C Practise and Tasks

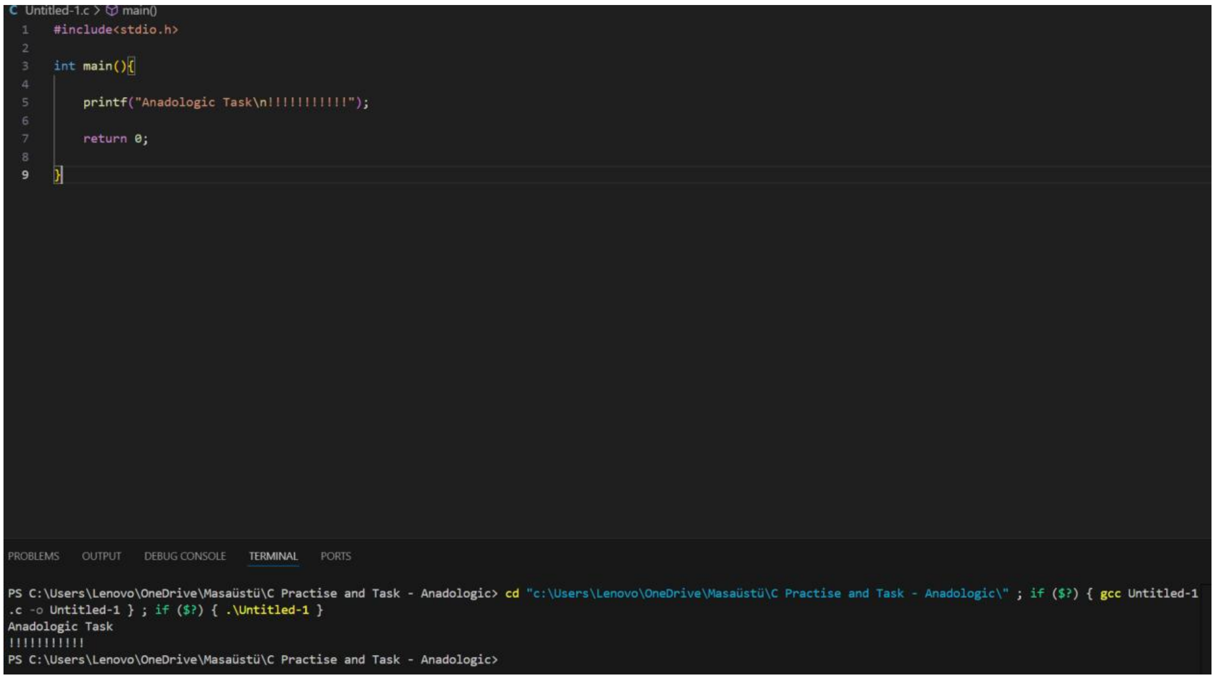
My First task is download visual studio and compile basic codes.,

1- Firstly, I downloaded visual studio code on https://code.visualstudio.com/download website.

2- Secondly, I added extension:

- C / C++ extension published by Microsoft.
- Code runner published by Jun Han.
- Then, I solved and fixed couple problems.

I compiled my first code on visual studio



```c
#include<stdio.h>

int main(){

    printf("Anadologic Task\n!!!!!!!!!!!");

    return 0;

}
```

```
PS C:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic> cd "c:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic\" ; if ($?) { gcc Untitled-1
.c -o Untitled-1 } ; if ($?) { .\Untitled-1 }
Anadologic Task
!!!!!!!!!!!
PS C:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic>
```

```c
#include<stdio.h>

int main(){

    printf("Anadologic Task\n!!!!!!!!!!!");

    return 0;

}
```

Memory In C Briefly

We can separate memory two parts that stack and heap.

1-Stack Memory

Quick, short-term memory for small, temporary data.

Purpose: Used for storing local variables and function call data.

Characteristics:

- Automatically managed: Memory is allocated and freed by the system when variables go in and out of scope.
- Fast but has a limited size.
- Temporary storage: Data is erased when the function exits.

2-Heap Memory

Flexible, long-term memory for big or dynamic data.

Purpose: Used for dynamic memory allocation (manually managed memory).

Characteristics:

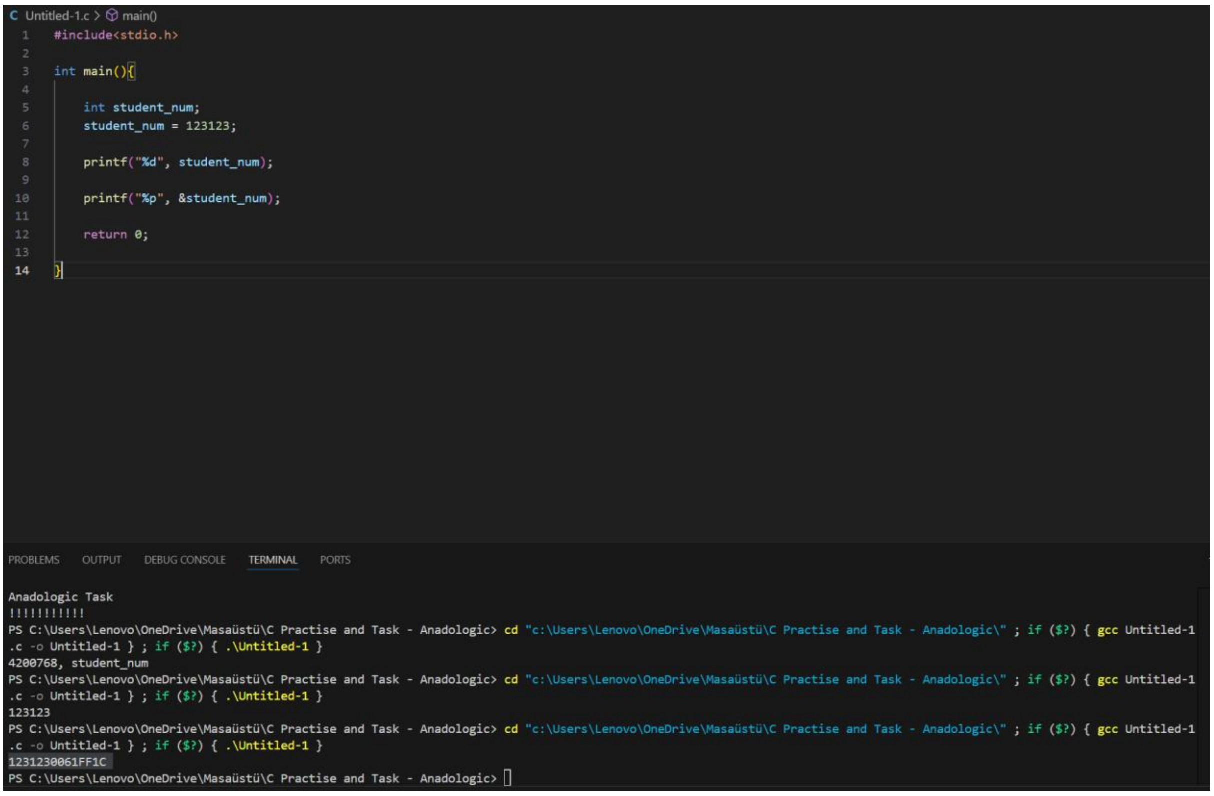
- Slower than stack but can handle large amounts of data.
- Memory persists until freed by the programmer.

In C, we have to manage memory ourself. Properly managing the computer memory optimizes the performance of the program, so it is useful that you know how to release memory when it is no longer required and only use as little as necessary for the task. There are two crucial topics for memory management. These are "Memory Address" and "Pointers"

Memory Address

When a variable is created in C, a memory address is assigned to the variable. The memory address is the location of where the variable is stored on the computer. When we assign a value to the variable, it is stored in this memory address.

To access it, use the reference operator (&), and the result represents where the variable is stored.



```c
#include<stdio.h>

int main(){

    int student_num;
    student_num = 123123;

    printf("%d", student_num);

    printf("%p", &student_num);

    return 0;

}
```

```
Anadologic Task
!!!!!!!!!!!
PS C:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic> cd "c:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic\" ; if ($?) { gcc Untitled-1
.c -o Untitled-1 } ; if ($?) { .\Untitled-1 }
4200768, student_num
PS C:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic> cd "c:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic\" ; if ($?) { gcc Untitled-1
.c -o Untitled-1 } ; if ($?) { .\Untitled-1 }
123123
PS C:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic> cd "c:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic\" ; if ($?) { gcc Untitled-1
.c -o Untitled-1 } ; if ($?) { .\Untitled-1 }
1231230061FF1C
PS C:\Users\Lenovo\OneDrive\Masaüstü\C Practise and Task - Anadologic>
```

Input

```c
#include<stdio.h>

int main(){

    int student_num;
    student_num = 123123;

    printf("%d", student_num);

    printf("%p", &student_num);

    return 0;

}
```

Output

```
123123

0061FF1C //address
```

“&student_num” often called a “pointer”. A pointer basically stores the memory address of a variable as it’s value. To print pointer values, we use the (%p) format specier.

Why it’s useful to know the memory address?,

Pointers are significant in C, because they allow us the to manipulate the data in **computer’s memory.** (This can reduce the code and improve the performance.).

Pointers

A **pointer** is a variable that **stores** the **memory address** of another variable as its value. A **pointer variable points** to a **data type** (like int) of the same type, and is created with the (*) operator.

```c
#include<stdio.h>

int main(){

    int student_num;
    student_num = 123123; //an int variable
    int* ptr = &student_num; //A pointer variable, with the name "ptr" that stores the address of student_num

    //Output the value of Student Number
    printf("%d\n",student_num);

    //Output the memory address of student_num
    printf("%p\n",&student_num);

    //Output the memory address of student_num with the pointer
    printf("%p\n",ptr);

    return 0;

}
```

Output

```
123123

0061FF18

0061FF18
```

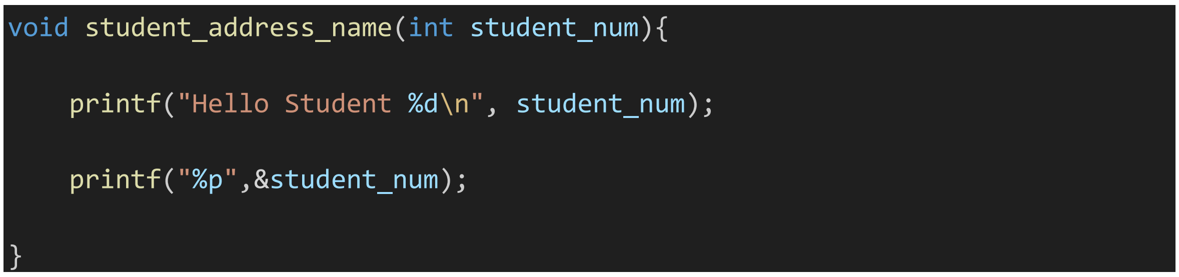
C Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: **Define the code once, and use it many times.**

Call A Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called. To call a function, write the function's name followed by

```
#include<stdio.h>

void student_adress_name(){} // Declare the function

int main(){

    int student_num;
    student_num = 123123; //an int variable

    student_address_name(student_num);

    return 0;

}
```



```
void student_address_name(int student_num){

    printf("Hello Student %d\n", student_num);

    printf("%p",&student_num);

}
```

Parameters and Arguments

Parameters act as variables inside the function. Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

Return Values

The “void” keyword indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as int, char or float etc.) instead of void, and use the “return” keyword inside the function.

Local Scope

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function. A local variable cannot be used outside the function it belongs to.

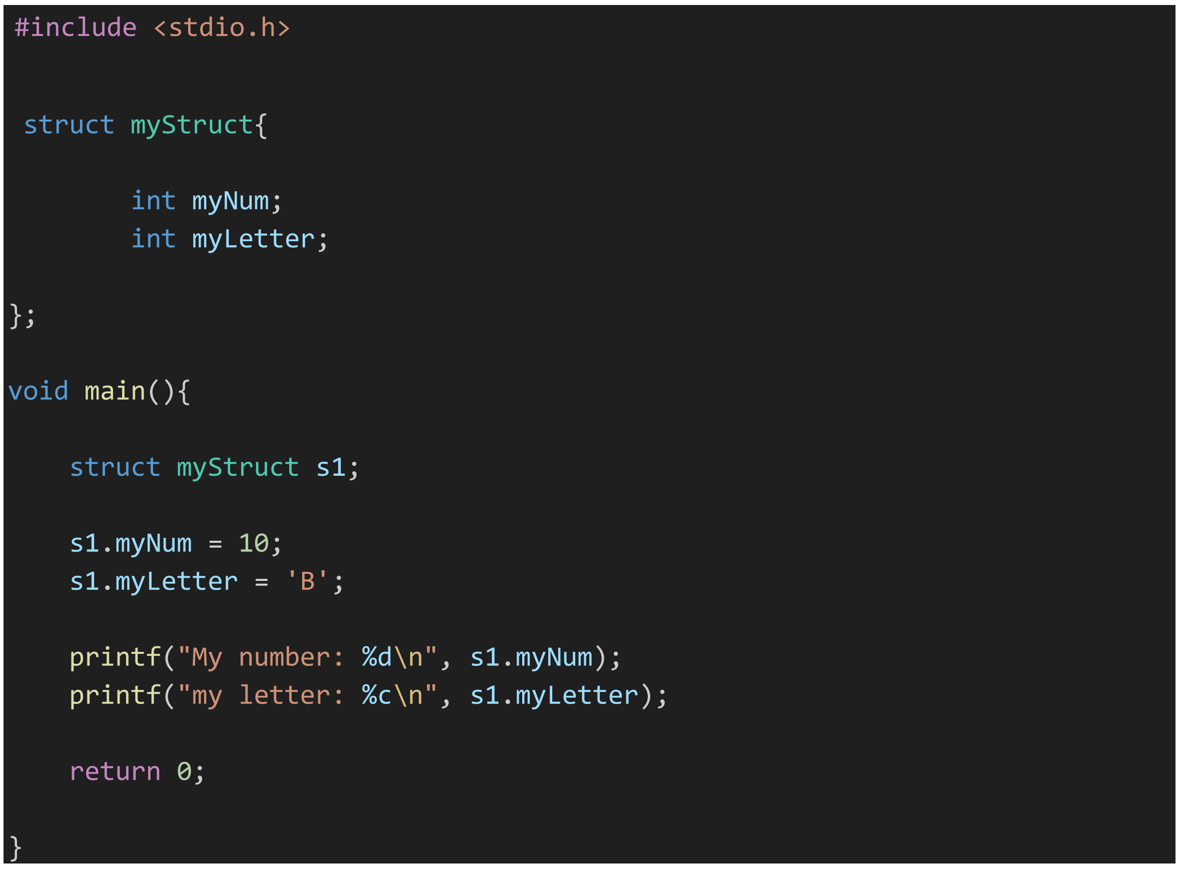
Global Scope

A variable created outside of a function, is called a global variable and belongs to the *global scope*. Global variables are available from within any scope, global and local.

Structures

A structure can contain many different data types (int, float, char, etc.).

Create a Structure



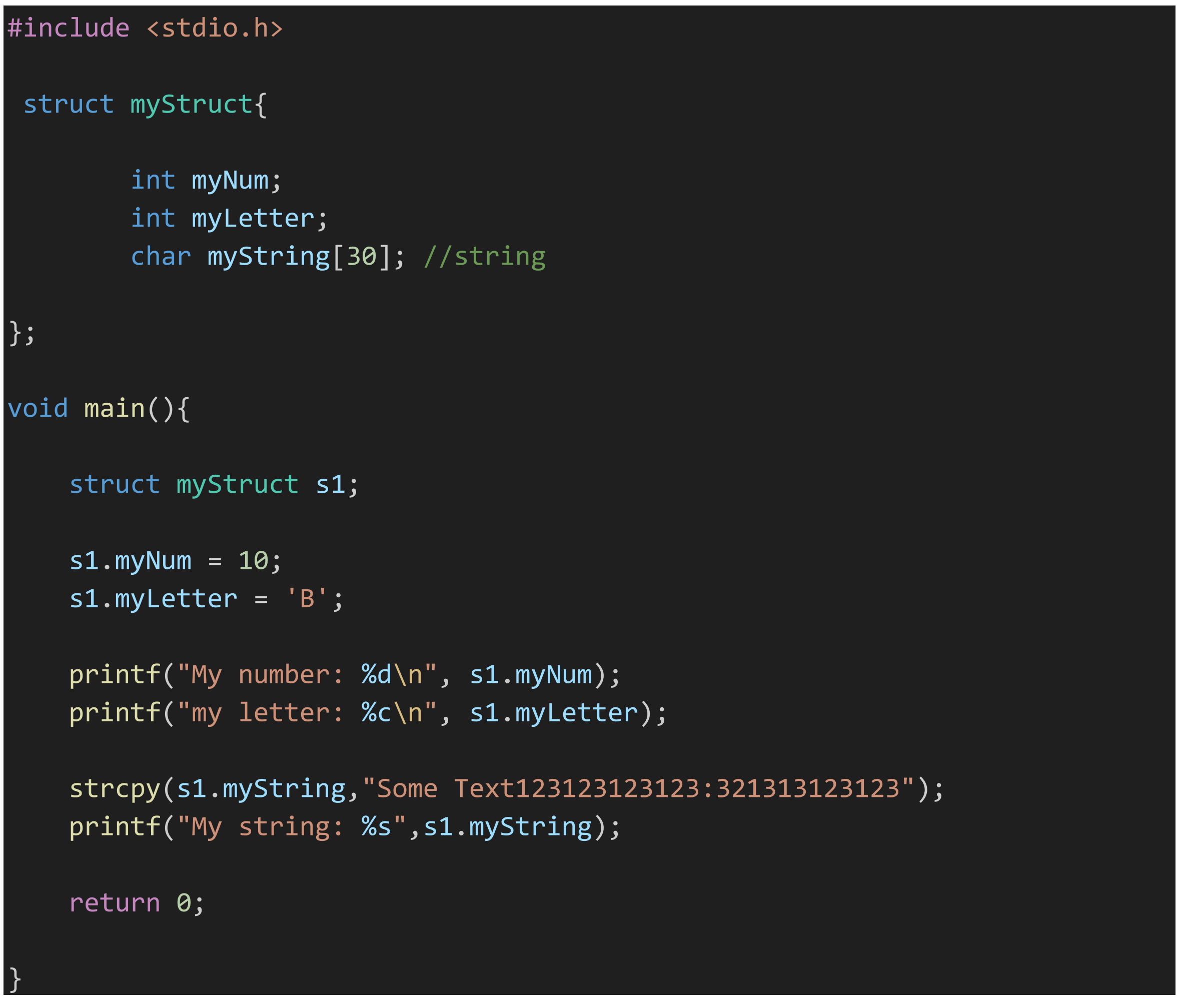
```c
#include <stdio.h>


 struct myStruct{

        int myNum;
        int myLetter;

};

void main(){

    struct myStruct s1;

    s1.myNum = 10;
    s1.myLetter = 'B';

    printf("My number: %d\n", s1.myNum);
    printf("my letter: %c\n", s1.myLetter);

    return 0;

}
```

We can easily create a multiple structure variables with different values, using just one structure.

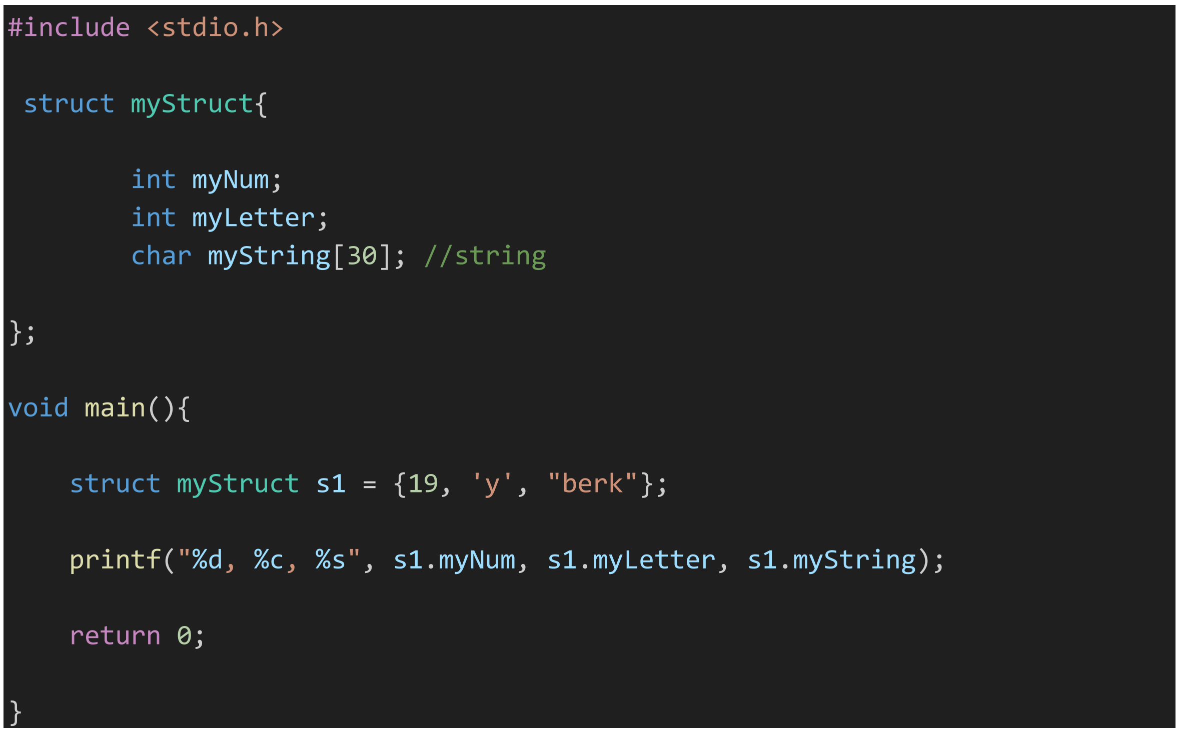
What About Strings in Structures?

Remember that strings in C are actually an array of characters, and unfortunately, you can't assign a value to an array.

We can use the strcpy() fuction and assign the value to s1.myString, like this.



```c
#include <stdio.h>

 struct myStruct{

        int myNum;
        int myLetter;
        char myString[30]; //string

};

void main(){

    struct myStruct s1;

    s1.myNum = 10;
    s1.myLetter = 'B';

    printf("My number: %d\n", s1.myNum);
    printf("my letter: %c\n", s1.myLetter);

    strcpy(s1.myString,"Some Text123123123123:321313123123");
    printf("My string: %s",s1.myString);

    return 0;

}
```



```c
#include <stdio.h>

 struct myStruct{

        int myNum;
        int myLetter;
        char myString[30]; //string

};

void main(){

    struct myStruct s1 = {19, 'y', "berk"};

    printf("%d, %c, %s", s1.myNum, s1.myLetter, s1.myString);

    return 0;

}
```

C Enums

An enum is a special type that represent a group of constant (unchanable values). To create a enum, use the enum keyword.

C Memory

Understanding how memory work in C is very important. When you create a basic variable, C will automatically reserve space for that variable. If a program uses excessive or unnecessary memory, it can lead to slow performance and inefficiency.

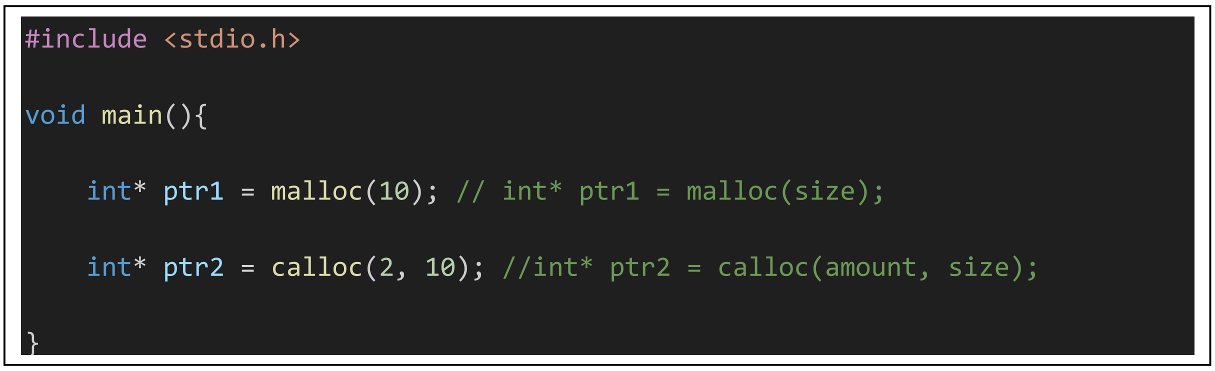
Static Memory

Static memory is memory that reserved for variables **before program runs**. Allocation of static memory is also known as compile time memory allocation. C automatically allocates memory for every variable. As a result of this,

Dynamic Memory

Dynamic memory is memory that is allocated after the program starts running. It's a **runtime memory** allocation. You can determine how much memory you need, and allocate it. Dynamic memory does not relate to a variable, it can only be accessed with pointers. To allocate dynamic memory, you can use the "malloc()" or "calloc()" functions that allocate some memory and return a pointer to it's address.

- "malloc()": Function has one parameter, size, which specifies how much memory to allocate, measured in bytes.
- "calloc()": Function which has two parameters.



```
#include <stdio.h>

void main(){

    int* ptr1 = malloc(10); // int* ptr1 = malloc(size);

    int* ptr2 = calloc(2, 10); //int* ptr2 = calloc(amount, size);

}
```

```c
#include <stdio.h>

void main(){

int *Ages;
int numAges = 5;
Ages = calloc(numAges, sizeof(*Ages));
printf("%d", numAges * sizeof(*Ages)); // 20 bytes

}
```

Stack Memory

Stack memory is a type of dynamic memory which is reserved for variables that are **declared inside the function**. Variables declared inside the function use stack memory rather than static memory.

C Access Memory

Dynamic memory behaves like an array**.** To access an element in dynamic memory, refer to it's index number.

```c
#include <stdio.h>

// Read from and write to dynamic memory:

void main(){

int *ptr;

ptr = calloc(5, sizeof(*ptr));

//Let's write the memory

*ptr = 2;

ptr[1] = 4; ptr[2]= 6;


//Read from memory
printf("%d\n", *ptr);
printf("%d, %d, %d, %d", ptr[1], ptr[2], ptr[3], ptr[4]);

return 0;

}
```

Output

```
2,

4, 6, 0, 0
```

C Reallocate Memory

If the amount of memory you reserved is not enough, you can *reallocate* it to make it larger.

```c
int *ptr2 = realloc(ptr1, size);
```

```c
#include <stdio.h>
// Increase the size of allocated memory

void main(){

int *ptr1, *ptr2, size;

//Allocate memory for four integer
size = 4 * sizeof(*ptr1);
ptr1 = malloc(size);
printf("%d bytes allocated at address %p \n", size, ptr1);

//Resize the memory to hold ten integer
size = 10 * sizeof(*ptr1);
ptr2 = realloc(ptr1, size);
printf("%d bytes reallocated at address %p \n", size, ptr2);

return 0;
}
```

Output

```
16 bytes allocated at address 00BB2F38

40 bytes reallocated at address 00BB2F38
```

C Deallocate Memory

When you no longer need a block of memory you should deallocate it. Deallocation is also referred to as “freeing” the memory. Dynamic memory stays reserved until it is deallocated or until the program ends.

Once the memory is deallocated it can be used by other programs.

Free Memory

To deallocate memory, use the “free()” function.

```c
#include <stdio.h>
// Increase the size of allocated memory

void main(){

int *ptr;
ptr = malloc(sizeof(*ptr));

free(ptr);
ptr = NULL;

return 0;
}
```