization for your classes, but only if benchmarking identifies this as a botttleneck

HOMEWORK

Submitting Homework

- See instructions at

  http://cspp51044-wiki.cs.uchicago.edu/index.php/Main_Page#Use_HWSubmit.

HW 1.1

- All referenced code is available on chalk
- Build, compile, and run the "Frame" programs on your compiler of choice.
- Send something to demonstrate that you’ve done this successfully (e.g., screenshots, any files you’ve written, including C++ files, makefiles, Visual Studio project files, a transcript of your shell session, etc.)

HW 1.2

- Build, compile, and run the vector program from slide 23
  - You can download it from chalk
- This will make sure your compiler supports at least some C++11
  - g++ >= 4.6
  - Clang
  - Visual Studio >= 2010

HW 1.3

- Print out the first 8 rows of Pascal's triangle. This assignment is most easily completed by using a nested container, e.g., a vector<vector<int> >. Note in particular the need for the space in "> >", to avoid confusing the lexing stage of the compiler.
- If you’ve never seen C++ vectors, look at slide 17 for a simple example(or ask me or Paul)

HW 1.3 – Extra Credit

- For additional credit on the previous problem, it should be tastefully formatted. In particular
  - Use "brick-wall" formatting, in which the numbers of each row are presented interleaved with the numbers on the rows above and below.
  - The brick size should bethe maximum size of any integer in the triangle. For aesthetic as well as technical reasons, it is useful if the brick size is odd, so you may increase the size by one if necessary to make this true.
  - Each number should be centered on it's brick.

HW 1.4

- This exercise has two purposes
  - Make sure Boost is properly installed in your environment
  - Show that a well-designed interface is easy to use even if its implementation is very complicated internally.
- Build, compile, and run any program using the optimized_copy function in optimized_copy.h (download from chalk). You may have to download Boost from www.boost.org and figure out how to get it working on your include path
- Note that optimized_copy is used exactly the same way as std::copy, so you should be able to directly adapt any sample code you can find that uses std::copy.
- Send something to demonstrate that you've done this successfully (e.g., any files you've written, including C++ files, makefiles, a Visual Studio project files, a transcript of your shell session, etc.)

HW 1.5 (Extra Credit)

- Write a valid C program that is not a valid C++ program.
- Hint: There are ways to do this that don’t require prior experience with C. Look for some simple “thinking-out-of-the-box” solutions.

HW 1.6 (Extra Credit)

- Why is C++ called C++ and not ++C?
- Note: If you are new to languages with the "++" operator, see
  - http://cplus.about.com/od/glossar1/g/preincdefn.htm
  - http://cplus.about.com/od/glossar1/g/preincdefn.htm