C Programming

This guide will cover the C programming language written by Dennis Ritchie. It assumes you are familiar with a programming language or two already and will not go into detail on basics like functions or data types. That is not to say this guide won't explain these things at all, but it assumes you are familiar with the core concepts. You should also have an intermediate understanding of UNIX or GNU/Linux as Unix was written entirely in C and shares a lot in common with it. Without further ado, allow us to delve into arguably the most famous programming language of all time.

The C programming language was inspired by the B programming language but fixes a lot of issues with that language. There are different variants of C (C99 released in 1999, C11 in 2011 and C18 in 2018, as of 2020). We will be using C99 as it is the most commonly used and least buggy. You may think of C as the intermediate step between machine code/assembly language and higher-level languages like JavaScript. C often requires more out-of-the-box thinking since it is a much lower level language than something like JavaScript that has plenty of libraries/APIs/frameworks. This is the blessing and curse of C, as it is fantastic for writing very low-level applications like program engines or operating systems since it is so easy to interface between C and the hardware of the computer. C begins to become less ideal when you are working with high level functions like GUI applications or web development. By far the most difficult aspect of C is the fact that you must manage memory which is not second nature to most programmers. There are key words built in which can tell the system to store variables in fast lanes such as the CPU registers, or for less commonly used variables, we can recommend that the system store them in RAM. We must create our own garbage collection, that is to say, remove stale or unused data and variables to free up space. We must carefully consider the length of arrays and so-forth.

**Intro**

As you should know if you are familiar with Unix utilities, to compile and run a C program, we compile it using the GCC (the GNU C compiler) from the GNU compiler collection. The file must have a .c extension (sometimes .c files are referred to as source files). Run the command gcc <file.c> [-o <output\_name>] where -o is an option for a new output filename instead of the default which is a.out. When a c file is compiled, what’s called an object file (a file ending with .o) is generated. We will discuss this more in the Makefile section. Almost every C file will require the stdio.h header file (standard input/output) which contains basic functions like printf() for outputting text. Header files are C files used for function, macro, and extern variable definitions. Notice I say definitions/declarations and not initialization. Since functions need to be defined before they are called, this often creates unnecessary clutter in our .c files, thus the need for .h files. The stdio.h file belongs to the C standard library, also called libc or glibc if you are using the GNU variant that comes with all Linux distros.

![A screenshot of a cell phone

Description automatically generated]()

Figure 1 All libc .h files

Libraries are a collection of both object files and header files. In other words, for us to be able to use the stdio.h file in our code, we must have the libc/glibc library installed on our pc. Virtually all computers will have this library by default, but for libraries which are not installed, you may install them with your package manager (on Linux) or online as an archive, and then copy them to /usr/lib/ in order for the library to be visible for all users on the system. To “include” (import if you are familiar with Java suntax) a .h file in our code, the keyword ‘include’ is used in conjunction with the special symbol # which we’ll go into more detail on later. For example: #include <stdio.h> where the <> braces signify that the library either exists in /usr/lib or /lib. The main function in C is aptly called main and the compiler expects this function to exist. Similar to Java, we must declare the return type of each function. main() should return an int indicating the exit status of the program. It is good practice to enter return 0; at the end of main so that the program exits.

**Datatypes**

Data types in C are sort of strange, but you must understand that a primitive language will have primitive datatypes. For example, String obviously does not exist (since you should know it is an object reference type ie. non-primitive), char is part of the integer family since C reads chars by their hex ASCII value, Boolean doesn't exist since 1 and 0 are sufficient for true and false respectively, arrays are viewed as multiple variables of the same type residing next to each other in memory, void exists, pointers are something all languages use but they are most common in C, function return types exist, structure types exist (sort of like an object but not identical) and union types exist. We will look at all of these in time. C places a large emphasis on integer types. By default, integer types (aside from char) are signed but also have unsigned counterparts. Here is a table describing the storage size and value range of each integer type.

![A screenshot of a cell phone

Description automatically generated]()

In professional C code, you will often see explicit declarations for signed and unsigned, and this is because a lot of these values are platform dependant. When no keyword is given, the variable is always implicitly signed. On the other hand, unsigned variables should always be declared as unsigned (please note that all floating point integers must be signed). Another side note about C that we will look into later is that char variables are really just unsigned integer values (0 - 255) which have corresponding ascii symbols. For this reason the int data type can represent a char variable. This should be avoided when possible for clarity and other reasons that we will look into later, although it does have certain use-cases, so just keep that in mind.

**Functions**

There is not much to discuss about functions but there are a few things to note. First of all, functions should be declared before their bodies are “initialized” ie. populated with code, or called ex. int returnNum(int i); If a function does not accept parameters, it should use void as a parameter ex. char returnNothing(void); This is for cross platform functionality since an older computer with, for instance, Unix installed on it may use an older version of C which requires that every function have a parameter. Functions that do use parameters should have their parameter's data-type defined in the braces, as you would in Java.

**Integer Literals**

Decimal: a simple number ex. 54

Octal: Begins with 0 and must be a valid octal digit

Hexadecimal: begin with 0x or 0X

Unsigned int: Ends in u

Long: Ends in l

Unsigned long: Ends in ul

**Floating-point Literals**

double or single precision float: 34.565

We can also store floats like the following: 314159E-5L Where E-5L means move the decimal place negative 5 decimal places (or you can read it as multiplying the float by 10 to the power of -5). You can see this by making the following simple script:

#define <stdio.h>

int main() {

float f = 314159E-5L;

printf("The value of literal variable f is: %f", f);

return 0;

}

which will output 3.141590.

**#define and const**

In C we can create what are called macros using #define. Keywords that begin with a hashtag ‘#’ are known as preprocessor directives. Preprocessors are part of the compilation *process* but are ran before the program is actually compiled. #define is one such preprocessor directive which are used for defining macros, as I already stated. Macros can be thought of as aliases or special variables, though they do have additional use-cases. This is useful for say, booleans. These are typically written in capital letters, the assignment operator is optional (=), and you do not need a semi-colon to end the statement since macros are normally limited to one line ex. #define TRUE 1. In order to have multi-line macros, we use a backslash ‘\’ at the end of the line before going to the next. If you are familiar with JavaScript, you probably know that const refers to a variable which is unable to be altered. This is also similar to the final keyword in Java. You may also use the word const in C. There are 3 primary differences between a macro and const: Firstly, the constant variable will have a datatype, whereas macros are sort of dynamically typed rather than statically. Secondly, constants reserve a location in memory, whereas macros are local litterals. Finally, excecuting code libraries can potentially change the value of a constant at runtime even though they should not be updated. Const is almost always preferred over macros though macros exist for a reason, so don’t completely discard them.

**Scope**

Scope is strange in C. There are 4 different keywords we can use to define the range of a variable. These 4 words are: auto, register, static, and extern. auto does not need to be declared for local variables since it is the default for variables declared within a function or block. Values with the auto scope are deleted from memory after the function/block is finished executing. The register keyword will suggest to the system to place the value of a variable within one of the registers of the CPU instead of RAM. These variables must actually fit within the registers of the CPU which means that a 32-bit machine would not be able to have a double declared with the register keyword. It should be noted that the CPU will *try* to place this value in one of it's registers, but this is not guaranteed depending on hardware restrictions. The static keyword instructs the compiler to keep a local variable in existence during the entire lifetime of the program instead of deleting it every time it exits it's scope. In other words, we essentially make local variables accessible from anywhere. Finally, extern is used for any variable which must be accessible from another file entirely. extern variables are declared in a header file, which must then be included in the source file to have access to the variable. When a variable is declared with no assignment, if the variable is either external or static, they will be initialized to 0. On the other hand, automatic variables and register variables will have garbage values so make sure you initialize them before you use them!

**Operators**

Operators are the same as you would expect and should be familiar with. Since we will be doing lower level programming in C, you should be familiar with the more uncommon operators like << and >> for bit shifting, ~ for one's compliment, and ^ for XOR. The & symbol can be used as a binary AND ie. int a = b & c; or to represent the address of a variable ie. int addressOfY = &y; The \* symbol most often refers to a pointer in C. The technical term for the asterisk when referring to pointers is the ‘dereference operator’ and is used to dereference a memory address. Whereas ‘&’ specifies the address itself, the dereference operator (\*) tells the program that we are dealing with a pointer value ie. int \*val = &y; sets a pointer of type int equal to the address of y. Pointers always store addresses and never actual values.

**Arrays**

An array in C is defined as follows: int arr[10]; with 10 being the size of the array (normally required). If the initial data is known to us, we can omit the size field and simply fill in the values within curly braces. ex. double arr[] = {21.3, 2.7, 99.65, 1.6754}; The *size* of an array initialized like this will be set to the number of variables within the curly braces and ignore whatever number is placed in the square brackets. Additionally, in most versions of C, we are unable to set size via a variable ex. int arr[size]; This also applies even if the variable is declared as constant. While constant prevents a variable from updating during runtime, at compile time the const variable’s value is still unknown, thus we cannot use it to initialize the size of an array. The best way to get around this is to use pointers and malloc which we wil look at later. In C it is important to think about arrays differently than we would in Java. arr, &arr and &arr[0] all refer to the first index of the array aka the same address in memory (note: while ‘arr’ technically points to the first index of the array, it is different from &arr and &arr[0] for reasons that we will discuss in a bit). This is because each index following the first index (0) is simply an offset of bytes. At the beginning of this document I described an array as a collection of variables of the same data type next to each other in memory. And this is exactly true. Let’s say that your computer defines the length of integer data types as being 2 bytes (32 bit machines usually have 2 bytes for int while newer 64 bit ones usually define int as 4 bytes). Well the value at address &arr or &arr[0] would be 2 bytes in length and each consecutive index would be an additional offset of another 2 bytes. So, in order to reach index 5, we could say: arr[0] + 4; or \*arr + 4; since we are moving 4 indexes to the right of index 0 (4 \* 2 bytes). The reason we don’t have to say arr[0] + 4 \* sizeof(int) is because the compiler knows that when we add numbers to pointers or arrays, that we are adding offsets, not decimal values. In other words, the compiler reads ‘arr + 4’ as ‘arr + 4 \* sizeof(array\_datatype);’ and similarily, whenever we retrieve an index using the square braces, the compiler is just adding the offset of the number we provide under the hood. Bringing back the point I raised earlier about arr being a special case apart from &arr and &arr[0], in C, arrays themselves are considered to be a “datatype” of sorts (although not technically true but it help us to think of it that way). And so, when we try to perform arithmetic on the array itself (arr), the compiler is considering arr to be the entire chunk of memory occupied by the array. In other words, the array name without any specific address referencing encapsulates the array as a whole, and adding to it will offset by the entire size of the array. This is useful, for example, in for loops. Since there is no .length attribute in C to get the size of an array, we can instead set and int len = (sizeof(arr)/sizeof(arr[0])); This will divide the entire length of the array by the datatype of its first element (which will also be the data type of all elements in the array), thus giving us the size of the array.

**Pointers**

Now we reach the real meat and potatoes of C: pointers! As you should be aware, a variable is a space in memory which contains a value and is referenced by the computer using the variable’s assigned address. A pointer, as was briefly mentioned will be declared using the \* symbol, so \*int means pointer to an int and \*char[] or more simply, \*char, means pointer to a string. Pointers must have a datatype and must point to a valid address (ie. one that contains a value of the same datatype). You can think of pointers as a form of transportation. While they are technically variables, they **always contain addresses** and act as a passage towards another variable. It is good practice to declare pointers as NULL if an address is not assigned right away. This will prevent accidental pointer exceptions. As you know, the & symbol returns the address of a variable, and pointers are no exception. A variable cannot store the address of another variable since addresses are hexadecimal literals. In other words, in order to access the address of a variable, we can either use the & sign to reference the original variable’s address, or we can output the value stored in our pointer variable (the address of the variable the pointer is pointing to). The dereference operator (\*) acts as a sort of key:value return feature. By *dereferencing* a pointer with \* before it’s name, we return the value associated with the address stored in the pointer variable. In other words, the pointer variable borrows the address of another variable of the same datatype, and the dereference operator returns the value stored in sed address. Here is a script which will print the address of a variable var, the address held in a pointer ip (which points to var), the value associated with the pointer ip's address and the address of the ip pointer. Ex.

int main () {

int var = 20; /\* actual variable declaration \*/

int \*ip = &var; /\* store address of var in pointer variable\*/

printf("Address of var variable: %p\n", &var);

/\* address stored in pointer variable \*/

printf("Address stored in ip variable: %p\n", ip);

/\* address of the pointer variable itself \*/

printf("Address of ip pointer: %p\n", &ip);

/\* access the value using the pointer \*/

printf("Value of \*ip variable: %d\n", \*ip);

return 0;

}

Output:

Address of var variable: 0x7ffcf2676b1c

Address stored in ip variable: 0x7ffcf2676b1c

Address of ip pointer: 0x7ffcf2676b20

Value of \*ip variable: 20

Notice that in printf we use %p for the address of &var since & points to the address of the variable (we can also use %u which is the sequence for an unsigned integer). In the last printf, we enter \*ip since we want the value that ip is pointing to as opposed to the address it is storing (which would just be ip without the asterisk).

If you thought that was confusing, please make sure you understand the basic concept because I am going to be addressing even more about pointers now. To start, it may already be a question in your mind, whether or not we can have pointers to other pointers; and the answer is yes, we can have an infinite amount of pointers pointing to other pointers. It is very important when looking at pointers to take their syntax one step at a time and remind yourself of what a pointer is actually doing. If we declare a pointer: int \*p1 which is set to the address of variable x: int \*p1 = &x; we know that p now stores the address of x, and by dereferencing p1, we return the value associated with the address of x. We also know that p1 has its own address (it is a variable after all, and all variables are stored somewhere in memory). Therefore, we may declare pointer int \*\*p2 which stores the address of p1: int \*p2 = &p1; and so by dereferencing p2, we get the value associated with p1’s address, which happens to be x’s address. Notice that we declare int \*\*p2 with the appropriate amount of asterisks so that we know it is a double pointer. This number increases as we continue to add pointers to pointers. We can now return the value of x by dereferencing p2 twice ie. printf(“The value of x is: %d”, \*\*p2);

Now what about pointers to arrays? Yes, those are a thing too. I explained in the array section before this that by using the array’s name by itself, we get the address of the first index of the array, which also happens to represent the array itself, as a whole. By setting a pointer of the same datatype of an array equal to the address that the array starts at, we can simply use pointer arithmetic to jump to the appropriate address and dereference it.

ex:

int main(void) {

int arr[] = {2, 4, 6, 8};

int \*ptr = arr;

printf(“Value at index 2 is: %d”, \*(ptr + 2)); //prints 6 since we jump 2 addresses and then deref

}

The last thing that I should mention about pointers is what are called function pointers. This is probably the weirdest part about pointers, but as we’ll come to discuss, functions also have addresses in memory (specificallty in a portion of memory called to stack) and the cpu jumps to these function addresses when a funtion is called. Since we know that pointers are all about storing addresses, we can naturally point to functions. The syntax for this is strange though, because we must cast each and every parameter to the pointer as well. The type of the pointer must match the return type of the function. Please also note that the brackets surrounded the pointer matter. Here is some sample code to examine:

#include <stdio.h>

void fun(int a, char b, long c);

void fun(int a, char b, long c) {

    printf("Value of a is %d value of b is: %c value of c is %ld\n", a, b, c);

}

int main(void) {

    void (\*fun\_ptr)(int, char, long) = &fun;   //pointer to function

    //void (\*fun\_ptr)(int, char, long); These two lines are equivellants to the above statement

    //fun\_ptr = &fun;

    (\*fun\_ptr)(10, ‘b’, 234456476); // call the function via the pointer with matching arguments

    return 0;

}

**2D-Arrays**

You should be aware of how 2D arrays function under normal circumstances, which is why I will be talking about 2D arrays in regards to pointers specifically. As we now know, we can look at arrays as a set of consecutive blocks of memory (based on datatype), containing some value. We also know that the array itself is a sudo-datatype, and represents the address of the entirety of the array (which happens to share the same memory address as it’s first index). By adding 1 to the array, we end up at the beggining of the memory block proceeding the array. Now think of an array of arrays, and how that would really just look like a consecutive sequence of arrays. In order to “go down a row” (fancy talk for accessing the next array), we add 1 to the array like I mentioned. Then, in order to advance in the columns, we add the offset to the address of the array we jumped to. This might sound confusing but study the code I have provided here:

main()

{

int i, arr[3][4] = {{10, 11, 12, 13}, {20,21,22,23}, {30,31,32,33}};

int (\*ptr)[4];

ptr = arr;

printf("%u %u %u", ptr, ptr+1, ptr+2); //Output (if ptr = address 5000): 5000 5008 5016

printf("%u %u %u", \*ptr, \*(ptr+1), \*(ptr+2)); //Output (if ptr = address 5000): 5000 5008 5016

printf("%d %d %d", \*\*ptr, \*(\*(ptr+1)+2), \*(\*(ptr+2)+3); //Output: 10 22 33

printf("%d %d %d", ptr[0][0], ptr[1][2], ptr[2][3]); //Output: 10 22 33

}

In the second last printf() call we output \*(\*(ptr+1)+2); Think long and hard about what is happening here. ptr is a pointer array, that is to say, an array filled with pointers of type int. If it were simply an int pointer, the datatypes would be incompatable, but by creating a pointer array which has the same amount of indexes as columns in the array, we set each pointer contained in the pointer array equal to the address of the corresponding value in arr. ptr itself is set to be the address of arr. Effectively, we have created a duplicate array. Now by adding 1 to the ptr, we are really adding 1 to the address of ptr, which happens to be the address of the encapsulating array ie. we are advancing in row count towards the second set array. We then dereference this, and this is because again, the pointer contains only the array address which is a reference type ie. it’s value *is* an address (hence why to get index arr[0][0] we use a double pointer, to dereference the address of the array which returns the address of its first index, which when dereferenced gets our first element. This is also equivellant to typing \*(\*(ptr+0)+0)). We get the address, add 2 to it (which is our offset in columns), and then dereference that. Keep in mind that the same applies for normal 2D arrays, not just pointers. Notice that in the code, when we declare ptr, we must surround the pointer part in brackets. This is because the index specifier takes precedence over the dereference operator. In other words \*int[5] = \*(int[5]) which would dereference the value of a regular array at index 5 rather than cast the index to a pointer of type int (which is what we want).

**Pointer Precedence**

I mentioned before that certain actions take precedence when operating on pointers. For example, pre and post increment/decrement operators have varying relationships with the dereference operator. You may know that postfix operators have precedence over unary operators and so, something such as \*x++ would be parsed as \*(x++). In the case that two unary operators such as the pre-increment/decrement operator and the dereference operator were put together, the compiler would read them left to right. For example, \*++x would be read as \*(++x) and ++\*x would increment the value of the dereferenced pointer x.

**Strings**

A string in C, while not a real data type, strings can be created in variety of ways. The classic example of creating a string in c is by using an array of characters ie. char greeting[6] = {'h','e','l','l','o','\0'}; The null character \0 indicates the end of a string and is required by the compiler. A simpler alternative to this is to create an array and have the compiler interpret the string as a String literal. A string literal is perhaps an odd concept to grasp, but in essence it is as it sounds: A litteral representation of a string. In other words, string litterals do not at all function like arrays. They are surrounded in double quotes “”, are appended with our null terminator automatically, and occupy twice as many bytes as the previous char array method eg. char s[] = “Hello”; The best way to declare strings however, is to declare them as a pointer to a string literal. In fact, we prefer to declare strings as pointers since it is very tedious to write out strings as we did in the greetings example. For example: char \*s = "I am considered a string"; When strings are declared as pointers, they act more like a string normally would in Java than a pointer would normally act in C ie. you usually don’t usually have to dereference the pointer. Instead, you may call the string pointer by name to refer to its value. This is because strings are primarily used for printing. The format specifier “%s” in printf expects a string literal value. If we were to try and only print the first character of the string however, it would need to be dereferenced since the “%c” format specifier does not expect a string literal ie. printf(“%c”, \*s);. There are other instances where pointer strings should be dereferenced but I won’t go over these edge cases since they just come with practice. It should additionally be noted that strings have their own header file, <string.h> (also from libc) which includes methods such as strcpy() for copying one string into another, strcat() to concatenate two strings, strlen() to return the length of a string in bytes - 1, strcmp() to compare the equivellance of two strings, strchr() to return a pointer to the first instance of a given character within a string, and strstr() to return a pointer to the first occurence of a string within a string.

**Structures**

Here is an interesting portion of the C language! A structure is a collection of related data. It is defined similar to objects in Javascript or items in a hashmap but does not contain key-value pairs (not directly at least). It is really a list of variables that may have differing datatypes. For example, I can create a Book datatype:

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

And then create an object of type struct Book:

struct Books HarryPotter;

strcpy(HarryPotter.title, "The Chamber of Secrets");

strcpy(HarryPotter.author, "JK. Rowling");

strcpy(HarryPotter.subject, "Fiction");

HarryPotter.book\_id = 23423455;

You may declare a new struct directly after the closing curly brace for the structs definition. In other words, I could have removed the line “struct Books HarryPotter” and placed it after my struct Books definition:

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} HarryPotter;

**Accessing Structure Elements**

To access the value of a structures element, we can simply print what we declared ex. printf("Author: %s\n", HarryPotter.author); Things get stranger with pointers however. To create a pointer to our Books structure we would naturally declare it as ‘struct Books \*struct\_pointer;’ We can then store the address of a child structure in this pointer ex. struct\_pointer = &HarryPotter; To access the members of a structure using a pointer, we use an arrow operator ex. struct\_pointer->title; which would be the equivalent of typing \*struct\_pointer.title; Here is some example code:

#include <stdio.h>

#include <string.h>

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

/\* function declaration \*/

void printBook( struct Books \*book );

int main( ) {

struct Books Book1; /\* Declare Book1 of type Book \*/

struct Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* book 2 specification \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* print Book1 info by passing address of Book1 \*/

printBook( &Book1 );

/\* print Book2 info by passing address of Book2 \*/

printBook( &Book2 );

return 0;

}

void printBook( struct Books \*book ) {

printf( "Book title : %s\n", book->title);

printf( "Book author : %s\n", book->author);

printf( "Book subject : %s\n", book->subject);

printf( "Book book\_id : %d\n", book->book\_id);

}

Output:

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book\_id : 6495700

Notice in this example that we set the parameter \*book in the function printBook equal to the address of the argument simply by calling the function. In other words, there is no need to set \*book = &Book(n) since we called the function using &Book(n) as an argument.

**Bit Fields**

A bit field allows data to be compressed within a structure. This is useful when memory is at a premium. For example, if you are familiar with machine words ie. a 32 bit or 64 bit value (depending on architecture), we can pack several objects into one machine word eg. 1 bit flags can be compacted. This is done by specifying the bit length after each variable. Ex.

struct packed\_struct {

unsigned int f1:1;

unsigned int f2:1;

unsigned int f3:1;

unsigned int f4:1;

unsigned int type:4;

unsigned int my\_int:9;

} pack;

This structure contains 6 members. Granted that the each member is smaller or equal to the integer word length of the computer, the compiler will pack each member as much as possible. If this is not the case, some compilers may allow memory overlap for fields, while others would store the next field in the next word.

**Unions**

Unions are the same principle as structures. The difference between the two is that a struct allocates memory for each field, whereas a union can store multiple variables of varying types into *one memory location.* This causes issues however so it must be used properly. The size of a union will be the size of it's largest member. This means that if I define a union:

union Data {

int i;

float f;

char str[20];

}

then the size of Data will be 20 bytes since the array is the largest item in our union and is 20 bytes long. Another issue with unions is that their values may overlap if the programmer is not careful. For instance, using printf after the union is declared will overlap data and output incorrect values. Here is an example:

#include <stdio.h>

#include <string.h>

union Data {

int i;

float f;

char str[20];

};

int main( ) {

union Data data;

data.i = 10;

data.f = 220.5;

strcpy( data.str, "C Programming");

printf( "data.i : %d\n", data.i);

printf( "data.f : %f\n", data.f);

printf( "data.str : %s\n", data.str);

return 0;

}

Output:

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming

#include <stdio.h>

#include <string.h>

union Data {

int i;

float f;

char str[20];

};

int main( ) {

union Data data;

data.i = 10;

printf( "data.i : %d\n", data.i);

data.f = 220.5;

printf( "data.f : %f\n", data.f);

strcpy( data.str, "C Programming");

printf( "data.str : %s\n", data.str);

return 0;

}

Output:

data.i : 10

data.f : 220.500000

data.str : C Programming

**Bit Fields**

Suppose we declare a small variable like a Boolean for example where we don't need the entire length of the data structure. We can tell C to only use the required amount of memory space necessary using bit fields. For example:

struct {

unsigned int widthValidated:1;

unsigned int heightValidated:1;

} status;

This structure requires 8 bytes of memory space since an unsigned int takes 4 bytes of space on modern hardware. Only 2 bits will actually be used since each variable will store either a 0 or 1. You may then declare a struct containing 32 1 bit values but once this limit reaches an overflow, the 33rd variable is allocated to the next slot in memory and the structure will consume 8 bytes of memory so be careful.

#include <stdio.h>

#include <string.h>

/\* define simple structure \*/

struct {

unsigned int widthValidated;

unsigned int heightValidated;

} status1;

/\* define a structure with bit fields \*/

struct {

unsigned int widthValidated : 1;

unsigned int heightValidated : 1;

} status2;

int main( ) {

printf( "Memory size occupied by status1 : %d\n", sizeof(status1));

printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

return 0;

}

Output:

Memory size occupied by status1 : 8

Memory size occupied by status2 : 4

**typedef**

Similar to how we can create macros using #define, we can create our own datatypes using the typedef keyword. For example, we can enter typedef unsigned char BYTE; and then declare a variable using the macro’s name BYTE ex. BYTE b1, b2; You can also use typedef in conjunction with unions and structs. Going back to our book struct example, I can place typedef in front of the struct declaration. The only thing that this changes is removing the need to type the keyword ‘struct’ before each and every Book variable, and is often done by C programmers to save time. Ex.

#include <stdio.h>

#include<string.h>

typedef struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} Book;

Book book1; //Normally would have to type: struct Book book1

**Function Parameters**

I figured that it would be helpful to dedicate a section towards explaining how we pass parameters in C, because this can get really messy and confusing if you don’t understand what’s going on. In C, we either pass arguments to functions using **call by value**, or **call by reference**, and this will depend on what you are trying to accomplish. We know that addresses are references to variables, and that they can be dereferenced using pointers. We also know that variables which are local to functions (at least ones declared with an automatic or register scope) are only temporarily in existence while the function is active. What I am getting at is that, if we pass a variable to a function using pass by value ie. passing the variable itself through the function, then when the function recieves it’s value, it can only be stored temporarily, and thus does not affect the variable outside of the function. In other words, if I passed the variable x as an argument int a function foo(int x), and then added 5 to x inside foo(), x would not actually increment by 5. Rather, it’s local copy within foo() would. In order to actually increment x, we would be required to use pass by reference, in which we pass x’s address to foo, and have a pointer dereference it for us. In order to do this, we make our parameters pointers, and we pass addresses as our arguments for sed parameters. Ex.

/\* declare function with pointer parameters \*/

void addTwo (int \*a, int \*b);

void addTwo(int \*a, int \*b) {

\*a += 2;

\*b += 2;

}

int main(void) {

int a = 3;

int b = 8;

addTwoNumbers(&a, &b);

printf(“Value of a after func: %d | Value of b after func: %d”, a, b);

return 0;

}

printf will be 5 as the value of a and 10 as the value of b. When it comes to passing reference type variables as arguments such as strings, arrays, or structs, the logic is the same, but we must be careful to remember that reference types act differently than normal variables. I think it will be better to simply provide an example rather than explain the logic.

Ex.

#include <stdio.h>

void pass(char \*\*s2, int arr[]);

void pass(char \*\*s2, int arr[]) {

\*s2 = "new and improved";

arr[0] = 4;

arr[1] = 5;

arr[2] = 6;

}

int main(void) {

char \*s1 = "original";

int arr[] = {1, 2, 3};

printf("%s: %d, %d, %d\n", s1, arr[0], arr[1], arr[2]);

pass(&s1, arr);

printf("%s: %d, %d, %d\n", s1, arr[0], arr[1], arr[2]);

return 0;

}

Output: 0riginal: 1, 2, 3

new and improved: 4, 5, 6

Here we see that the array does not need to have the address symbol when passed by reference because we know that calling an array by name refers to the address of its first element (in 2D arrays we do not require the row count but the column count is necesarry). Strings on the other hand, need the address specifier when using pass by reference, but note that the parameter also must be a pointer to a pointer since s1 is a string literal (remember, do not confuse character arrays and character pointers pointing to string literals). Just keep in mind that when we pass anything as an argument, we are setting the parameter equal to sed argument. With that in mind here is what is happening with our string:

1. s1 is a char pointer and is set to be equal to the string literal “original”

2. its address is passed on to the pass function which is the same as saying char \*\*s2 = &s1;

3. we dereference s2 and set its value = to “new and improved”.

**Standard IO**

As with all systems, standard IO is composed of stdin stdout and stderr. In C, there are many many functions for IO streams. To summarize IO streams if you are unaware, you can think of a stream as a one way tunnel by which we can send data to the system and recieve data. A stream must be opened to let data in, and should similarily always be closed after use to prevent data/memory leaks. In Linux/Unix systems, when the user enters some command into the terminal, the kernel stores the input into a buffer (a temporary array). This buffer can then be parsed accordingly (usually whitespace separates the command from additional arguments and the newline character when the user presses enter signifies the end of that command). There is also a system macro called EOF (end of file) which contains the value -1. EOF is used when characters are read one at a time (this is called serial reading as opposed to parallel). Since characters are essentially an unsigned int, that means they do not normally have negative values. There are a few ways in which we can use this EOF macro. As I mentioned, there are many functions used for IO. The function getchar() defined in stdlib will read the first character from stdin. stdin will either be the buffer which I mentioned ealier, or the programmer may put a file as a parameter and have it read from the file. getchar() will then convert the char which it reads into its integer equivellant. It will terminate when either EOF is generated by the user (by pressing ctrl+d) or when an error occurs. When getchar is used in a loop ie. while((c = getchar()) != EOF) and the user enters a number/word which is more than 1 character long, the while loop will continue to process every consecutive character in the input buffer. Please note that it will not wait for the user to enter another character so long as there are still characters in the buffer (this includes \n), and therefor this method should only be used when we are sure that the user will be inputting 1 character at a time (very very difficult to do error handling if user enters more than 1 character). putchar() is the function equivellant for printing one character at a time. Here are 2 snippets from stack overflow which I think highlight some important things about getchar() and EOF

![Graphical user interface, text, application, email

Description automatically generated]()

Here’s a tidbit showing to never use fgetc to check EOF because fgetc returns a char which can never be equal to EOF (-1)

![Text, letter

Description automatically generated]()

gets() and puts() are two more functions for IO which will read one line at a time (separated by \n) instead of 1 character at a time. gets() is deprecated and if you are using compiler flags such as -ansi -Wall -pedantic, you will get a message telling you that it is dangerous and should not be used. This is because we must provide it with a buffer (a char array) and if the users input is longer than the amount of space available, the user can end up overwriting memory that should not be accessed. Instead of gets(), we can use fgets() which takes 3 parameters (char \*str, int n, FILE \*stream) and is generally much safer than gets(). The first parameter is you buffer which can now be a char pointer rather than an array in gets(). The int n specifies the length of the buffer and the function will stop reading from the IO stream when it reads n amount of characters. Finally FILE \*stream is a common parameter which specfies the stream being used (stdin, stdout, stderr). Here is some example code:

#include <stdio.h>

int main () {

FILE \*fp;

char str[60];

/\* opening file for reading \*/

fp = fopen("file.txt" , "r"); //r signifies read

if(fp == NULL) {

perror("Error opening file");

return(-1);

}

if( fgets(str, 60, fp)!=NULL ) {

/\* writing content to stdout \*/

puts(str);

}

fclose(fp); //Make sure to close the stream

return(0);

}

fgets() is generally the safest and best option for user input if possible. On the other hand, it is also inconvenient due to the programmer needing to strip the newline character if they are creating a string and also figuring out how to deal with lines which are longer than the buffer. One last IO function which I will cover is printf() and scanf(). You are already familiar with printf(), and scanf() works similarily. scanf() will use a format specifier (“%[c,d,i,o,u,p,etc.]”) to determine how the parameter should be interpreted. As we discussed in the parameter/argument section, in order for scanf() to update values, primitive types must pass their address. Since our string is an array in this case, it does not need to pass it’s address (since the array intrinsically contains a reference to itself), although if our string was a char pointer, we would need to initialize it using malloc before passing it.

Ex.

#include <stdio.h>

#include <stdlib.h>

int main( ) {

char str1[100];

char \*str2 = malloc(sizeof(str1));

int i;

printf( "Enter a String :");

scanf("%s, str1);

printf( "Enter another String :");

scanf("%s, str2);

printf(“Enter an integer:”);

scanf(“%d”, &i);

printf( "\nYou entered: %s %s %d ", str1, str2, i);

return 0;

}

**Files and I/O**

First order of business is to open a stdin stream for the specified file. This is done using fopen() which will operate on a specified file. If the file does not exist, fopen() makes a system call and creates the file for us. Using this function, we initialize an object of type FILE which is a structure containing all the information necessary for controlling the stdio streams. FILE \*fopen(const char \*filename, const char \*mode); filename is a string literal which can be replaced with the file being created or opened. mode can also be replaced with r,w,a,r+,w+,a+ which can be viewed on the man page for fopen. To close the file stream, use fclose(). It is defined as: int fclose(FILE \*fp); where fp stands for file pointer/ pointer to file. The function returns 0 on success, or EOF if there is an error, thus closing the file. We've already looked at fputs() but let's view it again. fputs() writes user input to a file via stdout. It returns the input on success or EOF on failure. It is defined as: int fputs(const char \*s, FILE \*fp); Here is an example of fputs();

#include <stdio.h>

main() {

FILE \*fp;

fp = fopen("/tmp/test.txt", "w+");

fprintf(fp, "This is testing for fprintf...\n"); //Anytime you see ‘f’ in front of a function name, it usually stands for file

fputs("This is testing for fputs...\n", fp);

fclose(fp);

}

This should then output the two strings we wrote to our file test.txt. To read files, we use fgets() which is a bit more complicated than fputs(). fgets() reads a stream (in our case a file), and stores it to a buffer. A \0 is appended to the end of the last char in the buffer. It is defined as char \*fgets(char \*s, int size, FILE \*stream); where \*s is the buffer, size is the amount of characters which should be read, and \*stream is the input file.

#include <stdio.h>

main() {

FILE \*fp;

char buff[255];

fp = fopen("/tmp/test.txt", "r");

fscanf(fp, "%s", buff);

printf("1 : %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("2: %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("3: %s\n", buff );

fclose(fp);

}

This code assumes that the file from earlier still contains the contents we last viewed. Here is the output:

1 : This

2: is testing for fprintf...

3: This is testing for fputs...

fscanf() only searches for the first string as specified by "%s". fgets() only reads till the \n character so the last two lines are normal.

**Preprocessors**

The C preprocessor (CPP) is not a part of the compiler, but rather a separate step in the compilation process. In simple terms, a C preprocessor is a text substitution tool which instructs the compiler to do required pre-processing before actual compilation. Preprocessor commands begin with # such as #define. Here is a list of each command:

#define: Substitutes a preprocessor macro

#include: Inserts a particular header from another file (a library).

#undef: Undefine a preprocessor macro.

#ifdef: Returns true if macro is defined.

#ifndef: Returns true if macro is undefined.

#if: Tests is a compile condition is true.

#else: The alternative for #if.

#elif: #else and #if in one statement.

#endif: Ends preprocessor conditional.

#error: Prints error message on stderr.

#pragma: Issues special commands to the compiler, using a standardized method.

**Predefined Macros and Macro Operators**

\_DATE\_: MM DD YYYY

\_TIME\_: HH:MM:SS

\_FILE\_: Contains the current filename as a string literal.

\_LINE\_: Contains the current line number as a decimal constant.

\_STDC\_: Defined as 1 when the compiler complies with the ANSI standard.

Macros are normally confined to one line. The macro continuation operator \ is used to continue a macro that is too long for a single line.

Ex.

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n)

int main(void) {

message\_for(Carole, Debra);

return 0;

}

The token pasting operator ##, when included in a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token.

#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {

int token34 = 40;

tokenpaster(34);

return 0;

}

When compiled it will produce the output: token34 = 40. We first define a function tokenpaster() with parameter n. It prints token(n) = followed by a decimal value from the variable token##n. Here, we've used the pasting operator to append the value of #n to "token". Therefor, when we call tokenpaster(34), it returns token34's value (40).

The defined() operator is a function which checks if an identifier has been defined using #define. This is the same as using #ifdef and #ifndef. Ex.

#include <stdio.h>

#if !defined (MESSAGE)

#define MESSAGE "You wish!"

#endif

int main(void) {

printf("Here is the message: %s\n", MESSAGE);

return 0;

}

Macros with arguments must be defined using #define before they can be used. This essentially means that we can not operate on an argument passed to a Macro function. If we want to do this, we must enclose the parameter in brackets as opposed to using the # symbol. Ex.

#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {

printf("Max between 20 and 10 is %d\n", MAX(10, 20));

return 0;

}

Rather than

#include <stdio.h>

#define MAX(x,y) (#x > #y ? #x : #y)

int main(void) {

printf("Max between 20 and 10 is %d\n", MAX(10, 20));

return 0;

}

In general, if you have a function, it is best to simply declare the function rather than storing it as a macro.

**C Header Files**

As you know, .h or header files are like libraries with functions and defined variables for us to use. We can actually easily write our own header files. We typically like to keep all of our constants, macros, global variables and function prototypes in a header file which we can then access from our source code. If we define a header file "header.h" which contains the line: char \*test (void); and then a main program called program.c, we can use #define to "import" our function. For local headers (which exist in the same directory as our .c program, we can use "" rather than <>. Ex.

int x;

#include "header.h"

int main(void) {

puts(test());

}

which would be read by the compiler as:

int x;

char \*test(void);

int main(void) {

puts(test());

}

It is possible to accidentally include the header file twice which will result in the compiler processing its contents twice. We can prevent this as follows:

#ifndef HEADER\_FILE

#define HEADER\_FILE

/\*The entire header file\*/

#endif

**Computed Includes**

Sometimes it is necessary to select one of the several different header files to be included into your program. For instance, they might specify configuration parameters to be used on different sorts of operating systems. You could do this with a series of conditionals as follows −

#if SYSTEM\_1

# include "system\_1.h"

#elif SYSTEM\_2

# include "system\_2.h"

#elif SYSTEM\_3

...

#endif

But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a computed include. Instead of writing a header name as the direct argument of #include, you simply put a macro name there. Ex.

#define SYSTEM\_H "system\_1.h"

...

#include SYSTEM\_H

SYSTEM\_H will be expanded, and the preprocessor will look for system\_1.h as if the #include had been written that way originally. SYSTEM\_H could be defined by your Makefile with a -D option.

**Type Casting**

Converting a datatype into another is known as type-casting or type-conversion. You have most likely seen this at some point when doing integer division. In order to retrieve an accurate value, we can cast the return value of an int division to a double as follows:

int sum = 17, count = 5;

double mean;

mean = (double) sum/count;

printf("Value of mean: %f\n", mean);

As you can tell, the cast operator has precedence over the division, so the value of sum is first converted to double and gets divided by count yielding a double. Similarly, we can cast a char to int since char is an int type. By simply storing a char value as an int variable, we can retrieve its ascii value. Ex.

int c = 'z'

printf("Value of z is: %d", c);

**Error Handling**

There is no error handling libraries or framework in C since it is a low level language. Instead we use the standard posix streams. You are familiar with the standard io streams, but there are others we will be using such as errno, perror(), and strerror(). errno produces the exit status value of the error, perror() is a function which allows us to input our own cause of error and returns an error message from stderr, and strerror() is a function which will allow us to return the stderror message of a given errno. Here is an example:

#include <stdio.h>

#include <errno.h>

#include <string.h>

extern int errno ;

int main () {

FILE \*pf;

pf = fopen("unexist.txt", "rb");

int errnum;

if (pf == NULL) {

errnum = errno;

fprintf(stderr, "Value of errno: %d\n", errno);

perror("Error printed by perror");

fprintf(stderr, "Error opening file: %s\n", strerror(errnum));

}

else {

fclose(pf);

}

return 0;

}

Output:

Value of errno: 2

Error printed by perror: No such file or directory

Error opening file: No such file or directory

**Recursion**

Recursion is doing something repeatedly in laymans terms. The most popular example is a function calling itself. Here is a function called factorial which will take a number and multiply itself by n-1 until it reaches 1. So passing 4 as an argument would perform the equation 4x3x2x1. Ex.

#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

if(i <= 1) {

return 1;

}

return i \* factorial(i - 1);

}

int main() {

int i = 12;

printf("Factorial of %d is %d\n", i, factorial(i));

return 0;

}

Another classic example of recursion is the Fibonacci series:

#include <stdio.h>

int fibonacci(int i) {

if(i == 0) {

return 0;

}

if(i == 1) {

return 1;

}

return fibonacci(i-1) + fibonacci(i-2);

}

int main() {

int i;

for (i = 0; i < 10; i++) {

printf("%d\t\n", fibonacci(i));

}

return 0;

}

Output:

0

1

1

2

3

5

8

13

21

34

More information on recursion can be found in my data structures notes which goes much further in detail on recursion.

**Variable Arguments**

We've gone over var args before but they are defined differently in C. The first parameter is always an int representing the amount of arguments passed to the function, and elipses represent the arguments passed. This ends up looking something like: double foo(int num,...) {} Varargs are not actually something included in stdio.h so we have to include stdarg.h. Things get more complicated inside of our vararg function. First, we declare va\_list which is a structure defined in stdarg.h. Then we initialize va\_list with va\_start() by giving it our list and our num parameter. We can use va\_arg within a for loop to read out each argument and finally use va\_end to clean memory reserved for valist. Here is an example code:

#include <stdio.h>

#include <stdarg.h>

double average(int num,...) {

va\_list valist;

double sum = 0.0;

int i;

/\* initialize valist for num number of arguments \*/

va\_start(valist, num);

/\* access all the arguments assigned to valist \*/

for (i = 0; i < num; i++) {

sum += va\_arg(valist, int);

}

/\* clean memory reserved for valist \*/

va\_end(valist);

return sum/num;

}

int main() {

printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2, 3, 4, 5));

printf("Average of 5, 10, 15 = %f\n", average(3, 5, 10, 15));

}

Output:

Average of 2, 3, 4, 5 = 3.500000

Average of 5, 10, 15 = 10.000000

**Memory Management**

stdlib.h is an important header which also contains functions for memory management. Here are the main ones:

void \*calloc(int num, int size) : allocates an array of 'num' elements which are each 'size' bytes big.

void free(void \*address) : Releases a block of memory specified by the address.

void \*malloc(int num) : Allocates an array of num bytes and leaves them uninitialized.

void \*realloc(void \*address, int newsize) : Re-allocates memory extending it up to newsize (makes more room).

In the case that we don't know how large an array should be (which is common), we can dynamically create enough space using malloc which stands for memory allocate. Here is an example:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char name[100];

char \*description;

strcpy(name, "Zara Ali");

/\* allocate memory dynamically \*/

description = malloc(200 \* sizeof(char));

if(description == NULL) {

fprintf(stderr, "Error - unable to allocate required memory\n");

} else {

strcpy(description, "Zara ali a DPS student in class 10th");

}

printf("Memory address allocated by malloc: %p\n", description);

printf("Name = %s\n", name);

printf("Description: %s\n", description);

}

In this example, we create a static array called name which is 100 characters long, and enough to fit the name we copy to it using strcpy(). After that, we dynamically put aside memory using malloc without having to declare an array. \*description is a pointer of type char ie. a string. The relevance of this is explained in the next section. We set the value associated with it's address equal to the return value of malloc. If description is NULL, we know that malloc failed to allocate enough memory, otherwise we copy our sentence to the address of description.

**The Call Stack**

If you are at all familiar with assembly code you are probably familiar with the call stack. A stack is a concept in programming which conforms to the FIFO (first in last out) structure. This means that we push an item onto the stack (like putting a plate on top of a stack of plates), which is the ‘first in’ portion of FIFO, and then we pop off the first item (like taking the top plate off the stack of plates), which is the ‘first out’ part. A stack is allocated for each thread of a program and stored in memory (RAM). Local variables, as well as functions (called subroutines in assembly) are stored in the call stack. The CPU contains special system registers such as the stack pointer (SP) which points to the beggining of the stack. Another register called the program counter or more commonly, the instruction pointer (IP) points to the current instruction. It will start at SP and then decrement itself by 1 after each instruction is excecuted (since the stack goes down not up). By decrementing by the size of the instruction, IP always contains the next address it needs to go to.

**The Heap**

malloc is a command which utilizes nmap and brk which are Linux sys-calls (ie. they communicate with the system through the kernel directly). nmap will request that the kernel gives us a new memory segment whether it be through virtual memory like swap memory while brk can resize memory segments for programs. malloc is able to use these sys-calls to create new memory for our own program called the heap or heap space. The programmer has no control over where the computer decides to create this space, and the space that it allocates will be of datatype void (calloc is another version of malloc which will initialize the space to be all 0s, but this is rarely used). malloc() then returns a void pointer, which points to the newly allocated address. While the call stack may contain pointers to global variables, heap space is where the actual global variables are kept. Thus, if we initialize a global variable within a function, the global variable is stored in heap space. This is problematic because once the functions exits, the local pointer to our global variable is destroyed, and since we do not know where the global variable is stored in heap space, it will occupy that space until the program is finished running and we will have no access to it. The method free() should always be used to dereference the space occupied in the heap once it is no longer required. This is known as garbage collection; something that higher level languages like java do for us, but that is ultimately the programmers problem in C. As an additional note, you may hear people say that you must cast the void pointer returned by malloc to the user-defined pointer’s datatype. This is only true for C++. In C, void pointers are automatically cast to the datatype of the user-defined pointer, whereas in C++, we must perform a cast. To give you an example, I can create an int pointer p which points to the address returned by malloc ex. int \*p = malloc(sizeof(int)\*2); This is fine in C, but in C++ we must cast the void pointer returned by malloc to match the user-defined pointer ex. int \*p = (int \*)malloc(sizeof(int)\*2);

**Command Line Arguments**

As in Linux, we can pass arguments when we execute the program. This is done similarly to varargs but we use a pointer array instead of ellipses in main to store pointers to our arguments. For example ./a.out arg1 arg2

#include <stdio.h>

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

if(argc == 2) {

printf("The argument supplied is %s\n", argv[1]);

}

else if(argc > 2) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

**Makefile**

In order to create an executable which joins all of our files together, we must use Linux’s make command in unison with a file which the user creates that must be called Makefile. A Makefile will contain a similar pattern to that of running gcc. In gcc, to compile multiple files, we could simply list each one. For example: gcc hellomake.c hellofunc2.c -o main -I.. In this case, we compile 3 files into one binary called main. The -I option stands for link. Certain libraries must be linked in order for the executable to be made. For example, a really good library that provides a sort of API for terminal-based GUI applications is ncurses, which requires the ncurses library to be linked first. In our case we use .. to represent our parent directory, which tells the compiler that we are using the parent directory as our library. This can be rewritten inside the makefile as the following:

main: hellomake.c hellofunc.c

gcc hellomake.c hellofunc.c -o main -I.

The make command will scan the first line and understand that any changes to hellomake.c or hellofunc.c will mean that main must be recompiled. Notice the tab on the second line where we tell the make command what to do when it recompiles. This tab in necessary simply due to how the make command functions (there must be a tab before each command). In this build, only main is recompiled if changes in hellomake.c or hellofunc.c are detected.

CC=gcc

CFLAGS=-I.

main: hellomake.o hellofunc.o

$(CC) hellomake.o hellofunc.o -o main

In this version, we make a couple adjustments. The CC variable is recognized by make as the preferred C compiler (gcc). CFLAGS is the variable used to pass arguments to the compilation. We also use .o instead of .c on all of our c files to indicate that they should be recompiled before main. There is still some things missing, thus let us continue our adventure of building makefiles.

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

main: hellomake.o hellofunc.o

$(CC) hellomake.o hellofunc.o -o main

DEPS is a variable used for dependencies ie. files which other files rely upon to function eg. any header files. We then ask make to create an object (.o) file for each .c file. The first line states that any .o files will rely on their .c counterparts, as well as any files listed in DEPS (in our case, only hellomake.h). This is indicated using the colon (:) to state that all .o files (%.o) rely on all .c files and any dependencies (: %.c $(DEPS)). The second line is pretty confusing at first glance but it’s important to remember that we are using the gcc command. We call gcc using the -c option which means create an individual object file. It is important to understand the bigger picture of what we are doing. We are compiling each .c file individually so that we can then compile each .o file into one executable. The output name for each .c file will be $@ which is the name of the file on the left of the colon separator (in our case %.o). $< represents the first file on the right side of the colon (%.c) and $(CFLAGS) is just the -I option. This can then be read as: gcc –create\_object\_file (-c) for $< (%.c) with\_output\_name (-o) $@ (%.o) using current directory ($(CFLAGS)) as the library. To recap, the first line states that all .o file rely on their .c counterparts and any dependencies ie. they must be compiled first. The second line gives the structure for the compilation of each .o file by calling gcc on the individual file, giving it an output name of $@ which is just equivalent %.o and the rest is self explanatory. The first command creates our object files and the second command creates our executable. Minor changes can rewrite this more neatly by using $^ which refers to the entirety of the right side of the colon ie. the opposite of $@. To reiterate. $@ = left of colon, $^ = right side of colon, $< = first element of $^

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

OBJ = hellomake.c hellofunc.c

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

main: $(OBJ)

$(CC) -o $@ $^ $(CFLAGS)

The final form of our makefile will describe how to create a full deliverable where code is placed in a src directory, header files are placed in an include directory, and libraries are included in a lib directory. We also want to remove the .o files which are created for each file since we don’t actually need them, gcc just requires that an object file is created when we compile. Here is the daunting result:

IDIR =../include

CC=gcc

CFLAGS=-I.

ODIR=obj

LDIR =../lib

LIBS=-l

\_DEPS = hellomake.h

DEPS = $(patsubst %,$(IDIR)/%, $(\_DEPS))

\_OBJ = hellomake.o hellofunc.o

OBJ = $(patsubst %, $(ODIR)/%, $(\_OBJ))

$(ODIR)/%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

main: $(OBJ)

$(CC) -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:

rm -f $(ODIR)/\*.o \*~ core $(INCDIR)/\*~

So now that you’re officially confused let me attempt to explain what’s going on here. All of our header files will be placed in our include folder which in our case is in the parent directory and assigned to a variable called IDIR (include directory). ODIR (object directory) will contain all of our object files ie. our .c and .o files. LDIR is the library directory and the LIBS variable contains the option -l which will search for the lib or library folder. patsubst is a command which follows the order $(patsubst pattern, replacement, text) where text is a given string and matching patterns from pattern are placed into replacement. Essentially, we are looking for any header files in the current directory that match our \_DEPS list and placing them into the include folder. The same is repeated for our object files into the obj folder. The .PHONY hidden file declares file names which do not actually exist. In other words, when we run make clean, we do not want make to create a file called clean, but rather to run the command associated with clean. In our case, clean will remove all object files from obj if we don’t want them. Finally, we use the command make with the -f option followed by our makefile and each of our files should be properly compiled into one executable: main.