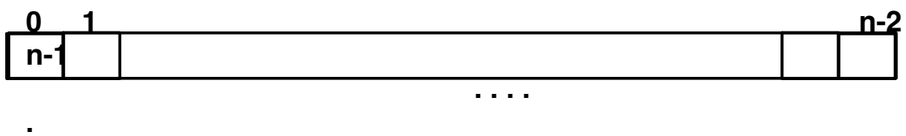
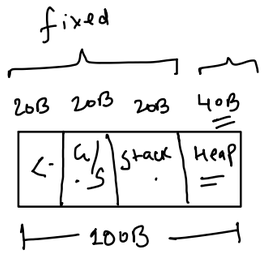
Introduction to Dynamic Memory Allocation

- We can visualize the system memory **(RAM)** as a sequence of memory cells such that each memory cell is of **1B (8 bits)** and is addressable.



- When we run a C++ program, a portion of system memory is allocated for program execution which is known as **application memory.**

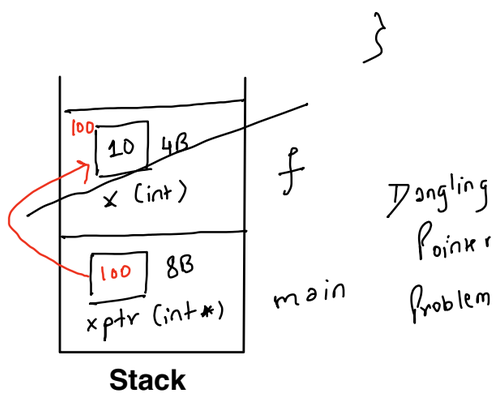
- The application memory is divided into **four** segments
  - Code or Text : to store program instructions
  - Global/Static : to store global and static variables
  - Stack : to store local variables
  - Heap or Dynamic *



Stack Segment

- The memory allocated for stack segment of application memory is **fixed**.
- The size of the stack frame for a function must be known at compile time.
- The process of allocation and deallocation of memory is handled by **OS***.

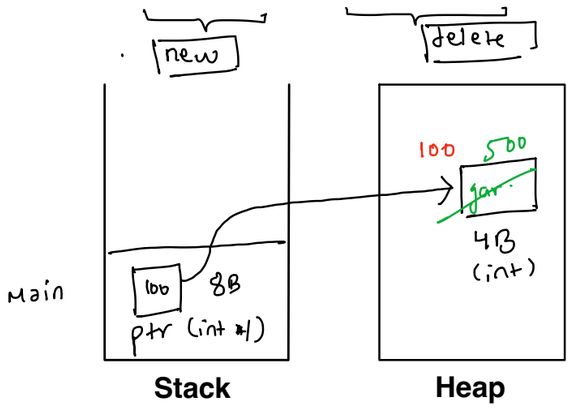
```
int* f() {
    int x = 10;
    return &x;
}

int main() {
    int* xptr = f();
    cout << *xptr << endl;
    return 0;
}
```



Heap or Dynamic Memory

- The memory allocated for the heap segment is dynamic in nature.
- The programmer handles the process memory allocation and deallocation.

```
main() {
    int * ptr = new int;
    *ptr = 500;
    cout << *ptr; //500
}
```



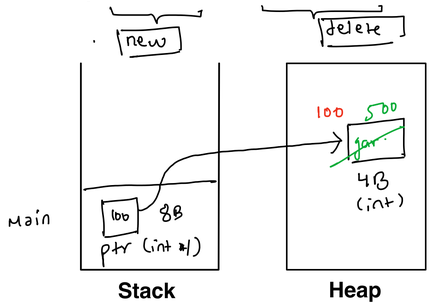

```
type * ptr = new type;
```

Stack

Heap or Dynamic Memory

- The memory allocated for the heap segment is dynamic in nature.
- The programmer handles the process memory allocation and deallocation.

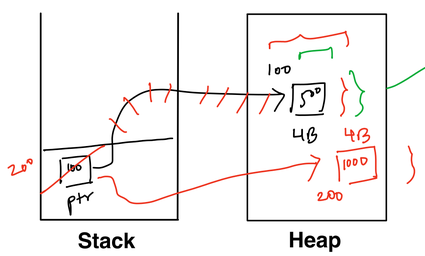
```
main() {

    int * ptr = new int;
      *ptr = 500;
    cout << *ptr; //500
}
```



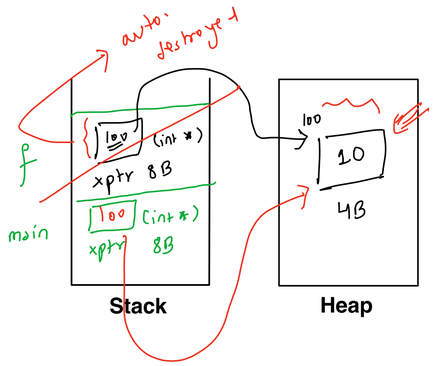
type * ptr = new type ;

Memory Leak

```
main() {
    int * ptr = new int;
      *ptr = 500;
    delete ptr; // do this to avoid memory leak
    { ptr = new int;
      *ptr = 1000; }
}
```




```
int* f() {
    int* xptr = new int;
    *xptr = 10;
    return xptr;
}

int main() {
    int* xptr = f();
}
    cout << *xptr; // 10
```

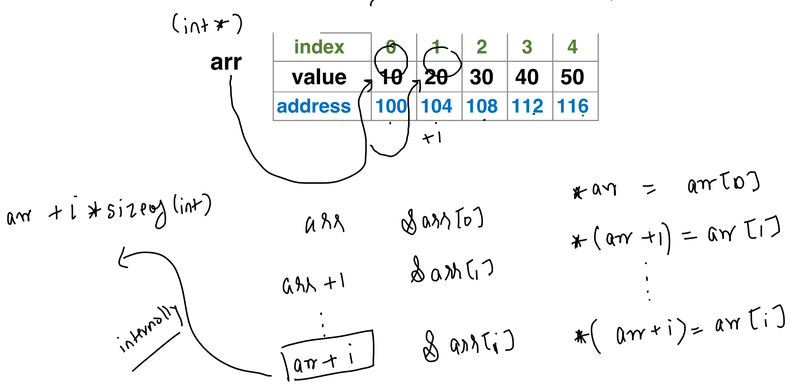


1D arrays on the Stack or Static Memory

An array is a **linear** data structure, referred by a single name, which is used to store a **sequence** of values of the **same type**.

```
// syntax for array declaration

type name[size];
```

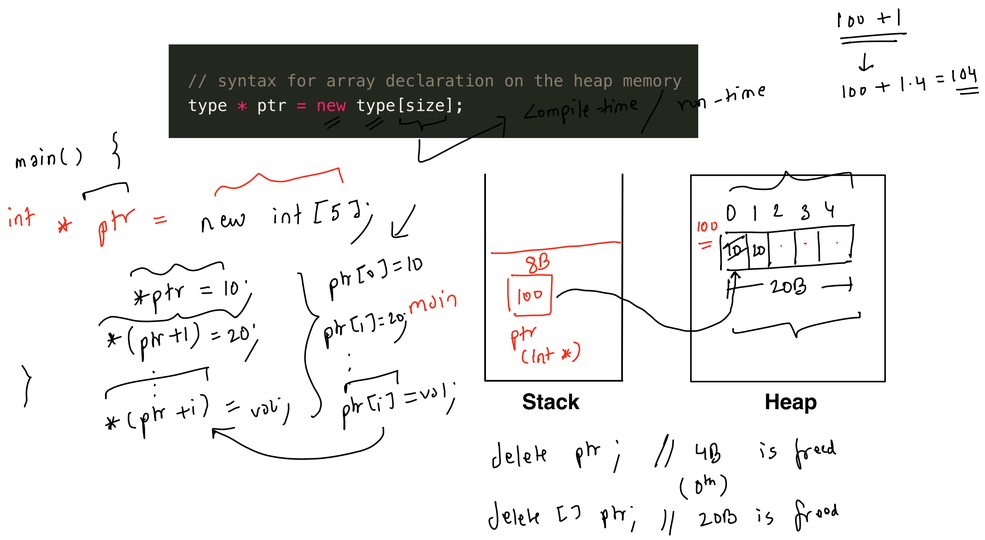
In C++, we can think of **name** of an array as a **pointer** to the element at the **0th** index.



1D arrays on the Heap or Dynamic Memory

```
// syntax for array declaration on the heap memory
type * ptr = new type[size];
```

100 + 1 → 100 + 1·4 = 104



By default, when we declare an array on the heap memory, it contains **garbage** value.

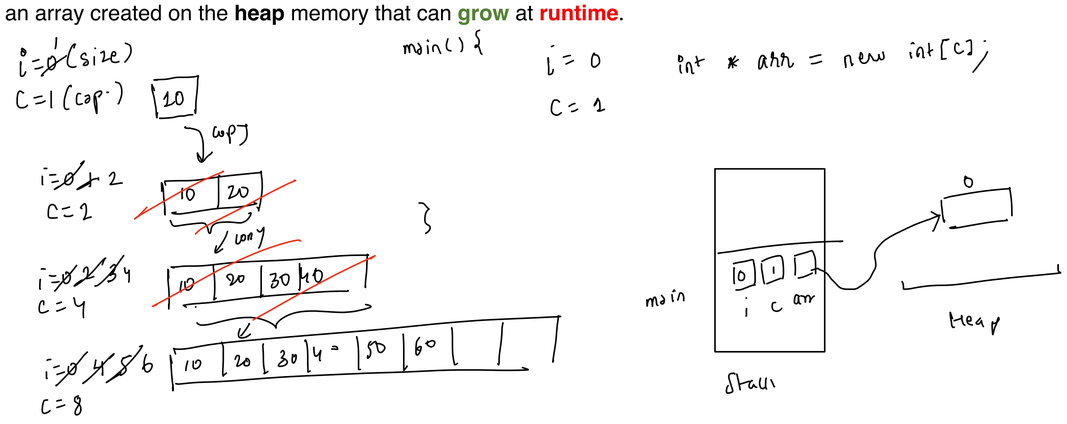
```
int* arr = new int[5]{10, 20, 30, 40, 50};
```

During the **initialization** of an array created on the heap memory, specifying the size of the array is **optional**. Also, the size of the **initializer list** should not exceed the array size.

To access elements of a 1D array created on heap or dynamic memory, we can use the same syntax that is used to access elements of a 1D array created on the stack memory.

Dynamic Array (Vector)

It an array created on the **heap** memory that can **grow** at **runtime**.



Static Allocation

2D arrays on the Stack or Static Memory

A 2D-array is an **array of 1D arrays**, referred by a single name, which is used to store a **sequence** of values of the **same type** and can be visualized as a **matrix**.

```
// syntax for a 2D-array declaration

type name[rows][cols];
```

compile-time

By default, when we declare an array on the heap memory, it contains **garbage** value.

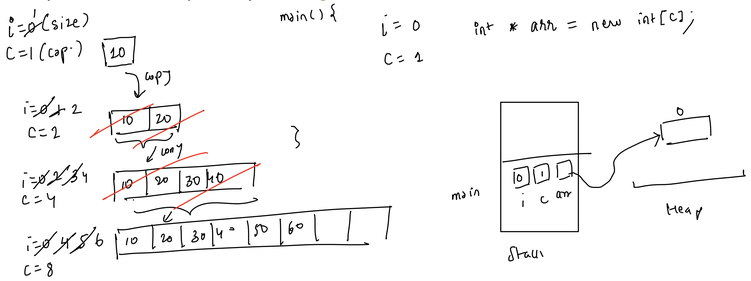
```
int* arr = new int[5]{10, 20, 30, 40, 50};
```

During the **initialization** of an array created on the heap memory, specifying the size of the array is **optional**. Also, the size of the **initializer list** should not exceed the array size.

To access elements of a 1D array created on heap or dynamic memory, we can use the same syntax that is used to access elements of a 1D array created on the stack memory.

Dynamic Array (Vector)

It an array created on the **heap** memory that can **grow** at **runtime**.



Static Allocation

2D arrays on the Stack or Static Memory

A 2D-array is an **array of 1D arrays**, referred by a single name, which is used to store a **sequence** of values of the **same type** and can be visualized as a **matrix**.

```
// syntax for a 2D-array declaration

type name[rows][cols];
```

compile-time

arr[i][j] → *(arr + i) → *(arr[i] + j)

) 2D arrays on the Heap or Dynamic Memory

m × n → <b1n

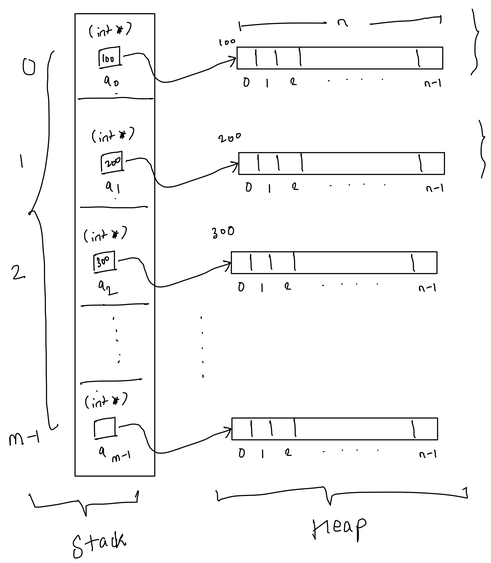
rows

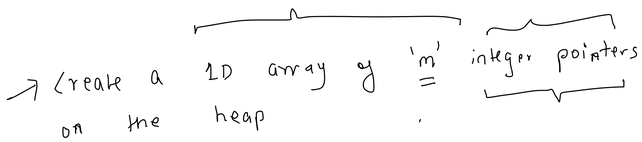
A 2D-array is an **array of 1D arrays.**

<read an array of size 'm' whose each element is a 1D array of size 'n'

```
int * a0 = new int[n];
int * a1 = new int[n];
int * a2 = new int[n];
        .
        .
        .
int * am-1 = new int[n];
```

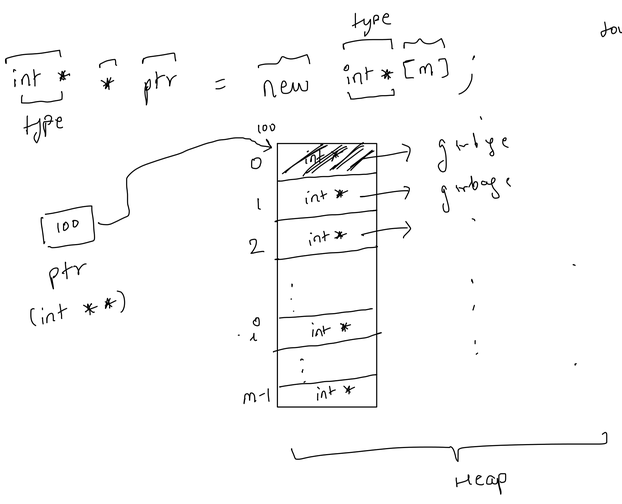
m times






```
int * ptr = new int[5]
char * ptr = new char[
double * ptr = new double[
```

```
int * * ptr = new int*[m];
type
```

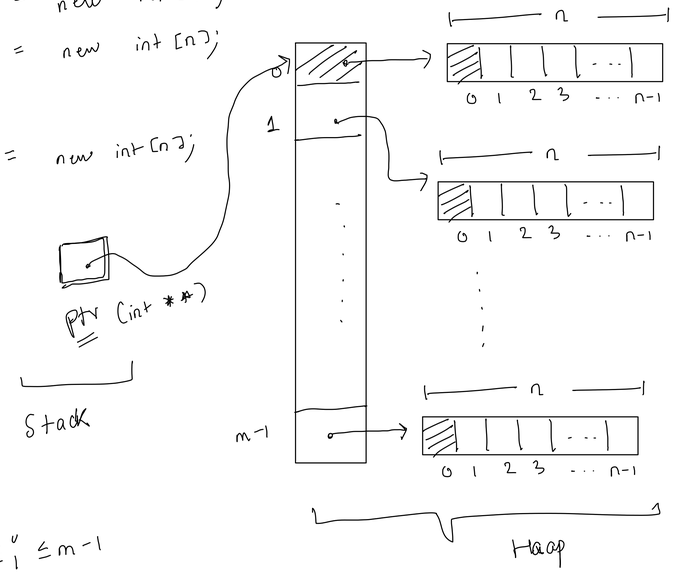


ptr[0] = new int[n];

ptr[1] = new int[n];

⋮

ptr[m-1] = new int[n];



$0 \le i \le m-1$

ptr[i] = new int[n];

ptr[i][j]

ptr[i][j]

internally

(ptr + i)

ptr + i * sizeof(int *)

deref

*(ptr + i)

ptr[i] + j * sizeof(int)

(ptr[i] + j)

deref

*(ptr[i] + j)

ptr[i][j]