Introduction

This document is intended to provide a very basic introduction to the C programming language, and to provide a few suggestions on how to setup one's development environment in the context of this course.

Contents

1 Fundamental UNIX Knowledge

This course is taught in a UNIX-like environment (Ubuntu Linux). It is assumed that most students enrolled will have previously taken CSCE215, or otherwise used such an environment in the past. Basic UNIX knowledge will not be covered in this course. If you do not feel comfortable with these skills, you should consult your instructor and/or TA.

Individuals wishing to brush up on UNIX topics are suggested to review Pat O'Keefe's CSCE215 resources repository: `https://github.com/okeefe/215-resources`.

You are expected to understand how to do the following tasks from a terminal environment on Ubuntu Linux:

- Manage directories (create a directory, delete a directory, set your current working directory, etc.).
- Execute a shell command given instructions.
- Find and read a man (manual) page.
- Save and restore back-up copies of your work.
- Create a tar and/or zip file of a directory or collection of files.
- Extract a tar and/or zip file.

2 Text Editors

It is assumed that many of the students enrolled in this course will have a preferred text editor. Any text editor will do, but support for C syntax highlighting is definitely preferred. If you do not already have a preferred text editor, this would be a good time to find one.

Here are some text editors that we like:

- Vim[1] - one of the most popular text editors ever created, with a modal interface and rich plugin ecosystem.

[1]`https://www.vim.org/`

- Emacs[2] - also one of the most popular text editors ever created. Unlike Vim, Emacs does not use a modal interface. Emacs also features a rich plugin ecosystem and a wide collection of features.
- Visual Studio Code[3] - an easy to use, free, open-source code editor. Relatively new on the scene, but already has a large and robust ecosystem of plugins. **We recommend Visual Studio Code, if you don't already have an editor that you like.**.

There are many, many more text editors available. Choice of text editor is very personal, and we encourage you to explore and experiment with different solutions, if you have not done so already.

3 Version Control

You should use some kind of version control for your projects. At a basic level, this might look like simply creating a time-stamped ZIP file after each editing session. How you choose to do this is up to you, but you are strongly encouraged to learn and use git[4] if you do not already know how to do so. Git is widely used in both academia and in industry, and is a skill that you will no doubt find useful far beyond this course.

There are also many other version control systems, such as Mercurial, CVS, SVN, Darcs, Bazaar, and many more. However, git is far and away the most popular and widely used VCS.

Based on our experience, we strongly discourage the use of git "front-ends" or GUI tools until you have learned to use the git command-line interface at a basic level. These tools often claim to make using git easy, but we often see beginners back themselves into a corner using such tools, which they lack the underlying knowledge to escape from. These types of tools include the likes of SourceTree, GitHub Desktop, GitKraken, Tower, Sublime Merge, and more.

Note for non-majors: if you do not intend to work in computing or a related field in the long run, learning a VCS such as git may give you far less mileage than your peers who plan to do so. All VCSs, and especially git, are complicated beasts that take a considerable time investment to learn. For computing majors, this investment is worthwhile due to the ubiquity of these tools within the field. You may want to stick with creating a time-stamped ZIP file of your project after each session in the lab.

4 Build System

There are many build systems available for compiling C code. In this course,a all projects must use GNU Make. Project templates with a pre-written Makefile will be provided. However, you should take the time to gain a basic level of familiarity with Make. You may wish to modify the Makefile in your project (for example, to add more files to your project). Make is also widely used in both industry and academia, and so is a skill you will find valuable far beyond this course.

If you are not already familiar with Make, you may want to consider Charles's *A Practical Guide to Make*[5].

5 C Basics

5.1 Memory Allocation

C requires you to manage your own memory utilization. Fortunately, in the context of this course, there will be little in the way of complex memory management. With only 2KB of memory, there is little memory to manage.

There are two ways to allocate memory in C. The first is **stack allocation**. This is usually done automatically when a function is called (or more accurately, when a scope is entered). Most scalar variables you will use in C are stack-allocated.

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

int main(void) {

        int a; // a stack allocated integer variable; a is said to be scalar
        int b[10]; // a stack allocated array of 10 integers; b is said to be a vector
```

[2]https://www.gnu.org/software/emacs/
[3]https://code.visualstudio.com/
[4]https://git-scm.com/
[5]http://cdaniels.net/2017-07-15-guide-to-make.html

```c
        int* c = malloc(sizeof(int) * 10); // a heap-allocated array of 10 integers
        int* d = malloc(sizeof(int) * 1); // a heap-allocated integer

        // Notice that a scalar in C is also equivalent to an array of one item

        /* Also keep in mind that c and d are in fact stack-allocated pointers
         * to heap-allocated memory regions. A statement such as:
         *
         * c = 7;
         *
         * Would in fact modify c to point to memory address 7, which would
         * likely cause a segfault.
         *
         * To set the zeroth element of c to 7, we could use any of the
         * following forms:
         *
         * dereferencing c:
         *
         * *c = 7;
         *
         * dereferencing c with an explicit offset
         *
         * *(c + 0) = 7;
         *
         * using array notation (this way is most preferred)
         *
         * c[0] = 7;
         */

        /* Also notice that the type of b will in fact be int*, not int. This
         * is because b it is a stack-allocated pointer to a stack-allocated
         * memory region. All of the above remarks about element access also
         * apply to b. */

        /* ... program code that does things with a, b, c and d ... */

        /* Release our allocation of c and d. a and b will automatically be
         * released at the end of our scope (i.e. when `return` is called). */
        free(c);
        free(d);
}
```

Listing 1: Example showing stack and heap allocation in C.

5.2 Pointer Manipulation

Unlike Java, which you are likely familiar with, C allows direct manipulation of raw pointer values. This can be an incredibly powerful tool when wielded with care, but is also once of C's largest footguns. A **pointer** is simply a memory address. Your system's memory can be thought of as a large linear array of bytes[6], and a memory adders as a subscript into that array.

To **dereference** a pointer means to follow or "chase" it to the location that it points to. More specifically, dereferencing implies that you are using a value stored in a memory location to refer to another memory location.

Pointers have a few key legitimate use cases:

- To pass a variable by reference (this is a common way to "return" several values, and is also ideal for avoiding spurious memory copies).
- To keep track of an allocated memory region, such as an array.
- To perform reinterpretation of a value (it is sometimes desirable to interpret a memory value as a different type without actually typecasting it).
- To perform certain kinds of string manipulation.
- To access hardware registers at known, fixed memory addresses.

[6]Or more accurately, machine-words. On AVR a machine word is one byte, but on a modern AMD64 processor a machine word is 64 bits.

It is also important to note that array subscription and pointer arithmetic are closely linked, **but are not identical**. Array subscripts correctly account for the size of the items stored in the array, wheres direct pointer arithmetic is not guaranteed to do so.

```c

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

int main() {
        printf("program started\n");

        int arr[8];
        // stack-allocated equivelant of:
        // int* arr ; arr = (int*) malloc(8 * sizeof(int));
        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;
        arr[3] = 4;
        // same as: (arr + 3 * sizeof(int))[3] = 4;
        arr[4] = 5;
        arr[5] = 6;
        arr[6] = 7;
        arr[7] = 8;

        printf("address of arr is %x\n", (void*) arr);
        printf("address of arr[3] is: %x\n", (void*) &(arr[3]));
        printf("address at &arr + 3 * sizeof(int): %x\n", (void*) (arr + 3));
        printf("value at &arr + 3 * sizeof(int): %x\n", (void*) *(arr + 3));
        printf("value at arr[3] is: %x\n", (void*) arr[3]);

}
```

Listing 2: `pointers.c`: A demonstration of some pointer arithmetic.

```
$ cc pointers.c
$ ./a.out
program started
address of arr is cc0a3730
address of arr[3] is: cc0a373c
address at &arr + 3 * sizeof(int): cc0a373c
value at &arr + 3 * sizeof(int): 4
value at arr[3] is: 4
```

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

void myfunc(int a, int* b) {
        printf("myfunc: value of a is %i, address of a is 0x%p\n", a, &a);
        printf("myfunc: value AT b is %i, value of b is 0x%p\n", *b, b);

        a = 34;
        *b = 23;

        printf("myfunc: returning... \n");
}

int main(void) {
        int a;
        int b;

        a = 2;
        b = 4;

        printf("main: value of a is %i, address of a is %p\n", a, &a);
        printf("main: value of b is %i, address of b is %p\n", b, &b);
        printf("main: calling myfunc(a, &b)...\n");
        myfunc(a, &b);
        printf("main: value of a is %i, address of a is %p\n", a, &a);
        printf("main: value of b is %i, address of b is %p\n", b, &b);
}
```

Listing 3: `passbyref.c`: A demonstration of pass-by-reference.

```
$ cc passbyref.c
$ ./a.out
main: value of a is 2, address of a is 0x7fff4e3aab6c
main: value of b is 4, address of b is 0x7fff4e3aab68
main: calling myfunc(a, &b)...
myfunc: value of a is 2, address of a is 0x0x7fff4e3aab4c
myfunc: value AT b is 4, value of b is 0x0x7fff4e3aab68
myfunc: returning...
main: value of a is 2, address of a is 0x7fff4e3aab6c
main: value of b is 23, address of b is 0x7fff4e3aab68
```

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

int main(void) {
        float a;
        int b;
        int c;

        a = 123.456;

        /* this implies a type-cast, b will get the value 123 */
        b = a;

        /* Reinterpret a as an integer, by casting a pointer to it to an int
         * pointer, then dereferncing. c takes the actual raw value of a as it
         * is in memory.
         *
         * IMPORTANT: when doing this type of stuff, make sure that the
         * sizeof() the left-hand side is the same as the right-hand side. */
        c = *((int*) &a);

        printf("a is %f\n", a);
        printf("b is %i\n", b);
        printf("c is %i\n", c);

        printf("sizeof(int)=%lu\n", sizeof(int));
        printf("sizeof(float)=%lu\n", sizeof(float));

}
```

Listing 4: `reinterp.c`: A demonstration of pointer reinterpration in C.

```
$ cc reinterp.c
$ ./a.out
a is 123.456001
b is 123
c is 1123477881
sizeof(int)=4
sizeof(float)=4
```

5.3 Input and Output

A common task in any language is to input and output data. In C, this is most easily done via `printf()`, and `scanf()`, which make it easy to respectively output and input formatted data. There are other ways in which data may be read or written, and if you would like more information beyond this basic level overview, you should review the following man pages:

- READ(2)[7]
- OPEN(2)[8]
- CLOSE(2)[9]
- PRINTF(3)[10]
- SCANF(3)[11]

A **format-string** is a way of describing how data should be substituted into a string (or in the case of `scanf()`, how it should be parsed). Most characters in a format string are reproduced literally, but sequences of characters starting with `%` are special – they signify that a value is to be substituted into that location in the string, rather than the actual character sequence (a `%` literal is generated by using `%%` in a format string). Some of the most common format specifiers include:

- `%i` - formats an integer value
- `%x` - formats a hexadecimal value
- `%s` - formats a string
- `%f` - formats a floating point value

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

int main(void) {
        int intval = 42;
        float fval = 3.14159;
        char* strval = "hello, world!";
        char str[512];
        int num;
        int returncode;

        printf("intval as an int is: %i\n", intval);
        printf("intval as hex is: %x\n", intval);
        printf("fval is: %f\n", fval);
        printf("strval is: %s\n", strval);
        printf("you can use several format specifiers in one string: intval=%i, fval=%f\n",
                        intval, fval);

        /* Notice that we are passing the address of the variables to be parsed
         * by scanf */

        printf("enter a string> ");
        returncode = scanf("%s", &str);
        printf("your string was: %s (returncode=%i)\n", str, returncode);

        printf("enter number> ");
        returncode = scanf("%i", &num);
        printf("your number was %i (returncode=%i)\n",
                num, returncode);

        /* Also note: scanf returns the number of items parsed - you should
         * probably check this and make sure it read all of them. This error
         * checking is omitted in this simple example. */


}
```

Listing 5: `format.c`: Demonstration of `printf()` and `scanf()`

```
$ cc format.c
$ ./a.out
intval as an int is: 42
intval as hex is: 2a
```

[7] https://man.openbsd.org/Linux-5.03/read
[8] https://man.openbsd.org/Linux-5.03/open
[9] https://man.openbsd.org/Linux-5.03/close
[10] https://man.openbsd.org/Linux-5.03/printf
[11] https://man.openbsd.org/Linux-5.03/scanf

```
fval is: 3.141590
strval is: hello, world!
you can use several format specifiers in one string: intval=42, fval=3.141590
enter a string> abc
your string was: abc (returncode=1)
enter number> 123
your number was 123 (returncode=1
```

`printf()` and `scanf()` respectively write to standard out, and read from standard in. In the context of this course, that's all we're really interested in. Should you want to scan or print to a file, you should instead use `fscanf()` and `fprintf()`, which allow a file descriptor to operate on to be passed as a parameter.

A common design pattern is to write a program which must read input one line at a time. `scanf()` is not suitable for this, because it assumes that any space character is the end of the input (to one instance of `%s`) it is to read (in effect, it scans one word at a time). If you want to read input one line at a time, you should consider using `getline()`[12] to read each line, then using `scanf()` or another method to parse the contents of each line. String parsing will be discussed in more detail later in this document.

5.4 Structures

You may be used to languages that include objects, such as Java or C++. C is not object oriented, and does not allow the declaration of classes and objects. However, C does permit the creation of custom data types with multiple separate fields, which are commonly used by C programmers in contexts where a Java or C++ programmer might use an object.

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

/* declare a new type named point, which includes two float values */
typedef struct {
        float x;  /* x and y are said to be members of the structure point */
        float y;
} point;

float distance(point* p1, point* p2) {

        /* This is one of the trickiest parts of C syntax. When you have a
         * pointer to a structure, you access structure members using
         * somestruct->somemember. This is equivalent to
         * (*somestruct).somemember. */
        return sqrt(pow(p2->x - p1->x, 2) + pow(p2->y - p1->y, 2));
}

int main(void) {
        /* create a point a on the stack and initialize it */
        point a;
        a.x = 3;
        a.y = 4;

        /* create a point on the stack, with inline initialization */
        point b = {.x = 5, .y = 6};

        /* this shows how to access structure members when you have the
         * structure itself */
        printf("point a is (%f,%f) and point b is (%f,%f)\n", a.x, a.y, b.x, b.y);

        /* pass by reference, to demonstrate accessing structure members when
         * you have a reference to a structure. */
        printf("distance is: %f\n", distance(&a, &b));

}
```

Listing 6: `structures.c`: Demonstration of structure utilization in C.

[12] https://man.openbsd.org/Linux-5.03/getline

```
$ cc -lm structures.c
$ ./a.out
point a is (3.000000,4.000000) and point b is (5.000000,6.000000)
distance is: 2.828427
```

Both the `a` and `b` structure instances shown are stack-allocated. We can just as well heap-allocate a structure instance using `malloc()`. For example, we could have declared a third point like so `point* c = malloc(sizeof(point));`. However, we would later need to `free()` that `point`, and we would also need to use the `->` notation (e.g. `c->x`) to access it's members, rather than `.` notation (e.g. `c.x`).

Something else to keep in mind is that `point` in this example is just another type like `float` or `int`, but larger, since it includes more values. This means that if were to use pass-by-value, all of the members of the structure would have to be passed by vale individually. This is probably OK for a small structure like `point`, but can introduce considerable overhead for large and complex structures.

Be careful when passing structures between functions - a common mistake is to pass a structure instance that should not be mutable by reference, or one that should be by value. Such mistakes can be very difficult to debug.

5.5 Callbacks

TODO: should talk about function pointers

5.6 Strings

TODO: should talk about strings

5.7 Includes

5.8 Macros

5.9 Linked Lists

TODO: should demonstrate linked lists

5.10 Ring Buffers

TODO: should demonstrate how to do a ring buffer

5.11 Best Practices & Style

TODO: should provide a style guide, DOs, and DONTs that will be used for grading. Probably whatever cpplint / clang-tidy says is fine?