Introduction to C : Basics

50.005 Computer System Engineering

Materials taken from various resources

Installing C
- Ubuntu
- MacOS (Catalina)

Hello World!

Learning Objectives

Part 1: .c and .h file extensions
- Learning points

Part 2: Primary Data Types
- Learning Points

Part 3: Derived Data Types
- Arrays
- Pointers
- Strings
- Structures
- Learning Points

Part 4: Loops and Iterations
- Learning Points

Part 5: Functions
- Learning Points

Part 6: Dynamic Memory allocation
- Learning Points

Summary

Natalie Agus
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Installing C

Ubuntu

To install C compiler and its manual on Ubuntu, type the following into the terminal:

1. `sudo apt-get update`
2. `sudo apt-get upgrade`
3. `sudo apt install build-essential`
4. `sudo apt-get install manpages-dev`

Then you can check if gcc is installed successfully using: `gcc --version`
And see output similar to below:

```
gcc (Ubuntu 7.4.0-1ubuntu1~18.04) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

MacOS (Catalina)

For MacOS users, **simply install Xcode.** It automatically comes with Command Line Tools that include the C compiler. If not, you can always download it separately. There's plenty of guide on the internet on how to do this easily.

You can type `gcc --version` in the terminal and see output similar to below upon successful installation:

```
Configured with: --prefix=/Library/Developer/CommandLineTools/usr
--with-gxx-include-dir=/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
/usr/include/c++/4.2.1
Apple clang version 11.0.0 (clang-1100.0.33.12)
Target: x86_64-apple-darwin19.0.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

Hello World!

Now install your favourite editor (one recommendation is VSCode) and create the following helloworld.c file. **If you are using WSL please setup VSCode for WSL environment first. You can find the info in edimension > information tab.**

```c
#include <stdio.h>
int main()
{
   printf("Hello, World!");
   return 0;
}
```

Save it in a directory of your choice, and then `cd` to that directory. Type the following to compile and run them:

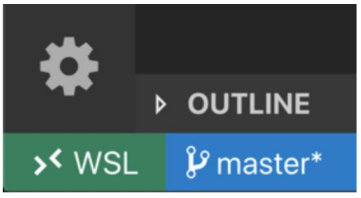
```
$ gcc -o hello hello.c
$ ./hello
```

And you should see the following output in your terminal:

```
Hello, World!
```

Note: for MacOS users, it is NOT recommended for you to use XCode IDE.

For WSL users, remember you need to install and *activate* your WSL remote server when you use VSCode to edit your code. Go to the bottom left corner of your VSCode and ensure it has the WSL indicator (sometimes with Ubuntu written if you use it) like this.



Otherwise, click open the green remote window logo on the bottom left corner (ensure you have installed the Remove Development extension pack for VSCode) and select Remote-WSL: new Window. A new window in your WSL home directory will pop up. You can save or open new file and navigate the folders from there.

Use the terminal in VSCode instead: Terminal >> New Terminal. A new terminal will be created at your home directory. Type `pwd` to find out where you are. If you are not familiar with terminal commands, **do homework 1 first**.

Learning Objectives

In this intro class, we will learn how to:

1. Create .c and header files
2. Create, manipulate, print different primary data types
3. Create, manipulate, print different derived data types
4. Loops and iterations
5. Create functions, and pass value by reference
6. Allocate and manage memory dynamically

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: .c and .h file extensions

Typically, a C program comes with .c and .h file extensions. The files with .h extension is called the **header files,** and typically contain:

1. Functions declaration
2. Constants declaration
3. Global variables
4. Struct declaration
5. Library imports

And so on that's required by the .c files. These header files are then `#included` in .c files where the bulk of your main code lies. You can think of .c files as the files that contain the **actual work**.

For example, create a file cclass_part.c with the following content:

```c
#include "cclass.h"
int main(int argc, char** argv){
  printf("Hello World!\n");
  printf("Constant BUFFERSIZE has a value of %d \n", BUFFERSIZE);
}
```

And create a file cclass.h with the following content:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>



#define BUFFERSIZE 1024
```

Save both of them in the same folder and compile it: **`gcc cclass_part1.c -o out`**
Run it, and you will see such output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % gcc cclass_part1.c -o out
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Hello World!
Constant BUFFERSIZE has a value of 1024
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

Notice how the .c can print the amount of BUFFERSIZE, which is a constant that is declared earlier in the header file imported.

Learning points

1. You can import header file using the #include keyword and the header file name using the quotation marks, e.g: `#include "cclass.h"`
2. You can import **standard libraries** using the #include keyword and the arrow brackets: `#include <stdio.h>`
3. There are many different C standard libraries. These three are the most basic ones that allow you to perform *simple system calls* such as `print`.
4. You can define `int/float` constants using the keyword `#define NAME value`

Technically, you can dump everything within a single .c file, but you may find that you may want to share some function **declarations** for many different .c files. It is convenient then to **declare** them earlier in a single header file that's **imported** by different .c files.

To give you some context, a single program that's written in C can be comprised of **millions of lines**. One of such examples is the interpreter of your favourite language, Python. **Python interpreter** (the program that you call when you type `python / python3` in command line to run a .py script) is written in C.

Part 2: Primary Data Types

These are the basic data types in C that we will encounter in this class: `int, char, float,` and *`void`*. Void means *nothing or undefined type*. We will talk more about this later. **The size of int is 32 bits (4 bytes), float is 32 bits (4 bytes), and char is 8 bits (1 byte).** You can increase the precision of the float into 8 bytes using `double` (also a float but with double precision) type instead of float. However, it's not required for this course.

Add these lines into your main function:

```
   int x = 5;
   float y = 3.0;
   char a = 'a';
   char b = 'b';
   char c = 'c';

   printf("Printing integer x: %d \n", x);
   printf("Printing float y: %f \n", y);
   printf("Printing characters abc: %c %c %c \n", a,b,c);
   printf("Printing characters as ASCII: %d %d %d \n", a,b,c);

   printf("Size of int is %d bytes, size of float is %d bytes, size of char is %d bytes\n",
sizeof(int), sizeof(float), sizeof(char));
```

Compile it and you see the output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Printing integer x: 5
Printing float y: 3.000000
Printing characters abc: a b c
Printing characters as ASCII: 97 98 99
Size of int is 4 bytes, size of float is 4 bytes, size of char is 1 bytes
```

As you can see, primary types are pretty basic. The way you print them needs the "formatting" (similar to Java): %d for int, `%f` for floats, and `%c` for characters, and you supply the content later on. You can read more about it in `printf` documentation. Meanwhile, here's the cheatsheet you need:

%c	character
%d	decimal (integer) number (base 10)
%e	exponential floating-point number
%f	floating-point number
%i	integer (base 10)
%o	octal number (base 8)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)
%%	print a percent sign
\%	print a percent sign

Don't underestimate the power of print. It saves you a lot of time when debugging. Although it is *technically not* the *professional* way of debugging, it works better than nothing at all. **Don't be lazy of printing things and checking your work every now and then.**

WARNING: Debugging C is pretty painful if you DO NOT do incremental debugging.

Learning Points

1. Primary data types: `int, float, char`
2. Use print formatting
3. Variable naming follow the same rule as Java:
    a. You can't start with a digit
    b. Upper and lower case is treated differently (Case sensitive)
    c. You can't use keywords as variable names

Part 3: Derived Data Types

These are the derived data types in C that we will encounter in this class: `array (of int, chars - strings, etc), pointer, and structure.`

Arrays

Creating arrays of basic types are easy, using the common [ ] array declaration. Add these lines to your main function:

```
   int vector_int[3] = {1,2,3};
   float vector_float[3] = {0.3,0.4,0.5};
   char characters[5] = {'a','i','u','e','o'};

   printf("Contents of vector_int %d %d %d \n", vector_int[0], vector_int[1],
vector_int[2]);
   printf("Contents of vector_float %f %f %f \n", vector_float[0], vector_float[1],
vector_float[2]);
   printf("Contents of the second char: %c\n", characters[1]);
```

Compile it and you see the output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % gcc cclass_part1.c -o out
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Contents of vector_int 1 2 3
Contents of vector_float 0.300000 0.400000 0.500000
Contents of the second char: i
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

Notice that the array size **cannot be dynamic! It has to be a CONSTANT.**

If you try to compile stuff like this:

```
  int array_size = 3;
  int vector_int[array_size] = {1,2,3};
```

You will be met with such error:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % gcc cclass_part1.c -o out
cclass_part1.c:23:20: error: variable-sized object may not be initialized
    int vector_int[array_size] = {1,2,3};
                   ^~~~~~~~~~
```

This is because `array_size` is a variable.

Also, with the [ ] you are declaring an object with automatic storage duration (in stack). This means that the array **lives only as long as the function that calls it exists.**
If you want to create an array with a size that's determined **later** during **runtime**, or **persists outside the function calling it** (in heap)**,** then you should use `malloc` or `calloc` that's going to be discussed in the later part. Before we can learn this, we need to know what a *pointer* is first.

Pointers

A **pointer** data type is indicated by the * (star) sign. You can make a *pointer* of any primary / derived data types. The **pointer** simply means the **address of the data you are pointing to.**

Paste the following code in your main file:

```
   // recall int vector_int[3] = {1,2,3};
   int *vector_int_pointer = vector_int;
   printf("Address of vector_int array is 0x%llx\n", vector_int_pointer);
   printf("Address of the first element in vector_int array is 0x%llx\n", &vector_int[0]);
   printf("Address of the second element in vector_int array is 0x%llx\n", &vector_int[1]);
   printf("Address of the third element in vector_int array is 0x%llx\n", &vector_int[2]);

   printf("Printing address using pointer : \n");
   printf("Address of the first element in vector_int array is 0x%llx\n",
vector_int_pointer);
   printf("Address of the second element in vector_int array is 0x%llx\n",
vector_int_pointer+1);
   printf("Address of the third element in vector_int array is 0x%llx\n",
vector_int_pointer+2);
```

You'll find such output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Address of vector_int array is 0x7ffee26129bc
Address of the first element in vector_int array is 0x7ffee26129bc
Address of the second element in vector_int array is 0x7ffee26129c0
Address of the third element in vector_int array is 0x7ffee26129c4
Printing address using pointer :
Address of the first element in vector_int array is 0x7ffee26129bc
Address of the second element in vector_int array is 0x7ffee26129c0
Address of the third element in vector_int array is 0x7ffee26129c4
```

Let's digest this output:

1. We create a pointer (the star can stick to either side, doesn't matter where it is so long it is between the data declaration and variable name) of type int, meaning that it is pointing to the **address of the first integer element in vector_int.**

2. Notice that **an array is basically a pointer,** therefore we can simply do:
   ```
   int *vector_int_pointer = vector_int;
   ```

3. As seen in the print format, we can print the *content* of the pointer and we have 0x7ffeec216129bc (address space is huge as its a 64 bit system)
4. This is the **address of the first element of the vector_int.** You can also manually print an **address** of your data using the & operator, e.g: `&vector_int[0]`

5. That's why we have the printed output of `&vector_int[0]` and `vector_int_pointer` as the same thing at 0x7ffeec216129bc.

6. Notice the **address** of the next two elements in the array differ by FOUR bytes: `0x7ffeec216129bc, 0x7ffeec216129c0,` and `0x7ffeec216129c4.`

7. This is because recall from Part2, the **`sizeof(int)` is 32 bits (4 bytes).** The size of a pointer however is 64 bits (8 bytes). **This means that although it is a 64 bit system,** we can **access memory address like a 32 bit system where it differs by just 4 bytes instead of 8!**

8. We also know that array is a **contiguous block of memory.** Therefore we also can obtain the **address** of the other elements by incrementing the pointer value by 1 (word of 4 bytes), which is what we are doing here (obtaining the same output at `0x7ffeec216129c0`:
   ```
   printf("Address of the second element in vector_int array is 0x%x\n",
   vector_int_pointer+1);
   ```

Now let's examine how we can change the content of the integer array using the pointer. Paste the following code:

```
//change the second element of vector_int
printf("The original second element is %d\n", vector_int_pointer[0]);
vector_int_pointer[2] = 5;
printf("The new second element is %d\n", vector_int_pointer[1]);
printf("The new second element is %d\n", *(vector_int_pointer+1));
```

And you have the output:

```
The original second element is 1
The new second element is 2
The new second element is 2
```

So this teaches you that:

1. You can *access* the content pointed by the pointer using the [i] just like how you access element in array. In essence, the [i] means that we:
    a. Compute the effective address EA using `vector_int_pointer + i*4`
    b. Obtain the element at `Mem[EA]`
2. OR you can also perform it manually by computing the EA yourself:
    a. Add 1 *word (4 bytes, not 8!)* into the value of vector_int_pointer: `vector_int_pointer+1`
    b. Obtain the element pointed to by the *content* of the variable explicitly using the * operator. You can read this * operator as "value at"

Can we create pointers to other primary data types that's not an array? Paste the following code:

```c
int z = 5;
int *z_pointer = &z;


printf("Value of z is %d \n", z);
printf("Z is stored in address 0x%llx\n", z_pointer);
printf("The pointer to Z is stored in address 0x%llx\n", &z_pointer);
printf("Size of Z pointer is: %d \n", sizeof(z_pointer));


// change value of z through pointer
*z_pointer = 6;
printf("The new value of z is %d\n", *z_pointer);
```

Compile it and this is the output you'd get:

```
Value of z is 5
Z is stored in address 0x7ffeeab4c97c
The pointer to Z is stored in address 0x7ffeeab4c970
Size of Z pointer is: 8
The new value of z is 6
```

Here's why:

1. The integer z is stored at address 0x7ffeeab4c97c
2. We create a *pointer* into it, called z_pointer
3. **z_pointer itself is stored at 0x7ffeeab4c970. The content of 0x7ffeeab4c970 is 0x7ffeeab4c97c, that is the *address of z*. Therefore, we can say that z_pointer IS A POINTER because it does not store a content but rather, an address.**
4. Recall its a 64 bit system, so the size of pointer is 8 bytes.
5. We can change the value that z_pointer is pointing to using the * operator too. **Recall the other operator we learn, the & operator means *obtaining the address of the variable it is operated on (address of).***

Strings

What about strings? Well, strings are none other than an array of characters.

```
char hello_world[12] = {'h','e','l','l','o',' ','w','o','r','l','d', NULL};
printf("%s", hello_world);
```

You will find that you have such output, with some garbage content at the end.

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
hello world▯▯▯>▯▯▯>%
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

This is because printf with %s format will keep printing until it finds a NULL terminated character.

You can easily fix this by adding a null termination character (as NULL or '\0'):

```
char hello_world[12] = {'h','e','l','l','o',' ','w','o','r','l','d', '\0'};
printf("%s\n", hello_world);
```

And printf will just print "hello world".

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
hello world
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

Of course it is tedious to type characters one by one. Since a string is just an array of characters and a pointer works the same way as array identifiers, we can simply create a string using char pointer:

```
//allocates in static memory, NOT modifiable, READ only
char *hello_world_readonly = "hello world";
printf("%s\n", hello_world_readonly);
printf("Size of hello_world_better pointer %d\n",sizeof(hello_world_readonly));
```

Conveniently, the NULL termination is automatically added for you.

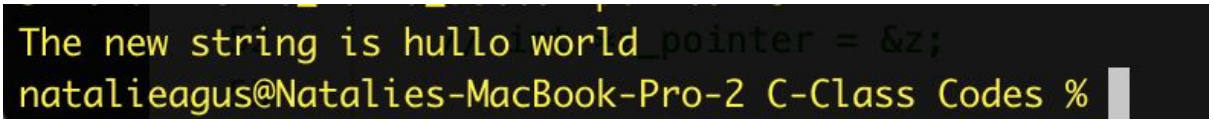
```
hello world
Size of hello_world_better pointer 8
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

You can also initialize string this way:

```
char hello_world_init[] = "hello world";
//change the letter in the string
```

```
hello_world_init[1] = 'u';
printf("The new string is %s\n", hello_world_init);
```

The difference between this and the method previously is that this is allocated in **static memory** which **cannot be modified during runtime**.
Whereas initializing it with the [ ] implies that it will be allocated in heap/stack (no formal documentation on this but this is how it is usually implemented) that's modifiable during runtime.

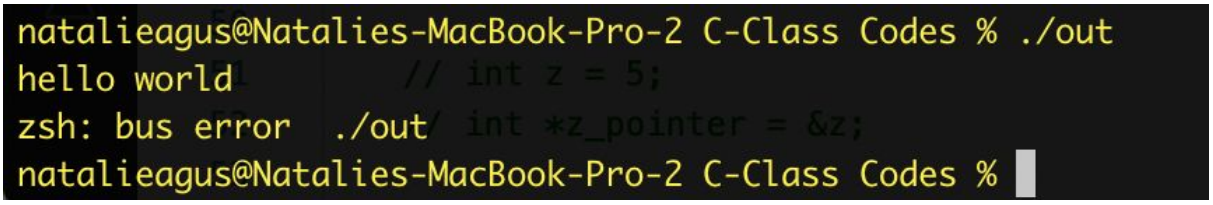


If you try to modify the static-allocated helloworld:

```
//allocates in static memory, NOT modifiable, READ only
char *hello_world_readonly = "hello world";
printf("%s\n", hello_world_readonly);

hello_world_readonly[1] = 'u'; //this results in unpredictable behavior
printf("The new string is %s\n", hello_world_readonly);
```

You will be faced with errors:



A very common way to initialise a string is by defining a size to the array declaration as an empty string. Then we can *print* strings into it using `sprintf`. It overwrites the entire buffer.

```
char sentence[BUFFERSIZE] = "";
sprintf(sentence, "Hello World");
printf("The sentence is: %s \n", sentence);
sprintf(sentence, "This is another sentence overwriting the previous one. Lets write a number %d. ", 5);
printf("The sentence now is modified to: %s \n", sentence);
```

You can use the same print formats as you'd do with `printf`. Of course the size of the buffer has to be big enough. Recall BUFFERSIZE is a constant that's defined in the header file.

If you want to concatenate between two strings, you can use `strcat(char *dest, char *source)`:

```
char sentence_append[64] = "The quick brown fox jumps over a lazy dog";
strcat(sentence, sentence_append);
printf("%s \n", sentence);
```

The content of the second array is appended at the NULL termination of the first sentence, hence forming a new sentence:

```
This is another sentence overwriting the previous one. Lets write a number 5. The quick brown fox jumps over a lazy dog
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

There's a whole lot of string operations. You don't have to memorize them. We aren't testing you in C like how Digital World tests you on Python. We just want you to be able to apply and code in C at your own pace.

For what its worth, the nine most commonly used functions in the string library are:

- `strcat` - concatenate two strings
- `strchr` - string scanning operation
- `strcmp` - compare two strings
- `strcpy` - copy a string
- `strlen` - get string length
- `strncat` - concatenate one string with part of another
- `strncmp` - compare parts of two strings
- `strncpy` - copy part of a string
- `strrchr` - string scanning operation

You should be able to search by yourself how to use these operations when the need arises.

Structures

In C there's no classes / objects. The closest that we have to objects and classes are `structs`. These are data structures that contain a collection of primary and derived data types.

You can declare structures using the `struct` keyword, and then declare its members. Paste the following to your main function:

```
// defining struct
struct Vector_Int{
    int x;
    int y;
    int z;
    char name[64];
};
```

```c
// structure variable declaration, empty member values
struct Vector_Int v1;

// manual member initialization
v1.x = 2;
v1.y = 3;
v1.z = 10;
sprintf(v1.name, "Vector 1");

// structure variable auto member initialization
struct Vector_Int v2 = {3,5,11, "Vector 2"};

printf("Values of v1 is x:%d y:%d z:%d name: %s\n", v1.x, v1.y, v1.z, v1.name);
printf("Values of v2 is x:%d y:%d z:%d name: %s\n", v2.x, v2.y, v2.z, v2.name);
```

You will have such output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Values of v1 is x:2 y:3 z:10 name: Vector 1
Values of v2 is x:3 y:5 z:11 name: Vector 2
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

You can also declare struct within struct:

```c
struct Info{
    char name[32];
    int age;
    struct address{
        char area_name[32];
        int house_no;
        char district[32];
    } address;
};

struct Info my_Info = {"Alice", 25, "Somapah Road", 8, "Upper Changi"};

printf("Name: %s, age %d, area name %s, house number %d, district %s\n", my_Info.name,
my_Info.age, my_Info.address.area_name, my_Info.address.house_no,
my_Info.address.district);
```

And we can print out the members:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Name: Alice, age 25, area name Somapah Road, house number 8, district Upper Changi
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

Since address is a struct that's already defined within Info, we can also now create a `struct` variable `addrss`:

```
   struct address my_Addrs = {"Another Road", 15, "Lower Changi"};
   printf("Another address %s %d %s \n", my_Addrs.area_name, my_Addrs.house_no,
my_Addrs.district);
```

```
Another address Another Road 15 Lower Changi
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

Of course equivalently we can just define two struct and declare it as a member:

```
   struct address
   {
       char area_name[32];
       int house_no;
       char district[32];
   };

   struct Info
   {
       char name[32];
       int age;
       struct address address; //now this is a member
   };

   struct Info my_Info = {"Alice", 25, "Somapah Road", 8, "Upper Changi"};

   printf("Name: %s, age %d, area name %s, house number %d, district %s\n", my_Info.name,
my_Info.age, my_Info.address.area_name, my_Info.address.house_no,
my_Info.address.district);

   struct address my_Addrs = {"Another Road", 15, "Lower Changi"};
   printf("Another address %s %d %s \n", my_Addrs.area_name, my_Addrs.house_no,
my_Addrs.district);
```

And we still have the same output as above.

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Name: Alice, age 25, area name Somapah Road, house number 8, district Upper Changi
Another address Another Road 15 Lower Changi
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

The byte size of structures are roughly the sum size of its members. For example,

```
struct Vector_Int{
    int x;
    int y;
    int z;
    char name[64];
};

struct Vector_Int vector_sample;

printf("Size of Vector_Int struct is %d bytes\n", sizeof(struct Vector_Int));
printf("Size of its members are x %d bytes, y %d bytes, z %d bytes, and name %d bytes\n", sizeof(vector_sample.x), sizeof(vector_sample.y), sizeof(vector_sample.z), sizeof(vector_sample.name));
```

Results in the output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Size of Vector_Int struct is 76 bytes
Size of its members are x 4 bytes, y 4 bytes, z 4 bytes, and name 64 bytes
```

This is because each int is 4 bytes, and the char is 64 bytes : 12 + 64 = 76 bytes.

Of course you can have an array of structures. Just treat structs like a new data type that's a collection of primary and derived data types.

```
struct Info many_info[3] = {{"Alice", 25, "Somapah Road", 8, "Upper Changi"},
                            {"Bob", 22, "Somapah Road", 19, "Upper Changi"},
                            {"Michael", 30, "Another Road", 25, "East Changi"}};

for (int i = 0; i < 3; i++)
{
    printf("Name: %s, age %d, area name %s, house number %d, district %s\n", many_info[i].name, many_info[i].age, many_info[i].address.area_name, many_info[i].address.house_no, many_info[i].address.district);
}
```

Here's the output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Name: Alice, age 25, area name Somapah Road, house number 8, district Upper Changi
Name: Bob, age 22, area name Somapah Road, house number 19, district Upper Changi
Name: Michael, age 30, area name Another Road, house number 25, district East Changi
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

Tips for neater code: use `typedef`. `typedef` is a keyword used in C language to assign alternative names to existing datatypes. Its mostly used with user defined datatypes, when

names of the datatypes become slightly complicated to use in programs. So in the example below, we can *rename* `struct Info` into just `InfoData` so that things look neater:

```
typedef struct Info InfoData;
InfoData many_info[3] = {{"Alice", 25, "Somapah Road", 8, "Upper Changi"},
                         {"Bob", 22, "Somapah Road", 19, "Upper Changi"},
                         {"Michael", 30, "Another Road", 25, "East Changi"}};

for (int i = 0; i < 3; i++)
{
    printf("Name: %s, age %d, area name %s, house number %d, district %s\n",
many_info[i].name, many_info[i].age, many_info[i].address.area_name,
many_info[i].address.house_no, many_info[i].address.district);
}
```

The output is of course the same as the above.

More about Structure Size

Up above, we say the structure size is **roughly** the sum of its members, but may not be exact. The members of a struct are allocated as **contiguous block of memory.** However**,** remember that the size of an int is 32-bit, float is 32-bit and a char is 8-bit. However in many 32-bit machines, you cannot retrieve *different parts of words*. So what happens if you have 3 chars and 1 float inside a struct? The total number of "bytes" used is 1 byte each for each char and 4 bytes for a float = 7 bytes. Just like playing *tetris,* the size of the struct depends on the *order of declaration of the attributes.*

The compiler will **pad** unused bytes accordingly.

We leave it up to you to figure this out by yourself using this toy example:

```
#include <stdio.h>
#include <string.h>
/*  Below structure1 and structure2 are same.
   They differ only in member's allignment */
struct structure1
{
      int id1;
      int id2;
      char name;
      char c;
      float percentage;
};
struct structure2
{
      int id1;
      char name;
      int id2;
      char c;
      float percentage;
};
```

```c
int main()
{
   struct structure1 a;
   struct structure2 b;
   printf("size of structure1 in bytes : %d\n",
          sizeof(a));
   printf ( "\n   Address of id1        = 0x%llx", &a.id1 );
   printf ( "\n   Address of id2        = 0x%llx", &a.id2 );
   printf ( "\n   Address of name       = 0x%llx", &a.name );
   printf ( "\n   Address of c          = 0x%llx", &a.c );
   printf ( "\n   Address of percentage = 0x%llx",
                &a.percentage );
   printf("   \n\nsize of structure2 in bytes : %d\n",
                sizeof(b));
   printf ( "\n   Address of id1        = 0x%llx", &b.id1 );
   printf ( "\n   Address of name       = 0x%llx", &b.name );
   printf ( "\n   Address of id2        = 0x%llx", &b.id2 );
   printf ( "\n   Address of c          = 0x%llx", &b.c );
   printf ( "\n   Address of percentage = 0x%llx\n",
                &b.percentage );
   return 0;
}
```

The output is:

```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
size of structure1 in bytes : 16

   Address of id1        = 0x7ffeed60aa48
   Address of id2        = 0x7ffeed60aa4c
   Address of name       = 0x7ffeed60aa50
   Address of c          = 0x7ffeed60aa51
   Address of percentage = 0x7ffeed60aa54

size of structure2 in bytes : 20

   Address of id1        = 0x7ffeed60aa30
   Address of name       = 0x7ffeed60aa34
   Address of id2        = 0x7ffeed60aa38
   Address of c          = 0x7ffeed60aa3c
   Address of percentage = 0x7ffeed60aa40
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Learning Points

1. Declare arrays, modify arrays
2. Print strings, create and modify strings
3. Pointers data type: difference between addressing and content
4. Addressing in arrays and 64 bits system
5. Declaring structs, initializing and changing its values

Part 4: Loops and Iterations

Loops and iterations can simply be done in C using the for-loop and while-loop, just like any other language. There's really nothing fancy in here.

```c
float array_floats[8];
for (int i = 0; i<8; i++){
    array_floats[i] = (float) i/8;
    printf("%f, ", array_floats[i]);
}
printf("\n");

int i = 0;
while(i < 8){
    array_floats[i] += 0.5f;
    printf("%f, ", array_floats[i]);
    i ++;
}
printf("\n");

i = 0;
do{
    array_floats[i] -= 0.5f;
    printf("%f, ", array_floats[i]);
    i ++;
}while(i<8);
printf("\n");
```

Here's the output for sanity check:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
0.000000, 0.125000, 0.250000, 0.375000, 0.500000, 0.625000, 0.750000, 0.875000,
0.500000, 0.625000, 0.750000, 0.875000, 1.000000, 1.125000, 1.250000, 1.375000,
0.000000, 0.125000, 0.250000, 0.375000, 0.500000, 0.625000, 0.750000, 0.875000,
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

The only new thing is the do-while loop. Unlike the while-loop, the do-while loop executes the body **at least once**, since it doesn't perform check first. It might be handy in some

situations, but most of the time it doesn't make much difference in terms of output if you meant for your while loop to be executed at least once.

In terms of execution time, the do-while loop might be a little faster. In the code above, the check i<8 is done exactly 8 times in the do-while loop, but it is done 9 times in the while-loop.

You can also break out of the loop using the keyword `break`.

What about variable initialization inside loops? For example, compare the two:

```
for (int i = 0; i<128; i++){
    char c = i;
    printf("%c ", c);
} // c does not exist out of the for-loop scope



char c;
for (int i = 0; i<128; i++){
    c = i;
    printf("%c ", c);
}
//c exists, as 127
printf("final c: %c.\n", c); //its a space
```

The difference between the two for-loops is the char c initialization. It is reinitialized in each loop for the first one, but initialized only once in the second loop. It might seem like the second loop is *more efficient*, **but it does not make any much difference with modern compilers, since it is able to optimize the code above.** In fact, the code above is safer because the scope of each variable **is limited for within that ONE loop only.**

Learning Points

1. Three ways to perform loops: for-loop, while-loop, do-while loop
2. Do-while loops execute the body at least once because it performs checks *after*
3. `break` loops, or `continue` (same like Java, Python)
4. Scope of loops for variables initialized inside or outside the loop

Part 5: Functions

The functions in C must be *declared first* before being utilized, so that the compiler knows the return type. For example, such usage:

```c
int main(int argc, char **argv)
{
   float output = square(3.f);
   printf("Output is %f \n", output);
}

float square(float a){
   return a*a;
}
```

Results in compilation error:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % gcc cclass_part1.c -o out
cclass_part1.c:179:20: warning: implicit declaration of function 'square' is invalid in C99 [-Wimplicit-function-declaration]
    float output = square(3.f);
                   ^
cclass_part1.c:184:7: error: conflicting types for 'square'
float square(float a){
      ^
cclass_part1.c:179:20: note: previous implicit declaration is here
    float output = square(3.f);
                   ^
1 warning and 1 error generated.
```

This is because the compiler doesn't know the return type of `square`, so it will set it as an `int` by default. Then later on when you declare it as `float` return type, the error is generated.

Obviously if you try to call a function that does not exist there's the *linker* error because the compiler doesn't know where does this function comes from. For example calling this `function1()` when you haven't implemented it:

```c
   function1();
```

Results in such error:

```
Undefined symbols for architecture x86_64:
  "_function1", referenced from:
      _main in cclass_part1-b7e5ac.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

The **correct way** is to declare your function first in the header file:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define BUFFERSIZE 1024
float square(float a);
```

And then implementing it in the .c file as above.

*Otherwise, you need to implement the function **ABOVE** the main() declaration:*

```
float square(float a){
   return a*a;
}

int main(int argc, char **argv)
{
   float output = square(3.f);
   printf("Output is %f \n", output);
}
```

To be neat, it is recommended that you always declare your functions in the header file and only implement those in the .c files. Also, declare your structs in the header file.

To declare a function, you need to define the:

1. Return type
2. The type of each argument

For functions without any return type, we just define it as *void*. For example, add these to your header file:

```
// defining struct
typedef struct Vector_Int
{
   int x;
   int y;
   int z;
   char name[64];
}Vector;

void print_vector(Vector input);
```

Implement the function in your .c file:

```
void print_vector(Vector input){
   printf("{x:%d, y:%d, z:%d}\n", input.x, input.y, input.z);
}
```

And call it in the main function:

```
Vector v1 = {3,7,10};
print_vector(v1);
```

We shall have the output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
{x:3, y:7, z:10}
```

As you can see in the given examples above for functions `square` and `print_vector`, C functions can receive any primary or derived data type as argument.

The two examples of functions above receives argument by *value* and **not by reference**. To understand this, suppose we want to implement another function that's job is to zero all the members of Vector. Declare this in the header file:

```
void clear_vector(Vector input);
```

And the implementation:

```
void clear_vector(Vector input){
   input.x = 0;
   input.y = 0;
   input.z = 0;
}
```

Calling these in the main function:

```
Vector v1 = {3,7,10};
print_vector(v1);
clear_vector(v1);
print_vector(v1);
```

Results in:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
{x:3, y:7, z:10}
{x:3, y:7, z:10}
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

That is because the input vector is a new COPY of v1. Hence modifying input does NOT affect v1. In order to return the cleared vector, you need to change the return type into Vector:

```
Vector clear_vector(Vector input);
```

And the implementation:

```
Vector clear_vector(Vector input){
   input.x = 0;
   input.y = 0;
   input.z = 0;
```

```
    return input;
}
```

We call this in the main function:

```
    Vector v1 = {3,7,10};
    print_vector(v1);
    v1 = clear_vector(v1);
    print_vector(v1);
```

We have the output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
{x:3, y:7, z:10}
{x:0, y:0, z:0}
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

Now while **this works, it is NOT efficient.** We require creation and destroy of memory space during runtime. If we just want the function to *modify* the created structure or array, then we are better off by creating a function and passing its argument **by reference.** Add this method into the header file:

```
void clear_vector_byreference(Vector *input);
```

And implement it in the main function:

```
void clear_vector_byreference(Vector *input){
    input->x = 0;
    input->y = 0;
    input->z = 0;
}
```

Note how to access the member of a pointer to a struct, we can use the arrow -> instead of a dot.

And add these in the main function:

```
    Vector v2 = {31,99,21};
    print_vector(v2);
    clear_vector_byreference(&v2);
    print_vector(v2);
```

We have the similar output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
{x:3, y:7, z:10}
{x:0, y:0, z:0}
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

However, the difference becomes clear if we print the **address** of the members of input, and compare it with the address of the members of v1 and v2. Add the address print statement on the functions:

```
Vector clear_vector(Vector input){
   printf("Address of clear_vector input members: 0x%llx, 0x%llx, 0x%llx\n", &input.x,
&input.y, &input.z);
   input.x = 0;
   input.y = 0;
   input.z = 0;
   return input;
}

void clear_vector_byreference(Vector *input){
   printf("Address of clear_vector_byreference input members: 0x%llx, 0x%llx, 0x%llx\n",
&input->x, &input->y, &input->z);
   input->x = 0;
   input->y = 0;
   input->z = 0;
}
```

And call these in the main:

```
   Vector v1 = {3,7,10};
   printf("Address of v1 members: 0x%llx, 0x%llx, 0x%llx\n", &v1.x, &v1.y, &v1.z);
   print_vector(v1);
   v1 = clear_vector(v1);
   print_vector(v1);

   Vector v2 = {31,99,21};
   printf("Address of v2 members: 0x%llx, 0x%llx, 0x%llx\n", &v2.x, &v2.y, &v2.z);
   print_vector(v2);
   clear_vector_byreference(&v2);
   print_vector(v2);
```

The output is:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
Address of v1 members: 0x7ffeef03c978, 0x7ffeef03c97c, 0x7ffeef03c980
{x:3, y:7, z:10}
Address of clear_vector input members: 0x7ffeef03c820, 0x7ffeef03c824, 0x7ffeef03c828
{x:0, y:0, z:0}
Address of v2 members: 0x7ffeef03c8d8, 0x7ffeef03c8dc, 0x7ffeef03c8e0
{x:31, y:99, z:21}
Address of clear_vector_byreference input members: 0x7ffeef03c8d8, 0x7ffeef03c8dc, 0x7ffeef03c8e0
{x:0, y:0, z:0}
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

As you can see, the `clear_vector's input` members is located in totally different address as v1's. To be specific, they're created in the **stack space** of `clear_vector` function. However for `clear_vector_byreference`, this isn't the case. We save on memory read/write and stack space if we use passing arguments by reference.

When to pass arguments by value and when to pass arguments by reference?

1. Depends on the size of the arguments. For primary data types and small structs, it doens't make much difference
2. It makes a lot of difference in runtime for deeply recursive functions and large structs

Learning Points

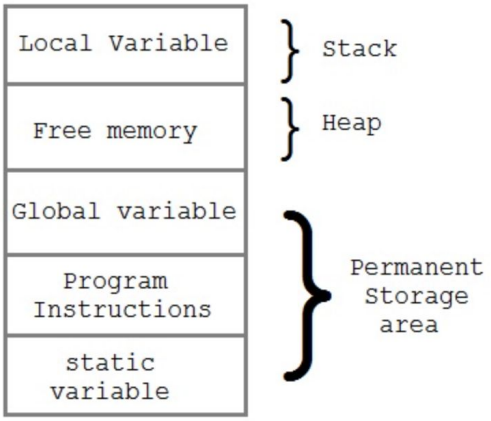
1. Declare functions and structs first in the header files, before using it in the .c files
2. Function declarations must include return type and argument types
3. Note the difference between passing argument by value vs by reference

Part 6: Dynamic Memory allocation

The final part of this tutorial is to introduce you to C's dynamic memory allocation using `malloc` or `calloc.` This memory is allocated in the process' heap (and not stack), and therefore it **persists** even after the caller function exits. You need to *explicitly* `free` the memory when it is no longer needed, otherwise it will populate your heap space unnecessarily.

`Global` variables, **`static`** variables and program instructions get their memory in **permanent storage area** whereas **local** variables are stored in a memory area called Stack.

See the figure below for basic memory space of a process:



The memory space between the stack and permanent storage area is known as the Heap area. This region is used for **dynamic memory allocation** during **execution** of the program, while the permanent storage area stays **the same**. **The size of heap keeps changing.**

The difference between global, static, and local variables:

1. Global variables are visible in the entire process (across different .c file modules). **In the other modules, we can access the global_variable declared once in any of the .c files and accessed by all other using the extern keyword. This is a more advanced knowledge and not required for our course.** If you're interested, you can read more about it here.
2. Static variables are only visible within the module (.c file) itself
3. Local variables are only visible within the function scope

The simple program below shows the difference. Add these to a header file:

```
int global_variable;

void test_global(void);
int test_static(void);
int test_local(void);
```

And these implementations to the .c file:

```
int test_static(void){
   static int static_variable = 20;
   static_variable += 1;
   return static_variable;
}

int test_local(void){
   int local_variable = 20;
   local_variable += 1;
   return local_variable;
}

void test_global(void){
   global_variable ++;
}
```

Then call them in the main function:

```
   printf("The global variable is %d \n", global_variable);
   test_global();
   printf("The global variable is now %d \n", global_variable);
   printf("The static variable is %d \n", test_static());
   printf("The static variable is %d \n", test_static());
```

```
printf("The local variable is %d \n", test_local());
printf("The local variable is %d \n", test_local());
```

We have the output:

```
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes % ./out
The global variable is 10
The global variable is now 11
The static variable is 21
The static variable is 22
The local variable is 21
The local variable is 21
natalieagus@Natalies-MacBook-Pro-2 C-Class Codes %
```

As you can see, global variables are typically defined in the header file, and can be accessed anywhere. Static variables can be defined within functions and still are visible outside of it. Calling test_static the second time skips the initialization of static_variable. Hence we have the value of 22 in the second call.

We can also declare static arrays, but we *cannot have dynamic memory allocation.* For example, as shown in the earlier part, an attempt to compile this,

```
int test_static(void){
    static int static_variable = 20;
    static_variable += 1;
    static int static_array[static_variable];
    return static_variable;
}
```

Results in the error:

```
cclass_part1.c:215:16: error: variable length array declaration cannot have 'static' storage duration
    static int static_array[static_variable];
               ^           ~~~~~~~~~~~~~~~
```

So in order to have an array which size is *dynamic*: **meaning that its size is determined later during runtime execution, we need to use `malloc` (or `calloc`):**

```
int buffersize;
    printf("Enter total number of elements: ");
    scanf("%d", &buffersize);

    //allocates memory in heap
    int *x = (int*) malloc(sizeof(char)*buffersize); //type cast it
    //print the address x is pointing to
    printf("Memory address allocated by malloc starts at 0x%llx\n", x);
```

```c
    //print the address of the pointer x
    printf("This pointer is stored at address 0x%llx\n", &x);

    // do something with the array
    for (int i = 0; i<buffersize; i++){
        x[i] = i;
    }

    printf("Enter additional number of elements: ");
    scanf("%d", &buffersize);

    //resize the array, buffersize can be smaller than original amount. The remainder is
automatically freed
    //the unused memory initially pointed by x is also automatically freed
    int *y = realloc(x, buffersize);
    printf("Memory address allocated by realloc starts at 0x%llx\n", y);
    printf("This new pointer is stored at address 0x%llx\n", &y);
    for (int i = 0; i<buffersize; i++){
        printf("Original content element %d is %d \n", i, x[i]);
        x[i] += i; //do something with the array
    }

    //free heap manually
    free(y);
```

The output is:

```
natalie_agus@Natalies-MacBook-Pro C-Class Codes % ./out
Enter total number of elements: 3
Memory address allocated by malloc starts at 0x7fdd3fd00000
This pointer is stored at address 0x7ffee23329f0
Enter additional number of elements: 5
Memory address allocated by realloc starts at 0x7fdd3fd00000
This new pointer is stored at address 0x7ffee23329e0
Original content element 0 is 0
Original content element 1 is 1
Original content element 2 is 2
Original content element 3 is 0
Original content element 4 is 0
natalie_agus@Natalies-MacBook-Pro C-Class Codes %
```

Let's digest this a little bit:

1. The program asks for user to key in array size, initially it was 3.

2. Then it allocates memory using `malloc(total_byte_size)`, hence the argument is:
    - `sizeof(int) * buffersize`
    - `malloc` returns a generic pointer to this allocated place in the heap
    - We need to type cast it

- **Note that malloc is NOT a system call. It is C standard library, and the memory is allocated in the process' local space (VM space)**

3. The address for the array is initialized at `0x7fdd3fd00000`. The pointer to this address is stored at `0x7ffee23329f0`.

4. Afterwhich, we **resize** the array into size 5 using `realloc(original_pointer, new_size)`:
   - If `new_size` is smaller than original size, then the remainder is automatically freed
   - Otherwise, `realloc` will increase the size of the memory allocated for the array
   - The new array location may or may not overlap the old array
   - `realloc` migrates the old value of the array to the new one with the new size
   - The example above shows that the new array location starts at the same place as the old one, that is `0x7fdd3fd00000`, but it may not be the same as well depending on whether there's enough space in the heap.

5. It seems like the two new locations at index 3 and 4 has initial values of 0. **Note that this is NOT always the case for `malloc`.** `Malloc` can initialize the array with garbage values as well. You might want to use `calloc(number, sizeof(type))` for auto initialization to zero. Otherwise with malloc you need to loop through the array after initialization.
6. Note that the memory block allocated by the malloc or calloc MUST BE EXPLICITLY freed using `free(pointer)`. Otherwise, your program might run out of heap space.

The nice thing about malloc is that it persists outside the scope of the calling function. As a demonstration, add this declaration in your header file:

```
int* test_malloc(int size_array);
```

Add this implementation in the .c file:

```
int* test_malloc(int size_array){
   int *x_local = malloc(sizeof(int)*size_array);
   for (int i = 0; i<size_array; i++){
       x_local[i] = i*i;
   }
   printf("Local pointer is at address 0x%llx\n", &x_local);
   printf("Pointer is pointing to address 0x%llx \n", x_local);
   return x_local;
}
```

Then call the function in your main file:

```
   int *pointer = test_malloc(10);
   printf("Returned pointer is at address 0x%llx \n", &pointer);
```

```
    printf("Pointer is pointing to address 0x%llx \n", pointer);
    // test print content
    for (int i = 0; i<10; i++){
        printf("%d ", pointer[i]);
    }
    printf("\n");

    //free the memory allocated
    free(pointer);
```

The output is:

```
natalie_agus@Natalies-MacBook-Pro C-Class Codes % ./out
Local pointer is at address 0x7ffee9f0c9c0
Pointer is pointing to address 0x7fa41e402670
Returned pointer is at address 0x7ffee9f0c9f8
Pointer is pointing to address 0x7fa41e402670
0 1 4 9 16 25 36 49 64 81
natalie_agus@Natalies-MacBook-Pro C-Class Codes %
```

Notice how the array is initialized inside the function test_malloc. However, in the main function, we can still print out its content and the array persists. It is located at address starting at `0x77fa41e402670`. The *address of the pointer pointing to* `0x77fa41e402670` varies (was `0x7ffee9f0c9c0`, then changed to `0x7ffee9f0c9f8`), as it *was the local variable* inside `test_malloc`.

Another common way of using malloc is to **pass the pointer (returned by malloc)** to another function to modify. This is very similar to the example in this part (pass by reference).

Declare this function in the header file:

```
void modify_array(int* array, int array_size);
```

Implement this in .c file:

```
void modify_array(int* array, int array_size){
    for (int i = 0; i<array_size; i++){
        array[i] += i;
}
```

And call it in the main function:

```
    int buffersize;
    printf("Enter total number of elements: ");
    scanf("%d", &buffersize);

    //allocates memory in heap
```

```c
int *x = (int*) malloc(sizeof(char)*buffersize); //type cast it

//initialize to some value
printf("The original array value is : ");
for (int i = 0; i<buffersize; i++){
    x[i] = i;
    printf("%d ", x[i]);
}
printf("\n");

//pass it to the function to modofy
modify_array(x, buffersize);

//print its content
printf("The new array value is : ");
for (int i = 0; i<buffersize; i++){
     printf("%d ", x[i]);
}
printf("\n");

//free it
free(x);
```

The output is as expected, where the array is modified by the function:

```
natalie_agus@Natalies-MacBook-Pro C-Class Codes % ./out
Enter total number of elements: 5
The original array value is : 0 1 2 3 4
The new array value is : 0 2 4 6 8
natalie_agus@Natalies-MacBook-Pro C-Class Codes %
```

Learning Points

- Dynamically allocate memory using malloc or calloc
- Resize the memory using realloc during runtime
- Difference between static memory vs heap vs stack
- Freeing memory after usage
- Scoping between functions

Summary

Congratulations. You have completed the basic training for C. Next, we will explore more advanced topics such as function pointers, making various system calls in C, error handling, File I/O, process control, as well as inter-process communication means. If you'd like to test your knowledge up until now, head to e-dimension and do the quiz (Part 1 and Part 2). The grade is not going to be computed for your overall grade.