

Using the C Language to Understand Memory

Final Report for CS2263: System Software Development, Fall 2020



<https://xkcd.com/138/>

Isaac Shoebottom (3429069)

*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 covers data, process memory and its intricacies from the stack to the heap, recursion, structs, typedefs, file handling, abstracting using function pointers, and the tools you need to make it all happen. I explain how memory is laid out, what type of data goes where, how to group data together and how you can turn it into your own custom type. I also explain how you can handle files and read them very easily with ASCII encoded files or straight raw binary. This report explains how C isn't so limiting in terms of abstraction after all. A language doesn't need real objects to have polymorphism after all.

Data Types

A list of the built-in data types and their sizes on a 64-bit processor:

char – 8 bits

short – 16 bits

int – 32 bits

long – 64 bits

float – 32 bits

double – 64 bits

string – $8n+1$ bits (where n is the number of characters in the string)

All data types except string also have a pointer counterpart, which acts as a store of the memory location where the data being pointed to is located. Strings do not have pointers because they are simply arrays of the char type whose last element is a null terminating character. All of the other data types can be stored in arrays, which is a section of contiguous memory where each element is accessed by getting the index element multiplied by the size of the data type in addition to the head element's location.

The language has also defined newer types that can be included, like the Boolean type, which provides additional value as to simply using int's as 1 and 0.

The C standard library also provides utilities to help construct and manage strings mainly in the header `string.h`, which provides methods relating to copying, finding chars in strings, concatenating, finding the length and providing macros for the null terminating character.

Process Memory

Process memory consists of executable code, data such as static and global variables, the heap, and the stack. Each name is fairly descriptive. The executable section stores the binary executable code from your program. The data section stores immutable data. The heap is the section of memory where dynamically allocated memory, such as variables that you wish to exist out of the scope they are defined in are stored here. Stack memory stores the variables that are declared in the stack you currently are executing in. When a program has just begun execution, the executable code is in the executable section, the immutable data is loaded into the data section and in the main stack frame there are the arguments of main starting from left most defined arguments to the right most defined arguments. The state of standard in, out and error are stored in the data region.

Stack Memory

Stack memory in C is handled by moving down from the stack. The first variable on the stack will be the highest position on memory. The subsequent values will be below them by the size of the value that was put on the stack. If entering a new function (stack frame), it is below the first one by the required values of the stack frame such as the return address and other such values. The argument variables are copied from the passed in values and any new values created in said function are added to the current stack frame below the copied argument variables. This behavior is demonstrated in the file stackmem.c.

```

1  #include <stdio.h>
2  void print(int i) {
3      printf( format: "The location of i: %p\n", &i);
4      int j;
5      printf( format: "The location of j: %p\n", &j);
6  }
7
8  int main() {
9      int i;
10     i = 100;
11     printf( format: "The first variable on the stack: %p\n", &i);
12     int *ptr;
13     {
14         int j;
15         ptr = &j;
16         j = 200;
17         printf( format: "The first location of j: %p\n", &j);
18     }
19     printf( format: "The value of j out of its scope: %d\n", *ptr);
20     int j;
21     printf( format: "The new location of j: %p\n", &j);
22
23     print(i);
24 }

```

Common misuse of the stack might be expecting variables assigned in new stack frame to persist after returning, when they cannot because they are a copied value passed in. Another might be expecting any variable created within a scope to exist outside of the scope without the use of pointers. Another might be expecting to be able to reference a value that is outside of a stack frame by using a pointer, this behavior is referenced in stackmem.c.

Heap Memory

Heap memory in C is handled by going up the heap memory, the opposite of the stack. Any subsequent values past the first ones allocated will be above the first value. If entering a new function, the stack frame does not matter. All heap values are stored through pointers, and you are responsible for keeping track of memory on the heap. The state of the heap exists regardless of where you are executing. This is all demonstrated in heapmem.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  void print(int *i) {
4      printf( format: "The location of i: %p\n", i);
5      int *k = (int *)malloc( Size: sizeof(int));
6      printf( format: "The location of k: %p\n", k);
7  }
8
9  int main() {
10     int *i = malloc( Size: sizeof(int));
11     *i = 100;
12     printf( format: "The first variable on the heap: %p\n", i);
13     int *ptr;
14     {
15         int *j = malloc( Size: sizeof(int));
16         ptr = j;
17         *j = 200;
18         printf( format: "The first location of j: %p\n", j);
19     }
20     printf( format: "The value of j out of its scope: %d\n", *ptr);
21     printf( format: "The new location of j: %p\n", ptr);
22
23     print(i);
24 }

```

Common mistakes when managing heap memory might be not allocating enough room for whatever you want to store. If you do not allocate enough room, it will be written past the boundaries and things will go wrong. Losing track of memory on the heap would also be a mistake, Reusing a pointer without freeing the memory being pointed to is the cause of a memory leak. Another error might be trying to read out of bounds of your allocated memory. You will likely get garbage back.

Recursion

Recursion works by inputting the value of the same function, executed previously into the same function. Recursive functions end when a base case is executed. This base case does not call the recursive function, thus going down the stack frame, taking the return value that is now valid and using it to finish the function call on each stack frame. Each call of a recursive function adds on top of the stack frame. In my recursion.c example program,

it calculates factorials by multiplying the last recursive call's result with a recursive call of the last result minus 1. For an example input of 3, The factorial function would be called, giving us 3 times factorial (2), which would then be an input of 2 times factorial (1), which would give us a new input of 1 times factorial(0). This ends the chain, which following it up the stack frame would look like $1 * 1 * 2 * 3$, with each returned value being multiplied by the next n above it in the stack frame.

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int n;
    n = 5;
    printf("format: %d! = %d\n", n, factorial(n));
    return 0;
}
```

Structures and Type Definitions

Structures are a way of grouping together previously disconnected data into a structure that allows for easy of coding and coding readability. The size of a struct is the size of all the data inside the struct. Typedefs are a way to define a custom type, an example of this being Boolean values in C, where the Boolean type is a wrapped way of saying 0 for false and 1 for true, but additional functionality and methods are included. The way we define structs and typedefs is by using those keywords and wrapping them in curly braces along with the data we want to include. Like any other type, they can be put on the stack or the heap, but each have their own rules. When on the stack the structure's elements must be accessed by a period and then the variable you wish to manage. If on the heap, each member of a structure must be accessed with the pointer operator (\rightarrow). If a member of a structure is also a pointer, you will need to allocate memory for said pointer as well. An example of all these rules can be seen in structs.c where a sample 3d point

object is used to show how data in structs/typedefs needs to be accessed regarding the stack and heap.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct point{
5      int x;
6      int y;
7      int z;
8  }Point3D;
9
10 typedef struct point2 {
11     int *points;
12 }Point3DArray;
13
14 void printPoint(Point3D p){
15     printf( format: "(%d, %d, %d)\n", p.x, p.y, p.z);
16 }
17
18 void printPoint2(Point3DArray p){
19     printf( format: "(%d, %d, %d)\n", p.points[0], p.points[1], p.points[2]);
20 }
21
22 int main() {
23     Point3D p1;
24     p1.x = 1;
25     p1.y = 2;
26     p1.z = 3;
27     printPoint( p: p1);
28
29     Point3D *p2 = malloc( Size: sizeof(Point3D));
30     p2->x = 4;
31     p2->y = 5;
32     p2->z = 6;
33     printPoint( p: *p2);
34
35     Point3DArray *p3 = malloc( Size: sizeof(Point3DArray));
36     p3->points = malloc( Size: sizeof(Point3D) * 3);
37     p3->points[0] = 7;
38     p3->points[1] = 8;
39     p3->points[2] = 9;
40     printPoint2( p: *p3);
41
42     return 0;
43 }
```

Common misuse might be using the arrow or dot operator when pointing to a struct from the wrong section of memory. Another common misuse might be not initializing members of a structure that are pointers. Another common misuse might be the confusion of structs and typedefs, where when you have a bare struct, you cannot create one with just the struct name.

Files

A file is block of data encoded in a specific format. All files are stored as binary, but are usually encoded in a specific format, such as text files being ascii encoded, printable documents being encoded as a PDF, zip files being stored in the PKZIP format, and windows binaries stored in PE format, and Linux executables being stored in the ELF format. Files are loaded into programs often by representing them as an array of bytes originating at the files header and ending at the file's footer terminated by an EOF character. Files are handled by the operating system, and the files themselves are represented by a file system, common examples being FAT and NTFS. This file system is handled by the operating system in such a way that when programs ask for a file, the operating system hands a response back after handling the file system. Therefore you need not know complicated things like the hex address of a file on a disk, the operating system acts as a translation layer between your program and the underlying hardware of the system.

Text (ASCII)

C provides plenty of functions to manage dealing with ascii encoded text files. The `stdio.h` header provides lots of functions for reading/interpreting a file as ascii encoded text. Functions like `getc`, `fputc`, `fprintf`, `fscanf`, `fputs`, `getw`, `putw` etc., All may be used to interact with a file that is being interpreted as text. This demonstration of this can be seen in `iotext.c`, where a file called `iotext.txt` is opened, erased, a string is written to the file, the file pointer reset to the beginning of the file, the file's contents are then read to the console screen, and the file is closed.


```

4   #include <stdio.h>
5   int main() {
6       FILE *fp;
7       fp = fopen( Filename: "iotext.txt", Mode: "w");
8       if (fp == NULL) {
9           printf( format: "Error opening file\n");
10          return 1;
11      }
12      fprintf( stream: fp, format: "CS2263!\n");
13      fclose( File: fp);
14      fp = fopen( Filename: "iotext.txt", Mode: "r");
15      char c;
16      do {
17          c = getc( File: fp);
18          putchar( Ch: c);
19      } while (c != EOF);
20      fclose( File: fp);
21      return 0;
22  }

```

Binary

The same functionality as interpreting ascii encoded files can be used for binary files as well, you just need to understand the files to be encoded as binary and not ascii. There are functions to manage binary files by manually using the file pointer to access locations of a file, and you can use scanf to interpret sections of a file as a specific format. Opening a file as binary is almost the same except the write/read mode specifies b after the r/w, and the functions fread and fwrite are used. The example I provided demonstrates the way C can open a file as binary, write a number directly to the byte stream, reset the file pointer and read said binary and output said binary as ascii characters to the console.

```

1  #include <stdio.h>
2  int main(){
3      FILE *fp;
4      fp = fopen( Filename: "iobinary.txt", Mode: "wb");
5      if(fp == NULL){
6          printf( format: "Error opening file\n");
7          return 1;
8      }
9      int num = 0x12345678;
10     fwrite( Str: &num, Size: sizeof(int), Count: 1, File: fp);
11     fclose( File: fp);
12     fp = fopen( Filename: "iobinary.txt", Mode: "rb");
13     int num2;
14     fread( DstBuf: &num2, ElementSize: sizeof(int), Count: 1, File: fp);
15     printf( format: "%d\n", num2);
16     fclose( File: fp);
17     return 0;
18 }
19

```

Abstract Data Types

An abstract data type is a way of representing data that represents the purpose it was built for. In a system that is designed for a restaurant's incoming orders, you can use an array, but you could use a queue to represent the flow of data your code intends. It makes your code more readable by abstracting the data to fit the purpose of the program. Other kind of abstract data types we have discussed would be Lists, Stacks, Queues, Binary Trees, and Directed Graphs. In my code example I've chosen to represent a stack in stack.h and its usage in stack.c.

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  typedef struct stack {
5      int *array;
6      int total_size;
7      int size;
8      int top;
9  } Stack;
10
11 struct stack *createStack(int size) {
12     struct stack *s = malloc( Size: sizeof(struct stack));
13     if (s == NULL) {
14         printf( format: "Error: malloc failed\n");
15         return NULL;
16     }
17     s->array = malloc( Size: sizeof(int) * size);
18     if (s->array == NULL) {
19         printf( format: "Error: malloc failed\n");
20         return NULL;
21     }
22     s->total_size = size;
23     s->size = 0;
24     s->top = -1;
25     return s;
26 }
27
28 void deleteStack(struct stack *s) {
29     free( Memory: s->array);
30     free( Memory: s);
31 }

```

```

33 void printStack(struct stack *s) {
34     int i;
35     for (i = 0; i < s->size; i++) {
36         printf( format: "%d ", s->array[i]);
37     }
38     printf( format: "\n");
39 }
40
41 void push(struct stack *s, int item) {
42     if (s->size == s->total_size - 1) {
43         printf( format: "Stack is full\n");
44         return;
45     }
46     s->array[s->size] = item;
47     s->top = item;
48     s->size++;
49 }
50 void pop(struct stack *s) {
51     if (s->top == -1) {
52         printf( format: "Stack is empty\n");
53         return;
54     }
55     s->size--;
56     s->top = s->array[s->size];
57 }
58 int search(struct stack *s, int item) {
59     int i;
60     for (i = 0; i < s->total_size; i++) {
61         if (s->array[i] == item) {
62             return s->array[i];
63         }
64     }
65     return -1;
66 }

```

```

67 int peek(struct stack *s) {
68     if (s->top == -1) {
69         printf( format: "Stack is empty\n");
70         return -1;
71     }
72     return s->top;
73 }

```

```

1  #include "stack.h"
2  int main() {
3      Stack *stack = createStack( size: 8);
4      if (stack == NULL) {
5          printf( format: "Stack creation failed\n");
6          return 1;
7      }
8      int i = 7;
9      int j = 8;
10     push( s: stack, item: i);
11     push( s: stack, item: j);
12     printf( format: "%d\n", peek( s: stack));
13     printf( format: "%d\n", search( s: stack, item: i));
14     pop( s: stack);
15     int k = 9;
16     push( s: stack, item: k);
17     printStack( s: stack);
18     deleteStack( s: stack);
19     return 0;
20 }
21

```

In my stack program, when creating a stack, an array is created which holds the stack. The total size, the current size of the stack, and the top element are also stored. Methods control adding to the stack, and when adding, it is placed in the array at the "top" and overwrites the stored top value. When searching for a value it runs down the array from top to bottom and returns it if found. Popping a value removes it from the stack and sets the top value to be the value beneath it. Printing the stack runs over every value and prints the top to the bottom on the screen in order of left to right. Peek returns the top value. The stack.c program all of this functionality is demonstrated.

When managing memory with abstract data types, you need to be careful to allocate enough memory for the intended use of the data type, so you often need a size parameter, and when using an array (or pointer) you need to make sure to malloc enough memory using this size parameter, and when freeing the data type you need to make sure to follow all the pointers/arrays in your data type and free them individually. Freeing a struct does not mean you freed all the memory inside said struct.

Pointers to Functions

Polymorphism is the process of using an interface to ease the extendibility of code. This can be achieved in C by using function pointers. We can take a typedef struct and include a function pointer, which we can then make functions to create extensions of this typedef by overwriting said function pointer with a function of our own choosing. I demonstrate this in my C example by making a Shape interface which can be overwritten with examples such as a triangle or square, which when we call the area function on said extensions of Shape, returns a different value even with the same input.

```

1  #ifndef shape_h
2  #define shape_h
3
4  typedef struct{
5      int numSides;
6      double perimeter;
7      double (*area) ();
8  } Shape;
9
10 #endif

```

```

1  #ifndef square_h
2  #define square_h
3
4  #include ...
5
6
7  double squareArea(Shape* square) {
8      double side = square->perimeter / 4;
9      double area = side * side;
10     return area;
11 }
12
13 Shape* newSquare(double perimeter) {
14     Shape* square = malloc( Size: sizeof(Shape));
15     square->numSides = 3;
16     square->perimeter = perimeter;
17     square->area = squareArea;
18     return square;
19 }
20
21 #endif

```

```

1  #ifndef triangle_h
2  #define triangle_h
3
4
5  #include ...
6
7
8
9  double triangleArea(Shape* triangle) {
10     double side = triangle->perimeter / 3;
11     double area = (sqrt(3) / 4) * (side * side);
12     return area;
13 }
14
15 Shape* newTriangle(double perimeter) {
16     Shape* triangle = malloc(sizeof(Shape));
17     triangle->numSides = 4;
18     triangle->perimeter = perimeter;
19     triangle->area = triangleArea;
20     return triangle;
21 }
22
23 #endif

```

```

1  #include ...
2
3
4
5
6  int main(){
7     Shape *triangle = newTriangle(perimeter: 20);
8     Shape *square = newSquare(perimeter: 20);
9     printf(format: "Triangle area of perimeter 20: %f\n", triangle->area(triangle));
10    printf(format: "Square area of perimeter 20: %f\n", square->area(square));
11    return 0;
12 }

```

Tool Chain

Various tools used throughout the course would be, gcc, gdb, ddd, make, and valgrind. The purpose of GCC is to translate our code into assembly, which is then linked into an executable. GDB is used as our debugger to place breakpoints to then inspect memory, or to follow the stack trace. DDD is a nice graphical interface for GDB that makes it easier to interact with. Make is a build automation tool that you can describe commands which invoke GCC on your source files, or other command such as a clean command that will remove your object/executables and allow you to rebuild from a clean slate.

It can also be used to provide testing automation. Valgrind is a utility primarily used to help identify problems in memory, such as writing to freed memory, writing past the boundary of allocated memory and memory leaks that can occur from improperly freeing memory.

Conclusion

In summary, the data types, the recursion, the structs, the heap and stack, the files in the operating system and the tools that make it all happen are all C, C is the language definition, the code, it has the data types, and the recursion, but C is also about handling the memory, allocation and freeing, C is also using the tools to build it, to debug it, and to make sure you aren't leaking memory. C is all of these things, and it endures to this day as the mother of most modern syntax and language design.

The Course: Closing Thoughts

Answer at least one of the following two questions (you are welcome to answer both if you would like to) to help me improve the course:

- What was the most interesting thing you learned in this course and why was it so interesting? Pointers were the most interesting thing. The syntax was tricky to figure out but rewarding.
- What do you wish you had learned more about in this course and why? I wish I learned more about safety, I think it would have been cool learning about safer functions like these:
<https://www.youtube.com/watch?v=B9DouAlkZlc>

Materials Consulted

It would be too long of a list to include every resource, assume I used every slide and lecture at: <https://lms.unb.ca/d2l/le/content/195036/Home>