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
5. "typedef *var-type new-name*" allows you to call an existing variable by a different name
   (a) Typedef is typically used within the header to allow the term to be used throughout the program

1.2 C Compilation

1. They are compiled through "gcc *file.c* -o *program_name*", then run through "./*program_name*"
2. 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
3. The same is true for files called within other files, though those are compiled automatically
4. C programs must have exactly 1 main() function to compile, creating issues when multiple files have a main()
5. "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
   (a) Binary object files are used for files without a main method, and are non-executable
   (b) Binary object files can be linked to a .c file through compound compiling with the first file as the .c file

(c) 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

(d) The .o file can then be compiled identically to .c files

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

7. Projects with large numbers of dependencies can have a file called "makefile" to organize creation of the executable with dependencies

   (a) The makefile is ordered from the most dependent level to least dependent (ignoring non-dependent files and standard libraries)

   (b) Format is:
   Target-Filename: dependency1 dependency2
   <tab> *Terminal compilation code for file*

   (c) "make *target-filename*" command is used when in the directory to compule the project using the make file

   (d) The clean targets can be used to remove files from the directory after compilation

   (e) The all target can be used to compile the entire program

   (f) Any bash commands can have a makefile target made, such that it can be used with a "run" target to run commands faster (though any target can be used)

   (g) The first target in a makefile is the default if just "make" is written in terminal

1.3 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) item 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)

2. 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

3. 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

1.4 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 by other programs 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) Variable names are not stored, but rather substituted for memory locations during compilation

   (c) Once the function/scope under which the stack memory created is finished, the memory is automatically released

3. 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 $[0, 2^3 2 - 1]$ is possible, or 4 GB

4. Pointers are variables designed to store memory addresses, stored within the stack

   (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)

(e) Null pointers (set equal to the location 0) are often used to signify the end of a list

5. Other sections in memory are code (the executable section of an object file, often stored in ROM until run), bss (global or static uninitialized variables), and data (global or static initialized variables)

6. Variables marked by the "const" keyword have the value stored in the read-only portion of the same memory space they would normally exist in

7. String literals are generally stored within the code section, though it can occasionally be stored in the data section as a static variable

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

2.3 Dynamic Memory Allocation

1. (*type* *)malloc(int *byte-num*) allocates that number of bytes from the head, returning the location of the first byte, typecasting the pointer to the type specified

   (a) sizeof(*type*) is often used to allocate the correct amount of memory

    (b) calloc is a similar function that sets each bit to 0, otherwise acting like malloc, though with a first parameter to determine the number of data pieces created
    (c) malloc returns a void pointer, such that it can be typecast to any type of pointer

2. realloc(void *p, int *new-byte-num*) will return any extra bytes, or add additional bytes to the allocation

3. Normal memory allocation happens on the stack, automatically released after the function which created it ends

4. Dynamic memory allocation happens on the heap, kept even after the function that created it is removed from the stack, such that it must be released

    (a) free(*pointer*) releases the dynamically allocated memory which the pointer goes to
    (b) Dynamic memory should always be released in the program when created, and can prevent filling the memory
    (c) After the program is ended, the memory is freed automatically, but it can freeze the computer if filled before then, such as in an infinite recursion

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

3.2 Struct

1. Structs are a collection of values within a single data type, declared by struct{*type1 var1; type2 var2*} as the type name

    (a) Typedef is used to create a simple type name for it
    (b) If the struct type itself is needed within the struct, it can be declared implicitly within the typedef, by putting "struct type-name var" instead of just the type

2. *struct-name.subvar* is used to call a specific item within the struct

3. Since . operator has precedence over *, pointers to structs either must get the struct data before getting a specific piece of data, or use "*pointer-to-struct* -¿ *struct-var* to do

4 File Programming

4.1 Permissions

1. There are 3 permission areas, each with their own permission value, first the creator (user), then a specific group of users (group), then everyone else (others), each mutually exclusive
   (a) The owner always has the ability to delete and change the permissions of a file, even if the permission value is 0
2. ABC is the permissions of an area in binary, where A = read, B = write, C = execute, such that it can be converted into a number 0-7 in octal (which must be written in code with a 0 first, to tell the compiler it is octal)

4.2 Bitwise Operators

1. & is the bitwise AND operator, — is OR, is NOT, and îs XOR
2. Bitwise operators are used to modify the actual binary of equal lengths, going bit by bit

4.3 File Usage

1. The file table is a list of all files used by a program while it it running, containing basic information such as location and size
   (a) The file table has a limited size of 2^n, typically 256, where getdtablesize() in “unistd.h” returns the size value;
   (b) Each file is given a descriptor, or an integer index from 0, and the table records the path, location, and other data
   (c) File descriptor 0 (or STDIN_FILENO) always refers to stdin (standard command line input), and 1 (or STDOUT_FILENO) refers to stdout
   (d) File descriptor 2 (or STDERR_FILENO) refers to stderr (standard error), which contains all error messages produced by the compiler, similar to stdout
   (e) These are automatically opened on the opening of a C program
2. open(*file_path, flags, mode*) - Opens/adds the file to the first open file table space, returning the file descriptor
   (a) File descriptor -1 is returned when the file opened does not exist, or there is not permission to access the file
   (b) When open() fails, the variable “errno” in “errno.h” is automatically set for the specific type of error
   (c) strerror(*int*) in “string.h” returns a string describing the integer stored in errno
   (d) Found within “fcntl.h”
   (e) Flags are used to declare what the file is being used for, while mode is only used if creating the file, setting the permissions as an octal number
   (f) Flags can be:
      - O_RDONLY - Read only

- O_WRONLY - Write only
- O_RDWR - Read and Write
- O_APPEND - Write to end of file only
- O_TRUNC - Erase file and write over
- O_CREAT - Create file and open it if it exists
- O_EXCL - Can be combined with O_CREAT to return an error if it exists

3. close(*file_descriptor*) removes the file from the file table, found within "unistd.h"

4. read(*file_descriptor, buffer, amount*) reads text from a file, where buffer is a pointer to where the text is placed, and amount is the bytes read

   (a) Returns the number of bytes read, or -1 if it fails (setting the errno value)

5. unmask(*mask*) sets the file creation permission mask, found within "sys/stat.h"

   (a) Files are not initially given the permissions in the mode argument when opened, when they are created, such that a mask must be applied to modify

   (b) The new permissions when opened are thus mask & mode, shutting off all permission from the mask

   (c) The mask should be in octal form, adding a new mask to the permissions

6. lseek(*file_descriptor, offset, whence*) sets the current position in an open file, where offset is the number of bytes to move by

   (a) Whence can either be SEEK_SET (beginning of the file), SEEK_CUR (current position in the file), or SEEK_END (end of the file)

   (b) Returns the number of bytes from the current position to the start of the file, or -1 in case of error, setting errno

   (c) Found within "unistd.h"