**C Basics for Toddlers**

*50.005 Computer System Engineering*

***Materials taken from various resources. Suitable for 1-3 years old toddlers.***

[Learning Objectives](#_grkcv3epkn13)

[Part 1: Function Pointers](#_qhpa1oknj0g6)

[Learning Points](#_4emk14w3fu91)

[Part 2: Void\* Return Type](#_2w9pqc7ub4mz)

[Part 3: Command Line Arguments](#_ri18yh9lrn4i)

[Part 4: File I/Os](#_s5jktf59ubc1)

[Learning Points](#_s8ctrxitahvo)

[Part 5: Error Handling](#_640s97564aiw)

[Learning Points](#_1p2sflytg5a3)

[Part 6: C Threads](#_4q792v30tdkg)

[Learning Points](#_qysiajo1mxz1)

[Part 7: fork()](#_h4ytk5yh056z)

[Learning Points](#_h88063oxivjy)

[Part 8: Shared Memory](#_b06jq7jr3xc0)

[Learning Points](#_9lxc343zyd1x)

[Part 9: Socket](#_9c6ayp2xoglj)

[Learning Points](#_lwr4svrsut6i)

[Summary](#_37nx294qo77h)

# Learning Objectives

**In this class we will learn how to:**

1. Utilize function pointers
2. Create and utilize functions with void\* return type
3. Receive command-line arguments
4. Handle File I/Os
5. Error checking and handling

At the end of this class, you may head to e-dimension (Week 2) and answer the questions there to test your understanding. **The test grade is for personal achievement only and not included for computation of grades in 50.005.**

# 

# Part 1: Function Pointers

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. By definition, **pointers point to an address in any memory location**, they can also **point** to **at the beginning** of **executable** **code** as functions in memory.

They are declared in this format: return\_type (\*function\_name)(argument types)

For example, create the following header file:

| #include <stdio.h> #include <stdlib.h>   int sum(int\* array, int size); //declaration of a function  //legal declaration and initialization of pointer to function int (\*sum\_function\_pointer)(int\*, int) = &sum; |
| --- |

Here we declare two things:

1. The function sum
2. The pointer sum\_function\_pointer, pointing to the address of the function sum

You can easily call the function through its pointer as follows:

| #include "morec.h"  int sum(int\* array, int size){  int sum\_value = 0;  for (int i = 0; i<size; i++){  sum\_value += array[i];  }  return sum\_value; }  int main (){  int array[10] = {1,2,3,4,5,6,7,8,9,10};   //call the function using pointer  int result = sum\_function\_pointer(array, 10);    printf("The result is %d \n", result);  return 0; } |
| --- |

Notice:

1. The function sum receives an array in the form of integer pointer, along with its size
2. You can call the function through its pointer using pointername(args)

One of the reasons why function pointers are handy is if you have a whole array of them:

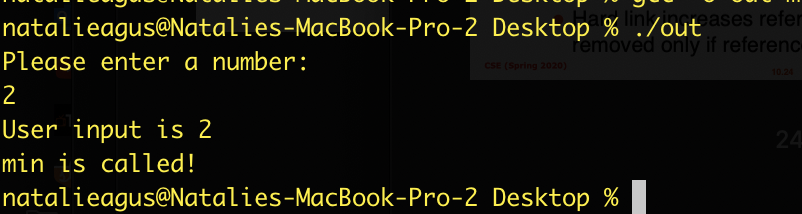
| #include <stdio.h> #include <stdlib.h>  //declaration of function int sum(int\* array, int size); int geometric\_sum(int\* array, int size); int min(int\* array, int size); int max(int\* array, int size); int stdev(int\* array, int size); int average(int\* array, int size);  //legal declaration and initialization of pointer to function int (\*function\_pointers[])(int\*, int) = {  &sum, //index 0  &geometric\_sum, //index 1  &min, //index 2  &max, //index 3  &stdev, // index 4  &average //index 5 |
| --- |

};

And that you wish to invoke them *by index*:

| int main (){  int array[10] = {1,2,3,4,5,6,7,8,9,10};  char input;  printf("Please enter a number: \n");  scanf("%s", &input);   int user\_input = atoi(&input);   printf("User input is %d \n", user\_input);  // select a function based on user input  function\_pointers[user\_input](array, 10);   return 0; } |
| --- |

Assuming you have implemented the functions, you should be able to invoke a particular function given an index:

****

**Functions in the same function pointer array must have the same return type and argument types.**

For functions that return a pointer, the pointer to that function can be written as:

| int\* test\_function(int\* array, int size); int\* (\*test\_function\_pointer)(int\*, int); |
| --- |

Which still follows the same format of return\_type (\*function\_name)(argument types).

Finally, the fun part about having function pointers is that we can **pass a function pointer** as an argument to another function.

For example, declare and initialize the pointer to the sum function, and a function that receives such pointer as its argument:

| int (\*sum\_function\_pointer)(int\*, int) = &sum; int (\*min\_function\_pointer)(int\*, int) = &min; void func (int (\*f)(int\*, int), int\* array); |
| --- |

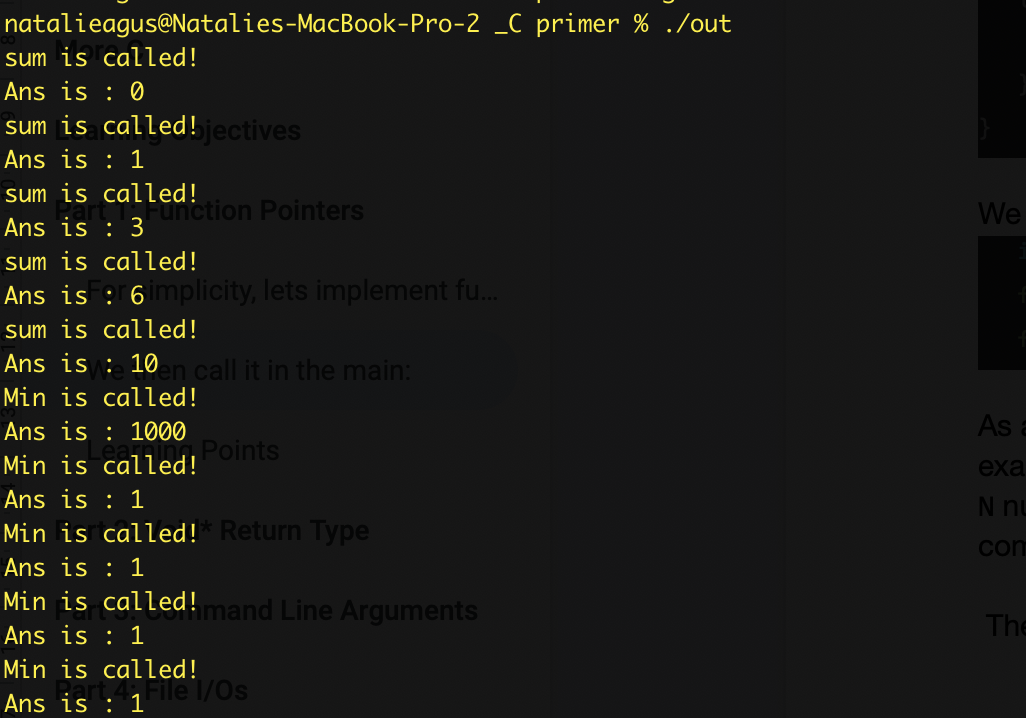
For simplicity, lets implement func as follows:

| void func(int (\*f)(int \*, int), int \*array) {  for (int ctr = 0; ctr < 5; ctr++)  {  printf("Sum is : %d \n", (\*f)(array, ctr));  } } |
| --- |

We then call it in the main:

| int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  func(sum\_function\_pointer, array);  func(min\_function\_pointer, array); |
| --- |

The output is:



In essence, we can ask the same function to run different other functions. In this simple example, of course it can be done by a simple if-else. In practice, this is useful if you have N number of functions with M different possible arguments. To automate executing all combinations, you’d want to pass function pointers plus its arguments around like this.

### Learning Points

1. Create and utilize function pointers (a pointer that points to the start of a function)
2. Understand how to use utilize function pointers in an array
3. Call functions through its pointers
4. Pass function pointers as arguments to other functions.

# Part 2: Void\* Return Type

Void pointers are used during function declarations. We use a void\* return type permits to return **any type.**

For example, consider this function declared at the header file, and its implementation:

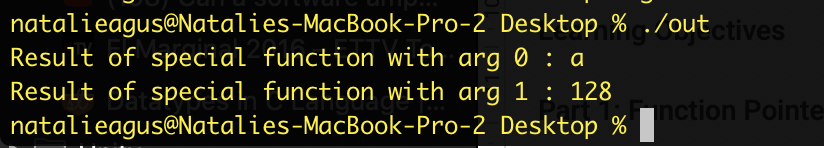
| void\* special\_function(int arg);  void\* special\_function(int arg){  if (arg == 0){  char\* c = malloc(sizeof(char));  c[0] = 'a';  return c;  }  else{  int\* i = malloc(sizeof(int));  i[0] = 128;  return i;  } } |
| --- |

This shows that the function can return either pointer to a char, or pointer to an int, depending on the value of arg. Therefore we declare its return type void \* (a generic pointer).

In the main function, we can call the function twice with different arguments:

| char c = \*((char \*)special\_function(0));  int i = \*((int \*) special\_function(1));  printf("Result of special function with arg 0 : %c \n", c);  printf("Result of special function with arg 1 : %d \n", i);   free(c);  free(i); |
| --- |

Which of course as expected, results in different output based on the arg given:



# Part 3: Command Line Arguments

Command line arguments can be supplied when you call the executable from your terminal. Each argument is separated by **spaces**. Every process has **at least one argument**, which is itself.

The main function can *optionally* receive argument in this format:

| int main (int argc, char\*\* argv) |
| --- |

Where:

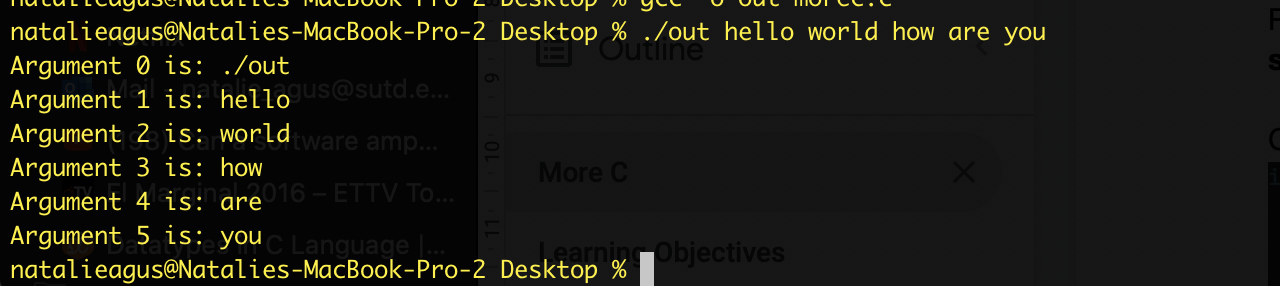
* argc: the number of arguments supplied in the command line
* argv: a double char pointer, where argv[i] points to the **start** of the argument

Recall that a **string** itself is a char\* type, since the pointer **points to the first char in the string**. Therefore an *array of strings* is naturally a char\*\* type.

Consider this main function:

| int main (int argc, char\*\* argv){   if (argc < 2){  printf("Please supply arguments!\n");  }   for (int i = 0; i<argc; i++){  printf("Argument %d is: %s \n", i, argv[i]);  } } |
| --- |

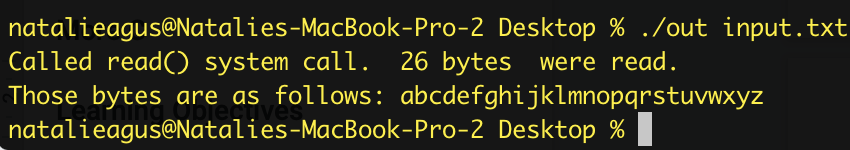
The following is the output:



Usually the arguments we supply are filenames, options, etc that are going to be **parsed** by the program (system programs for example allows many options) accordingly to do what we need it to do, e.g: ls -la (-la is an option to display file info in full format).

# Part 4: File I/Os

In this section, we will learn how to open a file, read, and write to it programmatically using C. Before we can **read** or **write** to *any file*, we need to **open** it. After usage, we need to **close** it.

Supposewe want to have a program that receives a filename from the command line argument, as argv[1], 

You need three more C standard libraries for this part. Include these in your header file:

| #include <unistd.h> #include <fcntl.h> #include <string.h> |
| --- |

We need to first check that there’s at least *two arguments:*

| //check arguments  if (argc < 2)  {  printf("Please key in filename\n");  return 0;  } |
| --- |

Afterwhich, you can use the open() system call:

| //open the file, with flag of O\_RDONLY if you only want to read  int fd = open(argv[1], O\_RDONLY);   //error checking  if (fd < 0)  {  perror("Failed to open file. \n");  exit(1);  } |
| --- |

* The open(filename, mode) system call returns an **integer**, which is something called a *file descriptor*. You will learn more about this in **week 6.** For now, let’s think of it as just an “id” to the file that you have opened.
* If fd < 0, it means that the open() operation is **not successful**, and you should handle this accordingly. We will talk more about *error handling* in the [next part](#_640s97564aiw).
* This mode O\_RDONLY is just to *read the file*. There’s another mode: O\_RDWR to read and write, and O\_CREAT to create (if a file that's being opened doesn’t exist).
* You can later on read() or write() to this file by putting the fd as the argument to the functions.
* The filename can be *absolute* or ***relative path* (meaning that the path starts at the same current working directory).**
* For now, let's just assume that both the program and the file are in the same folder.

The next step is just to read from the opened file:

| //initiallize a character array to contain what you will read later  char char\_buffer[128];   //byte offset  int byte = 0;   //read 1 byte by 1 byte, put it into the buffer  int check\_read = read(fd, char\_buffer + byte, 1);   //keep reading 1 byte until nothing else to read  while (check\_read > 0)  {  byte++;  check\_read = read(fd, char\_buffer + byte, 1);  } |
| --- |

* First, we need to **allocate** memory (char\_buffer) to contain the space we need to read the file. As we learned before, this is *a static memory allocation.*
* Statically allocated memory is **automatically** **released** on the basis of scope, i.e., as soon as the scope of the variable is over, memory allocated gets freed.
* *Note : we do not know the size of the file, but for now we know that it is less than 128 bytes.*
* We can then read the file using the read(file\_descriptor, array, amount) system call.
* The system call will return the number of bytes read (the argument amount is basically asking the program to read *up to the amount* of bytes).

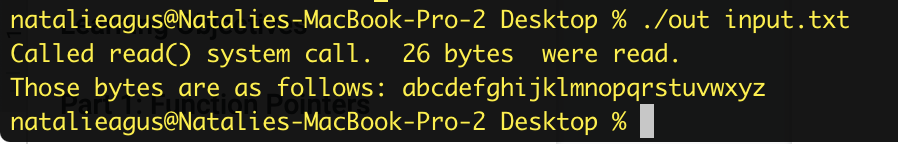
Then, we have the while loop there because we are primitively reading the contents of the file **byte by byte** and storing them in the array. That’s why we need the int byte to take note of the **offset address** from the beginning of the char\_buffer.

If we have reached the end of file (EOF character), read() will return 0, and we will break out of the while-loop.

Finally we can print the content and **close the file**:

| printf("Called read() system call. %d bytes were read.\n", byte);   //add terminating character so that you can print it  char\_buffer[byte] = '\0';   printf("Those bytes are as follows: %s \n", char\_buffer);   //close the file  close(fd); |
| --- |

With a sample input.txt file, the output of the program is:



To **write to a file**, a similar protocol is used. Assume we need another argument for the filename of the file *to write to*, so let's modify our argument checking:

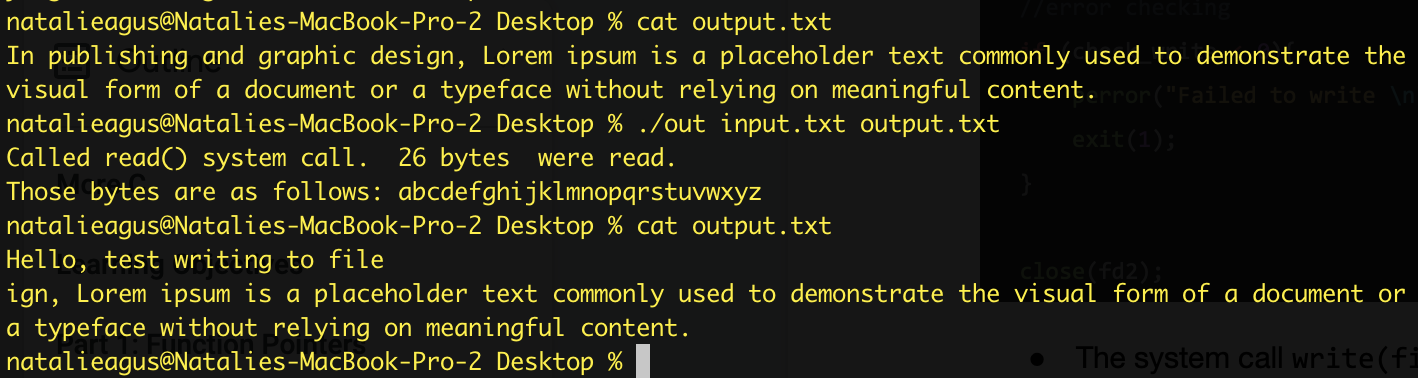
| //check arguments  if (argc < 3)  {  printf("Please key in filename\n");  return 0;  } |
| --- |

And the code to write to file:

| int fd2 = open(argv[2], O\_RDWR|O\_CREAT, 0666);  char sentence\_to\_write[128] = "Hello, test writing to file \n";   int check\_write = write(fd2, sentence\_to\_write, strlen(sentence\_to\_write));   //error checking  if (check\_write < 0){  perror("Failed to write \n");  exit(1);  }    close(fd2); |
| --- |

* The system call write(file\_descriptor, char\* sentence, int length) will write the stated number of length bytes to the file.
* It will return -1 upon *error*, hence it is good to check its status afterwards. More about error checking in the next [part](#_640s97564aiw).
* Note that this time round, we have the third argument to system call open(). These are **file permission**, so that in the case that the file doesn’t exist and that we create it, the value 0666 ensures that everybody has permission to read and write (all other users).

Notice that if output.txt **has already existed** with contents longer than the sentence we are intending to write, the **remainder** of the old content will still be there (it's not “cleared”).



If you would like to *clear* and *overwrite the file*, you need to use O\_TRUNC option instead:

| int fd2 = open(argv[2], O\_RDWR|O\_CREAT|O\_TRUNC, 0666); |
| --- |

This **truncates** (set its content to be 0 bytes) the file automatically as you open it.

If you would like to **append** to file:

| int fd2 = open(argv[2], O\_RDWR|O\_CREAT|O\_APPEND, 0666); |
| --- |

There’s also an alternative way to open the file, *which less primitive and more convenient to write:*

| FILE \*out = fopen(argv[2], "a");   if (out != NULL){  fprintf(out, "%s", "Hello Again! \n");   fclose(out);  }  else{  perror("File failed to be opened\n");  return 0;  } |
| --- |

* We use fopen(filename, mode) instead. The mode is in terms of string, of which “a” means *append*.
* The system call fopen returns a FILE pointer type.
* You can *write* using fprintf instead, this is like *printing to console,* but we do it to the file instead.

**Note: do NOT read AND write using the same file descriptor. You will NOT get the behaviour you expected because they share the *same pointer.***

Each time you read, the pointer will be advanced by n number of bytes which you tell it to read, and is *no longer at the beginning of the file.* You can set the absolute byte position (from the beginning of the file) of the file pointer using the following before attempting to read() or write() to the file again. You will learn more about it in **week 6.**

| lseek(file\_descriptor, BYTE\_absolute\_location, SEEK\_SET) |
| --- |

### Learning Points

1. Learn how to **open,** read, and write to file
2. Know the existence of different modes to open a file depending on our purpose
3. Understand the notion of file pointer

# Part 5: Error Handling

It is always a good habit to check for *errors* whenever we make a system call or other API calls. You can read the documentation first to check different return types. We have seen one above, for file open():

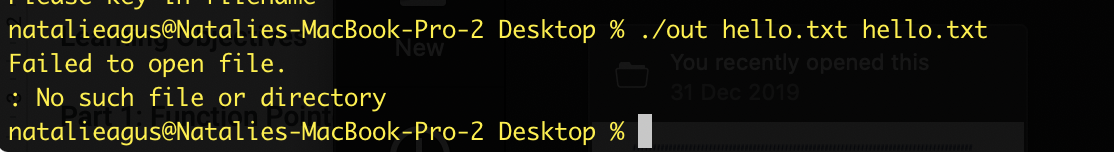
| //open the file, with flag of O\_RDONLY if you only want to read  int fd = open(argv[1], O\_RDONLY);   //error checking  if (fd < 0)  {  perror("Failed to open file. \n");  exit(1);  } |
| --- |

**It is very difficult to otherwise debug a C program that’s successfully compiled but encounter errors at runtime. There’s no default exception handling in C. All you can do is to check if the system call or API call is successful or not from its return value.**

How to check error:

* **Always check the return value** of your system call or API call. There’s plenty of these library-implemented functions that we have encountered so far regarding file operations and dynamic memory allocations.
* Have your program execute a perror(), that will print your custom error message plus the error message that the system has given you.

For example, an attempt to open non-existent file results in this error message:



If you have multiple file open(), then you **might want to print a more helpful error message** to indicate exactly which open(filename, mode) operation causes the error.

Here is another example involving dynamic memory allocation using malloc():

| int n = 5;  int\* ptr = (int\*)malloc(n \* sizeof(int));  // Check if the memory has been successfully  // allocated by malloc or not  if (ptr == NULL) {  printf("Memory not allocated.\n");  exit(0);  } |
| --- |

Upon memory allocation failure, the integer pointer ptr will *still* be pointing to a NULL location (only declared and not initialized).

In the next sections from now onwards, you will see such error handlings after each system call or API call. While it may seem tedious and clutter your code, it is very useful to do so to catch bugs during development. You will save a lot of time when debugging.

### Learning Points

1. Understand the importance of success-check in API calls
2. Handle errors efficiently without having to print each line to catch the bug
3. Handle errors efficiently without the usual try-catch convenience

# 

# Summary

Congratulations. You have completed the second training for C in this course. If you’d like to test your knowledge up until now, head to e-dimension and do the quiz (Part 2). The grade is not going to be computed for your overall grade.