

CSP 125 Lab 2: Programming Tools

1. Introduction

This lab will introduce you to the tools that you will be using to complete the programming exercises and projects for CSP-125. The new tools that you will be practicing with this week are:

- emacs
- the compiler: g++

Before you begin, remove the question and answer sheet from the back of this handout.

2. First C++ Editing Adventure

Before beginning you should **create a directory** to hold all of your assignments and projects for this course. This directory should be named `csp125`. Each week you can then create a subdirectory for each lab assignment. To create this course directory use the `mkdir` command as follows

```
$ mkdir csp125
```

Using the `chmod` command, **change the permissions on the `csp125` directory**. You should change the permissions on this directory so no one can read from the directory but you. This step is very important; it ensures that no one will be able to copy your work from your directory.

```
$ chmod go-rwx csp125
```

Change into this directory using the `cd` command. Now create a `lab2` directory inside of your `csp125` directory. You should create a new directory for each lab or project. This allows you to organize your files.

```
$ mkdir lab2
```

After you have changed directories, you will use the `cp` command, `cd`, to **copy the files used for this lab project**. This file is located at: `~goldwasser/csp125/demos/Hello/Hello.cpp` You can copy it into your directory using the `cp` command as follows. Remember, UNIX is case-sensitive.

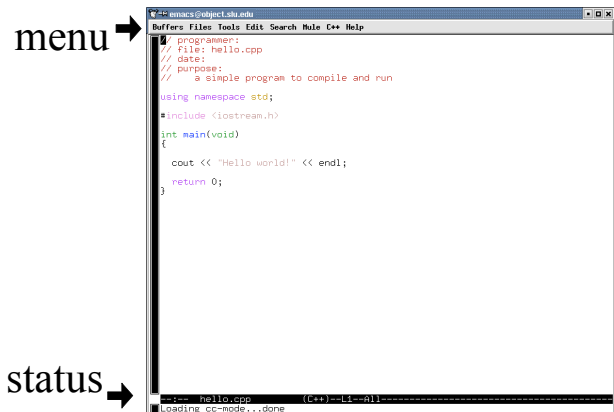
```
$ cp ~goldwasser/csp125/demos/Hello/Hello.cpp hello.cpp
```

You can verify that you successfully copied the file by using the `ls` command. Now **open the file for editing using the emacs program**. Make sure that you end the command with the `&` character. If you fail to do so, you will not be able to continue using the shell until you exit the emacs program.

```
$ emacs hello.cpp &
```

When you start emacs, a window will appear that looks like (but not exactly as) the one in the figure at the right. Since the name of a file was given when emacs was started, the file will appear (loaded) in the emacs window. Before you start editing the program, let's take a look at emacs.

Emacs is an editor that can be used to edit any text file. Emacs was used in the first lab to create a web page. Now emacs is being used to edit a program file. Both web pages and program source code files are just text.



Emacs has special capabilities when used to edit a program file. One thing you should notice is that different parts of the program are colored differently. Also notice the status line. The information there indicates the name of the file being edited and the line number the cursor is on. Knowing the line number will be valuable when debugging programs (later). Also, on the status line you can tell if a file needs to be saved. You will see in a moment.

STOP now and answer questions 1 and 2 (only) on the question and answer sheet.

Once you have opened the file, **add/change comments** at the beginning of the file to include your name and today's date. C++ comments begin with `//`. All source code files should begin with comments that include your name, the current date, the lab number or project name, and a short description. This will make it easier for you to identify the files later. After you have added the comments, **save the file** (Ctrl-x, Ctrl-s) and then exit emacs (Ctrl-x, Ctrl-c).

Emacs has a menu with familiar edit, cut, copy, and paste operations. You can also save and exit using the menu, but remembering the keystrokes for saving and exiting will save time (over and over).

STOP now and answer question 3 on the question and answer sheet.

3. First Compilation Adventure

This section of the lab introduces you to the C++ compiler that will be used in this course. We will be using a program called `g++` as our compiler. A compiler is a program that translates source code into a program that can be executed by a computer.

To use the compiler to **compile the hello.cpp** program, using the shell window, execute the `g++` program followed by the file name of the file containing the C++ source code that you wish to compile. (C++ files typically use the file extension '.cpp') Use the following command:

```
$ g++ -o hello hello.cpp
```

The `-o` argument specifies the name of the application that you wish to create--in this case, `hello`. You can remember the meaning of the `-o` option as the name of the *output* of the compiler.

Now view the contents of the directory using the `ls` command to see the executable file that results.

STOP now and answer questions 4 and 5 on the question and answer sheet.

You will see that there a file named `hello` is create by the compiler. To execute this program you must type the following command:

```
$ ./hello
```

Depending on the configuration of your account, if you only type the file name of the executable you will not be able to execute the program. You must pre-pend the command with `./` so that the program can execute in your current directory. If you don't, you will get the following error message:

```
$ hello
bash: hello: command not found
```

Open the hello.cpp file once again using emacs. Once you have opened the file, add a second output statement (the one that begins 'cout') to print your name to the screen. Follow the example given by the first output statement.

After you have added the statement, save the file (Ctrl-x, Ctrl-s) and then exit emacs (Ctrl-x, Ctrl-c).

Now recompile the source code file using g++. You must recompile before your changes will be reflected in the executable file (hello). Use the following command again:

```
$ g++ -o hello hello.cpp
```

Now execute the hello program again to ensure that your changes worked.

The exercise you have just completed was intended to introduce you to the basic use of the g++ compiler and the emacs editor. You may have noticed the Hello.cpp program used is the same one that is in the book.

4. Second Compilation Adventure

Now copy the second file that you will use for this project. This file can be copied from ~bouvier/pub/variables.cpp using the command `cp ~bouvier/pub/variables.cpp .`

Open this file with emacs and add comments that contain the same information that you added to the previous file: name, date, title/lab number, and description. Save your work, but you don't have to exit the editor.

We'll take this opportunity to mention two more options available with the g++ compiler: These parameters are:

- `-ansi` Disables certain features of g++ that do not follow the ANSI standard for C++
- `-Wall` Provide verbose warning messages for conditions that frequently cause bugs

These options should be used when you compile all of your programs. They perform extended error checking to ensure that your program compiles accurately.

Compile this file using the g++ compiler. The following command will do the trick:

```
$ g++ -Wall -ansi -pedantic -o variables variables.cpp
```

STOP and answer questions 6 and 7 on the question and answer sheet.

Obviously, this program as it is currently written does not compile successfully. There are a few bugs in it that you must fix. Edit the file in emacs and correct the errors. Be sure to save your changes and then recompile the source code before attempting to run the program.

When attempting to fix the errors work on the first error message first. Later errors messages may be caused by one of the first errors. When looking for an error, look on and above the line number indicated. When you compiled the program, you should have seen something like this (though the line numbers may vary depending on how many comment lines you have):

```
variables.cpp: In function `int main ()':
variables.cpp:16: parse error before `='
variables.cpp:25: non-lvalue in assignment
variables.cpp:31: parse error at end of input
variables.cpp:12: warning: unused variable `int e'
```

The first line of the output tells you the file and the function the messages are related to. Notice that the first three messages (the second through forth lines) are **syntax error messages** for lines 16, 29, and 38. The last line is a **warning message** related to line 16. Let's go after the error messages.

The first **error occurs on, or above, the line indicated** by the compiler. Can you find it?

It so happens that the first error does not occur on the line number indicated. Often when you get a parse error syntax error message, the problem is a missing semicolon on a preceding line. In this case, the variable declaration line is missing a semicolon at the end.

Make the change in the editor, save the change, and recompile.

The **error occurs on, or above, the line indicated** by the compiler. Can you find it?

The meaning of the 'non-lvalue in assignment' message is that the left-hand side of the assignment statement is not just a variable. The statement: `a * b = c;` should be `c = a * b;` While it is true these two statements mean the same meaning in algebra, they are very different in programming. One is a legal **assignment statement**; the other is not a legal statement in the languages (thus, an error).

Make the change in the editor, save the change, and recompile.

The **error occurs on, or above, the line indicated** by the compiler. Can you find it?

The meaning of the 'parse error at end of input' message is that the compiler did not recognize the end of a program before it found the end of the file. This can be a difficult error to find and fix. The error can be many lines above the line (last line of the file) indicated. In this case, the mistake is a missing close curly brace at the end of the `main()` function.

Make the change in the editor, save the change, and recompile.

You will see something like this:

```
variables.cpp: In function `int main ()':  
variables.cpp:13: warning: unused variable `int e'  
variables.cpp:18: warning: statement with no effect
```

There are no more errors, but one new warning appears. Though a program that compiles with warnings will produce an executable program, you should work to eliminate the source of the warnings. In the case of the unused variable, just delete the declaration for it.

While it is true the program you have just been working on does nothing useful, it was not intended to be useful as a program. The use of the program was to give you experience in working with a program that does not compile and have you experience the *edit-compile-execute cycle*.

STOP and answer questions 8 through 11 on the question and answer sheet.
--

Before you leave, you must check with the lab instructor or teaching assistant. They will ask to see your question and answer sheet for this lab and may ask you questions concerning the lab assignment and may ask you to demonstrate some of the work you have done.

QUESTION and ANSWER sheet for CS-P 125 Lab #2

Name:

Section:

Lab Day:

1. Fill in the following chart by identifying program words and symbols of different color as they appear in the `emacs` program.

color	example words	what it is
red		
green		
blue	<code>main</code>	name of a function
black		
pink		
gold		
purple		

2. What characters appear before the file name in the `emacs` status line (before editing)?
3. What characters appear before the file name in the `emacs` status line (after editing)?
4. What are the name, size, and permissions of the new file?
5. Which file is larger, the source code or the executable file?
6. What error messages are reported?
7. Did an executable file get created? (use `ls` to list the files)
8. Where can an error appear (with respect to the line number in an error message)?
9. What does `'non-lvalue in assignment'` mean?
10. Did an executable file get created when you compiled `variables.cpp`?
11. What is the difference between a syntax error and a warning?

CSP 125 Lab 2: Programming Tools

Take Home Lessons

emacs

save the file	Ctrl-x, Ctrl-s
exit	Ctrl-x, Ctrl-c

unix shell (commands by example):

\$ mkdir csp125

mkdir – **make a directory** named <something> in the current directory. In the example, ta directory named csp125 is created

\$ ls

ls – **list** the files in the current directory. The result is just a list of filenames in alpha-order.

\$ ls - l

ls – **list** the files in the current directory, use the **long** listing. In the long listing you might see something like this:

```
drwxr-xr-x 81 djb  staff    2754 18 Jan 08:46 .
-rwxr-xr-x  1 djb  staff   352316 18 Jan 08:46 a.out
-rw-r--r--  1 djb  staff    370 18 Jan 08:27 variables.cpp
```

The long listing includes the following information

-rwxr-xr-x	1	djb	staff	352316	18 Jan 08:46	a.out
mode						
owner						
group						
size (bytes)						
date and time						
filename						

\$ chmod go-rwx csp125

chmod – **change the mode (permissions)** of the named file or directory. The three permissions on each file are: read, write, and execute. There are three types of people that can have these permissions, the file's owner, other people in the group, and anyone not in the group else (called other) In the example, g = group, o=other, r=read, w=write, x=execute; the statement used here removes (because of the minus sign) the rwx permissions for group and other for the directory named csp125.

\$ cp ~goldwasser/csp125/demos/Hello/Hello.cpp hello.cpp

cp – **copy the file** or directory. The command is used like this: cp <old> <new>, where <old> represents the existing file to be copied, and <new> represents the name of the new (or copy) to be made. The ~ in the example means start with the home directory of the named user. In the example, the copy command is used to copy a file from Dr. Goldwasser's directory space. You can only do this with his permission (as set on the files and directories he controls).

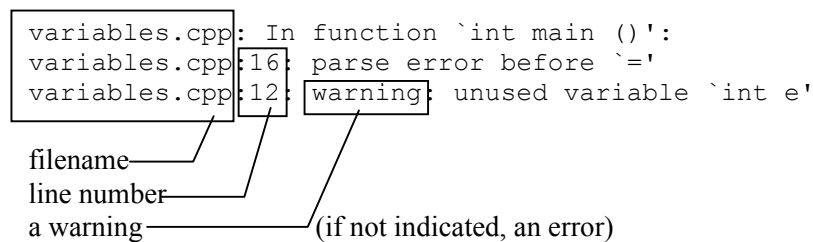
In the copy command cp ~bouvier/pub/variables.cpp . the dot represents your current directory.

compiler error and warning messages:

Compiler error and warning messages look like this:

```
variables.cpp: In function `int main ()':  
variables.cpp:16: parse error before `='  
variables.cpp:12: warning: unused variable `int e'
```

filename
line number
a warning (if not indicated, an error)



REMEMBER: The **error occurs on, or above, the line indicated** by the compiler.

Often when you get a **parse error message**, the problem is a missing semicolon on a preceding line

The meaning of the '**non-lvalue in assignment**' message is that the left-hand side of the assignment statement is not just a variable.

The meaning of the '**parse error at end of input**' message is that the compiler did not recognize the end of a program before it found the end of the file. This can be a difficult error to find and fix because it can be caused by a variety of mistakes and the mistake can be many lines above the line (last line of the file) indicated. Some possible mistakes that cause this error message: mismatched quote, mismatched parenthesis, mismatched curly braces.