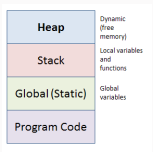
# DAY 01

## Memory allocation, pointers to members, references, switch statement

## Memory allocation :

When a program is run, the hard disk is used to load the program and its associated data into RAM. The CPU then executes the program's instructions, using the data stored in RAM to perform calculations and process information. As the program runs, data is exchanged between RAM and the CPU as needed, with the CPU reading and writing data from and to RAM. When the program finishes running, any data stored in RAM is discarded, and the CPU is free to run other programs.




Dynamically allocating memory refers to the process of allocating memory during runtime, rather than at compile time. In other words, memory is allocated dynamically when a program is running, based on the needs of the program.

Dynamically allocating memory is useful when you don't know in advance how much memory your program will need, or when the size of the required memory depends on user input or other dynamic factors.

```
int* ptr = new int;                delete ptr;

int* arr = new int[10];            delete[] arr;
```

## Stack vs heap :

The stack is used to store automatic variables, which are variables that are created and destroyed automatically as the program runs. These variables are created when a function is called and destroyed when the function returns. The stack is a limited resource, and its size is determined at compile time. If the stack overflows, it can cause a program to crash.

The heap, on the other hand, is used for dynamic memory allocation. Memory is allocated from the heap using the "new" operator, and it must be explicitly deallocated using the "delete" operator. The heap is a larger resource than the stack and can be used to allocate larger amounts of memory. However, if memory is not properly deallocated, it can lead to memory leaks and eventually cause the program to run out of memory.

It's important to note that while the stack and heap are different in their usage, they both ultimately allocate memory from the same pool of memory in the computer's RAM. The main difference is in how the memory is managed and accessed.

using the stack for storing variables and data can cause some problems, mainly related to memory usage and program performance. Here are some common issues with using the stack in C++:

1. Stack overflow: When too much data is pushed onto the stack, it can cause a stack overflow error. This occurs when there is not enough space in the stack to hold all the variables and data.
2. Limited stack size: The stack has a limited size, and if too much data is stored on the stack, it can cause the program to crash or become unstable.
3. Automatic deallocation: When a variable is declared on the stack, it is automatically deallocated when it goes out of scope. This can cause problems if the variable is needed outside of the scope in which it was declared.

```
int* returnPointer() {
  int stackVar = 10;                    // Allocate a variable on the stack
  return &stackVar;                     // Return a pointer to the stack variable
}

int main() {
  int* ptr = returnPointer();                                  // Get a pointer to the stack variable
  cout << "Stack variable value: " << *ptr << endl;            // Access the stack variable
  return 0;
}
```

```
main.cpp:6:10: warning: address of local variable 'stackVar' returned [-Wreturn-local-addr]  return &stackVar; // Return a pointer to the stack variable
```

```
int* returnPointer() {
  int* heapVar = new int(10);           // Allocate a variable on the heap
  return heapVar;                       // Return a pointer to the heap variable
}

int main() {
  int* ptr = returnPointer();                                  // Get a pointer to the heap variable
  cout << "Heap variable value: " << *ptr << endl;             // Access thhe heap variable
return 0;
}
```

## Reference & Pointers :

### Reference :

Reference is a type of variable that refers to an existing object or value. It is essentially an alias for another variable or value, and is often used as a way to pass arguments to functions by reference, allowing the function to modify the original value of the argument.

```
- int x = 10;          int &r = x;          x = 15; or r = 19;
```

** References in C++ are useful for several reasons:

1. Function parameters: When passing large objects to functions, passing by reference can be more efficient than passing by value. This is because passing by reference avoids making a copy of the object, and instead allows the function to directly modify the original object. This can be particularly useful when working with large objects or when modifying an object is a primary goal of the function.

2. Avoiding unnecessary copies: When returning large objects from functions, returning by reference can be more efficient than returning by value. This is because returning by reference avoids making a copy of the object, and instead allows the calling function to work directly with the original object. This can be particularly useful when working with large objects or when modifying an object is a primary goal of the function.

3. Aliasing: References can be used to create multiple names for the same object, allowing for more flexible and expressive code. This can make it easier to write clean, concise code that is easier to read and maintain.

### Pointer :

Pointer is a variable that stores the memory address of another variable or object. Pointers allow you to work with memory directly, giving you more control over your program's behavior and memory usage.

```
- int x = 10;          int *r = &x;          x = 15; or *r = 19;
```

** Dynamic memory allocation: Pointers are used to allocate memory dynamically at runtime using the new operator. This allows programs to allocate memory as needed and work with data structures of arbitrary size.

**---------- Differences between pointers and references ----------**

- **Pointer:**
**- A pointer can be initialized to any value anytime after it is declared.**
**- A pointer can be assigned to point to a NULL value.- Pointers need to be dereferenced with a *.**
**- A pointer can be changed to point to any variable of the same type.**

- **Reference :**
**- A reference must be initialized when it is declared.**
**- References cannot be NULL.- References can be used ,simply, by name.**
**- Once a reference is initialized to a variable, it cannot be changed to refer to a variable object.**

## Switch Statement :

The switch statement in C++ is a control flow statement that allows a program to execute different blocks of code depending on the value of an expression

```
switch (expression) {
  case value1:
    // code to be executed if expression == value1
    break;
  case value2:
    // code to be executed if expression == value2
    break;
  ...
  default:
    // code to be executed if none of the above cases are true
    break;
}
```

The expression in the switch statement can be of any integral type (such as int, char, or enum) or an expression that evaluates to an integral type. The case labels specify the different possible values of the expression, and the code to be executed for each case is placed in a block following the label. The default label is optional and specifies the code to be executed if none of the other cases match.

## Pointers to Members :

Pointer to a member is a way of referring to a member of a class or struct through a pointer. It allows you to store and manipulate the address of a specific member variable or member function of a class.

**Pointers to Data Members:**
° A pointer to a data member stores the address of a member variable within a class or struct. To declare a pointer to a data member, use the scope resolution operator :: to specify the class or struct name, followed by the member variable name, and then use the * operator to declare a pointer to that member variable.

```
class MyClass {
public:
  int myVar;
  void myFunc() {}
};

int MyClass::* ptr = &MyClass::myVar;
```

```
MyClass obj;

obj.*ptr = 42;          // set the value of myVar through the pointer
(obj.*ptr)++;           // increment the value of myVar through the pointer
(obj.*ptr)();           // call myFunc through the pointer
```

**Pointers to Member Functions:**
° A pointer to a member function stores the address of a member function within a class or struct. To declare a pointer to a member function, use the same syntax as declaring a regular function pointer, but with an additional class or struct name followed by the scope resolution operator :: and the function name.

```
class MyClass {
public:
  void myFunc() {}
};

void (MyClass::* ptr)() = &MyClass::myFunc;
```

### Exemple :

```
class MyClass {
public:
  void myPublicFunc() {
    std::cout << "Hello from myPublicFunc!" << std::endl;
  }
};
void (MyClass::* ptr)() = &MyClass::myPublicFunc;
int main() {
  MyClass obj;
  (obj.*ptr)();
  return 0;
}
```

### Output :

```
Hello from myPublicFunc!
```