Getting started with C++

Many computational assignments for this class use `C++`. Most Unix based operating systems (Linux, Ubuntu, OSX.x, ...) come with gnu C++ compilers. You start with a *source file*, which is `PoissonSolver.cpp` in this case. Download this from the class web site into a separate working directory and go to that directory in a command window. The command: `g++ -c -g PoissonSolver.cpp` (the `-c` and `-g` are explained below) will *compile* the source code and produce an *object* file `PoissonSolver.o`. Or, the compiler will give *error messages* that say something is wrong with your code. Only fix the first reported error and try the compile command again. You *link* the object file to make an *executable* using the command: `g++ -g PoissonSolver.o`. This makes a file called `a.out`. You *run* the executable by typing: `./a.out`. You should get output that starts:

```
Hello, from the Poisson solver
```

The Python language is interpreted, not compiled. The `python` program is the interpreter that runs Python source files. The C++ compile and link process creates an executable program, an *application*, that runs on its own without an interpreter. For that reason, it can do a computationally intensive task faster than the Python interpreter. See the `Resources` page for links to introductory material on C++.

The bullet numbers in the comments correspond to the bullets below:

1. A *line* of C++ code ends with a semi-colon. These 5 lines are *declarations* of the *signatures* of procedures that are used in the `main` program below but are not defined until after that. The C++ compiler is a *one pass* compiler, which means it reads the lines of code only once, in the order they have in the source file. Every variable and every procedure has to be defined before it is used.

2. Unlike Python, you have to specify the type of a variable when it is created. The Python interpreter determines the type from context. Normally, any *declaration* (saying the type and variable name) should have a comment saying what the variable is. The `const` (for *constant*) means that the value of this variable does not change after the variable is defined. It is not necessary, but it lets the compiler catch certain bugs at *compile time*. For example, the line `N = 2;` draws the error message: `error: read-only variable is not assignable`. *Reading* a variable means learning its value. *Writing* a variable means changing that value. A `const` variable is *read-only*.

3. This is a lot like defining `u` to be an array of doubles. To say exactly what this is, you have to know about *classes* and *templates* in C++. Basically,

the `vector` *template* defines a `vector` *class* for each variable type. You would define `Bp` (for B', our dual box) as a vector of N integers using: `vector<int> Bp[N];`.

4. This `#define` statement is a *macro*. The compiler substitutes the definition, which is `( (i) + (j)*(N) )`, for `l(i,j,N)` before it compiles. A macro is not a procedure. The parentheses are needed because the macro is *expanded* using direct text substutition. For example, this code has `l(i,j+1,N)`, which expands to `(i) + (j+1)*N`, which is what we want. If the definition had been `i + j*N`, it would have expanded to `i + j+1*N`, which is wrong.

5. The `l(i,j,N)` macro is a way to make a one dimensional C++ array look like a two dimensional array. We would like to refer to u_{ij} with code such as `u[i,j]`, but C++ does not support multi-dimensional array (a disadvantage over Python and Matlab). Defining a macro allows us to write `u[l(i,j,N)]`. The "`l`" stands for "linear", for the position in the "linear array" `u`.

6. You have to write lots of code taking care of boundary conditions even though most of the points are not at the boundary.