CS106B
J Zelenski

Handout #38
Mar 14, 2008

C++ sans CS106

Minor edits by Keith Schwarz

Congratulations! You've made it through CS106B. You now have a huge array of tools at your disposal, from linked lists and recursion to graphs and algorithmic analysis. Pat yourself on the back – you are ready to take on some pretty serious programming challenges!

Although CS106B is taught in C++, the skills you have developed are language-independent and will transfer to any language you choose to pursue, whether it's C++, Java, Python, or any of the other hundreds of choices. We've chosen to teach CS106B using C++ because C++ has excellent support for key concepts like pointers, data abstraction, recursion, and templates. However, CS106B is not designed to teach the C++ programming language, and if you want to pursue C++ professionally, there are a few language-specific features you will need to learn. This handout is a quick survey of some of those features and should serve as a launching point for further exploration of the C++ language.

Demystifying `genlib.h`

We've trained you to include **`genlib.h`** in every program, and if you've ever forgotten this, you were rewarded with a passel of compile errors. **`genlib.h`** is surprisingly small, but there is one key line stashed in there you need to know:

```
using namespace std;
```

C++ allows you to define *namespaces*, groups of symbols under a parent name that help reduce name clashes. All C++ standard library features are grouped into a namespace called **`std`**. For example, **`cout`** and **`isupper`** and **`string`** actually have the fully-qualified names **`std::cout`**, **`std::isupper`**, and **`std::string`**. Some C++ programmers make the habit of using the long names everywhere. Others find that tedious and instead introduce a **`using`** directive, which brings the entire contents of a namespace into the global scope. This allows you to use the names from that namespace without qualification. The **`genlib.h`** header also includes **`<string>`** to make the declaration of the **`string`** class visible and declares the prototype for our **`Error`** function. That's it!

What about the other CS106 libraries?

What else do I need to know to ditch my dependencies on the 106 headers?

`simpio`: You can use the input stream **`cin`** (the counterpart of **`cout`**) to read different objects from the console. However, you will have to take care to detect malformed input and deal with clearing the stream if it gets into an error state. Our implementation of the **`GetInteger`** and **`GetLine`** functions use the global **`getline`** function to read a string from the console and convert the string representation into an appropriate data type.

random: Standard C++ provides the **rand** and **srand** functions in the header file **<cstdlib>**. **rand** only returns integers in a small range (0 to the constant **RAND_MAX**), so if you want to generate random **double**s or **bool**s, you'll need to scale the value of **rand** appropriately.

strutils: You can convert numeric values to and from string format using insertion/extraction on a **stringstream** object from **<sstream>**. **stringstream**s are a stream variant that read/write to a **string** object instead of a file. Case conversion can be done in a loop using the **toupper** and **tolower** character functions from **<cctype>**.

graphics/**extgraph**/**sound**: C++ has no standard libraries for these platform-specific features. Each platform has its own toolkit, many of which are available in C++. There are also some cross-platform C++ libraries available as an add-on. If you want to build a graphical application, you'll first have to figure out which of these toolkits you're going to use and paw through its documentation to get oriented.

Replacements for the CS106 class libraries (**Vector**, **Queue**, **Map**, etc.) are discussed in the STL section below.

const

So far, you've only seen **const** used to declare constants. However, **const** has several different uses in C++ and is ubiquitous in professional code. In fact, it would not be a stretch to claim that **const** is the most-commonly used keyword in the entire language. The **const** qualifier can be applied to any variable declaration, including parameters, return types, local variables and object data members, and it even can be used to tag member function declarations. The **const** qualifier is particularly useful in conjunction with reference parameters, because it allows you to clearly document when you are using pass-by-reference for efficiency only and not for mutability. For example, consider this function prototype:

```
void Binky(const string &s, string &t)
```

The **Binky** function takes two string references. When invoking **Binky**, no new string objects are created or copied; instead, references are taken out to the two arguments passed by the caller. The first is a **const** reference, which means that the value of **s** cannot change within the call. The second is a non-**const** reference, so **t** is mutable inside **Binky**. Both have the efficient performance of the reference parameter, but using **const** allows the prototype to more clearly document that the first parameter is passed by reference only for efficiency, and it would be assumed that the second non-**const** parameter is a reference because it is modified.

The compiler enforces **const**-ness of parameters by disallowing modification. Within the body of **Binky**, trying to assign to **s** or calling string member functions like **insert** and **replace** (that modify the receiver string) will not be allowed on **s**. However, calls to functions like **length** or **substr** are still acceptable since they don't modify the receiver. You might ask: how does the compiler know which member functions change the receiver and which don't? The member functions of a class can themselves be declared **const**, indicating that they cannot change any state of the receiving object. When operating on a **const** object, you are only allowed to invoke those member functions that are declared **const**. For example, the **size** and **getAt** member functions of our **Vector** class should be declared **const**, whereas **setAt** should not, as shown in this excerpt below:

```
int size() const; // const following declaration makes function const
ElemType getAt(int index) const;
void setAt(int index, ElemType e); // Not const, modifies receiver
```

If a member function is not explicitly declared **const**, it is assumed non-**const** (regardless of what actually happens in the function body) and thus would not be callable on a **const** object.

Consistent use of **const** is a wonderful way to document where and if data is modified. It is particularly helpful when passing pointers/references that would otherwise suggest mutability. However, it's practically impossible to use **const** only a little bit. Once you start using it, it's rather contagious, and you find that you must follow-through everywhere and all functions/classes need to be designed for complete **const**-correctness. This is actually a great thing in production code, and a habit you will want to adopt in the future, but I feel the requirement to get it all correct adds more language syntax hurdles than it is worth for C++ novices, so I avoiding using it in CS106.

Overloading Operators

C++ provides the ability to add new meanings for the built-in operators (such as **+**, **=**, and **<**), known as *operator overloading*. This makes it possible for user-defined types to have the same conveniences as the standard built-in types. Operator overloading is just a form of "syntactic sugar". It doesn't change the functionality available, but allows you to access functionality using a more convenient/familiar/aesthetic syntax. You could write an **Add** function that takes two **Fraction** objects and returns their sum, or you could instead overload the **+** operator. The usage **sum = Add(f1, f2)** would be equivalent to **sum = f1 + f2**. To overload an operator, implement a function named **operatorXXX**, where **XXX** represents whatever operator you wish to overload (**+**, *****, **==**, etc.) For a class member function, the left-hand side is the receiver object and any other operands are passed as arguments, like this:

```
const Fraction operator+(const Fraction &rhs) const;
// left + right   compiled as   left.operator+(right)
```

You can overload almost any operator in C++ (including **++** ,**^**, **()**, *****, and so on). It can be abused to create cryptic and confusing code, so as a rule of thumb, operators should only be overloaded when the new use is obvious and consistent with existing meanings. For most classes, there are few reasons to overload operators (there is no obvious interpretation of **&&** applied to two queues, or ***** on two maps) save a few operators that are quite commonly overloaded; most notably **operator=**, the assignment operator overloaded to add deep-copying behavior, **operator<<** to define a print operation that allows a new type to be inserted into a stream, and **operator<** to store objects in standard container classes.

Note that operator overloading is *not* more efficient that function calls (in fact, it is implemented using function calls).

The standard template library is used in place of the 106 classes

You may weep at the prospect of facing a future without your now-beloved CS106 classes (**Vector**, **Map**, **Queue**, etc.) but fear not! All standard C++ implementations provide the *standard template library* (STL) of class and function templates implementing classic data structures and

algorithms. There is basically a one-to-one correspondence between our CS106 classes and the STL equivalent (with the notable exception of **Grid**). The STL version typically has a lowercase name and provides similar functionality but with slightly different syntax. All STL containers are **const**-correct and have appropriate deep-copying behavior. In general, the STL prioritizes efficiency over convenience and safety, so some operations are designed in a slightly awkward or unsafe way in the name of performance.

Here is some sample code using an STL **vector**:

```
#include <vector>
/* Plus others, using namespace std;, etc. */

int main()
{
   vector<int> nums;

   nums.push_back(4);  // push_back is equivalent to Vector's add
   nums.push_back(10);
   for (int i = 0; i < nums.size(); i++)
      nums[i] *= 2;
   for (vector<int>::iterator itr = nums.begin(); itr != nums.end(); itr++)
      cout << *itr << endl;
   return 0;
}
```

As you can see, STL **vector** is quite is similar to our **Vector**, with some name changes and syntactic differences. Basically, all of the conceptual knowledge you have about using container classes still applies, but you do have to learn the different names/syntax for working the STL versions.

Here's a quick list of the STL equivalents:

The STL **vector** class provides the "better" array. Note it does *not* provide bounds-checking on access (for efficiency reasons).

The STL **stack** and **queue** classes have a pretty standard interface although I find it odd that the **queue**'s enqueue/dequeue operations are called **push** and **pop**. Both **stack** and **queue** have undefined behavior when retrieving from an empty collection (as opposed to reporting an error).

The STL **priority_queue** class is a heap-based implementation that compares elements either with relational operators or via a client-supplied comparison callback.

The STL **map** class is a two-parameter template, allowing both keys and values to be client-chosen types. It typically is a tree implementation, not a hash table. Keys need to be comparable using relational operators or the client must supply a comparison callback.

The STL **set** class simply provides a container with quick tests for membership. Unlike the CS106 **Set**, though, the STL **set** does not provide mathematical set operators – those are handled by the STL algorithms. The type stored in a **set** must be comparable with relational operators or you will have to specify a comparison callback.

There are some additional STL containers: **deque** (double-ended queue), **list**, **multiset**, and **multimap**. Although many third-party libraries supply **grid**/**matrix**, **graph**, and **hashmap** classes, they are not present in the STL itself. In the upcoming standards revision to C++, though, some of these classes may be introduced to the library.

In addition, the STL provides many general-purpose algorithms (**binary_search**, **sort**, **for_each**, **transform**, **next_permutation**, **set_intersection**, and so on) in template form. Almost all of these work in terms of iterators, which allow them to seamlessly operate on all container classes (since all STL containers provide an iterator), and even on raw C++ arrays. The overall STL design is really quite clever, but has a bit of learning curve when trying to get your head around it for the first time.

Many C++ programmers used to be C programmers, and it shows

There is the full legacy of C within C++ and many projects and programmers began life using C and thus show ghostly remnants of times past. For example, in C, there is no **string** class, and strings are represented instead as null-terminated arrays of characters. There is a set of fairly primitive functions that operate on these **char *** C strings, most of which have cryptic names like **strcpy** or **strcspn**. You may encounter code that continues to use C-strings, perhaps because it was too much hassle to convert or because of the purported efficiency. I think old-style programmers like C-strings because it gives them more chance to write terse pointer code in the name of "speed" and "cleverness." In chapter 2 of the reader, we showed how pointers were equivalent to arrays in C++'s internal representation. The main loop within the ANSI C **strcpy** function, which copies a C-string from the character pointer **s** to the memory block at **t** can be written as

```
while (*t++ = *s++);
```

That this code can be written so compactly depends on four C features: (1) the fact that arithmetic applied to pointers is a valid way to advance a pointer through memory, (2) the fact that the **++** operator can be combined with the deferencing operator to return the original value, (3) that assignment returns the value assigned as the result of the assignment, and (4) that the character code used to terminate a string has integer value zero, which is interpreted as a boolean false. Wow!

Although some machines will implement the above construction more efficiently than a loop that makes the tests and increment operators more explicit, this extremely dense coding style is certainly overused in C. Whatever code turns out to be most efficient for copying a C-string (which might be the above code, but is more likely some specialized instruction sequence that takes full advantage of the hardware) has a definite place: it belongs inside the implementation of **strcpy**. Clients should then use **strcpy** and the other library functions to whatever extent is possible, rather than coding explicit loops in their own code. Presumably because it is possible to write the implementation of such a loop using only a very few characters, many C programmers embed nitty-gritty character manipulation at the very highest levels of the program.

C was designed to be a low-level language in which systems programmers could write exactly the code they needed to manipulate the internal structure of the machine. As such, C contains

many constructs that we have not seen in CS106 for working with the internal representation at a more detailed level. These features include:

- *Octal and hexadecimal constants*, written by preceding numbers with `0` and `0x`, respectively. For example, we can write the number twenty-four as `24` (in base 10), `030` (in base 8) or as `0x18` (in base 16).
- *Bit manipulation operators*. C includes a set of operations that work on integers and provide access to the individual bits (binary digits) that make up the word. The bit operators are discussed in Chapter 15 of the reader where it is shown how to use them to implement a bit vector implementation of our `Set` class.
- *Unsigned types*. The integer-like types (`int`, `short`, `long`, and `char`) each come in two flavors: `signed` and `unsigned`. When unsigned types are used, all bits in the word are considered to be part of the magnitude of that value, and there is no mechanism for negative numbers. This representation is sometimes required to specify precisely how to treat numerical quantities, particularly when they correspond to specific machine structures. The use of unsigned types can also affect the shift operators introduced above.
- *Bit fields within structures*. In a structure definition, it is possible to indicate that a field should take up less space than a full word. This was done in the bit-packed representation used by the `Lexicon` as a means of conserving space.

To understand these concepts in more detail, you need to learn more about machine architecture (for example, you might take EE108B) and read up on the definition of these operations in a C or C++ reference manual.

Objects support deep-copying

In CS106B, we provided you a `DISALLOW_COPYING` macro that prevented your custom-defined classes from accidentally shallow-copying one another. In many cases this is exactly what you want, but in practice you will want to define deep-copying behavior for your object.

C++ defines object deep-copying through two special functions, *copy constructors* and *assignment operators*. The first is a constructor whose prototype is `MyClass(const MyClass &)` that C++ uses to construct new objects as copies of older ones, and the second is the overloaded `=` operator (`operator =`) used for object assignment. These two functions are a bit difficult to write, so make sure that you consult a reference before trying to implement them yourself.

C++ books to consider

Disclaimer: I like these particular C++ books, however the list is in no way a complete listing of all worthwhile ones. The number of good CS books is large, and it's growing all the time, so don't be put off a book because I didn't list it here. When looking at CS books, be sure to check out the original copyright date; old is generally a warning sign.

Accelerated C++ by Andrew Koenig & Barbara Moo
Well-written, intelligent introduction to standard C++, targeted at the already proficient programmer. One of the best things about this book is that it is good about prioritizing what you need to know, rather than just trudging through all language features with equal coverage. As a result, it is surprisingly short (~300 pages) which is unheard of for an intro C++ book.

Thinking in C++ by Bruce Eckel
Super-thorough introduction to C++ programming (~1500 pages in 2 volumes, ack!) Opposite approach to above, in that it covers every little detail and then some, but the writing is good and has lots of example code. You may have to wade through a lot to find what you want, but everything you need to know is in there somewhere. This book is also available online for free at bruceeckel.com.

Professional C++ by Nick Solter and Scott Kleper
Another C++ tome (800 pages) but good writing and thoughtful in its discussion of modern software practice in C++. Written by two former students/section leaders (!), so I'm biased :-)

Effective C++, *More Effective C++*, and *Effective STL* by Scott Meyers
These three books are filled with "gems" of C++ wisdom on the key issues for using C++ well from a master C++ programmer. He identifies a lot of subtle things to watch for and provides solid advice for how to navigate the tricky features.

The C++ Standard Library: A Tutorial and Reference by Nicolai Josuttis
Great C++ reference, which I much prefer to the Stroustrup "official" version. This is a good reference book to have for looking up any standard feature of the language or its libraries.

The Design and Evolution of C++ by Bjarne Stroustrup
Stroustrup is the author of the C++ language and one of the best qualified to present this history of the C++ language, from birth through the standardization process and its rise to popularity. It's not a book to learn C++ from, but it does make interesting reading when you want to understand how its features came to be. C++ can be quirky and complex, but this book sheds light on the discussion and rationale that led to the modern version of the language and leaves you with immense respect for all the thoughtful effort involved.

In addition to the books listed above, feel free to check out the CS106L course handouts. Since CS106L is designed for CS106 students like you, the handouts should be pretty accessible. The course website is chock-full of useful C++ information, including handouts, sample code, and reference links, and if you're interested in learning C++ this might be a good place to start looking.