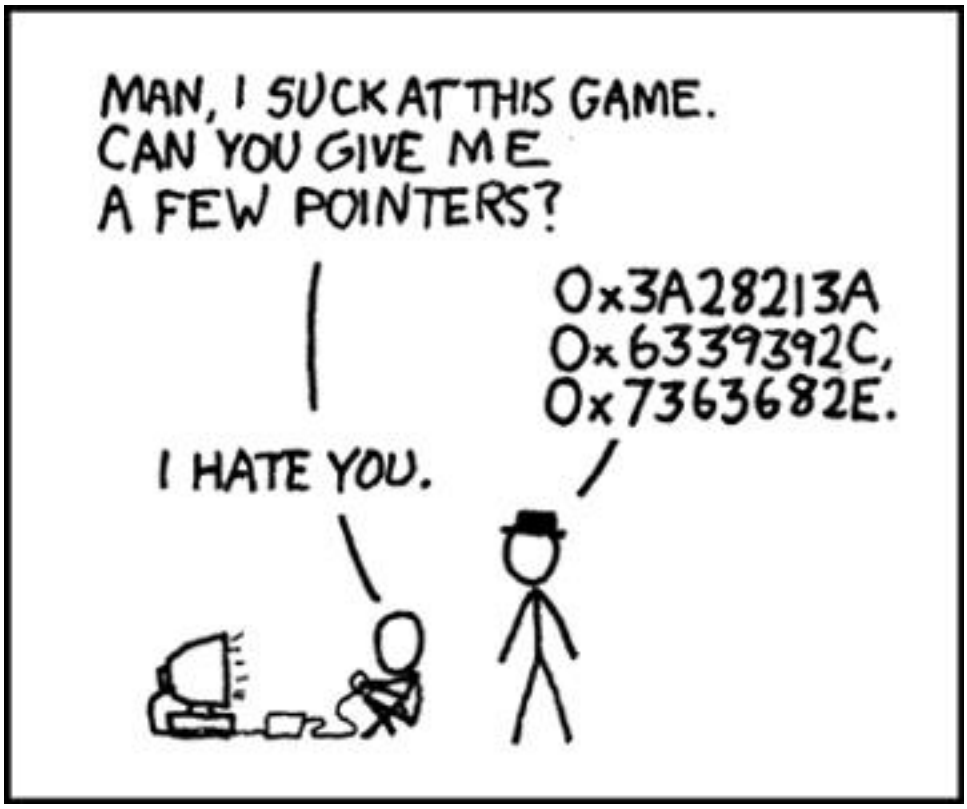
Using the C Language to Understand Memory

Final Report for CS2263: System Software Development, Fall 2020



https://xkcd.com/138/

Mahmoud Moustafa 3648276

This report is submitted in partial fulfillment of the requirements of CS2263
in the
Faculty of Computer Science
at the
University of New Brunswick

Introduction

This report will cover the data types we learned in this course, how much space they take up from the memory, and their functionalities. It will also cover process memory which is mainly how the memory allocated for a program will work upon its execution and will go through the anatomy of that memory. Then, it'll go through the stack memory and how variables and functions affect the stack memory and what happens to the stack upon different scenarios. It'll also mention what a stack contains and how elements in a stack are addressed and how might one misuse the stack memory. It'll then go through the heap memory and how we can manage and manipulate heap memory and how might one misuse the heap memory. It'll then go through and clarify recursion. After recursion, the report will go through structures and type definitions: what are they and how they can make the process of programming easier on us. Then, it'll go through files: what are they, how are data in files stored, the types of files, and the functionalities we have that allow us to manage and manipulate the files. After files, it will go into details about dealing with the text (ASCII) files, how we can manipulate them, and the functionalities that allow us to manipulate the data in those files. After the ASCII files, it'll go into the binary files, how to manage them, and the functionalities that we have that allow us to manipulate their data. After binary files, it'll go in deep details about abstract data types: what are they, how are they useful to programmers, the pitfalls that a programmer might fall into when dealing with them, the different types of ADTs and the functionalities of each type, and how each type of ADT is different from the others. After ADTs, it'll go through the pointers to functions: what are their uses, what is polymorphism in programming, and why is it useful to us. Finally, it will go through the tool chain that we learned in this course: which tools we learned to use, how and why are they useful, what is the function of each tool, and why should we use each tool.

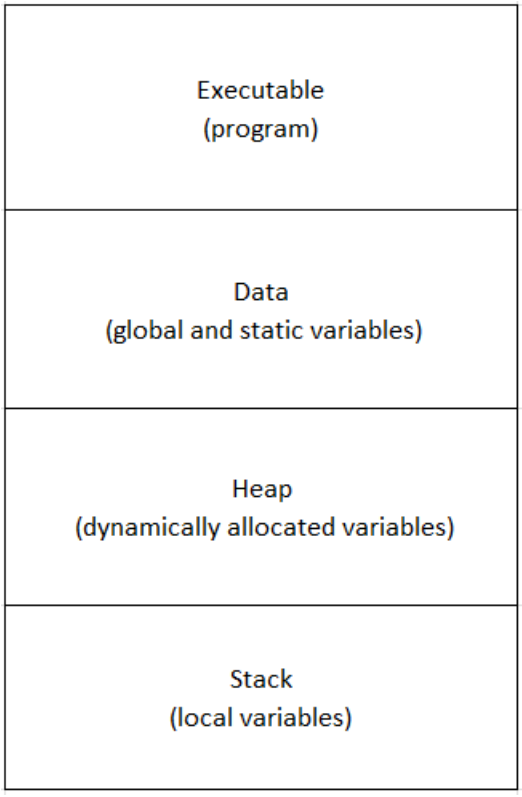
Data Types

When it comes to the fundamental data types of C, we have integers (int), characters (char), and floating-point values. Floating point values include doubles (double) and floats (float). Each of those fundamental data types has a size that it takes from the memory allocated to the program. The size of a char is 1 byte. The size of an int is 4 bytes. The size of a float is also 4 bytes. Finally, the size of a double is 8 bits. We also have pointers, arrays, and strings. Pointers basically are variables that store the addresses of another variables. They must be typed for the datatype they point to. They are used to access memory outside the current stack frame. In C, an array is a

collection of the same data types. Its elements usually take up contiguous memory. All its elements must be of the same type. The size of an array is the size of an element in the array times the number or elements it has. For instance, if it is an int array and it has 4 elements it will take 16 bytes. A string, fundamentally, is an array of chars that ends with a "NULL" (null-terminated string). C provides a lot of utilities to manage strings. For instance, when using the string library, one can copy a string to another using the strcpy function. One could also compare strings using the strcmp function. To get a substring, one could use the strncpy function.

Process Memory

The layout of a process memory has the text (executable program) on top. Underneath it exists the data which are (global and static variables) in the second layer. In the third layer, we have the heap memory (responsible for the dynamically allocated variables). Finally at the bottom layer we have the stack (local variables). This process memory takes up an assigned region of the RAM. The addressing of the process memory is top down, which means that they increase as we get closer to the bottom. This is a visual of the process memory.



Arguments in general are stored on the stack. They are in the stack frame of the functions they are arguments for. This applies to the arguments of main standard in, out, and error functions.

Stack Memory

A stack uses the LIFO principle (last in first out). A stack stores frames, symbols addresses, and values. The addresses of the elements in the stack increases as we go further down. When we want to add an element to a stack, we use the push function. When we want to remove and element, which will be the element at the top of the stack, we use the pop function. We can also look at the element at the top of the stack using the peek function. A stack frame is formed when a function is called. After a function is called, the return location is pushed in the frame of the called function. The return location is always the first thing pushed in the frame of a called function. If a function is called more than once, each call has its corresponding return location. If a variable is to store the return value of a called function, the variable would be given "garbage" value in the beginning. After the called function returns its value, the value of the variable will be modified from "garbage" to the return value. The address of such variable is called the value address. The value address is pushed in the stack frame of the called function directly above the return location. After the value address is pushed, if a called function has arguments, they are pushed in the stack frame of the called function. When pushing the arguments of a called function the function to be first pushed is the left-most argument then the next one and so on until all the arguments are pushed. After the arguments are pushed, the local variables of the called function are pushed. The local variable that was declared first will be pushed first and so on until all the local variables are pushed. Arrays are pushed on a stack element by element. The first pushed array element is the one at index 0, then the element at index 1, and so on until the array elements are all pushed. This scenario repeats with every function call.

Misuse of the stack occurs when someone tries to access a variable of a function that has been pushed. Another mistake is when trying to change the value of a variable using a function without storing the return value or passing in the variable address.

Below, in C, is an example of functions calling functions therefore adding frames to the stack. This example is from stack.c

```c
#include <stdlib.h>
#include <stdio.h>
int multiply(int a, int b);
int sum (int c, int d);
int main(){
    int x = 5;
    int y = 7;
    int multandsum = sum(x,y);
    return EXIT_SUCCESS;
}

int multiply(int a, int b){
    return a*b;
}
int sum(int c, int d){
    return (c+d) + multiply(c,d);
}

```

In this example the main class calls a function and the called function calls another function. The result of those functions is stored in the multandsum variable.

Heap Memory

Heap memory is a dynamically allocated memory. When using heap, memory is managed via malloc and free functions. Malloc function allocates memory. And the free function, well, frees the allocated memory. Unlike stacks, heap memory is managed by the programmer. When we malloc a memory we must free it, otherwise it is wasted memory. Heap memory management is done through pointers. Below is a simplified example of how a heap memory could be manipulated and managed from heap.c.

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    int* i = (int*) malloc(sizeof(int));
    *i = 14;
    printf("%d",*i);
    free(i);
    return EXIT_SUCCESS;
}
```

This is an example of manipulating heap memory by using malloc to allocate memory needed for the variable. Then we free the allocated memory to avoid memory leaks.

One of the common misuses of the heap memory is one that I am guilty of: forgetting to free the allocated memory after it has been used. Another common misuse of the heap memory is trying to access a variable (any value in a memory location) after it has been freed. Another misuse of the heap is when we have an array, and we move the head of the array. To avoid such a mistake, we should create a variable and make it equal to the head of the array and move that throughout the array to achieve our goal. A memory leak can be caused when using heap memory because of calling malloc more than once on the same variable without freeing it.

Recursion

Recursion is fundamentally used to address problems that can be broken down into simpler versions of themselves. In C, they are functions that call themselves. There are two cases in recursion: base case and general case. A base case is the case to which we have an answer. A general case is the case that expresses solution in terms of a call to itself with a smaller version of the problem. In recursion, every time a function calls itself a new stack frame is added on the stack. Since C is pass-by-value, arguments copies are used in each stack frame.

Below is an example of using recursion to find the factorial of a number from factorial.c.

```c
#include <stdio.h>
#include <stdlib.h>
int factorial(int n);
int main() {
    int n = 6;
    printf("%d", factorial(n));
    return 0;
}
int factorial(int n) {
    if (n >=1)
        return n*factorial(n-1);
    else
        return 1;
}
```

In this example the main function calls the factorial function which recurs itself until n = 1. When n = 1 each stack frame is popped, and the value of n is multiplied by the value from the popped frame. This will give us the factorial of the given number.

Structures and Type Definitions

A structure is a heterogeneous collection of data. Structures lay out on the stack just like arrays. We can access members of structures. A structure is used to group related data such as a bus-route name and a bus-stop number. The structures only hold values. The operations that we can perform on structures are copying, assigning, and member accessing. A pointer that points to a structure can be created on the heap memory. A type of definition is mainly used as a simple yet very much needed technique that reduces cognitive overload. It does this by giving us the ability to rename a type. Structures and type-definitions combined unlock a new and wide variety of possibilities for us as programmers because they mainly take away the complexity, allow us to program more efficiently, and more importantly make our code more readable and understandable.

Below is an example of using type-definitions and structures from struct.c

```c
#include <stdio.h>
#include <stdlib.h>
typedef char* String;
typedef struct car
{
    String name;
    int model
}Car;
int main(){
    Car* c;
    c = (Car*)malloc(sizeof(Car));
    c->name = "Lexus";
    c->model = 2022;
    free(c);
    return EXIT_SUCCESS;
}
```

In this example, we give a “nickname” to the char* to use instead of char* to make our lives easier. Then we create a car structure. This car structure has a name and model. In the main function, we declare and allocate the

memory needed for the Car pointer. Then we modify the name and model of the car. Finally we free the Car pointer.

A common misuse of typedef and structs is when trying to refer to data inside a struct with a "." instead of a "->". Another common misuse is forgetting to use "typedef" when defining a struct. If that happens, every time we call the struct, we will have to type struct before it.

Files

Files are arrays on an external device. There are two kinds of files. The first is a text / ASCII file and the second is a binary file. In an ASCII file, all data is stored as characters and / or numbers. In a binary file, data is stored in its native format (ASCII code [integers], 2's complement integers, and $IEEE_{754}$ floating point values). Files can be found on the disk or memory sticks. To open a file, we use the fopen function. The second argument of the fopen function is what gives us the ability to choose what we want to do with the data in the opened file. For instance, if we wanted to read a file we would pass in, as a second argument after the name of the file, "r" and if we wanted to write to a file, we would pass in "w". There are other functionalities of course. After we are done manipulating a file, we should close it with the fclose function. Compiling and running binary files are faster than compiling and running regular text files.

Text (ASCII)

When dealing with text files in C, we must open a file with the fopen function. The second argument gives us the ability to manipulate the file if we wanted. We can use "r" to read a file. In this case, it must exist. We can use "w" to write to a file. File does not need to exist. We can use "a" to append to a file. "r+" to update a file, a file must exist. "w+" to update a file, a file does not need to exist. "a+" to read and append. After opening a file, we can use multiple functions to manipulate it. We can use fprintf to print to a file. We can use fscanf to read from a file. we can use fputs to print a string to a file. fgets to read a string from a file. fputc to print a character to a file. And fgetc to read a character from a file. After manipulating the file, we should close it using the fclose function. This is an example of opening a file and reading from it, taken from openandread.c.

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen("openfile.txt", "r");
    int i;
    fscanf(f,"%d",&i);
    printf("%d",i);
    fclose(f);
}
```

In this file we open a file with the name openfile.txt to read. We then use the fscanf function to scan the data in that file. Then we print the data read from a file. Finally, we close the file.

Below is an example of opening a file and writing to it using fprintf from openandwritefile.c

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen("openwritefile.txt", "w");
    int i = 12345;
    fprintf(f,"%d",i);
    fclose(f);
}
```

In this example, we open a file to write data to it. We use the fprintf function to write the data to that file. Then we close that file.

Below is an example of writing a char to a file using fputc from writec.c

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen("writec.txt", "w");
    char c = 'c';
    fputc(c, f);
    fclose(f);
}
```

In this example we open a file to write a char to it. We use the fputc function to write the char to that file and we close that file.

Below is an example of reading a char from a file (writec.txt) using fgetc from read.c

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen("writec.txt", "r");
    char c = fgetc(f);
    fclose(f);
}
```

In this example, we open a file to read a char data type from it and store it in c. To read it, I used the fgetc function. I closed the file after I was done.

Below is an example of writing a string to a file from puts.c

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen("writes.txt", "w");
    char* s = "A7a";
    fputs(s,f);
    fclose(f);
}
```

In this example I opened a file to write a string (array of characters) to it. This “A7a” String is written in the file using fputs function. After I was done, I closed the file.

Below is an example of reading a string from a file (“writes.txt”) from gets.c

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen("writes.txt", "r");
    char s[4];
    fgets(s,10,f);
    printf("%s",s);
    fclose(f);
    return 0;
}
```

In this example I opened a file to read a String from it. I used the fgets function to achieve that goal. Then I printed the string that was read from the file. Finally, I closed the file after I was done.

Binary

When dealing with binary files in C, we will also use the fopen function to open a file, but we will add "b" to the second argument. For instance, if we want to write to a binary file we will use "wb" instead of just "w". When we are done manipulating a file as usual, we should use the fclose function. C provides a lot of functionalities to manage and manipulate binary files, for instance we have fread and fwrite. As the names suggest, we use fread when we want to read from a binary file and fwrite when we want to write to a file in binary. Below is an example of using fwrite to write data to a binary file from writebin.c.

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen ("writebin.txt","wb");
    int i = 5;
    fwrite(&i, sizeof(i), 1, f);
    fclose(f);
    return EXIT_SUCCESS;
}
```

In this example I opened a file to write to it in binary. I used the fwrite function to write to that file. I closed the file after writing to it.

Below is an example of reading from a binary file using fread from readbin.c

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    FILE* f = fopen ("writebin.txt","rb");
    int i;
    fread(&i,sizeof(i), 1, f);
    fclose(f);
    return EXIT_SUCCESS;
}
```

In this example I opened a file to read the binary data from it. I used the fread function to read the data in that file. The data read was stored in i. When I was done, I closed the file.

Abstract Data Types

An abstract data type (ADT) is a data structure that is defined by its behavior. ADTs can be reused in any number of programs. ADTs are great for creating templates for specific data structures and specific data types or templates that can be used repeatedly across different programs. In other words, they can be templates of templates. ADTs allow programs to become independent of the abstract data type's representation. Meaning the representation could be enhanced without breaking the entire program. This allows different parts of a program to become more independent from other parts. It also allows for more efficient implementation. We studied stack, queue, list, and binary tree ADTs. In list ADT we used create, read, update, and delete functionalities (CRUD). Create allowed us to create a pointer to the list. Read allowed us to process the list. Update allowed us to add and remove items. Finally, delete allowed us to remove items in the list and the list itself. For the stack ADT we have the read and update operations. We could only read / report the element on the top of the stack. Update allowed us to add an object by pushing it and to remove an object by popping it. Queues use FIFO principle (first in first out). For queue ADT, its read function allows us to report the element at the top of the queue. When updating, we can add an object via enqueue and remove an object via dequeue. We can neither create not delete. A binary tree ADT is like a linked-list whose elements have a parent-child relationship. They have Create, Read, Update, Delete functionalities. Meaning, we can create the binary tree list. We can read the items in the binary tree list. We can update it by adding and removing items. And we can delete the list. The order of the elements in the binary tree matters. Below is a header file for a stack ADT from stackADT.h.

```c
#ifndef STACK_H
#define STACK_H
typedef struct stack {
    int top;
    int capacity;
    int* array;
}Stack;
Stack *mallocStack();
void freeStack(Stack* stack);
void push(Stack* stack, int item);
int pop(Stack* stack);
int peek(Stack* stack);
#endif
```

In this example I used a stackADT to illustrate my point. I used a stack structure. I used the functions above because those are the functions that a stack is supposed to do. mallocStack and freeStack functions are supposed to allocate memory needed for the stack and free the memory that was taken by the stack, respectively. push and pop functions are the functions that are used to manipulate the data in the stack. push is used to add elements to the stack and pop is to remove elements from the stack. peek is used to "look" at the element at the top of the stack.

One very common pitfall is when we move the head node for any reason. That occurs a lot in list ADT, as you can manipulate whichever element you want which will cause memory leak. To avoid this, we should have another variable and make it equal to the head node. That pitfall most likely will not apply to the stack ADT since we can only manipulate the element at the top. Having the ability to manipulate only one element limits the pitfalls that a programmer might fall into when it comes to memory management.

Pointers to Functions

Pointers to function are used to store the memory address where a function starts executing from. Polymorphism is a technique that allows you to write completely general functions and then specialize them as needed. This technique is useful because we would not need to write similar things more than once. Instead, we write it as a general function and specialize as needed. For instance, we can create a car class and its type could be SUV, sedan, or coupe. Below is an example of this from pf.c

```c
#include <stdio.h>
#include <stdlib.h>
typedef char* String;
typedef String (*type)();
typedef struct car
{
    String name;
    type t;
}Car;
String SUV(){
    return "SUV\n";
}
String sedan(){
    return "Sedan\n";
}
String coupe(){
    return "Coupe\n";
}
int main(){
    Car* c = (Car*)malloc(sizeof(Car));
    c->name = "Rolls Royce Phantom";
    c->t = &sedan;
    printf("%s\t%s\n",c->name,c->t());
    Car* cc = (Car*)malloc(sizeof(Car));
    cc ->name = "Mercedes Benz G-Class";
    cc->t = &SUV;
    printf("%s\t%s\n",cc->name,cc->t());
    Car* ccc = (Car*)malloc(sizeof(Car));
    ccc->name = "BMW i8";
    ccc->t = &coupe;
    printf("%s\t%s\n",ccc->name,ccc->t());
    return EXIT_SUCCESS;
}
```

In this example we create 3 Car* elements and allocate the memory needed for each of them. We then add a name to each of those elements. Each one of them has a different type. Each type points to a different function.

Tool Chain

We used makefile, git, valgrind, and debugger. Makefile, even though annoying to make to me at least, saves a lot of time and effort. A makefile's

purpose is making the process of compiling programs easier. We can compile as many programs as we want using just one command: make.

The second tool we learned was git. Git is an open-source and free distributed version control system designed to handle projects of all sizes with speed and efficiency. We learned how to add, commit, and push a file to a repo. We also learned to clone a repo.

The third tool we learned is the debugger. It allowed us to go through the program step by step which was useful in figuring out where a bug was. It also allowed us to step into a function call or step over it.

The fourth tool is valgrind. It allowed us to check whether a program has memory leaks and if there were memory leaks how much and where. Knowing where the leaks were was crucial because it made it easier to go back and fix them. These memory leaks occurred because we malloced memory without freeing them.

Conclusion

In conclusion, we learned about the different data types we have in C, the functionalities of each data type, and the space each data type will take. Then we went through process memory: what it is and its anatomy. After process memory, we went through stack memory, what effects variables and function calls affect a stack and the addressing system of the elements in the stack. More importantly it went through mistakes that happen when using a stack memory. Then, we went through the heap memory: who manages the heap memory, how to manage it, and common mistakes that occur when using heap memory. Then, we went through recursion and clarified what recursion is and provided a different way to look at recursion with the hopes of making it easier to understand. Then, we learned what are structures and type definitions. We went through what they are and how they can make programming easier. Then we went through files. We discussed what they are, the different ways data is stored in a file, the different types of files, their functionalities that allow us to manage and manipulate the files. Then, we dove deeper in one kind of files (ASCII) files. We learned how we can manipulate and manage them and what functionalities allow us to do those actions. Then we dove deeper in the other kind of file: binary files. We learned how to manage them and the functionalities that we have that allow us to manipulate those files. Then we went through ADTs. We learned what they are, how can they be useful to programmers, the pitfalls that programmers need to watch out for when using ADTs, the different types of ADTs, the functionalities available for each type, and how are they different

from each other. Then, we learned about pointer to functions. We learned about their uses. We also learned about polymorphism in programming and why is it useful as a technique. Finally, we learned about a tool chain: makefile, debugger, git, and valgrind. We learned what is each tool of those, how and why they are useful. We also learned the function of each of those tools.

The Course: Closing Thoughts

I liked the forNextDays because they made it easier selecting my daily tasks that I had to for school. They also made it easier to stay on schedule with the lectures and go according to plan.

On the other hand, I would increase group work but not with a group project. Maybe a discussion every Friday.

Materials Consulted

I consulted mostly the lectures and the forNextDay exercises. I used labs 3 and 4.

But it would not be fair if I did not say I looked at the internet.

"Achieving polymorphism in C" codeproject,

https://www.codeproject.com/Articles/739687/Achieving-polymorphism-in-C

"Function Pointer in C", GeeksforGeeks,
https://www.geeksforgeeks.org/function-pointer-in-c/