

Compiling with Multiple Files

The Importance of Debugging

CS 16: Solving Problems with Computers I
Lecture #7

Ziad Matni
Dept. of Computer Science, UCSB

Lecture Outline

- Programming in Multiple Files
- The Magic of Makefiles!
- Design and Debug Tips
 - Designing and Debugging Loops
 - The Mighty TRACE
 - Designing and Debugging Functions

C++ Programming in Multiple Files

- Novice C++ Programming:
 - All in one .cpp source code file
 - All the function definitions, plus the `main()` program
- Actual C++ Programming separates parts
 - There are usually one or more **header files** with file names ending in **.h** that typically **contain function prototypes**
 - There are one or more files that **contain function definitions**, some with **main()** functions, and others that don't contain a **main()** function

Why?

- Reusability
 - Some parts of the program are generic enough that we can use them over again
 - Reuse is not necessarily just in one program!
- Modularization
 - Create stand-alone pieces of code
 - Can contain sets of functions or sets of classes (or both)
 - A library is a module that is in an already-compiled form (i.e. object code)
- Independent work flows
 - If we have multiple people working on a project, it is a good idea to break it into pieces so that everyone can work on their files
- Faster re-compilations & debug
 - When you make a change, you only have to re-compile the part(s) that have changed
 - Easier to debug a portion than the entire program!


```

#include <etc...>
#include <etc...>
float linearScale(...);
float quadraticScale(...);
float bellCurve(...);

float linearScale(...) { ... }
float quadraticScale(...) { ... }
float bellCurve(...) { ... }

int main()
{
    ...
}

```

```

// File: MyFunctions.h
#include <etc...>
float linearScale(...);
float quadraticScale(...);
float bellCurve(...);

// File: MyFunctions.cpp
#include "MyFunctions.h"
float linearScale(...) { ... }
float quadraticScale(...) { ... }
float bellCurve(...) { ... }

// File: main.cpp
#include "MyFunctions.cpp"

int main()
{
    ...
}

```

Compiling Everything...

```
g++ -c MyFunctions.cpp -o MyFunctions.o
```

(creates MyFunctions.o)

```
g++ -c main.cpp -o main.o
```

(creates main.o)

The -c option creates object code – this is machine language code, but it's not the entire program yet... The target object file here is always generated as a .o type

```
g++ -o ProgX main.o MyFunctions.o
```

(creates ProgX)

The -o option creates object code – in this case, it's object code created from other object code. The result is the entire program in executable form. The object file here is always generated with the name specified after the -o option.

What Do You End Up With?

MyFunctions.h	Header file w/ function prototypes
MyFunctions.cpp	C++ file w/ function definitions
MyFunctions.o	Object file of MyFunctions.cpp
main.cpp	C++ file w/ main function
main.o	Object file of main.cpp
ProgX	“Final” executable file

...and this is a simple example!!...

Wouldn't it be nice to have code that generates/controls this?

There Are Several Ways To Do This Piece-wise Approach

- See “example1” and “example2” in the demo code for this lecture (**demo_lecture07**)
- **example1**: similar to the one we just went through
- **example2**: by re-arranging headers, we can make one compile command (simpler, but also more limiting)

Make

- “Make” is a *build automation tool*
 - Automatically builds executable programs and libraries from source code
 - The instructions for **make** is in a file called ***Makefile***.
- Makefile is code written in OS-friendly code
 - Linux OS, to be precise...

Makefile

- The file must be called “makefile” (or “Makefile”)
- Put all the instructions you’re going to use in there
 - There is a syntax to follow for makefiles
 - Just type “make” at the prompt, instead of all the g++ commands
- Makefiles can easily be used to do other OS-related stuff
 - Like “clean up” your area, for example

Syntax of a Make

```
all: Exercise1 Exercise2
```

```
# This one forces the compile to use version 11 rules
```

```
# Also shows me all warnings (not just errors)
```

```
Exercise1: ex1.cpp
```

```
    g++ ex1.cpp -o ex1 -std=c++11 -Wall
```

```
# This next one's a doozy
```

```
Exercise2: ex2.cpp
```

```
    g++ ex2.cpp -o ex2
```

```
clean:
```

```
    rm *.o ex2 ex1
```

*Note: These are TAB
characters used for the
indents!
Don't use spaces!!*

Syntax of a Make

*Target "all" programs in
this project*

`all: Exercise1 Exercise2`

*Dependencies (macros)
that are declared below*

`# This one forces the compile to use version 11 rules
Also shows me all warnings (not just errors)`

`Exercise1: ex1.cpp`

`g++ ex1.cpp -o ex1 -std=c++11 -Wall`

Dependency files

`# This next one's a doozy`

`Exercise2: ex2.cpp`

`g++ ex2.cpp -o ex2`

is for commenting

`clean:`

`rm *.o ex2 ex1`

*Doesn't have to be compiling
instructions only!*

What's in Your Directory?

Before you run your make (and compile),
your directory should have *at least* these files
(per the example from the previous slide)

ex1.cpp

ex2.cpp

makefile

Using **make** in the Linux OS Environment

- Now that you have a **makefile**, you can execute a compiling process simply by issuing:

\$ make ← This is will create ALL the output executables

- Or you can execute make for one dependency (i.e. program) in particular, like this:

\$ make Exercise1 ← This is will create the output executable for
Exercise1 (i.e. the file **ex1** in our example)

- In our example, we even provided a way to “clean up” after we’re done by deleting all the executables that we created (in case we wanted to run the compiling again, let’s say)

\$ make clean ← This will delete all the executables that we created (like **ex1** and **ex2**)

Debugging Loops

Common errors involving loops include:

- *Off-by-one* errors in which the loop executes one too many or one too few times
- *Infinite loops* usually result from a mistake in the Boolean expression that controls the loop

Fixing Off-By-One Errors

- Check your comparison: **should it be < or <= ?**
 - Saw a few mistakes like this on the exam 😞
- Check that the var. initialization uses the correct value

Fixing Infinite Loops

- Common mistake: check the direction of inequalities:
should I use < or > ?
- Lean towards using < or > in your loop conditions
- Avoid equality (==) or inequality (!=)

More Loop Debugging Tips: **Tracing**

- Be sure that the mistake is really in the loop
- **Trace** the variable to observe *how* it changes
 - Tracing a variable is watching its value change *during* execution.
 - Best way to do this is to insert **cout** statements and have the program *show you* the variable at every iteration of the loop.

Debugging Example

- The following code is supposed to conclude with the variable “**product**” equal to the product of the numbers 2 through 5 – i.e. $2 \times 3 \times 4 \times 5$, which, of course, is 120.
- What could go wrong?! 😊

Where might **you** put a trace?

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
```

DEMO!

Using variable tracing

Loop Testing Guidelines

- **Every time a program is changed, it should be re-tested**
 - Changing one part may require a change to another
- Every loop should at least be tested using input to cause:
 - Zero iterations of the loop body
 - One iteration of the loop body
 - One less than the maximum number of iterations
 - The maximum number of iterations

*If all of these are ok,
you likely have a
very robust loop*

Starting Over

- Sometimes it is more efficient to throw out a buggy program and start over!
 - The new program will be easier to read
 - The new program is less likely to be as buggy
 - You may develop a working program faster than if you work to repair the bad code
 - The lessons learned in the buggy code will help you design a better program faster

Testing and Debugging Functions

- Each function should be tested as a separate unit
- Test functions by themselves: it make finding mistakes easier!
- “Driver” or “Test” Programs can help
 - Yes: create *another* program to test your original program...
- Once a function is tested, it can be used in the driver program to test other functions

Example of a Driver Test Program

```
int main()
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        → get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
              << wholesale_cost << endl;
        cout << "Days until sold is now "
              << shelf_time << endl;

        cout << "Test again?"
              << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}
```

Stubs

- When a function being tested **calls** other functions that are not yet tested, use a **stub**
- A stub is a *simplified version of a function*
 - A placeholder for the real thing...
 - i.e. **they're fake functions**
- **Stubs should be so simple**
that you have full confidence they will perform correctly

Stub Example

```
1  #include <iostream>
2  #include <cmath>
3  use namespace std;
4  double WeirdCalc(double x, double y);
5
6  int main( ) {
7      double n, m, w;
8      cout << "Enter the 2 values for weird calculation: ";
9      cin >> n >> m;
10     w = WeirdCalc(n, m) / (37 - pow(n/m, m/n) );
11     cout << "The answer is: " << w << endl;
12     return 0;
13 }
14
```

Stub Example

```
1  #include <iostream>
2  #include <cmath>
3  use namespace std;
4  double WeirdCalc(double x, double y);
5
6  int main( ) {
7      double n, m, w;
8      cout << "Enter the 2 values for weird calculation: ";
9      cin >> n >> m;
10     w = WeirdCalc(n, m) / (37 - pow(n/m, m/n) );
11     cout << "The answer is: " << w << endl;
12     return 0;
13 }
14
15 double WeirdCalc(double x, double y) // Make WeirdCalc a stub - just for testing!!
16 {
17     //return ( (sqrt(pow(3*x, y%(max(x,y)))) - sqrt(5*y/(x-6)) + 0.5*pow((x+y), -0.3));
18     return ( 7 );
19 }
```

Debugging Your Code: The Rules

- Keep an open mind
 - Don't assume the bug is in a particular location
- **Don't randomly change code** without understanding what you are doing until the program works
 - This strategy may work for the first few small programs you write
but it is doomed to failure for any programs of moderate complexity
- Show the program to someone else

General Debugging Techniques

- **Check for common errors**, for example:
 - Local vs. Reference Parameters
 - = instead of ==
 - Did you use **&&** when you meant **||**?
 - These are typically errors that might not get flagged by a compiler!!
- **Localize the error**
 - Narrow down bugs by using **tracing and stub techniques**
 - Once you reveal the bug and fix it, remove the extra **cout** statements
- Your textbook has great debugging examples

Pre- and Post-Conditions

Concepts of pre-condition and post-condition in functions

We recommend you use these concepts when making comments

Pre-condition: What must “be” before you call a function

- States what is assumed to be true when the function is called
- Function should not be used unless the precondition holds

Post-condition: What the function will do once it is called

- Describes the effect of the function call
- Tells what will be true after the function is executed (when the precondition holds)
- If the function returns a value, that value is described
- Changes to call-by-reference parameters are described

Why use Pre- and Post-conditions?

- Pre-conditions and post-conditions should be the first step in designing a function
- Specify what a function should do BEFORE designing it
 - This minimizes design errors and time wasted writing code that doesn't match the task at hand

Example

```
void write_sqrt(double x)
//   Precondition:    x  >=  0.
//   Postcondition:   The square root of x has
//   been written to the standard output
{
    cout << sqrt(x) << endl;
}
```

YOUR TO-DOs

- ☐ Finish Lab3 by next Monday
- ☐ Prepare Lab4 for next Wednesday
- ☐ Do HW7 by next Tuesday

- ☐ Visit Prof's and TAs' office hours if you need help!

- ☐ Hug a tree! (or a loved one will do)

</LECTURE>