

Sanad Alwerfali

201857760

Lab 1 - C Programming

Jan 20th, 2024

Table of Content

Symbols and Polymorphism.....	3
Part 1.....	3
Part 2.....	4
Part 3.....	5
Part 4.....	6
C-Style Strings.....	7
Part 1.....	7
Part 2.....	8
Part 3.....	9
I/O with libs.....	11
Part 1.....	11
Part 2.....	11
Part 3.....	12
Memory Management.....	13
Part 1.....	13
Part 2.....	13
Part 3.....	14
Part 4.....	15
Part 5.....	15
Part 6.....	16
Macros.....	17
Part 1.....	17
Part 2.....	17
Part 3.....	18
Part 4.....	20

Symbols and Polymorphism

Part 1

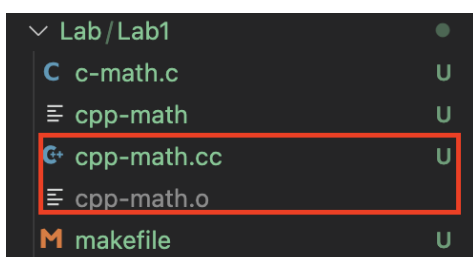
The first part of this lab involves evaluating polymorphism and function (as well as operator) overloading in both C and C++. To begin, we will create a C++ program that employs function overloading for an 'add()' method. This method will return either an integer or a float based on the parameters provided to the function:

```
Lab > Lab1 > G+ cpp-math.cc > ...
1  #include <iostream>
2
3  // PART 1: Symbols and Polymorphism
4
5  // functions declarations
6  int add(int, int);
7  float add(float, float);
8
9  int add(int a, int b) {
10     return a + b;
11 }
12
13 float add(float a, float b) {
14     return a + b;
15 }
```

We can then proceed by creating a *makefile* that allows us to compile the code above and review it using *nm*.

```
Lab > Lab1 > M makefile
1  cpp-math: cpp-math.cc
2      g++ -c cpp-math.cc -o cpp-math.o
3
```

Now we are able to run '*make cpp-math*' in the terminal and get an object file for our program.



```

v Lab/Lab1
C c-math.c
≡ cpp-math
G+ cpp-math.cc
≡ cpp-math.o
M makefile
```

This is the output when we run `nm --demangle cpp-math.o`

```
00000000000000020 T add(float, float)
00000000000000000 T add(int, int)
```

This output includes columns that represent various information about the symbols in the `cpp-math.o` file. The first column to the left shows us the memory address or value associated with the symbol. The next column is the type of the symbol, in our case it is T or Text Section Symbol. The next column shows the name of the symbol along with the parameters type (if the symbol is a function). Since our symbol is a function, the addresses on the left column represent the starting address of the functions code (i.e. 0X20 and 0X0 in hexadecimal)

Part 2

Now we can move on with creating a *header* file which we will include in our `cpp-math.cc` file:

```
Lab > Lab1 > C cpp-math.h > ...
1  extern "C" {
2
3  int add(int, int);
4
5  }
```

After recompiling and running `nm --demangle cpp-math.o`. We get the following output

```
00000000000000020 T add(float, float)
00000000000000000 T _add
```

As can be seen from the above output, the name of the symbol is different. That is because we are using `extern "C"` in our header file, which is used to specify that the function '`int add(int, int)`' should use *C-style linkage*. In our case, the difference in symbols name, shows us the impact of using `extern "C"` on the name mangling for C++ function '`add()`'.

Part 3

Now, let's try to do the same but in C. First let's save the code in a *c-math.c* file as shown below:

```
Lab > Lab1 > C c-math.c > ...
1  #include <stdio.h>
2
3  // PART 1: Symbols and Polymorphism
4
5  // functions declarations
6  int add(int, int);
7  float add(float, float);
8
9  int add(int a, int b) {
10 |     return a + b;
11 | }
12
13 float add(float a, float b) {
14 |     return a + b;
15 | }
```

As you can tell by the squiggly lines, there is an error. We also tried to compile the code and we get the following output:

```
c-math.c:7:7: error: conflicting types for 'add'
float add(float, float);
^
c-math.c:6:5: note: previous declaration is here
int add(int, int);
^
c-math.c:13:7: error: conflicting types for 'add'
float add(float a, float b) {
^
c-math.c:9:5: note: previous definition is here
int add(int a, int b) {
^
2 errors generated.
```

We can't compile our program because simply C doesn't have built-in support for Polymorphism nor function or operator overloading. Which creates a significant difference between C and C++.

Part 4

Now we can comment out the float version of the 'add()' method and we can compare the 2 object files using the 'colordiff' command and we got the following:

```
sanadalwerfali@Sanads-MacBook-Pro Lab1 % colordiff <(objdump -dS c-math.o | pygmentize -l c-objdump) <(objdump -dS cpp-math.o | pygmentize -l c-objdump)
1c1
< c-math.o:      file format mach-o 64-bit x86_64
-----
> cpp-math.o:   file format mach-o 64-bit x86_64
13a14,24
>      12: 66 66 66 66 66 2e 0f 1f 84 00 00 00 00      nopw    %cs:(%rax,%rax)
>
> 0000000000000020 <__Z3addff>:
>      20: 55                      pushq   %rbp
>      21: 48 89 e5                movq    %rsp, %rbp
>      24: f3 0f 11 45 fc          movss   %xmm0, -4(%rbp)
>      29: f3 0f 11 4d f8          movss   %xmm1, -8(%rbp)
>      2e: f3 0f 10 45 fc          movss   -4(%rbp), %xmm0      ## xmm0 = mem[0],zero,zero,zero
>      33: f3 0f 58 45 f8          addss   -8(%rbp), %xmm0
>      38: 5d                      popq    %rbp
>      39: c3                      retq
>
```

As we can see from the above screenshot, after comparing the implementation of the 'int add()' methods both in C and C++ object files, we noticed that the implementation is the same. Mainly because the 'add()' method is quite simple and doesn't use any features that are specific to either C or C++. It's a basic function that adds two integers, which is a feature common to both languages, hence the implementation in both object files is the same.

C-Style Strings

In this part in the lab we will examine strings in C.

Part 1

First we will write a program that prints out “*Operating systems rule*” to stdout and to achieve that we can do the following:

```
3  #include <unistd.h>
4
5  int main() {
6      const char *text = "Operating systems rule\n";
7
8      fwrite(text, sizeof(char), strlen(text), stdout);
9
10     return 0;
11 }
```

After compiling the program, it prints the following:

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % make c-string
gcc -g c-style-strings.c -o c-style-strings
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-style-strings
Operating systems rule
```

And we use the debugger tool ‘lldb’ to find the address of our string literal as such:

```
(lldb) break set -n main
Breakpoint 1: where = c-style-strings`main + 15 at c-style-strings.c:6:17, address = 0x00000000100003f3f
(lldb) run
Process 9103 launched: '/Users/sanadalwerfali/Desktop/W2024/OPSYS/Lab/Lab1/c-style-strings' (x86_64)
Process 9103 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  frame #0: 0x00000000100003f3f c-style-strings`main at c-style-strings.c:6:17
    3  #include <unistd.h>
    4
    5  int main() {
-> 6      const char *text = "Operating systems rule\n";
    7
    8      fwrite(text, sizeof(char), strlen(text), stdout);
    9
Target 0: (c-style-strings) stopped.
(lldb) print &text
(const char **) 0x00007ff7bfeff1d0
(lldb)
```

As can be seen from the above screenshot, the address is 0x00007ff7bfeff1d0

Part 2

Now we can change our code to add our initials to an char array and print them:

```
Lab > Lab1 > C c-style-strings.c > main()
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4
5  int main() {
6      const char *text = "Operating systems rule\n";
7
8      fwrite(text, sizeof(char), strlen(text), stdout);
9
10
11     char initials[] = "SA";
12
13     for (int i = 0; i < strlen(initials); ++i) {
14         printf("%c\n", initials[i]);
15     }
16
17
18     return 0;
19 }
```

We get the following output once the program is compiled and ran:

```
sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-style-strings
Operating systems rule
S
A
```

We can use the debugger to find the characters stored in that array by using 'lldb'. For this next step, we first need to figure out the address location where we are storing our initials, and we can do that using the following:

```
Current executable set to '/Users/sanadalwerfali/Desktop/W2024/OPSYS/Lab/Lab1/c-style-strings' (x86_64).
(lldb) breakpoint set --file c-style-strings.c --line 16
Breakpoint 1: where = c-style-strings`main + 167 at c-style-strings.c:21:1, address = 0x0000000100003f67
(lldb) run
Process 5363 launched: '/Users/sanadalwerfali/Desktop/W2024/OPSYS/Lab/Lab1/c-style-strings' (x86_64)
Operating systems rule
S
A
Process 5363 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  frame #0: 0x0000000100003f67 c-style-strings`main at c-style-strings.c:21:1
    18     return 0;
    19 }
Target 0: (c-style-strings) stopped.
(lldb) frame var -L initials
0x00007ff7bfeff1cd: (char[3]) initials = "SA"
```


We can then use the mem read command to find the characters stored in our initials variable:

```
(lldb) memory read -f c -c 2 0x00007ff7bfeff1cd  
0x7ff7bfeff1cd: SA
```

As we can see from the screenshot above, after reading the content of the address we find the initials we initialized the variable with.

Part 3

For this part we will use the 'strcpy()' method to copy our first name to the initials char array. We can do that using the following:

```
Lab > Lab1 > C c-style-strings.c > main()  
1  #include <stdio.h>  
2  #include <string.h>  
3  #include <unistd.h>  
4  
5  int main() {  
6      const char *text = "Operating systems rule\n";  
7  
8      fwrite(text, sizeof(char), strlen(text), stdout);  
9  
10  
11     char initials[] = "SA";  
12  
13     printf("The length of initials before strcpy is: %zu\n", strlen(initials));  
14  
15     strcpy(initials, "Sanad");  
16  
17     printf("The length of initials after strcpy is: %zu\n", strlen(initials));  
18  
19  
20     return 0;  
21 }
```

Which outputs the following:

```
Operating systems rule  
The length of initials before strcpy is: 2  
The length of initials after strcpy is: 5
```

We can then examine the number of bytes used by initials before and after we copied 'Sanad' to the initials array. We can do the following by using the GDB debugger as follows:

```
(gdb) break main
Breakpoint 1 at 0x11b6: file main.c, line 5.
(gdb) run
Starting program: /home/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main.c:5
5      int main() {
(gdb) next
6      const char *text = "Operating systems rule\n";
(gdb) next
Undefined command: "ext". Try "help".
(gdb) next
8      fwrite(text, sizeof(char), strlen(text), stdout);
(gdb) next
Operating systems rule
11     char initials[] = "SA";
(gdb) p sizeof(initials)
$1 = 3
(gdb) next
13     printf("The length of initials before strcpy is: %zu\n", strlen(initials));
(gdb) next
The length of initials before strcpy is: 2
15     strcpy(initials, "Sanad");
(gdb) next
17     printf("The length of initials after strcpy is: %zu\n", strlen(initials));
(gdb) p sizeof(initials)
$2 = 3
(gdb) p initials
$3 = "San"
```

As we can observe from the above screenshots, the 'strlen()' function returns the length of the string, which is the number of characters before the null character. However, the 'sizeof()' operator returns the amount of memory allocated for the 'initials' array, which is determined at compile time. In our code, we initialized our char array with enough space for the string "SA" and a null character. This means sizeof(initials) will return 3 as can be seen above. When we use 'strcpy()' to copy the string "Sanad" into initials, we are actually writing beyond the end of the array. The strlen(initials) after the strcpy operation returns the length of the string "Sanad", which is 5, but sizeof(initials) still returns 3 because the size of the initials array itself has not changed.

I/O with libs

For this part of the lab we will be using the formatted print methods to print in multiple ways.

Part 1

We will first print our favorite number using 'printf()' method as the following:

```
Lab > Lab1 > C c-io.c > ...
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <string.h>
4
5  int main() {
6      int fav = 7;
7
8      printf("My favorite integer is %d\n", fav);
9
10     return 0;
11 }
```

And we get the following output:

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % make c-io
gcc -g c-io.c -o c-io
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-io
My favorite integer is 7
```

Part 2

Now we move on to the next step, we modify the code as follows:

```
Lab > Lab1 > C c-io.c > main()
3  #include <string.h>
4
5  int main() {
6      int fav = 7;
7      float floatingPointNumber = 3.14;
8      int hexFormattedInteger = 255;
9      int *ptr_fav = &fav;
10
11     printf("My favorite integer is %d\n", fav);
12     printf("Floating-point number: %f\n", floatingPointNumber);
13     printf("Hex-formatted integer: %x\n", hexFormattedInteger);
14     printf("Pointer to favorite number: %p\n", ptr_fav);
15
16     return 0;
17 }
```

We get the following output:

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % make c-io
gcc -g c-io.c -o c-io
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-io
My favorite integer is 7
Floating-point number: 3.140000
Hex-formatted integer: ff
Pointer to favorite number: 0x7ff7b780c468
```

Part 3

Now to print the description using 'sprintf()' we can do the following:

```
Lab > Lab1 > C c-io.c > main()
5  int main() {
6      int fav = 7;
7      float floatingPointNumber = 3.14;
8      int hexFormattedInteger = 255;
9      int *ptr_fav = &fav;
10
11      char myName[] = "Sanad Alwerfali";
12      char description[100];
13
14      printf("My favorite integer is %d\n", fav);
15      printf("Floating-point number: %f\n", floatingPointNumber);
16      printf("Hex-formatted integer: %x\n", hexFormattedInteger);
17      printf("Pointer to favorite number: %p\n", ptr_fav);
18
19      sprintf(description, "My name is '%s', which is %zu characters long, and it is located at address %p\n",
20              myName, strlen(myName), (void *)myName);
21
22      fwrite(description, sizeof(char), strlen(description), stdout);
23      write(STDOUT_FILENO, description, strlen(description));
24      return 0;
25 }
```

We get the following output:

```
My favorite integer is 7
Floating-point number: 3.140000
Hex-formatted integer: ff
Pointer to favorite number: 0x7ffec79a3f0c
My name is 'Sanad Alwerfali', which is 15 characters long, and it is located at address 0x7ffec79a3f20
My name is 'Sanad Alwerfali', which is 15 characters long, and it is located at address 0x7ffec79a3f20
```

Memory Management

In this part of the lab we will look at how we can safely handle memory allocation and deallocation.

Part 1

We will start by allocating a memory space to store an integer as follows:

```
Lab > Lab1 > C c-memory.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main() {
6      int *one_integer = (int *)malloc(sizeof(int));
7      printf("Address of the allocated space is: %p\n", (void *)one_integer);
8
9
10     return 0;
11 }
```

We get the following output

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % gcc -o c-memory c-memory.c
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-memory
Address of the allocated space is: 0x7f7a5b705db0
```

Part 2

Then when we add the fsanitize flag we get the following:

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % gcc -o c-memory c-memory.c -fsanitize=address
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-memory
Address of the allocated space is: 0x6020000000b0
```

As we can see the address is different, mainly because the sanitize flag uses a different memory allocator that provides its own memory layout to make it easier to detect memory errors such as buffer overflows and use-after-free errors, which is why the addresses are different when you compile with and without the -fsanitize=address flag.

Part 3

For this part we will modify our code to allocate an array of size 1000 using malloc() and calloc() as shown here:

```
Lab > Lab1 > C c-memory.c > ...
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main() {
6      int *one_integer = (int *)malloc(sizeof(int));
7      printf("Address of the allocated space is: %p\n", (void *)one_integer);
8
9      int *arr_malloc = (int *)malloc(1000 * sizeof(int));
10     int *arr_calloc = (int *)calloc(1000, sizeof(int));
11
12     return 0;
13 }
```

Using the 'lldb' debugger we get the following for the arr_malloc:

```
(lldb) next
Address of the allocated space is: 0x7fad5a705db0
Process 7767 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0000000100003f3f c-memory`main at c-memory.c:9:30
    6      int *one_integer = (int *)malloc(sizeof(int));
    7      printf("Address of the allocated space is: %p\n", (void *)one_integer);
    8
->  9      int *arr_malloc = (int *)malloc(1000 * sizeof(int));
    10     int *arr_calloc = (int *)calloc(1000, sizeof(int));
    11
    12     return 0;
Target 0: (c-memory) stopped.
(lldb) memory read --format x --size 4 arr_malloc
0x7ff811435cac: 0x48d88948 0x5b08c483 0x4855c35d 0x8348e589
0x7ff811435cbc: 0x10b811fa 0x48000000 0x48d0420f 0x4811fe83
```

Using the debugger we get the following for the arr_calloc:

```
(lldb) next
Process 7767 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0000000100003f4d c-memory`main at c-memory.c:10:30
    7      printf("Address of the allocated space is: %p\n", (void *)one_integer);
    8
    9      int *arr_malloc = (int *)malloc(1000 * sizeof(int));
->  10     int *arr_calloc = (int *)calloc(1000, sizeof(int));
    11
    12     return 0;
    13 }
Target 0: (c-memory) stopped.
(lldb) memory read --format x --size 4 arr_calloc
0x7ff7bfeff1f0: 0xbfeff410 0x00007ff7 0x1140e386 0x00007ff8
0x7ff7bfeff200: 0x00000000 0x00000000 0x00000000 0x00000000
```

As can be seen from the above screenshots, 'malloc()' allocates memory but doesn't initialize it, so the values in the malloc-allocated array are random or whatever was already in that memory. On the other hand, 'calloc()' allocates memory and also initializes it to zero. So, all elements in the calloc-allocated array will be zero

Part 4

For this part we will modify our program to allocate and deallocate an array of 100 bytes as follows::

```
5  int main() {
6      int *one_integer = (int *)malloc(sizeof(int));
7      printf("Address of the allocated space is: %p\n", (void *)one_integer);
8
9      int *arr_malloc = (int *)malloc(1000 * sizeof(int));
10     int *arr_calloc = (int *)calloc(1000, sizeof(int));
11
12
13     char *array = (char *)malloc(100 * sizeof(char));
14     printf("Address of the allocated space for 100 bytes before de-allocation is: %p\n", (void *)array);
15
16     free(array);
17     printf("Address of the allocated space for 100 bytes after de-allocation is: %p\n", (void *)array);
18
19     return 0;
20 }
```

And we get this as an output:

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % gcc -g -o c-memory c-memory.c
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-memory
Address of the allocated space is: 0x7f838f705db0
Address of the allocated space for 100 bytes before de-allocation is: 0x7f838f705be0
Address of the allocated space for 100 bytes after de-allocation is: 0x7f838f705be0
```

As can be seen from the above outputs, the pointer value is the same before and after freeing the array. It is important to note that the array pointer itself still holds the address of the memory that was just freed. This is called a “dangling pointer”, and we can set the pointer to NULL to avoid future problems or unexpected behaviors.

Part 5

When we try to free an array that has already been freed, we get a Double Free error as shown here:

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % gcc -g -o c-memory c-memory.c
⊗ sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-memory
Address of the allocated space is: 0x7fd5f3f05db0
Address of the allocated space for 100 bytes before de-allocation is: 0x7fd5f3f05be0
Address of the allocated space for 100 bytes after de-allocation is: 0x7fd5f3f05be0
c-memory(7973,0x7ff854dbab80) malloc: Double free of object 0x7fd5f3f05be0
c-memory(7973,0x7ff854dbab80) malloc: *** set a breakpoint in malloc_error_break to debug
zsh: abort      ./c-memory
```

It is important that the developer ensures that free() is not called more than once on the same memory block.

Part 6

For this part we will set the pointer to NULL as follows:

```
int main() {
    int *one_integer = (int *)malloc(sizeof(int));
    printf("Address of the allocated space is: %p\n", (void *)one_integer);

    int *arr_malloc = (int *)malloc(1000 * sizeof(int));
    int *arr_calloc = (int *)calloc(1000, sizeof(int));

    char *array = (char *)malloc(100 * sizeof(char));
    printf("Address of the allocated space for 100 bytes before de-allocation is: %p\n", (void *)array);

    // set array to NULL
    array = NULL;
    free(array);
    printf("Address of the allocated space for array after setting it to NULL is: %p\n", (void *)array);

    return 0;
}
```

We get the following outputs:

```
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % gcc -g -o c-memory c-memory.c
● sanadalwerfali@Sanads-MacBook-Pro Lab1 % ./c-memory
Address of the allocated space is: 0x7fcbf6705db0
Address of the allocated space for 100 bytes before de-allocation is: 0x7fcbf6705be0
Address of the allocated space for array after setting it to NULL is: 0x0
```

As can be seen from the above code, the free method has no effect on a NULL pointer and it is safe.

Macros

For this part of the lab we will get to explore macros in C language.

Part 1

We will start by defining FOO as follows:

```
Lab > Lab1 > C c-macro.c > ...  
1  
2 #define FOO "this is the definition of foo"  
3
```

After we run the `cc -E` command we generate a `c-macro-verbose.c` file and the content inside of it is the following:

```
Lab > Lab1 > C c-macro-verbose.c  
1 # 1 "c-macro.c"  
2 # 1 "<built-in>" 1  
3 # 1 "<built-in>" 3  
4 # 385 "<built-in>" 3  
5 # 1 "<command line>" 1  
6 # 1 "<built-in>" 2  
7 # 1 "c-macro.c" 2
```

As can be seen from the above screenshot, the file doesn't contain the string FOO because it has not been used in our code.

Part 2

Now we write a program that uses the variable FOO as follows:

```
Lab > Lab1 > C c-macro.c > FOO  
1 #define FOO "this is the definition of foo"  
2  
3 void useFoo() {  
4     char character = FOO[5];  
5 }  
6
```

And we get the following:

```
Lab > Lab1 > C c-macro-verbose.c > ...
1  # 1 "c-macro.c"
2  # 1 "<built-in>" 1
3  # 1 "<built-in>" 3
4  # 385 "<built-in>" 3
5  # 1 "<command line>" 1
6  # 1 "<built-in>" 2
7  # 1 "c-macro.c" 2
8
9
10 void useFoo() {
11     char character = "this is the definition of foo"[5];
12 }
```

As can be seen from the above screenshot, since we used 'FOO' in our program, any occurrence of FOO gets replaced with its definition that we set inside of the c-macro.c file.

Part 3

Now we can modify our program to print FOO as follows:

```
Lab > Lab1 > C c-macro.c > ...
1  #include <stdio.h>
2  #include <stddef.h>
3
4  #define FOO "this is the definition of foo"
5
6  void useFoo() {
7      char character = FOO[5];
8  }
9
10 int main() {
11     printf("%s\n", FOO);
12     return 0;
13 }
```

And our verbose file is the following:

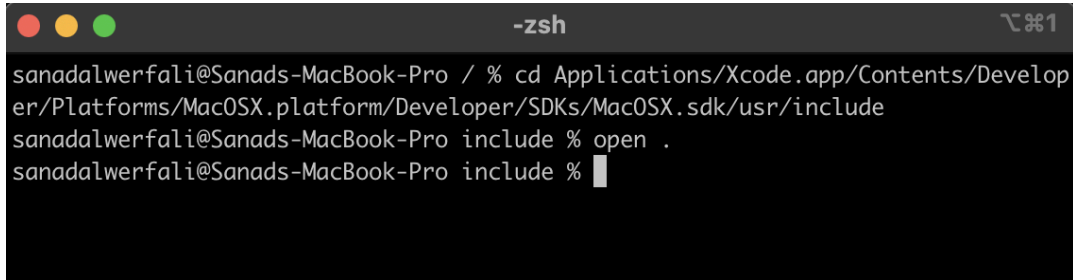
```
Lab > Lab1 > C c-macro-verbose.c > ...
553
554
555
556 extern int __vsprintf_chk (char * restrict, size_t, int, size_t,
557     const char * restrict, va_list);
558 # 410 "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdio.h" 2 3 4
559 # 2 "c-macro.c" 2
560 # 1 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/stddef.h" 1 3
561 # 35 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/stddef.h" 3
562 typedef long int ptrdiff_t;
563 # 60 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/stddef.h" 3
564 typedef long unsigned int rsize_t;
565 # 74 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/stddef.h" 3
566 typedef int wchar_t;
567 # 103 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/stddef.h" 3
568 # 1 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/__stddef_max_align_t.h"
569 # 16 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/__stddef_max_align_t.h"
570 typedef long double max_align_t;
571 # 104 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/15.0.0/include/stddef.h" 2 3
572 # 3 "c-macro.c" 2
573
574
575
576 void useFoo() {
577     char character = "this is the definition of foo"[5];
578 }
579
580 int main() {
581     printf("%s\n", "this is the definition of foo");
582     return 0;
583 }
584
```

First thing to notice is again, all instances of FOO get replaced with its definition. Moreover, as can be seen from the above screenshot, all included directives get replaced with the content of the respective files they point to (in our case we included `stdint.h` and `stdio.h`). This is part of what makes the output file more verbose. The output file will contain all the code from the included files, along with the code from the original file (with macros expanded and comments removed).

Part 4

Now we will look into the definition of NULL in C using the stddef.h file and will compare it to its definition in C++.

First we will find the stddef.h header file using our terminal as follows:

A terminal window titled '-zsh' with a dark background. The prompt is 'sanadalwerfali@Sanads-MacBook-Pro / %'. The user has entered the command 'cd Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include'. The prompt is now 'sanadalwerfali@Sanads-MacBook-Pro include %'. The user has entered 'open .' and the prompt is now 'sanadalwerfali@Sanads-MacBook-Pro include %' with a cursor.

```
sanadalwerfali@Sanads-MacBook-Pro / % cd Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include
sanadalwerfali@Sanads-MacBook-Pro include % open .
sanadalwerfali@Sanads-MacBook-Pro include %
```

Once we open the directory where the stddef.h file resides, we can open it using 'cat' and we check inside of the stddef.h file to find the macro for NULL as follows:

A dark rectangular box containing white text representing a code snippet from the stddef.h file.

```
#else
#define NULL ((void*)0)
#endif
```

We can see that In C, NULL is typically defined as an integer constant zero ((void*)0). This means that NULL is essentially an integer, and it can be implicitly converted to any pointer type. whereas, nullptr is a keyword used in C++ that represents a null pointer constant. It is of type nullptr_t, which is implicitly convertible and comparable to any pointer type or pointer-to-member type. This difference can lead to different behaviors in certain contexts, such as function overloading.