Lab #3

Intro to C and Debugging

In this week’s lab ...

- Make and compile your very first C program!
  - The forerunner to most modern languages
  - They’re called ‘C-like languages’ for a reason!
- Cleaning up some syntax errors
- A quick guide to basic debugging

Your First C Program

Using vi, create a file called ‘helloworld.c’ and type…

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]){
    printf(“Hello %s”,”world!”);
    exit(0);
}
```

So What Is All This Stuff?

The program explained

```
#include <stdlib.h>

#include <stdio.h>
```

#include directives let you use sets of functions in your program. Most functions in C require you to have a specific #include at the top of your file.

```
int main(int argc, char* argv[]){

    printf("Hello %s","world!");

    exit(0);

}
```

So What Is All This Stuff?

The program explained

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]){
    printf("Hello %s","world!");
    exit(0);
}
```

Every C program has a main( ) function— and every C program starts at the top of it.

So What Is All This Stuff?

The program explained

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]){
    printf("Hello %s","world!");
    exit(0);
}
```

These arguments to main( ) are for reading the full command used when your program is run on the command line.

You can ignore them for now.

So What Is All This Stuff?

The program explained

```
#include <stdlib.h>

#include <stdio.h>


int main(int argc, char* argv[]){

    printf(“Hello %s”,”world!”);

    exit(0);

}
```

printf( ) is a function that prints text.

If you ever don't know what some function does, type in '*man <name_of_function_I_need_help_with>*' in any command line.

So What Is All This Stuff?

The program explained

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]){
    printf("Hello %s","world!");
    exit(0);
}
```

The '%s' here is a placeholder. printf() replaces any placeholders in its first argument with the contents of the arguments that come after it.

'%s' indicates printf should expect a string here. If it was an integer number, you'd put '%d' in the first string instead.

So What Is All This Stuff?

The program explained

```
#include <stdlib.h>

#include <stdio.h>


int main(int argc, char* argv[]){

    printf(“Hello %s”,”world!”);

    exit(0);

}
```

exit( ) makes the program quit. The number 0 indicates the program ran successfully.

Any other number typically indicates disaster.

You don’t always need exit(). If a program reaches the end of its main() function, it’ll quit on its own.

Compiling your code ...

gcc is a C compiler (we'll be using --**version 4.8**). Basic Usage:

- `$ gcc [flags] <source_file(s).c> -o <compiled> [libs]`
- `$ gcc -Wall -g -std=c99 <source(s).c> -o <compiled> -I.`

The '-Wall' turns on warnings. The '-g' makes symbols for a debugger, which you'll use later on. The '-std=c99' uses the C99 spec. The '-o' lets you set the output file name. Without it, your program will be named 'a.out' instead!

- `$ gcc -Wall -g -std=c99 helloworld.c -o hello`
- `$ ./hello`

The Makefile

As a makefile is a list of shell commands, it must be written for the shell which will process the makefile, run by the program **make**. A makefile that works well in one shell may not execute properly in another shell.

The makefile contains a list of rules. These rules tell the system what commands you want to be executed. Most times, these rules are commands to compile(or recompile) a series of files. The rules, which must begin in column 1, are in two parts. The first line is called a dependency line and the subsequent line(s) are called actions or commands. The action line(s) must be indented with a tab.

Building your own Makefile

Rule format:

```
    target [target...] : [dependent ....]
        [ command ...]
```

Special Macros:

$@	The name of the file to be made.
$?	The names of the changed dependents
$<	The name of the related file that caused the action.
%	The argument passed by **make**
-D<flag>	Compiler flags

Let’s create a Makefile

```
DEBUG=-DDEBUG
CFLAGS =-Wall -g -std=c99
LIBS =-I.

% : %.c
    gcc $< -o $@  $(CFLAGS) $(LIBS)
    gcc $(DEBUG) $< -o $@.debug  $(CFLAGS) $(LIBS)
```

Usage:

```
$ make helloworld && ./helloworld
```

Common Syntax Errors

Or: Cap’n Ctrlaltdel Returns!

- The next few slides go over some common slip-ups people make when writing C, and why they’re bad / how to fix them.

- **Cap’n Ctrlaltdel’s Blunder:** In the Lab 3 materials, there is a program called ‘treasure_map.c’ written by Cap’n Ctrlaltdel from Lab 1. It doesn’t work. Using the following slides, can you fix it?

The text the treasure_map.c program is supposed to output is in the file ‘actual_treasure_map.txt’.

Common Syntax Errors

- Missing semicolon at end of line:

```
printf("This line doesn't end properly!")
printf("This line does, though.");
```

If you don't put a semicolon at the end of a line, C thinks the instruction continues on the next line.

- Using '==' instead of '=' or the other way around:

```
if(x = 15){ }
int y == 1;
```

'=' is for assigning values to variables. '==' is for comparing if two variables are equal. If you try to use '==' in an assignment, the assignment won't do anything. If you try to use '=' in a comparison, the comparison will use whatever value was on the right side as a truth value. Either way, it can cause a lot of confusion.

Common Syntax Errors

- Using a variable without declaring it:

```
int main(){
   x = 10;
}
```

  You have to say what type a variable is, either the first time you use it, or by declaring it before the first time. If you don't, you'll get a compiler error.

- Using a variable without setting it to a value:

```
int x;
printf("%d is my lucky number!",x);
```

  If you declare a variable without assigning a value to it, the variable's value will contain whatever random junk was in your computer's memory before the computer earmarked part of it for this variable. At that point, usually no one knows what will happen next!

Common Syntax Errors

- Mixing up single and double quotes for strings:

```
char* str = 'This is a string I wrote.'
char single_char = "a"
```

In C, strings (type **char***) are always surrounded with double quotes (" "). Using single quotes (' ') indicates that what's inside them is supposed to be a single character (type **char** , with no *).

- Semicolon after a loop's definition:

```
int i;
for(i = 0; i < 10; i++);{
printf("Hello!");}
```

does not run

Putting a semicolon after the definition of the conditions of a for or while loop makes that loop *do nothing* each time it loops.

Common Syntax Errors

- Arrays, and ending conditions of a loop off by 1:

```
int my_array[10]; int i;
for(i = 0; i <= 10; i++){
   my_array[i] = i+1;
}
```

When you write a loop, remember that the loop keeps running until the condition is false. In the example, the loop runs 11 times, in which i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (because $10 \leq 10$).

This is a problem in the example because: ***C doesn't check array boundaries.*** You can write to my_array[9] (in bounds), my_array[10] (out of bounds), or my_array[11] (even more out of bounds), and the program will just do it…

…until it crashes later because you wrote over some other variable that was sitting after my_array[] in memory.

Common Syntax Errors

- Forgetting opening/closing parentheses or braces:

```
printf(“I like math. Today’s number is %d”,(2+2) ;
if(i == 1){
   //do something
 else {
   //do something else
}
```

missing closing paren

missing closing brace

It’s easy to lose track of parentheses or braces if they’re stacked one after the other. Most modern programmer’s text editors will keep track of this for you.

The gcc compiler will also print a message if it catches this kind of error, since it confuses the structure of the program pretty drastically.

Debugging Bugs (the basics)

- Of course, sometimes your program might compile fine, just behave the wrong way... what to do?

- The idea behind debugging is to follow the control flow of your program, checking how it behaves compared to what you intended.

- For complex programs, you can use a debugger to follow a program as it runs (we'll cover this later). The next few slides will cover some simpler methods.

Debugging Bugs (the basics)

- **fprintf()** debugging is simply inserting fprintf() statements into your code at *key places*.
    - The difference between **printf()** and **fprintf()** is that you can set without output to channel to **stdout or stderr**.
    - Debug messages should sent to **stderr**.
- You can turn code on/off by using -D debug flags
    - -D[flag]   If this is passed to the compiler the code is compiled, otherwise, it’s left out.

```
#ifdef [flag]
    … your debugging code …
#endif
```

Piping and Diff

- Piping is the redirection of data between devices and channels in Linux.
  - `|` Change **stdout** from one program into **stdin** for another program
    - `$ ls | grep ‘pattern’`
  - `>` Redirect the **stdout** to another device
    - `$ ls | grep ‘pattern’  > my_program.log`
  - `2>` Redirect the **stderr** to another device
    - `$ ./myprogram 2> error.log > output.log`
  - `<` Direct **stdin** from device
- The program **diff** will tell you the difference between two inputs
  - `$ diff <(./myprogram) truth.txt`

Testing Process

- The key to good debugging and software development is iteration!
- So, let's combine everything we've learned so far!
  - Download the the C program **magic_number.c** and the expected output **real_magic_number.txt**.
  - Compile it using the Makefile we made earlier:
    - `$ make magic_number`
- **It compiles, but does it work?**
    - `$ diff <(./magic_number) real_magic_number.txt`
    - `$ echo $?   ← Returns the error code of the last command run`
- Download the bash test file **test.sh**:
  - `$ bash test.sh magic_number real_magic_number.txt`