C++ for statisticians, with a focus on interfacing from R and R packages

Chris Paciorek

February 6, 2014

These notes are the basis for a set of three 1.5 hour workshops on using C++ for statistical work. The first workshop focuses on the basics of C++ useful for statistical work. I don't expect to teach you C++ in that time, but to give you an overview so that you can go learn what you need more easily, or for those who know a bit of C or C++ already to help round out your knowledge. The second workshop focuses on calling C++ from R via a variety of methods. The third workshop focuses on creating R packages.

A few things to note in advance:

- My examples here will be silly toy examples for the purpose of keeping things simple and focused.
- I'll try to use *italics* to indicate names of things and `typewriter font` to indicate actual syntax. I'll likely slip up occasionally.
- A comment about the speed of R compared to C/C++. Oftentimes one will hear comparisons where R is orders of magnitude slower than C/C++ or some other software. I think these comparisons sometimes use R in naive ways (for example, avoidable for loops instead of vectorized calculations) and other times are not recognizing that a lot of the heavy numerical lifting in R, such as linear algebra, is done in compiled code and is likely to be comparable to doing it directly in compiled code. That said, there are lots of cases where you will want to use C/C++ to get substantial speedups. The motivation of this set of workshops is to enable you to use C/C++ from R for the slow parts and use R for the parts that don't involve serious computation, taking advantage of R's rapid coding, ease of use, input/output capabilities, and graphics.
- Also, most of this is focused on Linux as this is the environment in which most heavy-duty scientific computation gets done. Much of this should work on Macs since they run a variant

of UNIX under the hood. You'll need *xcode* installed on the Mac.

Resources

- Statistical Computing in C++ and R by Randall Eubank and Ana Kupresanin
- R extensions manual
- Dirk Eddelbuettel's Rcpp tutorial at useR! 2012 and Rcpp paper
- Hadley Wickham's guide to packages as part of his devtools package: https://github.com/hadley/devtools/wiki/Package-basics
- Papers on RcppArmadillo (http://www.sciencedirect.com/science/article/pii/S0167947313000492) and RcppEigen (http://www.jstatsoft.org/v52/i05/) [both also linked from Dirk's website]

1 C and C++ basics

1.1 C vs. C++

C++ builds on standard C in a number of ways. These include:

- additional functionality such as function overloading
- object-oriented programming
- the Standard Template Library that provides a variety of data structures and algorithms to operate on them
- templates, which allow you to more easily write functions that deal with multiple types

In the following, I'll describe the basics of coding in C and C++. I'll mix together standard C with features specific to C++ below, without being explicit about it and will generally refer to it as C++ even if I'm just using pure C functionality

1.2 Structure of a C/C++ program

A program consists of several pieces. Let's look though the example program below and see what the main pieces are.

This program does my favorite numerically intensive calculation, calculating the Cholesky decomposition of $X^\top X$ for random square X. *dsyrk* is Lapack's crossproduct function and *dpotrf* is Lapack's Cholesky decomposition.

```
// this is test.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <math.h>
#include <time.h>
#include <R.h>
#include <Rmath.h>
#define PI 3.14159

using namespace std;

// these declarations are needed as I don't think there is a lapack.h
extern "C" int dpotrf_(char* uplo, int* n, double* a, int* lda, int* info);
extern "C" int dsyrk_(char* uplo, char* trans, int* n, int* k,
  double* alpha, double* a, int* lda, double* beta, double* c, int* ldc);

// this is a one-line comment
/* This is
a multi-line
comment.
*/

// compilation:
// g++ -o test test.cpp -I/usr/share/R/include -llapack -lblas
//    -lRmath -lR -O3 -Wall

int main(){
  int size = 8000;
  int info = 0;
  char uplo = 'U';
  char trans = 'N';
  double alpha = 1.0;
  double beta = 0.0;
  double* x = new double[size*size];
  double* C = new double[size*size];
  for(int i = 0; i < size*size; i++){
    x[i] = rnorm(0.0, 1.0);
    C[i] = 0.0;
  }
  cout << "done with rnorm" << endl;
  dsyrk_(&uplo, &trans, &size, &size, &alpha, x, &size, &beta, C, &size);
   cout << "done crossprod" << endl;
  dpotrf_(&uplo,&size,C,&size,&info);
  cout << "done chol" << endl;
  return 0;
}
```

At the top are 'preprocessor' directives that provide information to the the preprocessor that runs before compilation. In particular, if you call functions from other libraries, you need to include the header files that contain the function prototypes (the definition of the function without the body) so that the compiler can check that you are calling the functions correctly (i.e., in terms of the arguments) in your code. You include standard system header files using <>, e.g.,

```
#include <iostream>
#include <Rmath.h>
```

For C++ system header files, you generally don't need the .h.

For non-standard header files (e.g., for external libraries obtained from other sources and your own user-written files), you put the file name in double quotes and you need the full filename with .h, e.g.,

```
# include "myheader.h"
```

You can define constants with *#define*. This allows one to avoid having "magic" numbers sprinkled around your code and then having to remember what they mean. However you may want to do this via *const* variables (more later).

You can do lots of other stuff with preprocessor directives, but we won't go into them here.

The "using namespace" tells the compiler that you'll make use of objects and functions from the standard template library (STL). This is akin to saying `library(pkgName)` in R, which loads objects and functions from the package into your workspace so they're accessible to you, and to Python's *import* statement.

The strangely named *dpotrf_* is the Lapack Cholesky routine and *dsyrk_* the crossproduct function. Calling Lapack routines involves a bunch of strange stuff, including the use of *extern*, which has to do with scoping issues that I don't fully understand. the _ occurs because a Fortran function is being called behind the scenes and typically an underscore is attached to the name of Fortran functions during their compilation.

If you're creating a full-fledged program that you call from the command line, you need a *main()* function that is where your execution begins and ends. From *main()*, you can of course call other functions. The return value of *main()* can just default to 0 as below or you could have it indicate whether the program executed correctly and finished without errors (0 is standard) or not (1 is standard in this case). If you're just going to create a library file that contains functions you'll call from R, you don't need *main()*. Just create your functions and compile as discussed below when we talk about calling C++ from R.

One line comments begin with "//". Multi-line comments can be enclose in the following syntax, e.g.: `/* THIS is a comment */`

1.3 Compiling and linking

Creating a program from C++ code involves compiling the program into a binary executable. The standard C++ compiler is *g++*. The standard C compiler is *gcc*. You can use *g++* for purely C code, so I'll just use g++ throughout. Also for code that includes MPI calls, there is *mpicxx* for C++ and *mpicc* for C.

When you compile code, a couple things happen. The code gets compiled into binary and code from different binaries gets linked together.

Often your program will use code from other libraries (e.g., BLAS and LAPACK). In this case you need to link the other libraries into your executable. There are two options for linking. You can link in a static version of the library, called an archive. Such files have names of the form: *lib{libName}.a*. The binary is included directly in your executable at linking time. Alternatively, you can link to dynamically to a *shared object* library (also called a *dynamic link library* - such files in Windows are *DLLs*). In this case the binary is not included directly in the executable at linking time, but is only referenced, and when your program is run, the code is obtained from the *lib{libName}.so* file. This reduces the size of your binary and allows one to use updated versions of the .so without changing the program, but it also means that the .so files that are linked to need to be available at run-time. In general this is not an issue, so using dynamic linking is very common.

There are a number of dependencies that need to be accounted for in compiling and linking. In the first step, code is compiled (creating a .o file). At this stage, you need to make sure that the compiler can find the necessary header files (.h) containing the signatures of any external library functions that you call in your code. Often these files are in standard places on your filesystem and the compiler can find them, but sometimes you need to add a flag pointing to the directory(ies)