System-level Programming

Avery Karlin

Fall 2015

Contents

Teacher: Dyrland-Weaver

1 Learning C

1.1 C Primitive Variable Types

1. All C primitives are numeric, divided purely based on variable size, and integer or floating point
   (a) C variables have sizes based on the platform they were compiled by and for, such that sizeof(*type*) can be used to determine the size in bytes
   (b) On a standard computer, int = 4 ($-2^31, 2^31 - 1$), short = 2, long = 8, float = 4, double = 8, and char = 1 bytes (8 bits to a byte)
   (c) Types can also be specified as unsigned, such that it is not able to be given a negative value
   (d) Types can be placed within types of larger size and the same format without any form of conversion, but not of a different format
   (e) sizeof(*type*) returns the size in bytes of the type
2. Boolean values are numbers, such that 0 is false, and all nonzero numbers are considered true
3. Character literals can be represented inside single quotes rather than use a number, and Strings, though not an object, can use a double quotes literal
   (a) Strings are created by character arrays, using a null character (value 0), to show the end of the array, allowing it to be modified easier
4. Variables are able to be initialized within a for loop, but are not able to be declared, such that it must be before the loop

1.2 C Programming

1. All C programs are made up of a series of functions, run within the main function, which returns an integer (typically 0, or other values for errors)
   (a) They are compiled through "gcc *file.c* -o *program_name*", then run through "./*program_name*"
   (b) C programs can be compiled into one executable, such that there is just one file after another in command line, though since they are compiled in the same namespace, they cannot have the same global variables or function names
   (c) The same is true for files called within other files, though those are compiled automatically
   (d) C programs must have exactly 1 main() function to compile, creating issues when multiple files have a main()
   (e) "gcc -o *file.c*" creates a binary object .o file, which can be run through other exectuable files, and can be compounded similarly during compiling
      i. Binary object files are used for files without a main method, and are non-executable
      ii. Binary object files can be linked to a .c file through compound compiling with the first file as the .c file

iii. Thus, the .o file is dependant on all .h files and the .c file, while the executable is dependent on the .o files, .h files, and the main .c file

(f) If "-o *program_name*" is not put in, the default executable name is "a.out"

2. Libraries are added, either .h files from the current directory through #include "*file*.h" or through premade libraries by #include <file.h>

   (a) All files typically start with calling the C library with #include<stdio.c> (standard io) and <stdlib.h> (standard library)

3. The man pages, called by "man *command*" or "man *section command*", give information on both bash and C commands

   (a) (1) is user commands, (2) is system calls, (3) is library functions, such as the C libraries, (4) is devices, (5) is file formats, (6) is games and amusements, (7) is conventions and miscellany, and (8) is system admin and priveledged commands

   (b) (L) is used for local commands, installed by certain programs

4. C functions are pass by value, such that they put the value into a new variable created by the function, though if pointers are passed, it is equivelent to pass by reference, due to being a ppointer to the same location

   (a) C functions are written similar to java, with the exception of the lack of the protection

   (b) Due to C being functional, the functions are created in the order written, such that it should already have created all functions and commands used within the function being compiled

   (c) Failure to declare first leads to an implicit declaration warning that it has not been formally declared yet, though it will still work if it is declared later

   (d) Headers can also be placed at the top of the function in addition to where they are defined, to avoid implicit declaration, or in a seperate header file

1.3 C Structures

1. "printf(*text, var1, var2*)" is used to print a String in terminal, where the text is a formatted string, with placeholders for variables following

   (a) %f is a placeholder for a float, %d for double, %c for char, %s for string, %f for pointer, %lf for double, %ld for long, and %d for int

   (b) println can be used instead of printf for non-formatted strings (without variables)

   (c) Print functions do not automatically add "
   n" at the end of a line, and must be written in the string

2. Arrays in C are non-dynamic, such that they must have a fixed size, with no length function, and there are no errors for going outside boundries, rather going to a different point in memory

   (a) Arrays are declared by "*type*[*size*];" and must be initialized each part at a time

3. String functions are held within the string.h library, always assuming the strings are null-terminated

2 Memory Management

2.1 Memory Allocation

1. Memory allocation is either during compile time (static stack memory), or during runtime (dynamic heap memory)
2. Compiler allocated memory is packaged within the binary, unable to be overwritten due to protected memory, without a default value, where variables and arrays are allocated
   (a) Memory addresses of variables are fixed once they are placed, such that the data can be changed, but the location cannot be
   (b) String literals and variables marked by the "const" keyword have the value stored in stack memory as well
3. Runtime memory is temporary, used for values of variables
4. Systems have a bit limit which they can read at once, such that 32 bit systems are limited to 32 bit unsigned values, such that $[0, 2^32 - 1]$ is possible, or 4 GB
5. Pointers are variables designed to store memory addresses
   (a) *%variable* is used to get the address of a variable, such that the number returned can be the value of a pointer
   (b) When a pointer is incremented, the location moves the number of bytes of the variable type which the pointer applies to
   (c) * is used before a variable name to declare a pointer, and is also used when calling a variable to get the value of the item at that location, preceeding before numeric operators except ++ and –
   (d) Thus, for some array a, with *a as the pointer, a[i] = *(a + i)

2.2 Strings and Arrays

1. Strings can be declared by several methods, "char *str[byte_num]*" to do basic allocation, or it can be set on the same line, with a null put in the byte after the last letter
   (a) It can also be declared with an empty byte number, but set such that it will be given the exact amount of space needed
   (b) It can also be declared as a pointer to the array by "char **str = data*", created the array the exact correct size, and a pointer to the array under that variable name
   (c) After declaration, each character must be set individually, instead of using the equal sign
   (d) On the other hand, if a pointer is used, the pointer can be changed to apply to a seperate array, using an equal sign, even after declaration
2. The null character at the end is needed for string functions in string.h to work correctly, but is not a requirement
3. String/array variables are functionally immutable pointers to the first item in an array (such that the location cannot be changed)

4. Pointer-defined strings are literals, such that they are made in protected memory, where the pointer location can be redefined, but the string cannot be
    (a) Literals of the same string will point at the same location as previously made literals, stored fully in static stack memory

3 Structural Functions

3.1 String Functions

1. String functions are found within ¡string.h¿, assuming a null character at the end
2. int strlen(char *s) returns the length of s, ignoring the null character
3. int strcmp(char *s1, char *s2) returns 0 if equal, ¿0 if s1 ¿ s2, and ¡0 otherwise
4. char* strcpy (char *destination, char *source) copies the string to destination, assuming the allocated destination space is the same size or larger
5. char* strcat (char *destination, char *source) adds source to the end of destination
6. strncat and strncpy has an integer as a final parameter, using only the first n characters of the source string, such that if it is longer than the string, it uses up to the null character