CMSC 15200- Intro to Computer Science 2
Summer 2009
Lab #3 (8/12/2009)

Using make and Makefiles to build C++ programs

Here are some resources on the tool make that we will cover today:

* The GNU make manual:
  http://www.gnu.org/software/make/manual/html_node/index.html

* A make Tutorial (assumes you are writing C and not C++):
  http://users.actcom.co.il/~choo/lupg/tutorials/writing-makefiles/writing-makefiles.html

Review: Separate Compilation

In class we talked about how to organize source code into different files that can be compiled separately. In this lab, you will learn how to use the tool make that automates the compilation process for programs using multiple source files.

The zip file for this lab includes these source files for the list ADT we saw in class, and for a simply modified version of it that keeps track of the end of the list as well as the head. To decompress the files, run the command:

```
$ unzip list.zip
```

This should create two subdirectories, list and list_end, each of which has the following files.

* list.h - Defines the list structures and the functions to use the list.
* list.cpp - Implements functions in list.h
* main.cpp - Contains main(); calls list functions.

Plus list should contain a file called Makefile.

We can build a single program from these files with these three commands:

```
$ g++ -g -Wall -c list.cpp
$ g++ -g -Wall -c main.cpp
$ g++ -g -Wall main.o list.o -o main
```

Note that we are using two flags in the compilation: -g and -Wall. These are not required, but they can be useful: -g optimizes the code so it will run faster, and -Wall will give you all the warnings, which can be helpful for debugging.

Here the object file list.o must be recompiled whenever list.h or list.cpp is updated, so we say that list.o depends on list.h and list.cpp. Similarly, main.o depends on main.cpp and main.h.

You can use make to describe dependencies among files, and ensure that a file is recompiled only when its dependent files are changed. Below we will show how to use make to build the executable file main in a single step.

Basics: Calling make

You use make by creating a file named Makefile for your program. Makefile contains rules that describe how the program should be compiled.

You run make with this command:

```
$ make target
```

This command searches the current directory for a file named Makefile, and then tries to update target if it is out of date. You will read more about rules and targets below.

If no target is specified:

```
$ make
```

then make will use the first one in the makefile:

Change to the directory where you unzipped list, and run the command make. You will see that the program main is built. (If you already ran the above calls to G++, then remove binary files main.o and main.)

```
$ cd list
$ ls
Makefile  list.cpp  list.h  main.cpp
$ make
g++  -g  -c list.cpp -o list.o
$ ls
Makefile  list.cpp  list.h  list.o main.cpp
```

Note that this only compiled list.o. That is because that is the first rule in the Makefile. To make main, we need to say:

```
$ make main
g++ -g -c main.cpp -o main.o
g++ -g main.o list.o -o main
```

Now we created the executable main. Note how make did not recompile list.cpp, since list.o was up to date (we did not change list.cpp or list.h).

Note also that the -Wall flag is not used. We will fix this below.

Rules

Makefile rules have the following format:

```
 target: prerequisite1 prerequisite2 ...
	command
```

Here, target is (in most cases) the name of a file that will be updated by this rule. The list prerequisite1 prerequisite2 ... gives files that target depends on. These may be object files or source code files. On the second line, command shows how to update the target when one of the prerequisite files are modified.

As an example, we can write this rule to compile the object file mystrings.o:

```
# compile list.o
list.o: list.cpp list.h
	g++ -g -Wall -c list.cpp -o list.o
```

By listing mystrings.cpp and mystrings.h as the prerequisites, we tell make that mystrings.o must be recompiled whenever mystrings.cpp or mystrings.h is updated. To ensure that this happens correctly, make will check the timestamps on all files and recompile mystrings.o when either of the source files has a newer timestamp. If mystrings.o is not out of date then it will not be recompiled.

Command lines in a makefile must start with a tab character, not a sequence of spaces. So in this example g++ is preceded by a tab.

Lines that start with # are comments and will be ignored. Lines containing only whitespace are also ignored.

Our makefile should also include these two rules to build our program:

```
# compile main.o
main.o: main.cpp main.h
	g++ -g -Wall -c main.cpp -o main.o

# link main
main: main.o list.o
	g++ -g -Wall list.o main.o -o main
```

You can use multiple lines for one command by ending the previous line with a backslash ("\").

The command line need not be a compilation instruction; instead it can be any shell command or even empty, as we will see below.

Phony targets: all and clean

The target of a rule does not have to be an actual file that is created by make; instead it can correspond to an action, such as removing files or running tests. These targets are known as phony targets

It is conventional to include a target named "all" to update all outputs of the makefile:

```
# top-level rule
all: main
```

This rule has no command, instead it will call the rule with target "main" that we defined above. We make "all" the default target so that all outputs are updated when the user calls make without specifying a target:

```
$ make
```

Recall that we can make "all" the default by listing it as the first rule in the makefile.

It is also common to include a "clean" target to remove outputs of the makefile such as executables and .o files:

```
clean:
	rm -f list.o main.o main
```

Here we use the -f flag with rm to avoid generating error messages when a file does not exist.

The command make clean will do what we expect as long as we don't have a file in the current directory named "clean". To avoid problems when this file does exist, use the special target .PHONY to indicate that "clean" is not the name of an actual file:

```
.PHONY: clean
clean:
	rm -f list.o main.o main
```

- Add a clean target to the Makefile, and then test this target:

```
$ make clean
rm -f list.o main.o main
$ ls *.o
```

```
ls: *.o: No such file or directory
$ ls main
ls: main: No such file or directory
```

Variables

You can define variables in a Makefile so that it is easier to update parameters that appear in multiple commands. For example, we can add these variables to tell make to use g++ as the compiler and "-Wall -g" as the compiler options:

```
# use "g++" to compile source files
CXX = g++
# set compiler flags
CPPFLAGS = -Wall -g
```

We then update rules in our Makefile to use these variables. To access the value of a variable use $( ) as shown below.

```
# top-level rule
all: main

# compile list.o
list.o: list.cpp list.h
	$(CXX) $(CPPFLAGS) -c list.cpp -o list.o

# compile main.o
main.o: main.cpp list.h
	$(CXX) $(CPPFLAGS) -c main.cpp -o main.o

# link main
main: main.o list.o
	$(CXX) $(CXXFLAGS) main.o list.o -o main

# remove outputs
.PHONY: clean
clean:
	rm -f main.o list.o main
```

Here the CXXFLAGS variable is used to indicate linking options. Since this variable is not set in our makefile, it will be empty when teststr is linked.

You can use conditional statements when you define variables. For example:

```
EXE=
ifdef WINDIR #Variable specifying if we are in a windows environment
```

```
  EXE=.exe
endif

myprogs$(EXE): foo.o bar.o
    $(CXX) $(CPPFLAGS) -o myprog$(EXE) foo.o bar.o
```

Here, the ifdef statement updates EXE when the user has also defined a variable WINDIR. In this case the target of the subsequent rule becomes myprogs.exe instead of myprogs.

Variables can be used to keep lists of files. Here we use a variable to track the object files that the program myprog depends on:

```
OBJS = foo.o bar.o baz.o

myprog: $(OBJS)
     g++ -o myprog $(OBJS)
```

We can also use a variable for the name of the output executable:

```
OBJS = foo.o bar.o baz.o
EXE = myprog

$(EXE): $(OBJS)
     g++ -o $(EXE) $(OBJS)
```

• Update the CPPFLAGS variable in your Makefile as shown above so that the -Wall flag is included when object files (*.o) are compiled. Test your changes.

```
$ make
g++  -Wall -g  -c -o list.o list.cpp
g++  -Wall -g  -c -o main.o main.cpp
g++  mystrings.o main.o -o main
```

Variables can also be set from the command line, in which case they override the definitions in the makefile:

```
$ make clean
 rm -f main main.o list.o
 $ make "CPPFLAGS=-g"
 g++  -Wall  -c -o list.o list.cpp
 g++  -Wall  -c -o main.o main.cpp
 g++  list.o main.o -o main
```

Automatic variables and implicit rules
In addition to variables that you define, there are several automatic variables that can be used in

commands. Some of the most common are:

* $@ - target
* $^ - all prerequisites
* $< - the first prerequisite

Automatic variables come in handy especially when writing implicit rules. Implicit rules are extremely useful tools to specify how to deal with all files with the same suffix. For example, this rule:

```
%.o : %.cpp
        $(CXX) -c $(CPPFLAGS) $(CPPFLAGS) $< -o $@
```

specifies how to create an object file n.o from source file n.cpp. Note how no particular target is mentioned, instead the '%' character is used to match with any object file that ends in .o or .cpp.

You can include implicit rules such as the one above in your Makefile. However, there are a number of "built-in" implicit rules that make uses automatically unless you override them specifically. You can review these rules in the GNU Manual. In particular, note that object file n.o is compiled automatically from source file n.cpp with the command '(CXX) -c $(CPPFLAGS) $(CXXFLAGS)'.

The zip file for this lab contains the file Makefile, a makefile for the program described above. Note that these rules for .o files do not have command lines:

```
# compile - relies on built-in rule for .cpp files
list.o: list.cpp list.h

main.o: main.cpp main.h
```

Here, mystrings.o and teststr.o will be compiled from mystrings.cpp and teststr.c using built-in rules. The prequisite lists in each rule ensures that each will be recompiled when mystrings.h is updated.

Exercise 1 <<10 points>>

Write a make file for both of the list ADTs (simulataneously). Your makefile should be in a directory that contains the two subdirectories list and list_end. Note that you can name files in subdirectories in commands like this:

```
#compile list/list.o
list/list.o: list/list.cpp list/list.h
        g++ -c -Wall -o list/list.o list/list.cpp
```

All of the executable and oject files (.o) you create should be in the correct directory: e.g.

list/list.o if you compiled list/list.cpp.

Note that you could compile and test these two projects with two lines:

```
$ g++ list/list.cpp list/main.cpp -o list/main
$ g++ list_end/list.cpp list_end/main.cpp -o list_end/main
```

However a Makefile will avoid recompiling up-to-date files.

Your makefile must meet the following requirements:

* The makefile has rules to create two executables called list_end/main and list/main.
* The makefile creates object files list.o, and main.o in each of the directories list and list_end.
* The makefile includes rules for targets all and clean. all should compile both the list_end an the list project, and clean should clean both directories.
* The compiler and the compiler flags are set with variables CXX and CPPFLAGS.

Hint: start off simple, with just one of the two projects. List already has a makefile, move it one directory up: if you start from the directory list, do the following:

```
$ cd ..
$ mv list/Makefile .
$ ls
list list_end Makefile
```

Now edit the Makefile: change the names of the files to contain the subdirectory (e.g. change list.cpp to list/list.cpp), add the all and clean rules, and the variables. Finally, add the other project (list_end).

Be sure to represent dependencies correctly. For example, list/list.o should depend on list/list.h and list/list.cpp

Submit the Makefile for the program.

FLTK: A simple GUI library

In the second part of the lab you will be writing a series of C/C++ programs that use the FLTK library (http://www.fltk.org/), a simple library for building graphical user interfaces (GUI). This library will enable you to write programs that open windows, react to events like button clicks, etc. The FLTK library is already installed on the CS machines, so there is no need to install it first.

Exercise 2 <<5 points>>

In this first exercise, you are provided with C/C++ code that uses the FTLK library. The goal of this exercise is for you to see how the library is included in your program, and how this affects how your program is compiled. In the next exercise you will build on this code.

You do not need to hand in any code for this exercise. When you are done, raise your hand and the instructor will verify that you've compiled and run the FLTK program correctly.

The code you will compile and run is similar to the example included in the FLTK documentation:

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv)
{
    Fl_Window *window = new Fl_Window(300,200);
    Fl_Box *box = new Fl_Box(20,20,260,100,"Hello, World!");
    box->box(FL_UP_BOX);
    box->labelsize(36);
    box->labelfont(FL_BOLD+FL_ITALIC);
    box->labeltype(FL_SHADOW_LABEL);
    window->end();
    window->show(argc, argv);
    return Fl::run();
}
```

For now, don't worry about the code inside the main function. Nonetheless, notice the **#include** statements at the top of the file. These statements include the FLTK header files, so the compiler will be aware of the classes, functions, and other declarations included with the FLTK library. Since the FTLK library is already installed on the CS machines, the include files are already in a well-known standard location (in **/usr/include**; take a look at this directory, and you will see header files from a wide variety of libraries). So, the include statement specifies the path to the header files <between less-than and greater-than symbols>. This instructs the compiler to search for the files in standard locations (like **/usr/include**).

Now, try to compile the program:

```
g++ test_fltk.cpp -o test_fltk
```

You should see a lot of “undefined reference” errors like this:

```
/tmp/ccwUroJt.o: In function `main':
test_fltk.cpp:(.text+0x42): undefined reference to
`Fl_Window::Fl_Window[in-charge](int, int, char const*)'
test_fltk.cpp:(.text+0x12c): undefined reference to
`fl_define_FL_SHADOW_LABEL()'
test_fltk.cpp:(.text+0x146): undefined reference to `Fl_Group::end()'
...
```

These are all *linker errors*, meaning that the program compiled successfully, but failed in the linking step because the compiler could not find the implementation of the FLTK functions used in our program (in fact, if you try to compile without linking, using “**g++ -c test_fltk.cpp -o test_fltk.o**”, you will see no error messages at all).

What is the problem? Although we have correctly included the FLTK headers in our code, and those headers are correctly installed on our systems, we still have to tell the compiler to link with the FLTK libraries. This is done with the “**-l**” option:

```
g++ test_fltk.cpp -l fltk -o test_fltk
```

Now, your program should compile fine. If you run it, you should see the following:

Now, let's take a closer look at the code inside the main function:

```
Fl_Window *window = new Fl_Window(300,200);
Fl_Box *box = new Fl_Box(20,20,260,100,"Hello, World!");
```

First of all, we create a new window (of size 300 pixels by 200 pixels). This is done by creating a new **Fl_Window** object. Next, we create a “box” with some text in it. The box is created in the window we just created and, in particular, it is placed in coordinates

(20,20). In most GUI libraries, the origin point of a window is its upper-left corner. The box has size 260 pixels by 100 pixels, and its text is “Hello, World!”. As we can see when running the program, this box appears at the top of our window.

```
box->box(FL UP BOX);
box->labelsize(36);
box->labelfont(FL_BOLD+FL_ITALIC);
box->labeltype(FL_SHADOW_LABEL);
```

The elements in our GUI (windows, boxes, buttons, etc.) are usually called *widgets*. Widgets generally have a set of attributes that we can modify, to customize the look of our GUI. So, after creating the box, we modify a couple of its attributes using member functions of the **FL_Box** class:

- We indicate the type of box using the box() method. **FL_UP_BOX** indicates that this is a “raised” box (notice how the box stands out from the rest of the window).
- We set the font size of the text in the box using the **labelsize()** method.
- We set the font properties (bold and italic) using the **labelfont()** method.
- We set the type of label using the **labeltype()** method. In this case, by specifying **FL_SHADOW_LABEL** we have added a shadow effect to the text.

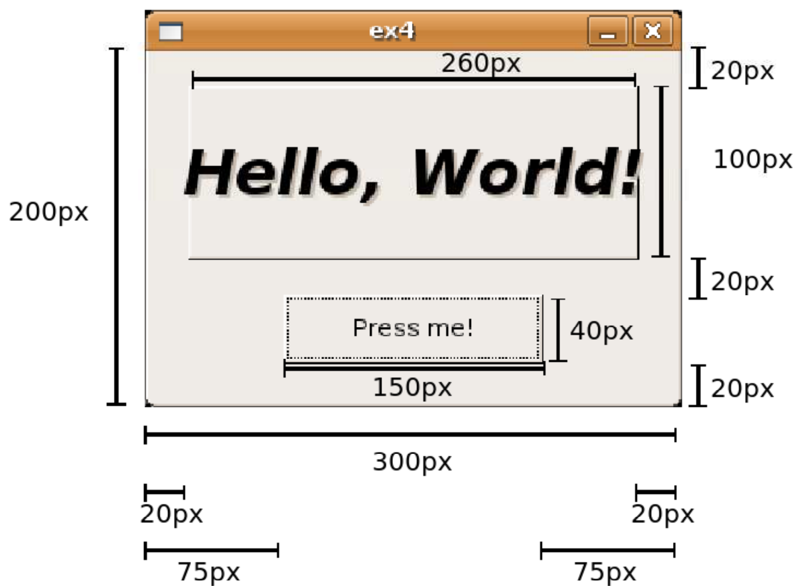
```
window->end();
window->show(argc, argv);
return Fl::run();
```

Finally, we indicate that we have finished specifying this window using the **end()** method. If we created more widgets, they would not be added to our window. Next, we make the window visible using the **show()** method. Finally, we call FLTK's **run()** function, which will create and visualize the GUI we just specified.

Exercise 3 <<10 + 5 points>>

The previous exercise showed a very simple example of a FTLK-based program. There are dozens of other widgets we could add to our program, and literally hundreds of other aspects we could modify in our GUI, and the example is just meant to illustrate how we can add new functionality to our programs (like a GUI) with relatively little effort (compared to writing a GUI library ourselves). If you want to learn more about FLTK, the best starting points are the FTLK website and the FLTK programming manual (http://www.fltk.org/doc-1.1/toc.html). In fact, in this exercise, you will have to look at the manual to find instructions on how to add a new widget to your program.

In this exercise, you must add a *button* widget, with the text "Press me!" on it. The dimensions and position of the button are specified in the following figure:



Note that adding this button involves adding a single additional line to your program.

Exercise 4 <<Extra credit: 5 points>>

Modify the program so that, when the button is clicked, its label changes to "You clicked me!".