C++ Lab 07 - Introduction to C++ Build Systems

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Ten Short CPP Labs

IAP 2024

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1 Lab Seven Overview and Objectives

We are at the point in our labs where our example programs are contained in multiple source code files, all passed as arguments to the `g++` compiler on the command line. If you're thinking that *building* an executable this way is starting to get unwieldy, we agree. And there are many tools and techniques out there to help manage this, some very simple and some more complex and sophisticated. The sophisticated tools generalize and in some cases *use* the simple tools, so it's worth introducing the simple build tools and concepts here and now. Partly because the message should resonate now that you are starting to have non-trivial programs, and partly to enable the exercises in later labs to remain simple and clear. So before we dive further into more advanced C++ language concepts, we push the pause button and discuss C++ build systems.

In this lab the following topics are covered.

- The Relationship between Compiling and Linking in the Build Process
- Basic Source Code Project Management
- The Construction of Archive Libraries for Holding Code Common to Multiple Apps
- An Introduction to Makefiles to Simplify and Quicken the Build Process
- A Discussion of Cross-Platform Build Tools such as CMake

2 A First Discussion on Source Code File Management

If you have been working through the exercises in all the prior labs, chances are you have them all in one big directory, and your situation looks something like this:

```
$ ls
arrays.cpp              filein.cpp             rand_seglist.cpp         string_split.cpp
arrays_nosmall.cpp      fileout.cpp            rand_vertices_class.cpp  string_split_v2.cpp
BiteString.cpp          function_hellocap.cpp  rand_vertices.cpp        str_search.cpp
BiteString.h            function_hellocmd.cpp  rand_vertices_file.cpp   vectors_nosmall.cpp
cmd_args.cpp            function_hello.cpp     rand_vertices_seed.cpp   Vertex.cpp
factorial_cmdargs.cpp   function_hellodef.cpp  rand_vertices_sleep.cpp  Vertex.h
factorial.cpp           function_main.cpp      SegList.cpp              VertexSimple.cpp
factorial_longint.cpp   function_sepdef.cpp    SegList.h                VertexSimple.h
FileBuffer.cpp          function_sepdef.h      string_bite.cpp          vertices.txt
FileBuffer.h            ParseString.cpp        string_deserial.cpp
filebuff_main.cpp       ParseString.h          string_parse.cpp
```

Ideally the above "flat" structure would instead be organized more in line with the order of the labs, and the top level view would look something like:

```
$ ls
lab01/     lab03/     lab05/
lab02/     lab04/     lab06/
```

Perhaps the only thing stopping you from organizing it this way is that source code files like `Vertex.cpp` and `FileBuffer.cpp` are needed in multiple labs and a little red flag popped up in your

head that having multiple versions of the same source code in different places is a bad idea (bravo to you if that was the case). This is indeed a bad idea, and there are ways to handle this.

Our goal in this lab is to allow you to organize things like the geometry source code (`Vertex.h/cpp` and `SegList.h/cpp`) into its own folder and the same with the string parsing utilities, e.g. `ParseString.h/cpp`. These dedicated utility folders will be called *libraries* and any number of apps or programs can access them. The resulting structure will look something like:

```
$ ls
lab01/     lab03/     lab05/    lib_geometry/
lab02/     lab04/     lab06/    lib_strings/
```

3 Digging a Bit Deeper into the Build Process

The term *build process* here refers to the steps starting with source code (a set of `.cpp` and `.h` files) ending with an executable file representing your program. So far we have been building from the command line where each argument is either a source code file and another argument indicating the desired name of the executable. For example: `g++ -o test main.cpp`. There are a few steps going on under the hood in this process, but you could be forgiven if you held a conceptual view of this process looking like:

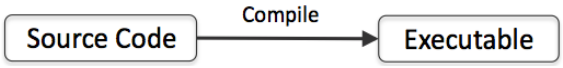

Figure 1: A simplistic view of the build process.

Here's another overly simplistic but more correct view that reveals an intermediate stage prior to the creation of the executable, the creation of object files:

Figure 2: A still simplistic but slightly more realistic view of the build process.

The *linker* stitches together pieces of code that reference other pieces of code. For example, one piece of source code can be compiled that invokes the `biteString()` function found in another piece of code. During the compilation, the compiler doesn't care that they live separately, but during linking the pieces are linked together into a single entitity, the executable. For now you can leave it at that, but you may want to check this out further at the below URLs:

- Read the section "1.4 GCC Compilation Process" found here: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html
- The whole page at this url:

- http://mylinuxbook.com/basics-of-gcc-compilation-process-explained/

The multiple stages are done automatically when building as we have been building, but they can be split up, and that is key for achieving an efficient build. The below example illustrates simple case.

3.1 A Simple Example to Try

As an example, the below is how we built the `string_bite` executable from the previous lab:

```
$ g++ -o string_bite BiteString.cpp string_bite.cpp
$ ls
BiteString.cpp string_bite.cpp string_bite*
```

If we wanted to, we could break this process up into first build the object files and then link them as below. First the compiling into object files. The `-c` command line switch tells g++ to only build the object files:

```
$ g++ -c BiteString.cpp string_bite.cpp
$ ls
BiteString.cpp string_bite.cpp BiteString.o string_bite.o
```

Then the linking of the object files into an executable:

```
$ g++ -o string_bite BiteString.o string_bite.o
$ ls
BiteString.cpp string_bite.cpp   string_bite*
BiteString.o   string_bite.o
```

3.2 How to Make Frequent Builds Efficient During Development

Here we begin to address the build process efficiency. In this case we're not talking about the time it takes to do a fresh build (no prior builds on this computer's recent history). Instead we're talking about efficiency between successive builds during development. The latter is something typically done many dozens or hundreds of times during the course of software development. In this scenario, the primary means for making the build process more efficient is to only re-compile what needs to be re-compiled since the previous build.

The situation is depicted below. An executable is comprised of many source code files. You're presently working on the code in one file. When you go to re-build the executable, this one file is the only file that has changed. The others have not. Yet if we continue to build like we have been doing, ALL source code files are re-compiled anyway:

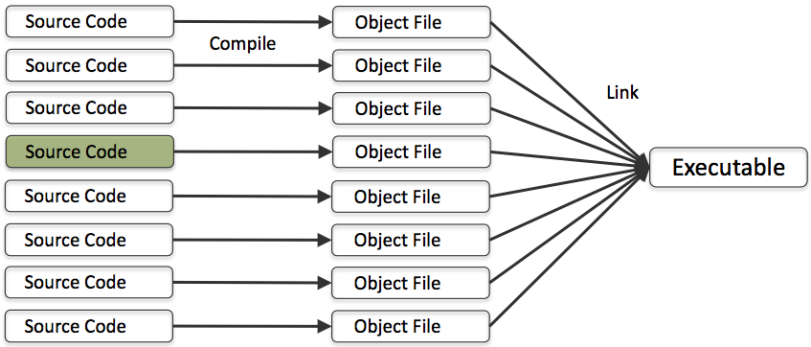

Figure 3: A simplistic view of a program having many source code files. In this case we highlight the situation where ONE of those files has changed since the previous build. In building the naive brute-force way, ALL source code is recompiled, even those source code files that have not changed.

In our exercises where the number and size of source files is small, the inefficiency isn't noticed. Before long, however, it will be noticable and progressively more painful. Instead we strive for a situation like that depicted below, where only the changed file is re-compiled:

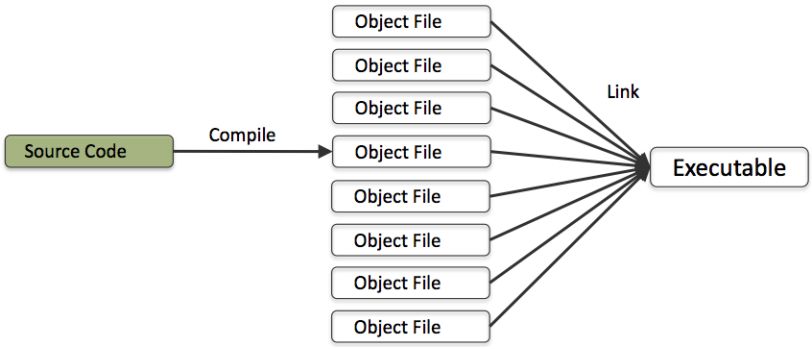

Figure 4: A simplistic view of a program having many source code files. Only one source file has changed since the last build. This is the only file re-compiled on the following build. The build otherwise uses the unchanged object files from the previous build, shortening the overall build time.

3.3 Exercise 1: Prepare Your Code Files for a Reorganization

As a preparation for later steps, we are going to re-organize the file structure of code you have developed so far. If you have been doing all the exercises and using the name suggestions, then the steps should pretty much match up exactly to what we show below.

First, make a copy of what you have done so far. Make a directory called `attic` and *move* everything into there with your current file structure. After that, create the below new directories as shown:

```
$ mkdir lab02  lab03  lab04  lab05  lab06 lib_geometry lib_strings
$ ls
attic/     lab03/     lab05/    lib_geometry/
lab02/     lab04/     lab06/    lib_strings/
```

Each of these new directories, except for the `attic`, will be initially empty, and the next step is to *copy* the source code in the `attic` from each of the labs into the right directory. Here's what it should look like if you followed the naming suggestions from the exercises.

```
$ cd attic

$ cp factorial.cpp factorial_longint.cpp factorial_cmdargs.cpp str_search.cpp ../lab02
$ cp function_* ../lab03
$ cp arrays*.cpp fileout.cpp filein.cpp vectors_nosmall.cpp filebuff_main.cpp ../lab04
$ cp rand_vertices* rand_seglist* ../lab05
$ cp string_*.cpp ../lab06
$ cp FileBuffer* BiteString* ParseString* ../lib_strings
$ cp Vertex* SegList* ../lib_geometry
```

Afterwards, in the top level of your directory, invoking `ls *` should produce something like that shown in the solution for this exercise shown in Section 7.1.

4 Building a Program with Source Code Across Different Directories

After organizing your code as per the previous exercise, the next question is how can this code be built now that all the source code files are not in the same directory. We'll use the code from the previous lab in our examples below. All this code should now be in the `lab06` directory. In this lab, the `string_bite` executable *was* built from the command line with:

```
$ g++ -o string_bite BiteString.cpp string_bite.cpp
```

This will no longer work from within the `lab06` directory since `BiteString.cpp` is not in that directory. It now resides in `lib_strings`. We could alter our build command with:

```
$ g++ -o string_bite ../lib_strings/BiteString.cpp string_bite.cpp
```

This would almost work, but the build will still fail because, inside `string_bite.cpp` there is this line of code (a pre-processor directive):

```
#include "BiteString.h"
```

It may be tempting to fix this by changing the line of code to:

```
#include "../lib_string/BiteString.h"
```

This is almost always a bad idea. It makes the source code in `bite_string.cpp` brittle. If the location of the string library is ever moved or renamed, the code no longer works. A better solution is to simply tell the g++ compiler, on the command line, where it should look for any files being include with the `#include` directive. This is done with the `-I` command line argument as follows:

```
$ g++ -o string_bite -I../lib_strings ../lib_strings/BiteString.cpp string_bite.cpp
```

Now `string_bite` should compile successfully as above. Naming a directory with the `-I` argument is referred to as setting your "include path". If your code needs to `#include` code from multiple places, the argument may be used multiple times on the command line. For example, the following is how `string_deserial` is now built with the new file structure. It needs to `#include` code from two different directories:

```
$ g++ -o string_deserial -I../lib_geometry -I../lib_strings   \
../lib_geometry/Vertex.cpp ../lib_strings/FileBuffer.cpp      \
../lib_strings/BiteString.cpp ../lib_strings/ParseString.cpp  \
string_deserial.cpp
```

The above build command solves the `#include` path problem, but it is still unwieldy. It *seems* like we should be able to get rid of the `../lib_geometry` part of `../lib_geometry/Vertex.cpp` by just specifying another *path* indicating where to look for source code files, in the spirit of how the `-I` argument works. There is indeed this option (the `-L` command line option), but it works with *archived object files*. So our next discussion is how these are made. The vast majority of non-trivial C++ objects in the world do things in this way, so it's worth discussing and start doing things this way.

NOTE: One final comment on the issue of the `#include` path. Notice that in earlier labs we have been including files like:

```
#include <iostream>
#include <cstdio>
```

And we have not needed to have any `-I` entries in our invocations of `g++`, but somehow these included files have been found. In C++ a few conventional locations are automatically part of the include path. These directories are usually "system" directories (accessible to all users). On your machine they are likely found in `/usr/include` or in one of the subdirectories like `/usr/include/c++/4.2.1/` like on my machine.

5 Creating a Library Archive of C++ Utility Source Code

An *archive* in C/C++ is a file that bundles a set of object files into a single file. This file almost always follows the naming convention of starting with `lib` and ending with `.a`, e.g., `libstrings.a` and `libgeometry.a` in our two library examples. An archive file can be build from object files using the `ar` command:

```
$ ar cr libarchive.a file1.o file2.o ... fileN.o
```

The `ar` command has several optional arguments. We use the two options `cr` here. You can find out what these mean by typing `man ar` on the command line to see the manual page for the `ar` command.

5.1 Exercise 2: Generation of Archive Files with a Script

In each of your two library folders `lib_geometry` and `lib_strings`, build a short script to generate an archive file in each folder. The archive names should be `libgeometry.a` and `libstrings.a` respectively. The script will have two lines, one for generating the object files, and one for generating the archive file.

We haven't discussed scripts in our labs so far, and there are several ways to write scripts and invoke them from the command line (bash scripts, perl scripts, python scripts to name a few). The simplest kind of script shouldn't be overlooked - a raw text file. Suppose you have a text file, named `test` for example, with the following few lines:

```
ls
mkdir one
ls
```

You can then "execute" this script using the `source` command as follows:

```
$ source test
```

It's as if you just typed these three commands on the command line. So, to make things easier for you, and to have a "deliverable" for this exercise, you should create a (raw text) script file in both the `lib_geometry` and `lib_strings` directory, and name the files simply `"build"` in both directories. Each file should have the two lines:

```
g++ ... (you fill this in)
ar  ...
```

You can even add a few lines beginning with `echo`, another command line utility that just echoes its arguments. For example, try adding `"echo Done!"` to be the last line of your script.

You should be able to invoke your scripts from the command line with the following results, first for the geometry library:

```
$ cd lib_geometry
$ source build
$ ls
SegList.cpp       Vertex.cpp        VertexSimple.cpp    build
SegList.h         Vertex.h          VertexSimple.h      libgeometry.a
SegList.o         Vertex.o          VertexSimple.o
```

And then for the strings library:

```
$ cd lib_strings
$ source build
$ ls
BiteString.cpp   FileBuffer.cpp   ParseString.cpp  build
BiteString.h     FileBuffer.h     ParseString.h    libstrings.a
BiteString.o     FileBuffer.o     ParseString.o
```

The solution to this exercise is in Section 7.2.

5.2 Linking to Archive Files in Building an Executable

Once the archive files have been created, linking against them is fairly easy. The `-L` argument to `g++` names a directory to look for archive files. The `-l` argument names an actual archive file to link to. The first example below shows what the compile command would look like without the use of an archive:

```
 $ g++ -o string_deserial -I../lib_geometry -I../lib_strings   \
 ../lib_geometry/Vertex.cpp ../lib_strings/FileBuffer.cpp      \
 ../lib_strings/BiteString.cpp ../lib_strings/ParseString.cpp  \
 string_deserial.cpp
```

The below command shows the equivalent build, using the archive and the `-L` and `-l` commands.

```
 $ g++   string_deserial.cpp -o string_deserial -I../lib_geometry \
-I../lib_strings -L../lib_geometry -L../lib_strings -lstrings -lgeometry
```

Notice that, for the `-l` option, there is no space between the switch `-l` and the name of the archive, as in `-lstrings`. Also note that an argument such as `-lfoobar` is seeking a library archive file named `libfoobar.a`. The `lib` and `.a` are thus filename conventions used for all static archive files.

Note also that we put the `string_deserial.cpp` argument first. In general, the order of arguments does not matter. In MacOS/Clang order does not matter. In some Linux distributions and versions of `g++`, order may matter and the above is most likely to work.

5.3 Exercise 3: Building a Set of Executables with a Script

In the previous exercise we created simple script files for building the archive files for our two libraries `lib_strings` and `lib_geometry`. We named both files `build`. In this exercise we will generate a build file for all executables in lab 06, for all five executables in that lab:

- `string_split`
- `string_split_v2`
- `string_bite`
- `string_parse`
- `string_deserial`

Your `build` file should have one line for each executable and use the `-I` include path, `-L` link path, and archive files already generated in the library directories.

You should be able to invoke your script from the command line with the following results:

```
$ cd lab06
$ source build
$ ls
build               string_deserial*     string_parse.cpp     string_split_v2*
string_bite*        string_deserial.cpp  string_split*        string_split_v2.cpp
string_bite.cpp     string_parse*        string_split.cpp
```

The solution to this exercise is in Section 7.3.

6 Using Makefiles and GNU Make for Building Projects

So far in this lab we have successfully:

- Moved away from a flat source code structure with all files in the same folder.
- Learned how to create utility libraries (archives) so source code common to multiple applications doesn't have to be duplicated in each application.
- Learned how to build an application with compiler directives to look for header files and archives in folders other than the current folder.

Build speed and efficiency aside, things are starting to feel more organized. We also have introduced some build efficiency too since presumably those library archives don't need to be re-built if they're not being changed and our iterative work is in the application code. The problem is, sometimes that library code is indeed being changed. When an application links to a library it is a *dependency*. All the non-library source code for an app are also dependencies. If *any* dependency has been modified between builds, that dependency needs to be re-compiled. Keeping track of what files have changed and need re-building can be daunting, and many an apparent bug has been found to be the result of failing to re-compile a dependency. So far in previous labs, we have side-stepped this problem by re-building everything, all the time (all source code and all dependencies) on each build, whether a file needs re-building or not. Simple, but not scalable unless you like working slow.

This is where the GNU 'make' utility comes in. The GNU 'make' utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. This utility is very well documented on the GNU website and we ask that you read the first two sections (Sections 1 and 2, 10 short pages) of the manual now before proceeding.

- https://www.gnu.org/software/make/manual/

If you'd like to have this material available off line, you can get the whole manual either in raw text or PDF format using `wget`:

```
$ wget https://www.gnu.org/software/make/manual/make.txt
$ wget https://www.gnu.org/software/make/manual/make.pdf
```

From the GNU Make manual we see a Makefile is comprised of a series of *rules*:

```
target ... : prerequisites ...
        recipe
        ...
```

As a simple example, we can replace each line in the `build` file created in Exercise 3 with a line like:

```
string_bite :
      g++ -I../lib_strings -L../lib_strings -lstrings -o string_bite string_bite.cpp
```

This works, but has the drawback that if any of the dependencies of the `string_bite` executable are modified, a subsequent `make` invocation wouldn't know do anything. So we add the dependencies to the prerequisites part of the rule:

```
string_bite : ../lib_strings/libstrings.a string_bite.cpp
      g++ -I../lib_strings -L../lib_strings -o string_bite string_bite.cpp -lstrings
```

Now if `libstrings.a` or `string_bite.cpp` have been modified more recently than the timestamp on the `string_bite` executable, `make` will re-execute this rule.

6.1 Exercise 4: Replacing our Build Script with a Simple Makefile

Replace the `build` script created in Exercise 3 with a `Makefile` containing a rule for each of the five executables. It should contain a first rule, `all`, that ensures that all five executables are built when the user simply types `make` in the lab06 directory. It should also contain a rule, `clean`, that removes all executables. For now, the rules corresponding to executables can be super-simple, having no prerequisites along the lines of the `string_bite` example above. While simple, it has serious drawbacks that we will improve on in the next exercise.

The Makefile should of course be in a file called `Makefile`. If this filename is used, `make` will automatically use it and the filename doesn't have to be passed in as an argument to `make`. FYI, if the lowercase `makefile` name is used instead, this also works, but the uppercase version takes precedent if both exist. Your `Makefile` with the seven rules discussed above should support the below command line invocations:

```
$ make                      // Builds everything (the "all" rule)
$ make string_split         // Builds just the string_split executable
$ make string_split_v2      // Builds just the string_split_v2 executable
$ make string_bite          // Builds just the string_bite executable
$ make string_parse         // Builds just the string_parse executable
$ make string_deserial      // Builds just the string_deserial executable
$ make clean
```

The first time `make` is run, you will likely see every line of every recipe in every rule echoed on the screen like:

```
$ make
g++ -o string_split string_split.cpp
g++ -o string_split_v2 string_split_v2.cpp
g++ -I../lib_strings -L../lib_strings -lstrings -o string_bite string_bite.cpp
g++ -I../lib_strings -L../lib_strings -lstrings -o string_parse string_parse.cpp
g++ -I../lib_geometry -I../lib_strings -L../lib_strings -L../lib_geometry \
     -lgeometry -lstrings -o string_deserial string_deserial.cpp
```

If make is invoked immediately again, all executables have already been made and you may see a message like:

```
$ make
make: Nothing to be done for 'all'.
```

The solution to this exercise is in Section 7.4.

6.2 Connecting the Makefiles - Building a Top-Level Makefile

The Makefile produced in Exercise 4 only concerns the source code for Lab 06. However, note that one of the rules contains a prerequisite that references an archive file in the lib_strings directory:

```
string_bite : ../lib_strings/libstrings.a string_bite.cpp
      g++ -I../lib_strings -L../lib_strings -o string_bite string_bite.cpp -lstrings
```

This implies that the libstrings.a prerequisite will have had a chance to be built or updated prior to this Makefile execution. Presumably if you had a Makefile in each subdirectory, you could proceed in this manner:

```
$ cd lib_strings
$ make
$ cd ..
$ cd lib_geometry
$ make
$ cd ..
...
$ cd lab06
$ make
$ cd ..
```

This would work, but as you can see it would be tedious to manually do this each time source code as changed. As we mentioned before, there are lots of choices for scripting such tasks, and you can add make to that list. The notion of a "top-level" Makefile is common and described here:

- https://www.gnu.org/software/make/manual/html_node/Recursion.html

6.3 Exercise 5: A Full Set of Makefiles with a Top-Level Makefile

In this exercise we will round out our build system for the full set of directories constituting our labs so far, up through Lab 06. We will need to create:

- A top-level Makefile in the lab root directory
- A Makefile for the two libraries, lib_geometry and lib_strings
- A Makefile for the five lab directories, lab02, ... lab05.

Verify that everything works by trying the few following steps:

First, at the top level after typing the following you should see something like:

```
$ make
make -C lib_geometry
g++ -c Vertex.cpp VertexSimple.cpp SegList.cpp
ar cr libgeometry.a Vertex.o VertexSimple.o SegList.o
make -C lib_strings
g++ -c BiteString.cpp FileBuffer.cpp ParseString.cpp
ar cr libstrings.a BiteString.o FileBuffer.o ParseString.o
make -C lab02
g++ -o factorial factorial.cpp
g++ -o factorial_cmdargs factorial_cmdargs.cpp
g++ -o factorial_longint factorial_longint.cpp
g++ -o str_search str_search.cpp
make -C lab03
g++ -o function_hello function_hello.cpp
g++ -o function_hellocap function_hellocap.cpp
g++ -o function_hellocmd function_hellocmd.cpp
g++ -o function_sepdef function_sepdef.cpp function_main.cpp
...
```

Next, after the initial full `make`, a subsequent call to `make` should result in the following:

```
$ make
make -C lib_geometry
make[1]: 'libgeometry.a' is up to date.
make -C lib_strings
make[1]: 'libstrings.a' is up to date.
make -C lab02
make[1]: Nothing to be done for 'all'.
make -C lab03
make[1]: Nothing to be done for 'all'.
make -C lab04
make[1]: Nothing to be done for 'all'.
make -C lab05
make[1]: Nothing to be done for 'all'.
make -C lab06
make[1]: Nothing to be done for 'all'.
```

Lastly, try "touching" a file in the `lib_strings` directory, and seeing how `make` reacts. (The `touch` command simply updates the timestamp for a named file. In effect it tricks `make` into regarding the file as having just been edited). After the `touch`, you should see something like:

```
$ touch lib_strings/ParseString.h
$ make
make -C lib_geometry
make[1]: 'libgeometry.a' is up to date.
make -C lib_strings
g++ -c BiteString.cpp FileBuffer.cpp ParseString.cpp
ar cr libstrings.a BiteString.o FileBuffer.o ParseString.o
make -C lab02
make[1]: Nothing to be done for 'all'.
make -C lab03
make[1]: Nothing to be done for 'all'.
make -C lab04
g++ -I../lib_strings -L../lib_strings -lstrings -o filebuff filebuff_main.cpp
make -C lab05
make[1]: Nothing to be done for 'all'.
make -C lab06
g++ -I../lib_strings -L../lib_strings -lstrings -o string_bite string_bite.cpp
g++ -I../lib_strings -L../lib_strings -lstrings -o string_parse string_parse.cpp
g++ -I../lib_geometry -I../lib_strings -L../lib_strings -L../lib_geometry \
        -lgeometry -lstrings -o string_deserial string_deserial.cpp
```

Note that only the targets that had `ParseString.h` as a dependency were re-built. Very efficient! Pat yourself on the back!

You can now safely just type `make` in the top-level directory as you continue coding and adding to this project. In fact, if you haven't yet learned about shell *aliases*, this is a good time to do so, and make yourself a build alias along the lines of:

```
alias mm  'cd ~/my_cpp_files/; make; cd -'
```

I chose `"mm"` here only as an example. But in this case you could type `"mm"` in any terminal window on your screen and your whole project would be (re-) built, with only the required steps given current edits. Very nice indeed. For more on aliases, see http://oceanai.mit.edu/ivpman/help/cmdline_aliases.

7 Solutions to Exercises

7.1 Solution to Exercise 1

After the file structure re-org, in the top level of your directory, invoking `ls *` should produce something like:

```
$ ls *
attic:
BiteString.cpp          VertexSimple.cpp         function_hello.cpp       rand_vertices_file.cpp
BiteString.h            VertexSimple.h           function_hellocap.cpp    rand_vertices_seed.cpp
FileBuffer.cpp          arrays.cpp               function_hellocmd.cpp    rand_vertices_sleep.cpp
FileBuffer.h            arrays_nosmall.cpp       function_hellodef.cpp    str_search.cpp
ParseString.cpp         factorial.cpp            function_main.cpp        string_bite.cpp
ParseString.h           factorial_cmdargs.cpp    function_sepdef.cpp      string_deserial.cpp
SegList.cpp             factorial_longint.cpp    function_sepdef.h        string_parse.cpp
SegList.h               filebuff_main.cpp        rand_seglist.cpp         string_split.cpp
Vertex.cpp              filein.cpp               rand_vertices.cpp        string_split_v2.cpp
Vertex.h                fileout.cpp              rand_vertices_class.cpp  vectors_nosmall.cpp

lab02:
factorial.cpp           factorial_cmdargs.cpp    factorial_longint.cpp    str_search.cpp

lab03:
function_hello.cpp      function_hellocmd.cpp    function_main.cpp        function_sepdef.h
function_hellocap.cpp   function_hellodef.cpp    function_sepdef.cpp

lab04:
arrays.cpp              filebuff_main.cpp        fileout.cpp
arrays_nosmall.cpp      filein.cpp               vectors_nosmall.cpp

lab05:
rand_seglist.cpp        rand_vertices_class.cpp  rand_vertices_seed.cpp
rand_vertices.cpp       rand_vertices_file.cpp   rand_vertices_sleep.cpp

lab06:
string_bite.cpp         string_parse.cpp         string_split_v2.cpp
string_deserial.cpp     string_split.cpp

lib_geometry:
SegList.cpp             Vertex.cpp               VertexSimple.cpp
SegList.h               Vertex.h                 VertexSimple.h

lib_strings:
BiteString.cpp          FileBuffer.cpp           ParseString.cpp
BiteString.h            FileBuffer.h             ParseString.h
```

7.2 Solution to Exercise 2

The lib_strings build file:

```
g++ -c FileBuffer.cpp BiteString.cpp ParseString.cpp
ar cr libstrings.a FileBuffer.o BiteString.o ParseString.o
echo Done!
```

The lib_geometry build file:

```
g++ -c Vertex.cpp VertexSimple.cpp SegList.cpp
ar cr libgeometry.a Vertex.o VertexSimple.o SegList.o
echo Done!
```

7.3 Solution to Exercise 3

The lab06 build file:

```
g++ -o string_split string_split.cpp
echo DONE compiling Exercise 01

g++ -o string_split_v2 string_split_v2.cpp
echo DONE compiling Exercise 01V2

g++ -I../lib_strings -L../lib_strings -o string_bite string_bite.cpp -lstrings
echo DONE compiling Exercise 02

g++ -I../lib_strings -L../lib_strings -o string_parse string_parse.cpp -lstrings
echo DONE compiling Exercise 03

g++ -I../lib_geometry -I../lib_strings -L../lib_strings -L../lib_geometry \
-lgeometry -lstrings -o string_deserial string_deserial.cpp
echo DONE compiling Exercise 04
```

The `echo` lines are optional. The order of the command line args is also optional in most cases.

7.4 Solution to Exercise 4

A simple lab06 Makefile: Compare the below `MakefileSimple` with the buildfile script from previous exercise. This accomplishes essentially the same thing. Notice that each `MakefileSimple` rule has no prerequisites (dependencies). After `make` has been invoked once on this file (`make -f MakefileSimple`), subsequent invocations will do nothing unless the executable has been deleted. It will not detect that a re-compile is necessary after a source code edit. Just like our simple build script, it is oblivious to changes in source code. This can be improved by adding the appropriate dependencies in the second `Makefile` example below.

```
# MakefileSimple for Lab06 (Ten Short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lab06/MakefileSimple
# Run: make -f MakefileSimple

all: string_split string_split_v2 string_bite string_parse string_deserial

string_split :
        g++ -o string_split string_split.cpp

string_split_v2
        g++ -o string_split_v2 string_split_v2.cpp

string_bite :
        g++ -I../lib_strings -L../lib_strings -lstrings -o string_bite string_bite.cpp

string_parse:
        g++ -I../lib_strings -L../lib_strings -lstrings -o string_parse string_parse.cpp

string_deserial:
        g++ -I../lib_geometry -I../lib_strings -L../lib_strings -L../lib_geometry \
        -lgeometry -lstrings -o string_deserial string_deserial.cpp

clean:
        rm -f string_split string_split_v2 string_bite string_parse string_deserial *~
```

A Better lab06 Makefile (with Dependencies):

```
# Makefile for Lab06 (Ten Short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lab06/Makefile

all: string_split string_split_v2 string_bite string_parse string_deserial

string_split : string_split.cpp
        g++ -o string_split string_split.cpp

string_split_v2 : string_split_v2.cpp
        g++ -o string_split_v2 string_split_v2.cpp

string_bite : ../lib_strings/libstrings.a string_bite.cpp
        g++ -I../lib_strings -L../lib_strings -lstrings -o string_bite string_bite.cpp

string_parse: ../lib_strings/libstrings.a string_parse.cpp
        g++ -I../lib_strings -L../lib_strings -lstrings -o string_parse string_parse.cpp

string_deserial: ../lib_strings/libstrings.a ../lib_geometry/libgeometry.a string_deserial.cpp
        g++ -I../lib_geometry -I../lib_strings -L../lib_strings -L../lib_geometry \
        -lgeometry -lstrings -o string_deserial string_deserial.cpp

clean:
        rm -f string_split string_split_v2 string_bite string_parse string_deserial *~
```

7.5 Solution to Exercise 5

```
# Top-Level Make Directory for programs through Lab 06
# wget http://oceanai.mit.edu/cpplabs/Makefile

all:
	make -C lib_geometry
	make -C lib_strings
	make -C lab02
	make -C lab03
	make -C lab04
	make -C lab05
	make -C lab06

clean:
	make -C lib_geometry clean
	make -C lib_strings clean
	make -C lab02 clean
	make -C lab03 clean
	make -C lab04 clean
	make -C lab05 clean
	make -C lab06 clean
```

```
# Makefile for lib_geometry (Ten Short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lib_geometry/Makefile

libgeometry.a : Vertex.cpp Vertex.h VertexSimple.cpp VertexSimple.h SegList.cpp SegList.h
	g++ -c Vertex.cpp VertexSimple.cpp SegList.cpp
	ar cr libgeometry.a Vertex.o VertexSimple.o SegList.o

clean:
	rm -f Vertex.o VertexSimple.o SegList.o libgeometry.a *~
```

```
# Makefile for lib_strings (Ten Short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lib_strings/Makefile

libstrings.a : BiteString.cpp BiteString.h FileBuffer.cpp FileBuffer.h ParseString.cpp ParseString.h
	g++ -c BiteString.cpp FileBuffer.cpp ParseString.cpp
	ar cr libstrings.a BiteString.o FileBuffer.o ParseString.o

clean:
	rm -f BiteString.o FileBuffer.o ParseString.o libstrings.a *~
```

```
# Makefile for Lab 02 (Ten short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lab02/Makefile

all: factorial factorial_cmdargs factorial_longint str_search

factorial: factorial.cpp
	g++ -o factorial factorial.cpp

factorial_cmdargs: factorial_cmdargs.cpp
	g++ -o factorial_cmdargs factorial_cmdargs.cpp

factorial_longint: factorial_longint.cpp
	g++ -o factorial_longint factorial_longint.cpp

str_search: str_search.cpp
	g++ -o str_search str_search.cpp

clean:
	      rm -f factorial factorial_cmdargs factorial_longint str_search *~
```

```
# Makefile for Lab03 (Ten Short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lab03/MakeFile

all: function_hello function_hellocap function_hellocmd function_sepdef

function_hello: function_hello.cpp
	g++ -o function_hello function_hello.cpp

function_hellocap: function_hellocap.cpp
	g++ -o function_hellocap function_hellocap.cpp

function_hellocmd: function_hellocmd.cpp
	g++ -o function_hellocmd function_hellocmd.cpp

function_sepdef: function_sepdef.h function_sepdef.cpp function_main.cpp
	g++ -o function_sepdef function_sepdef.cpp function_main.cpp

clean:
	      rm -f function_hello function_hellocap function_hellocmd
	rm -f function_sepdef *~
```

```
# Makefile for Lab 04 (Ten short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lab04/Makefile

all: arrays arrays_nosmall filein fileout vectors_nosmall filebuff

arrays: arrays.cpp
        g++ -o arrays arrays.cpp

arrays_nosmall: arrays_nosmall.cpp
        g++ -o arrays_nosmall arrays_nosmall.cpp

fileout: fileout.cpp
        g++ -o fileout fileout.cpp

filein: filein.cpp
        g++ -o filein filein.cpp

vectors_nosmall: vectors_nosmall.cpp
              g++ -o vectors_nosmall vectors_nosmall.cpp

filebuff: filebuff_main.cpp ../lib_strings/libstrings.a
        g++ -I../lib_strings -L../lib_strings -lstrings -o filebuff filebuff_main.cpp

clean:
              rm -f arrays arrays_nosmall fileout filein vectors_nosmall filebuff *~
```

```
# Makefile for Lab 05 (Ten short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lab05/Makefile

all: rand_vertices rand_vertices_file rand_vertices_class rand_seglist \
        rand_vertices_sleep rand_vertices_seed

rand_vertices: rand_vertices.cpp
        g++ -o rand_vertices  rand_vertices.cpp

rand_vertices_file: rand_vertices_file.cpp
        g++ -o rand_vertices_file  rand_vertices_file.cpp

rand_vertices_class: ../lib_geometry/libgeometry.a rand_vertices_class.cpp
        g++ -I../lib_geometry -L../lib_geometry -lgeometry -o rand_vertices_class  \
        rand_vertices_class.cpp

rand_seglist: ../lib_geometry/libgeometry.a rand_seglist.cpp
        g++ -I../lib_geometry -L../lib_geometry -lgeometry -o rand_seglist rand_seglist.cpp

rand_vertices_sleep: rand_vertices_sleep.cpp
        g++ -o rand_vertices_sleep  rand_vertices_sleep.cpp

rand_vertices_seed: rand_vertices_seed.cpp
        g++ -o rand_vertices_seed  rand_vertices_seed.cpp

clean:
              rm -f rand_vertices rand_vertices_file rand_vertices_class rand_seglist
        rm -f rand_vertices_sleep rand_vertices_seed *~
```

```
# Makefile for Lab 06 (Ten Short C++ Labs)
# wget http://oceanai.mit.edu/cpplabs/lab06/Makefile

all: string_split string_split_v2 string_bite string_parse string_deserial

string_split : string_split.cpp
	g++ -o string_split string_split.cpp

string_split_v2 : string_split_v2.cpp
	g++ -o string_split_v2 string_split_v2.cpp

string_bite : ../lib_strings/libstrings.a string_bite.cpp
	g++ -I../lib_strings -L../lib_strings -lstrings -o string_bite string_bite.cpp

string_parse: ../lib_strings/libstrings.a string_parse.cpp
	g++ -I../lib_strings -L../lib_strings -lstrings -o string_parse string_parse.cpp

string_deserial: ../lib_strings/libstrings.a ../lib_geometry/libgeometry.a string_deserial.cpp
	g++ -I../lib_geometry -I../lib_strings -L../lib_strings -L../lib_geometry \
	-lgeometry -lstrings -o string_deserial string_deserial.cpp

clean:
	rm -f string_split string_split_v2 string_bite string_parse string_deserial *~
```