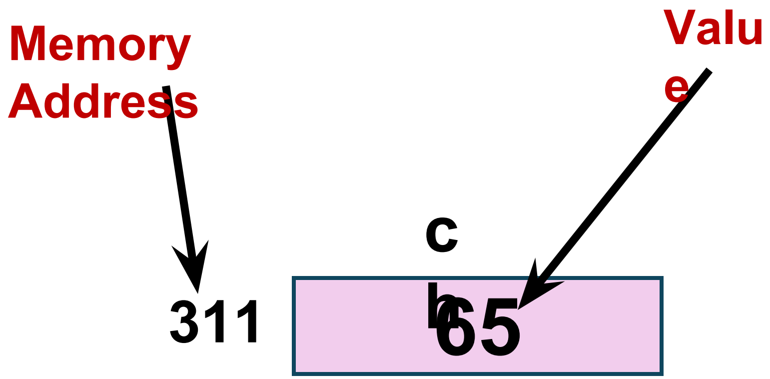
Pointers

Introduction to Pointers

- When we **declare** a **variable,** some **memory** is **allocated** for it.

- Thus, we have **two properties** for any **variable**:
  1. Its **Address**
  2. and its **Data value**

E.g., char ch = ‘A’;



Introduction to Pointers

- **How to get** the **memory-address** of a **variable**?

- Address of a variable can be accessed through the referencing operator "**&**"
  - Example: **&i** □ will return **memory location** where the data value for "**i**" is stored.

- A **pointer is a variable,** that **stores** an **address**.

Introduction to Pointers

- We can declare pointers as follows:

  Type* <variable Name>;

- Example:

  int* P;

  - creates a ***pointer variable*** named **“P”**, that will ***store address*** (memory location) of some **int type** variable.

The address of Operator &

- The **&** operator can be used to **determine** the **address of** a **variable**, which can be assigned to a **pointer variable**
  - Examples:

Dereferencing Operator *

- C++ uses the * **operator** in yet another way with pointers
  - "The variable values pointed to by p" □ *p
  - Here the * is the **dereferencing operator**

    p is said to be dereferenced

```
int v1=99;
int* p= &v1;
cout<<" P points to the value: "<<*p;
```

Dereferencing Pointer Example

```
int v1 = 0;
int* p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

v1 and *p1 now refer to
the same variable

Output:

```
42
42
```

Pointer Assignment and Dereferencing

- **Assignment operator** (**=**) is used to assign value of one **pointer** to another
- **Pointer stores addresses** so **p1=p2** copies **an address value** into another pointer

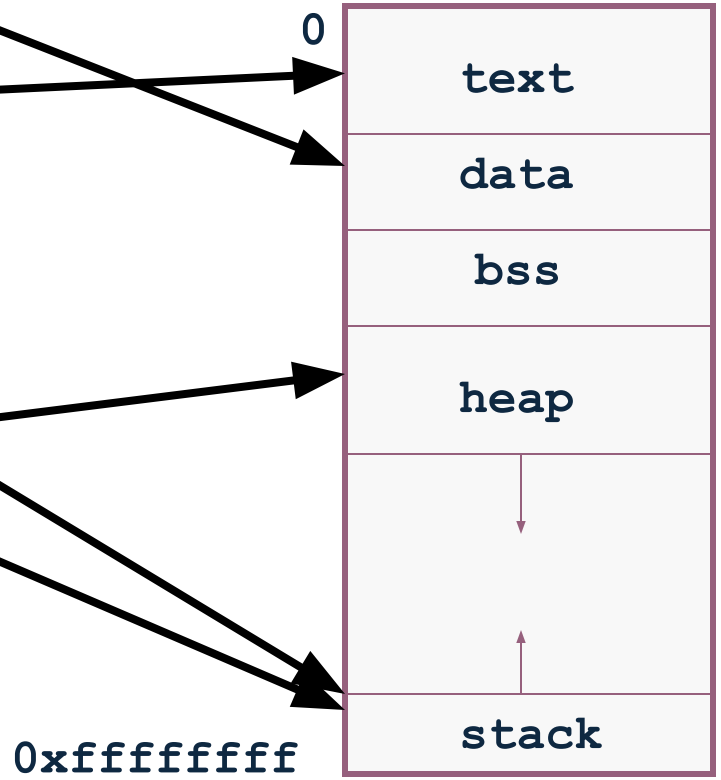
```
int v1 = 55;
int* p1 = &v1;
int* p2;
p2=p1;
cout << *p1 << endl;
cout << *p2 << endl;
```

<u>**Output:**</u>

```
55
55
```

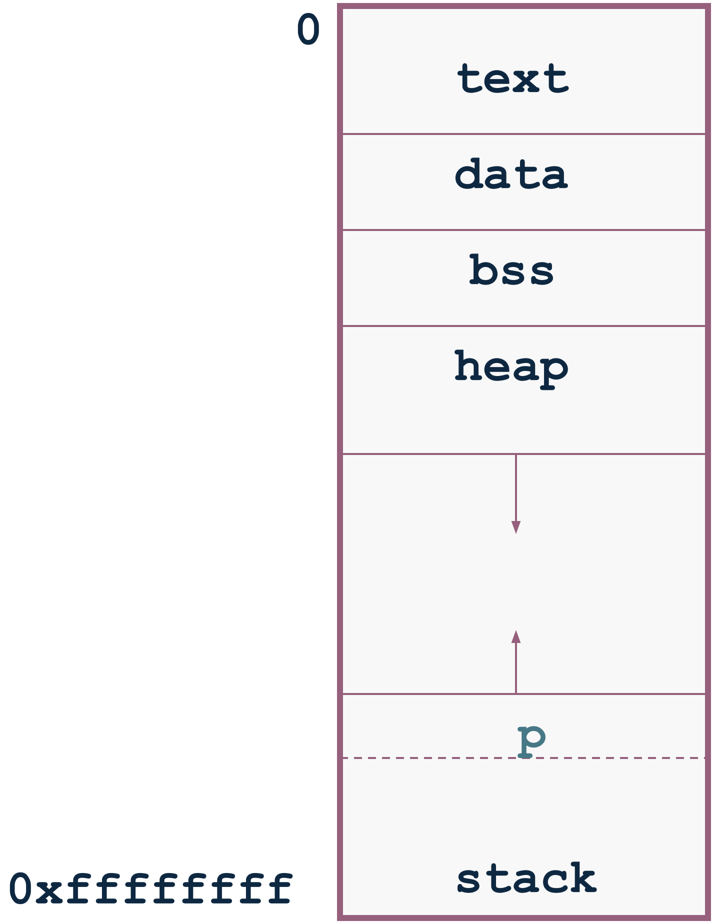
Example

```
char *string = “hello”;

const int iSize=8;

char* f(int x)
{
  char *p;

  p = new char[iSize];

  return p;
}
```



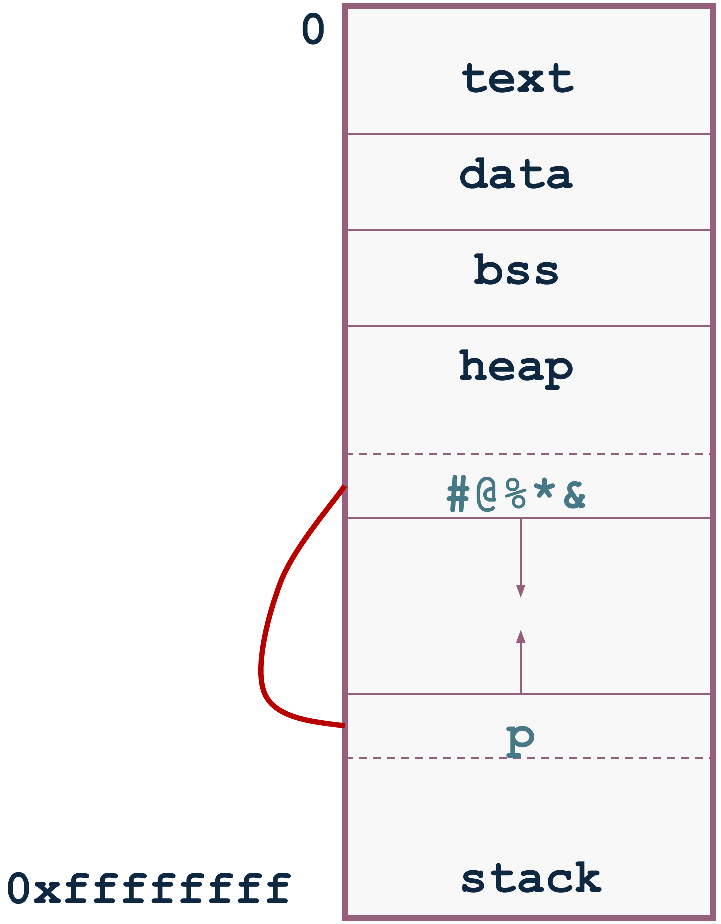
Example

```
int main()
{
  int *p;

  p = new int;

  return 0;
}
```



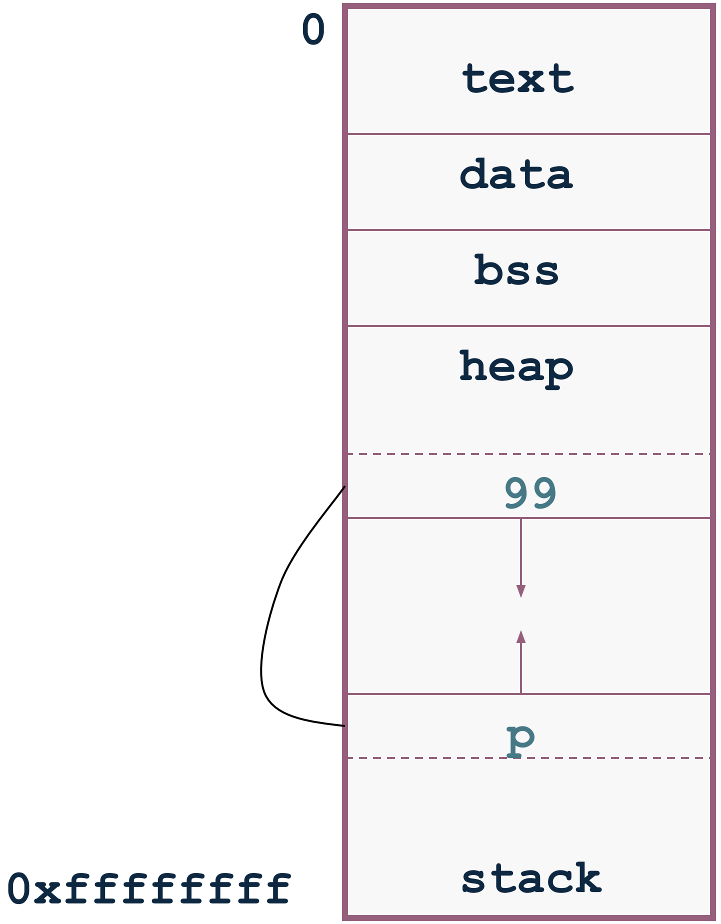
Example

```
int main()
{
  int *p;

  p = new int;


  return 0;
}
```



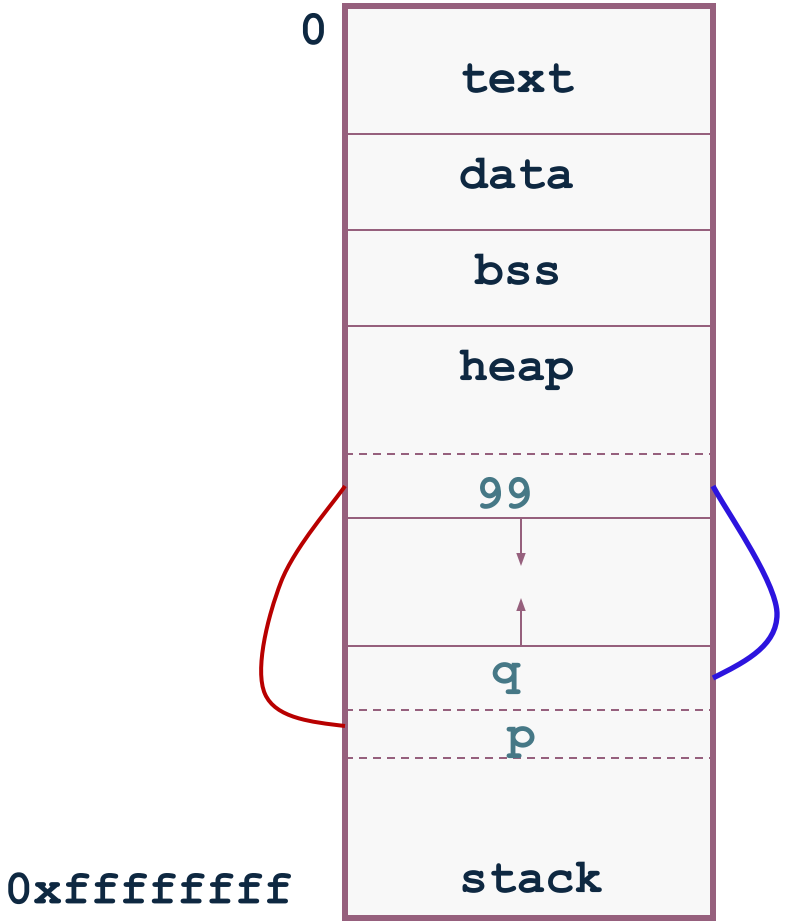
Example

```
int main()
{
  int *p;

  p = new int;
  *p = 99;

  return 0;
}
```



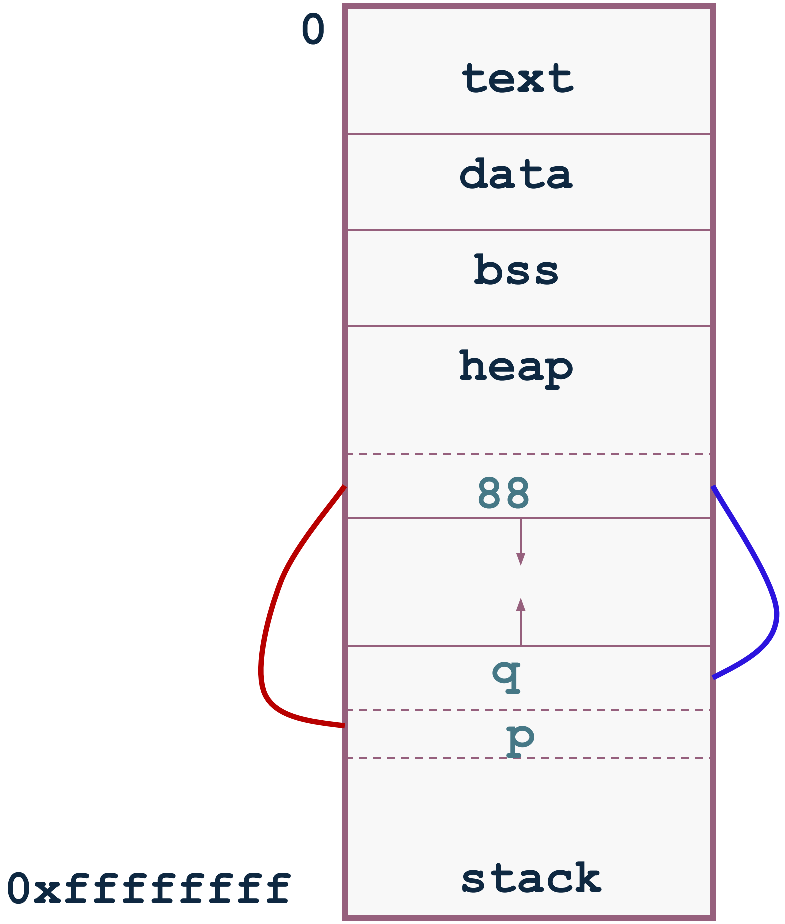
Aliasing

```
int main()
{
  int *p, *q;

  p = new int;
   *p = 99;
  q = p;

  return 0;
}
```



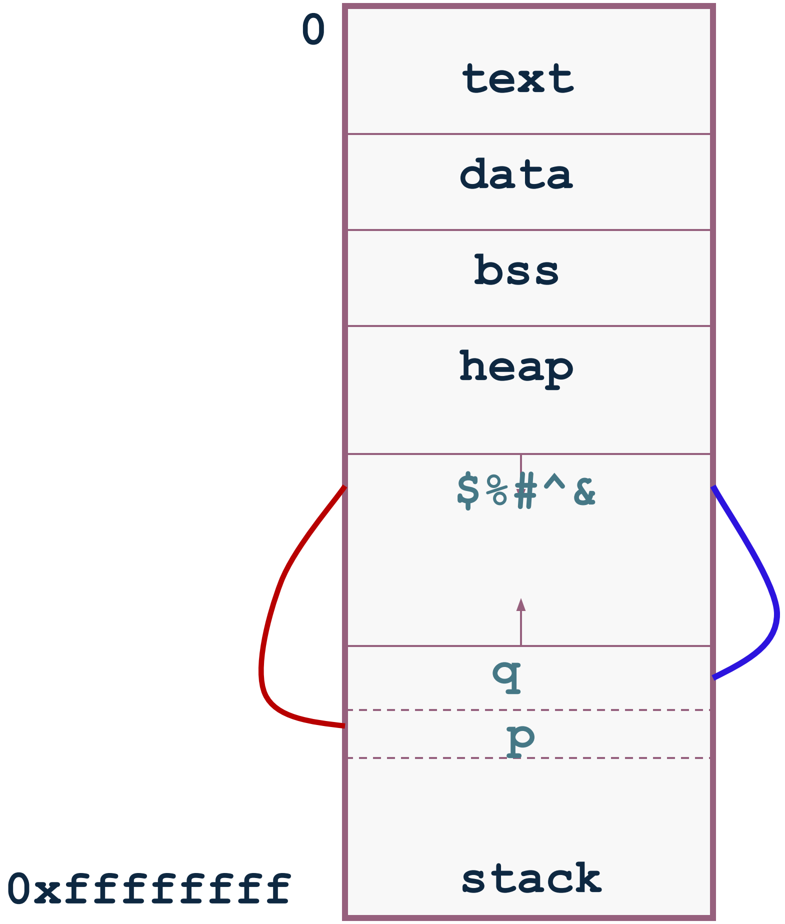
Aliasing

```
int main()
{
  int *p, *q;

  p = new int;
   *p = 99;
  q = p;

  *q = 88;

  return 0;
}
```



Aliasing

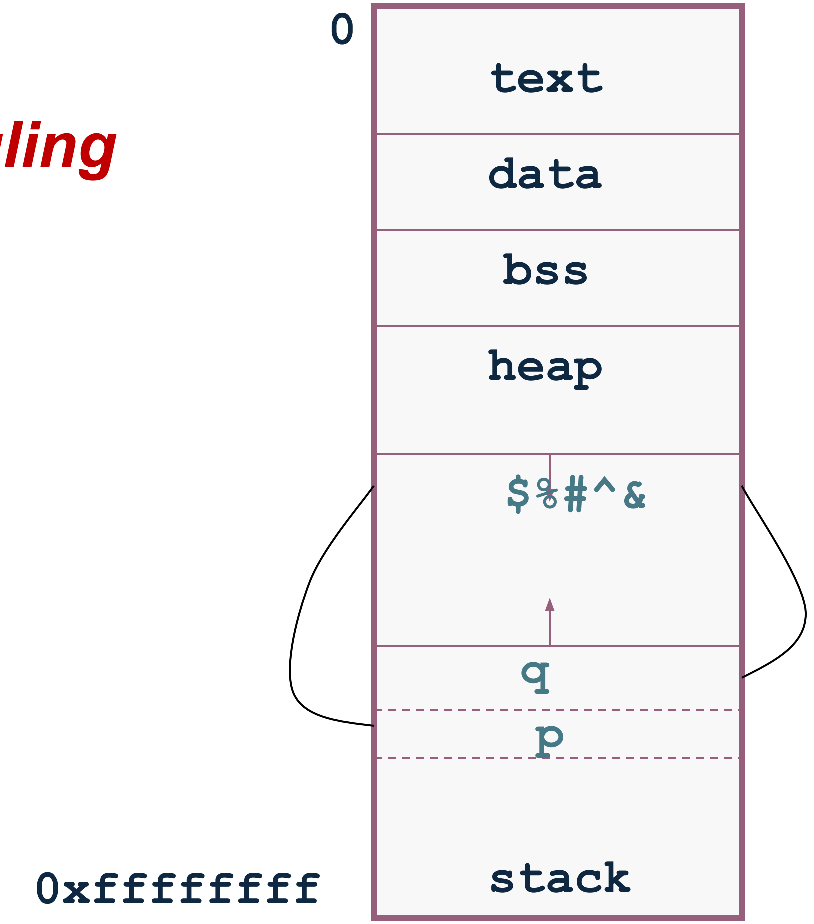
```
int main()
{
  int *p, *q;

   p = new int;
   *p = 99;
  q = p;

  *q = 88;

  delete q;

  return 0;
}
```



Dangling Pointers

```
int main()
{
  int *p, *q;

  p = new int;
   *p = 99;
  q = p;

  *q = 88;

  delete q;

  *p = 77;

  return 0;
}
```

P* and *q* are *dangling pointers
WHY?



Dangling Pointers

- The **delete operator does not delete the pointer**, it takes the **memory being pointed** to and **returns** it **to the heap**

- It **does not even change the contents** of the **pointer**

- Since the memory **being pointed to is no longer available** (and **may even be given to another application**), such a **pointer is said to be dangling**

Avoiding a Dangling Pointer

- **For Variables:**

```
delete v1;
v1 = NULL;
```

- **For Arrays:**

```
delete[ ] arr;
arr = NULL;
```

Returning Memory to the Heap

- **Remember:**
  - Return **memory to the heap before *undangling* the pointer**

- What's Wrong with the Following:

```
ptr = NULL;
delete ptr;
```

Memory Leaking

```
int main()
{

   int *p;

   p = new int;

   // make the above space unreachable; How?
   p = new int;


   // even worse…; WHY?
   while (1)
      p = new int;

   return 0;
}
```

Memory Leaking

```
void f ( )
{
    int *p;
    p = new int;

    return;
}



int main ( )
{
    f ( );
    return 0;
}
```

Memory Leaks

- Memory ***leaks*** when it is **allocated from the heap** using the **new** operator but **not returned to the heap** using the **delete operator**

Another Pointer Example

```
int i = 1;
int j = 2;
int* ptr;
ptr = &i; // ptr points to location of i
*ptr = 3; // contents of i are updated
ptr = &j; // ptr points to location of j
*ptr = 4; // contents of j are updated
cout << i << " " << j << endl;
```

```
Output:
     3
     4
```

Relationship Between Pointers and Arrays

- **Arrays** and **pointers** are **closely related**
  - **Array name** is **like** **constant pointer**
  - ***All arrays elements are placed in the consecutive locations.***
    - **Example:-** **int List [10];** ***List is the start address of array***
  - **Pointers can do array subscripting operations** We can access array elements using pointers.
    - **Example:-** **int value = List [2];** **//value assignment**

      int* p = List; **//address assignment**

Relationship Between Pointers and Arrays (Cont.)

Effect:-

- List is an address, no need for &
- The bPtr pointer will contain the address of the first element of array List.

• Element List[2] can be accessed by *(bPtr + 2)

Relationship between Arrays and Pointers

- **Arrays** and **pointers** are ***closely related***:

```
void main()
{
   int numbers[]={10,20,30,40,50};
   cout<<numbers[0]<<endl;
   cout<<numbers<<endl;
   cout<<*numbers<<endl;
   cout<<*(numbers+1);
}
```

10

Address e.g., &34234

10

20

Arrays and Pointers

Array name is the **starting address** of the **array**

- Let `int A[25];`
  `int *p; int i, j;`

- Let `p = A;`

- Then `p` points to `A[0]`
  `p + i` points to `A[i]`
  `&A[j] == p+j`
  `*(p+j)` is the same as `A[j]`

Arrays and Pointers

Expression	Assuming p is a pointer to a…	… and the size of *p is…	Value added to the pointer
p+1	char	1	1
p+1	short	2	2
p+1	int	4	4
p+1	double	8	8
p+2	char	1	2
p+2	short	2	4
p+2	int	4	8
p+2	double	8	16

Pointer Arithmetic

Only two types of arithmetic operations allowed:

1) **Addition : only integers can be added**
2) **Subtraction: only integers be subtracted**

Which of the following are valid/invalid?

I. pointer + integer (ptr+1) ✓
II. integer + pointer (1+ptr) ✓
III. pointer + pointer (ptr + ptr) ✗
IV. pointer – integer (ptr – 1) ✓
V. integer – pointer (1 – ptr) ✗
VI. pointer – pointer (ptr – ptr) ✗
VII. compare pointer to pointer (ptr == ptr) ✓
VIII. compare pointer to integer (1 == ptr) ✗
IX. compare pointer to 0 (ptr == 0) ✓
X. compare pointer to NULL (ptr == NULL) ✓

Void Pointer

- `void*` is a **pointer** to **no type** at all:
  - *Any pointer type may be assigned to void **

```
int iVar=5;
float fVar=4.3;
char cVar='Z';
int* p1;
void* vp2;
p1 = &iVar; // Allowed
p1 = &fvar; // Not Allowed
P1 = &cVar; // Not Allowed
vp2 = &fvar; // Allowed
vp2 = &cVar; // Allowed
vp2 = &iVar; // Allowed
```

This is a great advantage…
So, What are the limitations/challenges?

Accessing 1-Demensional Array Using Pointers

- We know, **Array name** denotes the **memory address** of its first slot.
  - Example:

```
int List [ 50 ];
int *Pointer;
Pointer = List;
```

- Other slots of the **Array (List [50])** can be accessed using by performing **Arithmetic operations** on **Pointer**.
- For example the address of **(element 4th)** can be accessed using:-

```
int *Value = Pointer + 3;
```

- The value of **(element 4th)** can be accessed using:-

```
int Value = *(Pointer + 3);
```

Address	Data
980	Element 0
982	Element 1
984	Element 2
986	Element 3
988	Element 4
990	Element 5
992	Element 6
994	Element 7
996	Element 8
…	
…	
998	Element 49

Accessing 1-Demensional Array

```
....
....
    int List [ 50 ];
    int *Pointer;
    Pointer = List; // Address of first Element

    int *ptr;
    ptr = Pointer + 3; // Address of 4th Element
    *ptr = 293; // 293 value store at 4th element address

}
```

Address	Data
980	Element 0
982	Element 1
984	Element 2
986	293
988	Element 4
990	Element 5
992	Element 6
994	Element 7
996	Element 8
...	
...	
998	Element 49

Accessing 1-Demensional Array

We can access all element of List [50] using Pointers and for loop combinations.

```
...
...
    int List [ 50 ];
    int *Pointer;
    Pointer = List;
    for ( int i = 0; i < 50; i++ )
    {
     cout << *Pointer;
     Pointer++; // Address of next element
    }
```

This is Equivalent to

```
for ( int loop = 0; loop < 50; loop++ )
 cout <<  Array [ loop ] ;
```

Address	Data
980	Element 0
982	Element 1
984	Element 2
986	Element 3
988	Element 4
990	Element 5
992	Element 6
994	Element 7
996	Element 8
...	
...	
998	Element 49

Accessing 2-Demensional Array

- Note that the statements

```
int *Pointer;
Pointer = &List [3];
```

- represents that we are accessing the address of 4th slot.

- In 2-Demensional array the statements

```
int List[ 5 ][ 6 ];
int *Pointer;
Pointer = &List [3];
```

Represents that we are accessing the address of 4th row

- or ***the address the 4th row and 1st column.***

Address	Data
980	Element 0
982	Element 1
984	Element 2
986	Element 3
988	Element 4
990	Element 5
992	Element 6
994	Element 7
996	Element 8
…	
…	
998	Element 50

Accessing 2-Demensional Array

- **int List [9] [6];**
- **int *ptr;**
- **ptr = &List [3];**

- To access the address of **4th row 2nd column:**
  - **ptr++;** **// address of 4th row 2nd column**
  - (**faster than normal array accessing Why?**)
  - Equivalent to **List [3][1] ;**

Column

Row	0	1	2	3	4	5
0	300	302	304	306	308	310
1	312	314	316	318	320	322
2	324	326	328	330	332	334
3	336	338	340	342	344	346
4	[illegible]	350	352	354	356	358
5	360	362	364	366	368	370
6	372	374	376	378	380	382
7	384	386	388	390	392	394
8	396	398	400	402	404	406

Memory address

Accessing 2-Demensional Array

- We know **computer can perform only one** **operation at any time** (**remember fetch-decode-execute cycle**).

- Thus to access List [3][1] element (**without pointer**) **two operations are involved:-**
  - **First to determine row List [3]**
  - **Second to determine column List[3][1]**

- But using pointer we can reach the element of 4^{th} row 2^{nd} column (directly) by **increment our pointer value (which is a single operation).**
  - **ptr+1; // 4^{th} row 2^{nd} column**
  - **ptr+2; // 4^{th} row 3^{rd} column**
  - **ptr+3; // 4^{th} row 4^{th} column**

Column

Row	0	1	2	3	4	5
0	300	302	304	306	308	310
1	312	314	316	318	320	322
2	324	326	328	330	332	334
3	336	338	340	342	344	346
4	348	350	352	354	356	358
5	360	362	364	366	368	370
6	372	374	376	378	380	382
7	384	386	388	390	392	394
8	396	398	400	402	404	406

Memory address

Swapping Variables using Pointers

```
void main()   {
   char a = 'A';
   char b = 'Z';
   char *Ptr1= &a;
   char *Ptr2= &b;

   char temp = *Ptr1;
   *Ptr1 = *Ptr2;
   *Ptr2 = c;

   cout << a << b << endl;
}
```

Pointer Arithmetic

```
int ar[]={20, 30, 50, 80, 90};
int *p;
p = ar;
p = p + 1;
*p = *p + 1;
```

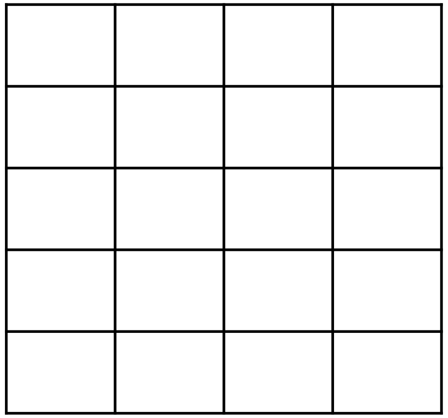
Creating Dynamic 2D Arrays

- Two basic methods:
  1. Using a single Pointer
  2. Using a Array of Pointers

Dynamic two dimensional arrays

1. Using a single Pointer
   - Total elements in a 2D Array:
     - m * n (i.e., rows * cols)

5 rows * 4 columns = 20 elements



Target Approach=

- allocate 20 elements using dynamic allocation
- Use a **single pointer** to point and access those items.

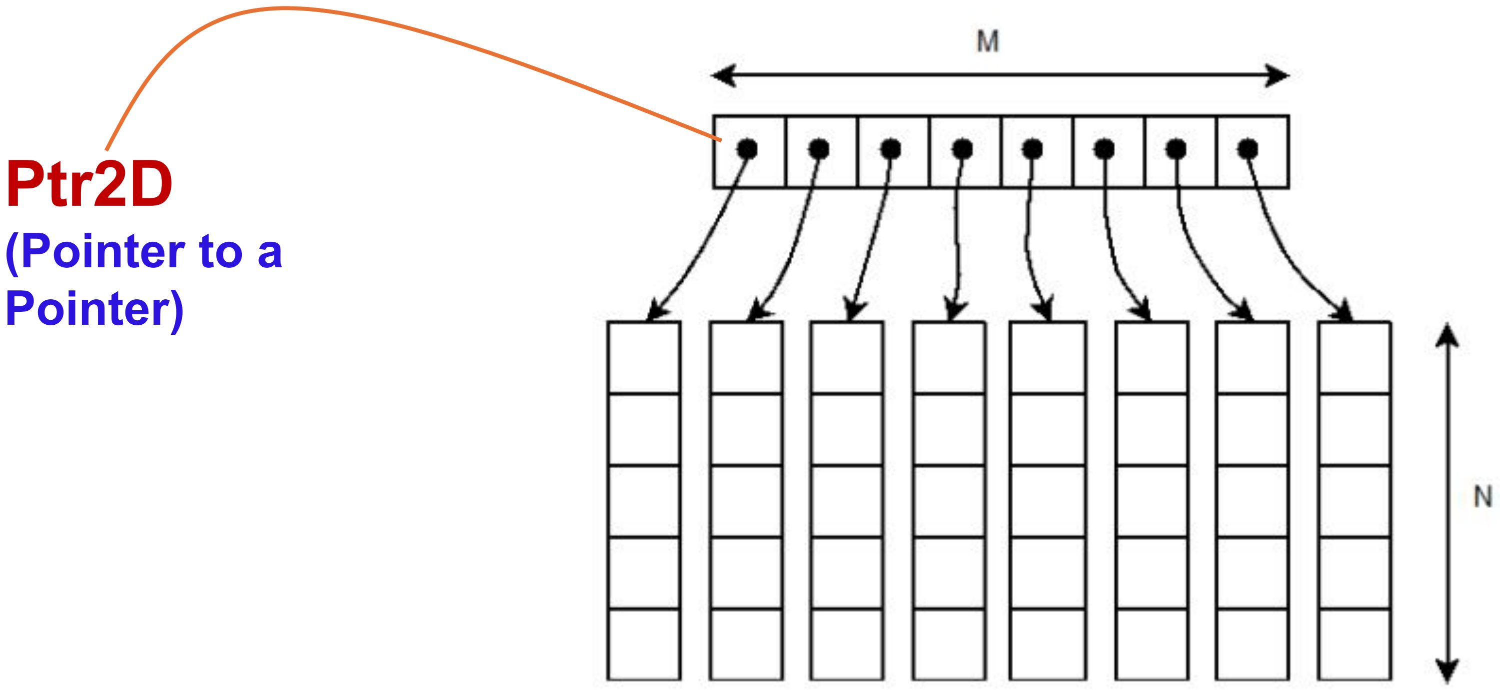
Dynamic 2D Arrays

```cpp
#include <iostream>

// M x N matrix
#define M 4
#define N 5

// Dynamically Allocate Memory for 2D Array in C++
int main()
{
    // dynamically allocate memory of size M*N
    int* A = new int[M * N];

    // assign values to allocated memory
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            *(A + i*N + j) = rand() % 100;

    // print the 2D array
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
            std::cout << *(A + i*N + j) << " ";    // or (A + i*N)[j])

        std::cout << std::endl;
    }

    // deallocate memory
    delete[] A;

    return 0;
}
```

Dynamic 2D Array – Double Pointer

2. Using a Pointer that points to Array of Pointer

- Total elements in a 2D Array: M_rows * N_coulmns



Dynamic 2D Array – Double Pointer

```
int **dynamicArray = 0;

//memory allocated for elements of rows.


dynamicArray = new int *[ROWS] ;

//memory allocated for  elements of each column.


for( int i = 0 ; i < ROWS ; i++ )
dynamicArray[i] = new int[COLUMNS];

//free the allocated memory


for( int i = 0 ; i < ROWS ; i++ )
delete [] dynamicArray[i] ;
delete [] dynamicArray ;
```

```
#include <iostream>

// M x N matrix
#define M 4
#define N 5

// Dynamic Memory Allocation in C++ for 2D Array
int main()
{
    // dynamically create array of pointers of size M
    int** A = new int*[M];

    // dynamically allocate memory of size N for each row
    for (int i = 0; i < M; i++)
        A[i] = new int[N];

    // assign values to allocated memory
    for (int i = 0; i < M; i+
        for (int j = 0; j < N
            A[i][j] = rand()

    // print the 2D array
    for (int i = 0; i < M; i+
    {
        for (int j = 0; j < N; j++)
            std::cout <
        std::cout << s
    }

    // deallocate memo
    for (int i = 0; i
        delete[] A[i];

    delete[] A;

    return 0;
}
```

Can we vary size of each column in Dynamic 2D Array (using double pointer)

PP □ start of array of pointers

*PP □ First Address pointed by first row (sub array)

*(*PP) □ First value of first array

(*PP)++ □ Move to next address in the first array

PP++ □ Move to Next row (second array address

Dynamic 2D Array

(Varying Row Size)

```
// Dynamically Allocate Memory for 2D Array in C++
int main()
{
    // dynamically create array of pointers of size M
    int** A = new int*[M];

    // dynamically allocate memory of size N for each row
    for (int i = 0; i < M; i++)
        A[i] = new int[N+i];

    // assign values to allocated memory
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N+i; j++)
            A[i][j] = rand() % 100;

    // print the 2D array
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N+i; j++)
            std::cout << A[i][j] << " ";

        std::cout << std::endl;
    }

    // deallocate memory using delete[] operator
    for (int i = 0; i < M; i++)
        delete[] A[i];

    delete[] A;

    return 0;
}
```

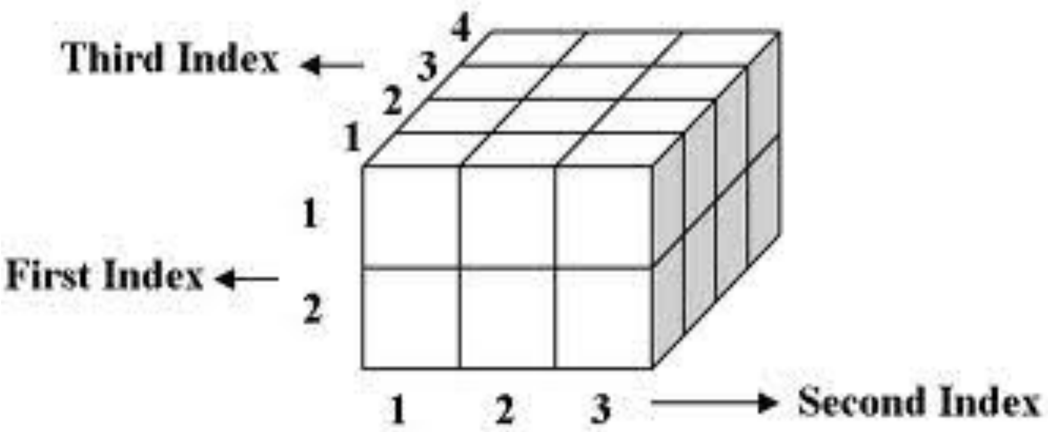
Output

```
83 86 77 15 93
35 86 92 49 21 62
27 90 59 63 26 40 26
72 36 11 68 67 29 82 30
```

Home Work

- Manipulating a 3D Array

1. Using a single pointer
2. Using a triple pointer



Three-dimensional array with twenty four elements

3D Array Using a single pointer

```cpp
#include <iostream>

// X x Y x Z matrix
#define X 2
#define Y 3
#define Z 4

// Dynamic Memory Allocation in C++ for 3D Array
int main()
{
    // dynamically allocate memory of size X*Y*Z
    int* A = new int[X * Y * Z];

    // assign values to allocated memory
    for (int i = 0; i < X; i++)
        for (int j = 0; j < Y; j++)
            for (int k = 0; k < Z; k++)
                *(A + i*Y*Z + j*Z + k) = rand() % 100;

    // print the 3D array
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            for (int k = 0; k < Z; k++)
                std::cout << *(A + i*Y*Z + j*Z + k) << " ";

            std::cout << std::endl;
        }
        std::cout << std::endl;
    }

    // deallocate memory
    delete[] A;

    return 0;
}
```

```cpp
#include <iostream>

// X x Y x Z matrix
#define X 2
#define Y 3
#define Z 4

// Dynamically Allocate Memory for 3D Array in C++
int main()
{
    int*** A = new int**[X];

    for (int i = 0; i < X; i++)
    {
        A[i] = new int*[Y];

        for (int j = 0; j < Y; j++)
            A[i][j] = new int[Z];
    }

    // assign values to allocated memory
    for (int i = 0; i < X; i++)
        for (int j = 0; j < Y; j++)
            for (int k = 0; k < Z; k++)
                A[i][j][k] = rand() % 100;

    // print the 3D array
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            for (int k = 0; k < Z; k++)
                std::cout << A[i][j][k] << " ";

            std::cout << std::endl;
        }
        std::cout << std::endl;
    }

    // deallocate memory
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
            delete[] A[i][j];

        delete[] A[i];
    }

    delete[] A;

    return 0;
}
```