C Programming

Module 05

POSIX

By now you should have secured access to a POSIX-based computer

Mac users: DONE!

Linux users: DONE!

Windows users: Need some work...

If you are a Windows user

3 strategies for installing Linux

1) Dual Boot
2) Virtual Machine with full Linux Installation
3) Linux runtime within Windows - **vagrant**

Choosing Linux Distribution

If you've never used Linux before, you want to pick a relatively familiar distribution

- **Ubuntu** - tries to keep UI experience similar to Mac and Windows 8/10
- **Linux Mint** - tries to keep UI experience similar to Windows 7

Both are very stable, I use Linux Mint.

Dual Booting

If you already have Windows installed:

- Both Ubuntu and Linux Mint can automatically configure your machine quite easily
- Modest risk (back up your files!)
- Best performance, but need to reboot to switch between Windows and Linux.

If you want to ***commit*** to learning Linux, good option.

Virtual Machine

Virtual Box (free)

VMWare Fusion (not free, but better)

- Allows you to install Linux and run from Windows in its own Window
- Some integration (file system)

Vagrant

See my tutorial on Vagrant (website)

- Installs a linux (ubuntu) system, but without the GUI
- You access the linux machine through the command prompt
- Complete file system integration
- Very good option if you want to stick with Windows, but compile/run POSIX code.

Simple test - name it `test.c`

```c
#include <sys/utsname.h>
#include <stdio.h>

int main() {
    struct utsname data;
    uname(&data);
    printf("This program is running on %s\n", data.sysname);
}
```

Compiling

On linux, you may already have gcc installed

- If not, `sudo apt-get install gcc`
- Type `gcc -o test test.c` to compile

Run by typing `./test`

```
> This system is running Linux
> This system is running Darwin
```

Access to POSIX

Ultimately - **this is on you**!

- Installing Linux can be tricky - you need to learn how to get it done!
- Working on Linux is a little different than Windows
- I’m here to help, and so is the internet.

C Programming

C and C++ are **not the same!**

- C++ is a derivative of C
- C++ and C **do not** share the same compiler!
  - Although nearly all C code will compile under the C++ compiler…
- C++ and C share the same performance, so why C?

Why C?

Google it… you'll find plenty of opinion

- C++ is way more complicated than C - sticking to the lowest common denominator seems to be safer
- A lot of OS code was written before C++ really fully took hold
- A lot of people who write OS code tend to prefer C
  - Search for "C++ is a horrible language Linus Torvalds"
- You could write an OS in C++, but no one really has, and there doesn't seem to be much advantage to doing so!

Forget about the why… if you are going to learn OS programming, you need to learn C!

The POSIX and Win32 APIs are C

Compatibility between C and C++

- Most C programs will compile under C++
- A C++ program has no problem calling functions in a C library
  - This means that yes, you can easily call POSIX system calls from C++!
- You can compile C and C++ files into one single executable using a C++ compiler

I recommend you stick to C in our programming. Its a great learning experience, plus it will be EASIER once you understand C.

C Quickstart

- First rule - if you are going to write C, use the C compiler!
  - On Mac/Linux:
    - g++ or clang++ compiles C++
    - gcc or clang compiles C
  - On Windows:
    - Visual Studio has both compilers, and chooses based on file extension (.c or .cpp)

C Include Files

You may have seen these in C++

```
#include <cmath>   // math functions
#include <cstdlib> // random, and more
#inlcude <ctime>   // time...
```

In C:

```
#include <math.h>
#include <stdlib.h>
#include <time.h>
```

C Variable Declarations

You must declare all variables at the top of your functions!

```
for (int i = 0; i < 10; i++ ) {
    int temp = i + 1
    ...
}
```

```
int i, temp;
for (i = 0; i < 10; i++ ) {
    temp = i + 1
    ...
}
```

C-style I/O

- C++ uses streams (cout, cin, iostream)
  - C++ is actually in the minority...
- C does **not**. Its procedural and uses file descriptors

C-style I/O

```c
#include <stdio.h>

int main() {

    char buffer[256];
    int input;
    float output;

    printf("Please enter a number:  ");
    fgets(buffer, 256, stdin);

    sscanf(buffer, "%d", &input);
    printf("You entered %d\n", input);

    output = input / 3.0;

    printf("That number, divided by 3, is %0.3f.  How about that!\n", output);
}
```

C-style I/O

```c
#include <stdio.h>

int main() {

    char buffer[256];
    int input;
    float output;

    printf("Please enter a number:  ");
    fgets(buffer, 256, stdin);

    sscanf(buffer, "%d", &input);
    printf("You entered %d\n", input);

    output = input / 3.0;

    printf("That number, divided by 3, is %0.3f.  How about that!\n", output);
}
```

C-style I/O

```c
#include <stdio.h>

int main() {

    char buffer[256];
    int input;
    float output;

    printf("Please enter a number:  ");
    fgets(buffer, 256, stdin);

    sscanf(buffer, "%d", &input);
    printf("You entered %d\n", input);

    output = input / 3.0;

    printf("That number, divided by 3, is %0.3f.  How about that!\n", output);
}
```

C-style I/O

```c
#include <stdio.h>

int main() {

    char buffer[256];
    int input;
    float output;

    printf("Please enter a number:  ");
    fgets(buffer, 256, stdin);

    sscanf(buffer, "%d", &input);
    printf("You entered %d\n", input);

    output = input / 3.0;

    printf("That number, divided by 3, is %0.3f.  How about that!\n", output);
}
```

C-style I/O

```c
#include <stdio.h>

int main() {

    char buffer[256];
    int input;
    float output;

    printf("Please enter a number:  ");
    fgets(buffer, 256, stdin);

    sscanf(buffer, "%d", &input);
    printf("You entered %d\n", input);

    output = input / 3.0;

    printf("That number, divided by 3, is %0.3f.  How about that!\n", output);
}
```

C-Style File I/O

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

int main() {
    FILE * out, * in;
    char buffer[256];

    out = fopen("my_file.txt", "w");
    if ( !out ) {
        fprintf(stderr, "Something bad happened\n");
        exit(1);
    }
    fprintf(out, "Hello file\n");
    fclose(out);

    in = fopen("my_file.txt", "r");
    fgets(buffer, 256, in);
    fprintf(stdout, "%s", buffer);
    fclose(in);
}
```

C-Style File I/O

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

int main() {
    FILE * out, * in;
    char buffer[256];

    out = fopen("my_file.txt", "w");
    if ( !out ) {
        fprintf(stderr, "Something bad happened\n");
        exit(1);
    }
    fprintf(out, "Hello file\n");
    fclose(out);

    in = fopen("my_file.txt", "r");
    fgets(buffer, 256, in);
    fprintf(stdout, "%s", buffer);
    fclose(in);
}
```

C-Style File I/O

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

int main() {
    FILE * out, * in;
    char buffer[256];

    out = fopen("my_file.txt", "w");
    if ( !out ) {
        fprintf(stderr, "Something bad happened\n");
        exit(1);
    }
    fprintf(out, "Hello file\n");
    fclose(out);

    in = fopen("my_file.txt", "r");
    fgets(buffer, 256, in);
    fprintf(stdout, "%s", buffer);
    fclose(in);
}
```

C-Style File I/O

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

int main() {
    FILE * out, * in;
    char buffer[256];

    out = fopen("my_file.txt", "w");
    if ( !out ) {
        fprintf(stderr, "Something bad happened\n");
        exit(1);
    }
    fprintf(out, "Hello file\n");
    fclose(out);

    in = fopen("my_file.txt", "r");
    fgets(buffer, 256, in);
    fprintf(stdout, "%s", buffer);
    fclose(in);
}
```

C-Style File I/O

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

int main() {
    FILE * out, * in;
    char buffer[256];

    out = fopen("my_file.txt", "w");
    if ( !out ) {
        fprintf(stderr, "Something bad happened\n");
        exit(1);
    }
    fprintf(out, "Hello file\n");
    fclose(out);

    in = fopen("my_file.txt", "r");
    fgets(buffer, 256, in);
    fprintf(stdout, "%s", buffer);
    fclose(in);
}
```

Pass By Reference?

Why do you use pass by reference in C++?

Pointer are our only option in C

Arrays are considered pointers

```
char * fgets ( char * str, int num, FILE * stream );
```

```
char * fail;
fgets(fail, 100, stdin)
```

```
char success[100];
fgets(success, 100, stdin)
```

```
char * success = malloc(100);
fgets(success, 100, stdin);
free(success);
```

No new operator?

Memory management is more low-level:

```
void * malloc(unsigned int bytes)
free(void * ptr);
```

Don’t confuse `void` with `void *` - common mistake.
`void *` means “pointer to anything”

malloc

To allocate a new float on the heap:

```
float * f = malloc(sizeof(float));
```

To allocate an array of 10 integers:

```
int * a = malloc(sizeof(int) * 10);
```

No Constants

```
const double PI = 3.14159    // C++

#define PI 3.14159           // C
```

This goes further though… remember const parameters? C doesn't care about your constant parameter...

Strings?

Nope… they are called C-strings for a reason

```
#include <string.h>
...
char my_string[50];
strncpy(my_string, “Hello World”, 50);
```

POSIX WILL NOT PLAY WITH C++ STRINGS

No Classes

C only includes structs, and they are used extensively in POSIX

```
struct my_struct {
    int field1;
    int field2;
};
```

Note, C++ structs are not the same as C structs:

- no methods
- no constructor/destructor
- no base classes
- all fields are public

Declaring struct variables

```
struct my_struct x;
struct my_struct *ptr;
ptr = malloc(sizeof(my_struct));
```

Lots of POSIX system calls accept ***pointers*** to structures, so they can return multiple pieces of information

- You must allocate the struct first!
- You are going to need to read documentation!!!

Up next...

Please read Chapter 2.1 on Processes