C Basics

Here are some basics on C that will help as you begin to use it more frequently. You might not need to be able to recall all the information from memory, but it is important to remember that certain characteristics exist to better design solutions or understand bugs.

Data Types

C offers various data types for you to use, it is important to remember their size as well as how that size limits what they can represent (i.e. integers overflowing or signed vs unsigned) This site shows different types, size, and minimum/maximum in tables.

Something you may notice is that C doesn’t have a boolean type, instead using 0 to represent false and 1 (or not 0) as true. PintOS *does* define a boolean type, allowing you to use the **true** or **false** keywords while working in it.

Operators

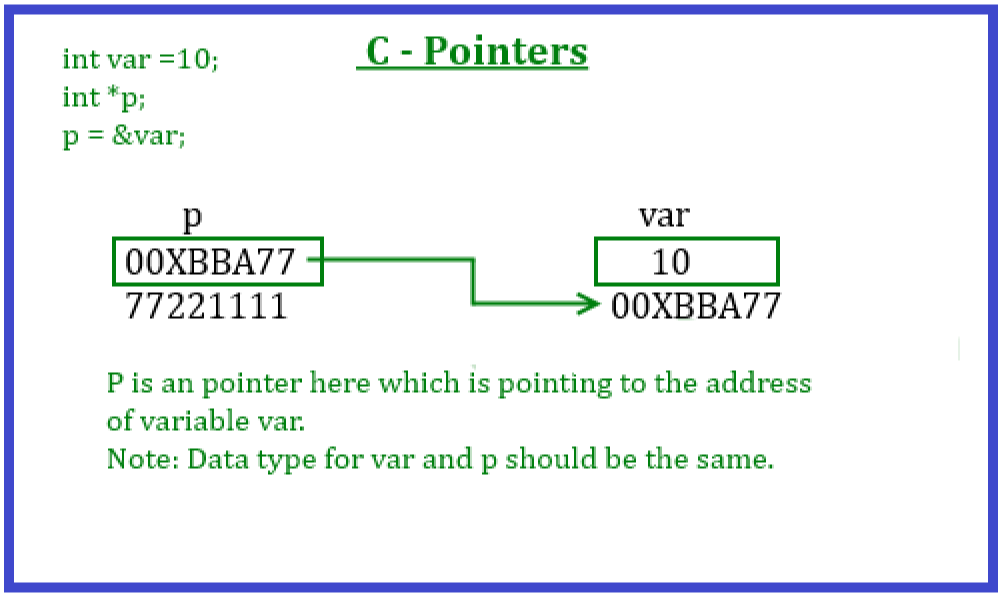
Operators are mostly the same as Java and many other programming languages. The assignment operator is still = and used to assign a value to a variable. The supported math operators are +, -, *, /, and %. +=, -=, *=, etc. are also allowed. Logical operators are also the same, with AND (&&), OR (||), and NOT (!). You can also make comparisons in the same way with >, <, >=, <=, ==, and !=.

C also allows you to use bitwise operators that treat variables like a collection of bits. The bitwise operators are explained well with examples here.

Pointers

One of the most challenging parts of C is getting accustomed to working with pointers. Remember that, behind a few levels of abstraction, variables are just values stored in a known spot in memory and that spots in memory are usually denoted by some long hexadecimal value.

Simply put, a pointer is just a variable that holds the address of another variable. Defining a pointer is done by declaring what data type the variable you want to point to is and then naming the variable, with the name starting with *.


C - Pointers
int var =10;
int *p;
p = &var;
p
00XBBA77
77221111
var
10
00XBBA77
P is an pointer here which is pointing to the address of variable var.
Note: Data type for var and p should be the same.

Note: Although it is syntactically correct to just define the pointer as “int *val;”, **you should always initialize your pointers to null.** It is good practice and may save you from random memory bugs in your projects.

When working with pointers, two important symbols are * and &. The asterisk (aka *) dereferences a pointer, which means that it reads the data at the memory address specified in the pointer (i.e. it will give you the actual data). The ampersand (aka &) gives the address of a variable (pointer or not). With these two together, you can effectively store and retrieve data in pointers.

When using pointers, **always check to ensure that they are not null before using them.** Dereferencing a null pointer will cause a segmentation fault and cause the program to stop (plus they’re just annoying).

For some simple examples on defining pointers and seeing how * and & affect what is returned, check out this site.

Pointer Arithmetic

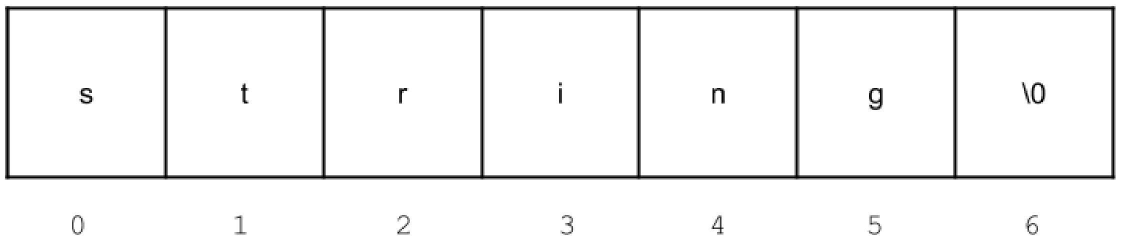
Pointer arithmetic lets you add or subtract to/from a pointer to either move it to the next or previous memory location. This is done simply by using ++ or -- on a pointer variable (without dereferencing).

This can be done once or iteratively and, among other things, can be used to iterate over arrays (because arrays are stored sequentially in memory) and strings (because C does not have objects, so strings in C are really just a pointer to an array of chars).

An important note with pointer arithmetic is that ++ will increment by the size of the data type that the pointer points to. So if it is a char pointer, ++ will increment by one byte, but if it is an int pointer, ++ will increment by 4 bytes.

Strings

Strings in C are actually just pointers to arrays of chars. They end with null terminators (the value 0) in order to signal that the string has ended. This means that there are several considerations to make when working with strings. We can use this example to make it easier to visualize.



This string is initialized using char *str = “string”;. The length of the string is 6, as we expect, but the actual length of the actual array is 7 (indexes 0 to 6).

This means that we cannot == strings, nor can we use string.equals(). To verify string equality, you have to iterate over the two strings you want to compare and compare char by char until the null-terminator.

You’ll notice that iteration is required when it comes to working with strings (comparing them, copying them, finding the length), but fortunately the string library (string.h) defines functions to do most of these tasks for you. Check this site for some of the most useful string.h functions, but feel free to find the rest of the functions and use the man pages to learn more about string manipulation.

Structs

C does not have objects, but allows you to create structs, which act similar to objects in the sense that they define a container to hold a certain set of variables in a certain order. In order to be as space efficient as possible, its best to define long variables first and short variables last.

There are two ways to access the variables within a struct, depending on if you have the struct itself or a pointer to the struct. If you have the actual struct, you get a certain variable using struct_name.variable_name (a dot operator). If you have a pointer to the struct, you get variables using struct_name->variable_name (the arrow operator).

For more information on structs, including how to define structs, check out this site.

Dynamic Memory

Declaring an array is easy when you know the exact size of it, however, if you don't know the exact size and need to define the size at runtime, you will have to allocate memory dynamically. This is done with malloc() or calloc() (their function is similar, except that malloc does not initialize the allocated range). Malloc and calloc both return pointers to the beginning of the allocated space, meaning that you will need a pointer to store the starting address.

C gives you the freedom to allocate memory as you would like, however this also means that it does not have a garbage collector to deallocate the memory when it is no longer in use. To avoid memory leaks, you as the programmer are required to free allocated memory using free().

Allocating space and never freeing it results in a memory leak, which is undesirable. It can be difficult to manage allocated space, so unless it is absolutely necessary, it's advisable to avoid using malloc or calloc and just store variables on the stack.

You do know how to store variables on the stack! Remember that a simple variable assignment like **int val = 5;** will store variables onto the program stack. This is opposed to using malloc/calloc, which will give you an address from the heap.

Conditionals

Conditionals in C use the same syntax as conditionals in Java, make sure to utilize the correct one in different scenarios (e.g. for loop vs while loop, switch statement vs if/else chain, etc.).

Functions and Include Statements

C allows you to write your own functions and use them in any order, so long as the declaration comes before it is first used. Functions have two parts to them, the function declaration and the function definition. The function declaration is one line that has the return type, function name, and type of each parameter. The function definition specifies what that function does. It has the return type, function name (must match), parameter types and names (order of data types must match), and function body (the actual code) to specify what the function will do.

Function declarations can either be done at the top of the .c file, after the include statements and before the code, or separately in a .h file, a header file. If you choose to define the function separately in a header file, the file must be included (using an include statement) at the top of the .c file.

Header files are not only used for function declarations, they are also used to store constants and system-wide global variables, among other things. Check out this page for more on header files and this page for more on functions in C.

`argv` and `argc`

The special function `main()` is the entry point of a C program. This function will most often use the predefined parameters `int argc()` and `char *argv[]`.

You may recall that Java's main method signature looks like this:
`public static void main(String[] args)`
Where `args` is an array of Strings containing the arguments passed to the program.

Similarly, `char *argv[]` or `char **argv`, which stands for "argument vector", is an array storing strings representing the arguments passed to the program from the host environment. Most often, this will mean the command line arguments you specify when you run the program.

For example, when you run `fib` in the command line using the arguments:

`fib 5 extra_args even_more_extra_args`

Then `argv` will be an array of char pointers that, when read as strings, *look* like:
`[“fib”, “5”, “extra_args”, “even_more_extra_args”]`
(By convention, the first argument is the name of the program)

These arguments can be accessed using `argv[0]`, `argv[1]`, and so on.

But what is `argc` for? Remember that unlike Java's `String[] args` parameter, C's arrays are not objects. Thus, C arrays do not have fancy features like a `length` property or out-of-bounds checking. Instead, we simply have an `argc` parameter, which stands for "argument count", that represents the number of arguments in the array pointed to by `argv`.

So, the example we have above will have `argc` be equal to `4`, and the last element can be accessed using `argv[3]`. We should not access the array beyond the last element, or we may have unexpected behavior.

Created by Edén Garza, August 2020, Updated September 2022 (argv, argc).