



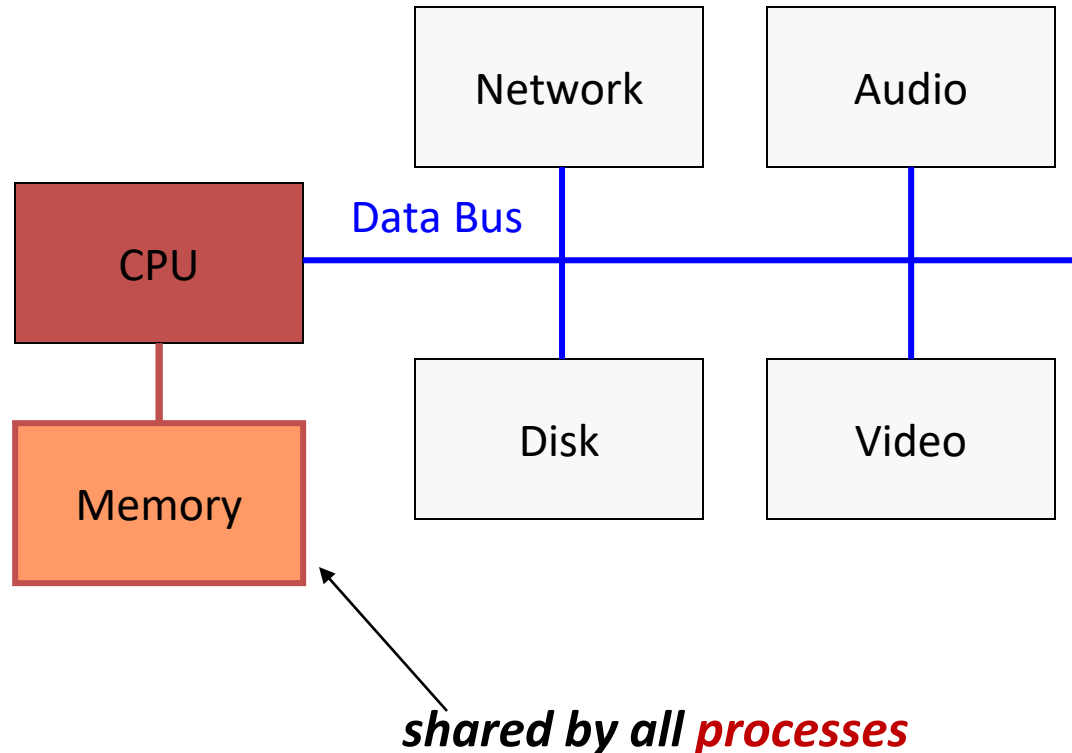
Pointers

Dr. Mudassar Aslam

Cybersecurity Department,
National University of Computer & Emerging Sciences,
Islamabad Campus



Main Memory



What is a process?

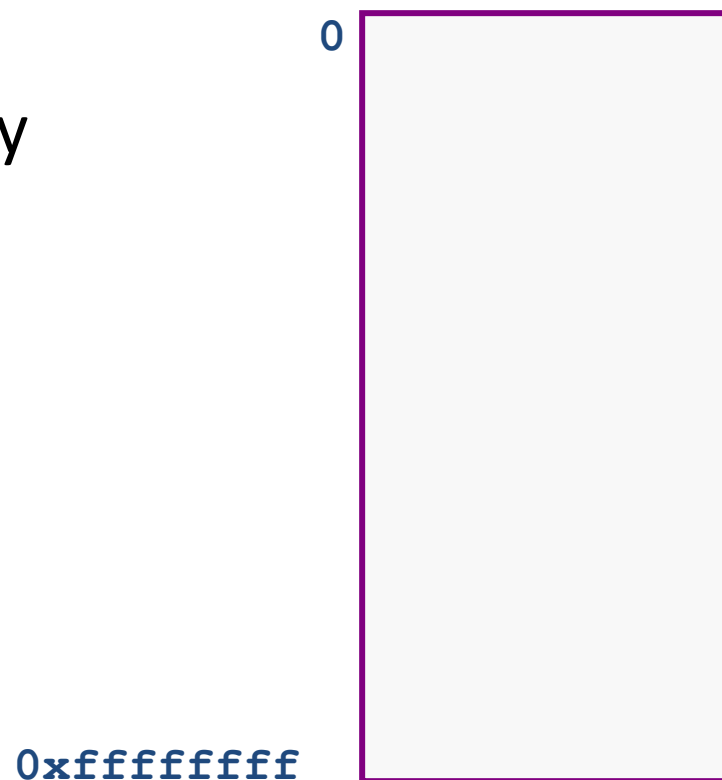
An executing program (loaded in memory) is called process...



Virtual Memory

(How a CPU see's a Process?)

- **Continuous** memory space for all process:
 - Set of locations as needed by a process

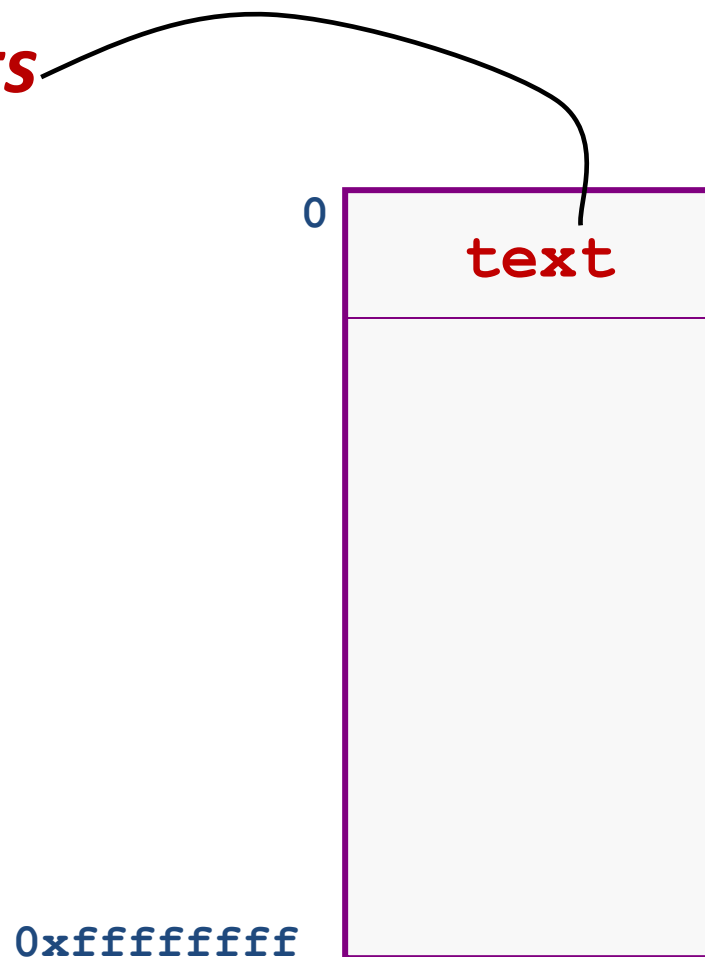




Organization of Virtual Memory: .text

- ***Program code*** and ***constants***

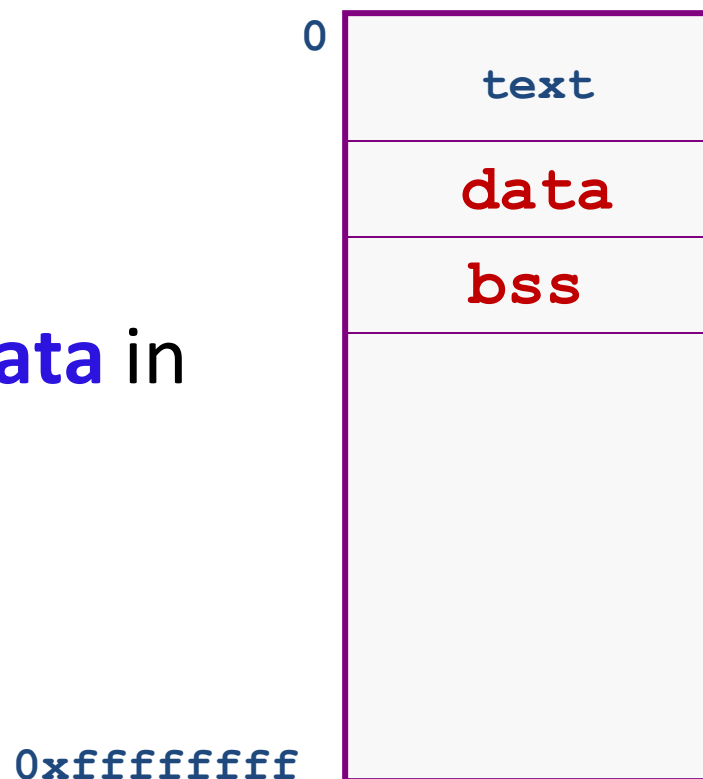
- binary form
- loaded libraries
- code instructions
- space calculated at compile-time





Organization of Virtual Memory: .data

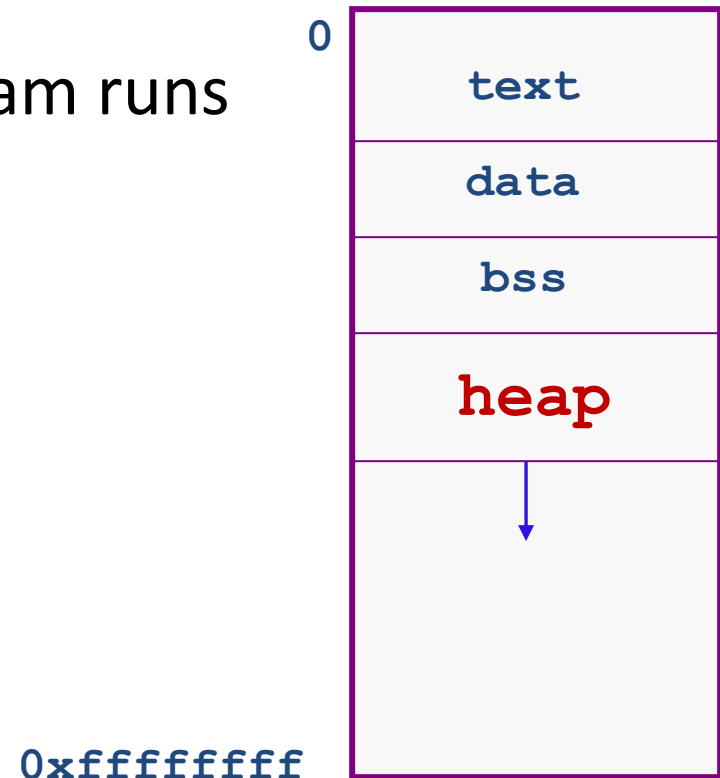
- **Data**: initialized global data in the program
 - Ex: `int size = 100;`
- **BSS**: un-initialized global data in the program
 - Ex: `int length;`





Organization of Virtual Memory: heap

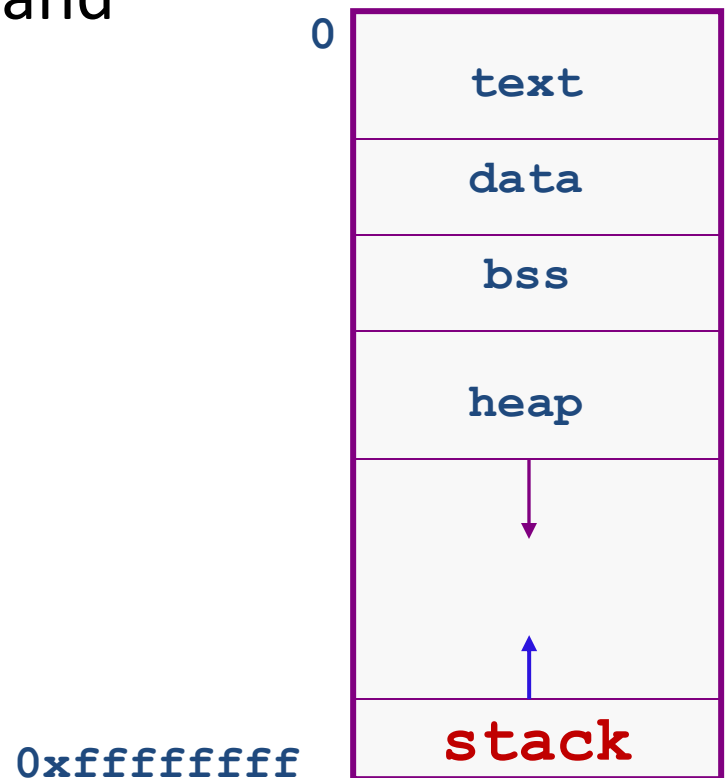
- **Heap**: dynamically-allocated spaces
 - Ex: **new**, **delete**
 - **dynamically grows** as program runs





Organization of Virtual Memory: stack

- **Stack**: local variables in functions
 - support function call/return and recursive functions
 - grow to low address

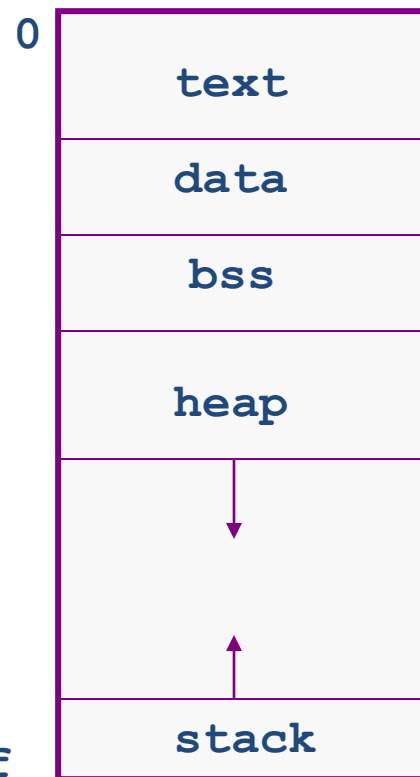




Summary: Process Address Space

- **text**: program text/code and constants
- **data**: initialized global & static data
- **bss**: un-initialized global & static data
- **heap**: dynamically managed memory
- **stack**: function's local variables

0xffffffff

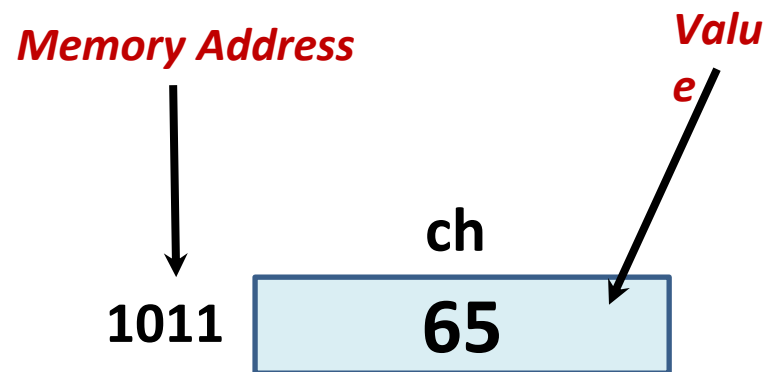




Introduction to Pointers

- 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
- **Pointer stores only address**



Creating a Pointer Variable

Type* <variable Name>;

Example:

```
int* P;
```

```
float* P2;
```

- 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" \square ***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;
```

Output:

55

55



Storage and Data Allocation of C++ Program

```
char *string = "hello";
```

```
const int iSize=8;
```

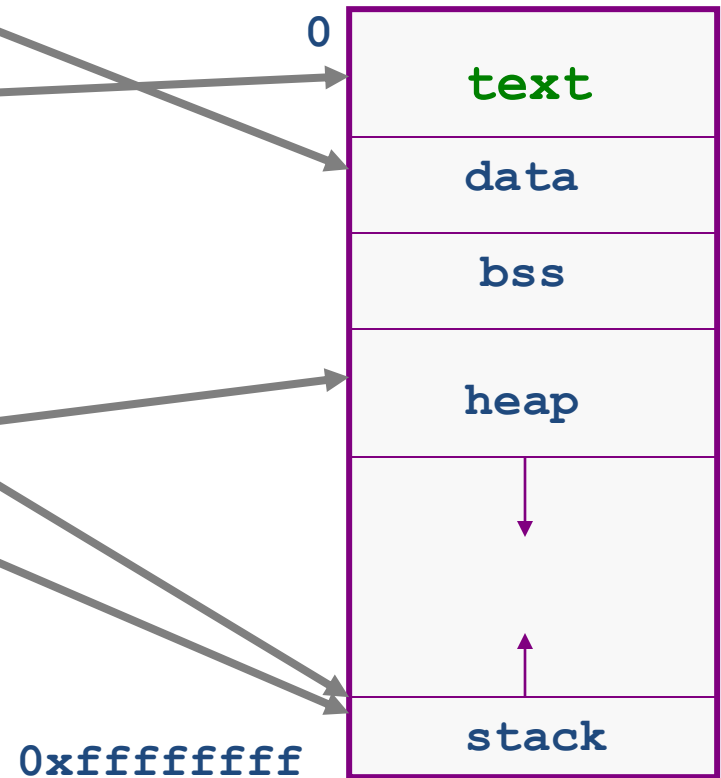
```
char* f(int x)  
{
```

```
    char *p;
```

```
    p = new char[iSize];
```

```
    return p;
```

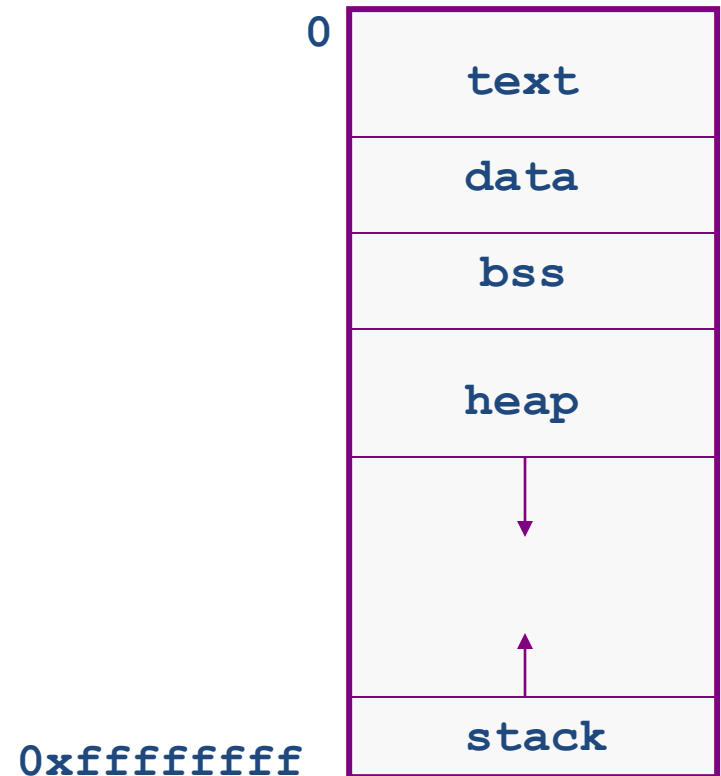
```
}
```





Variables' Lifetime

- **text:**
 - program startup
 - program finish
- **data, bss:**
 - program startup
 - program finish
- **heap:**
 - dynamically allocated
 - de-allocated (free)
- **stack:**
 - function call
 - function return





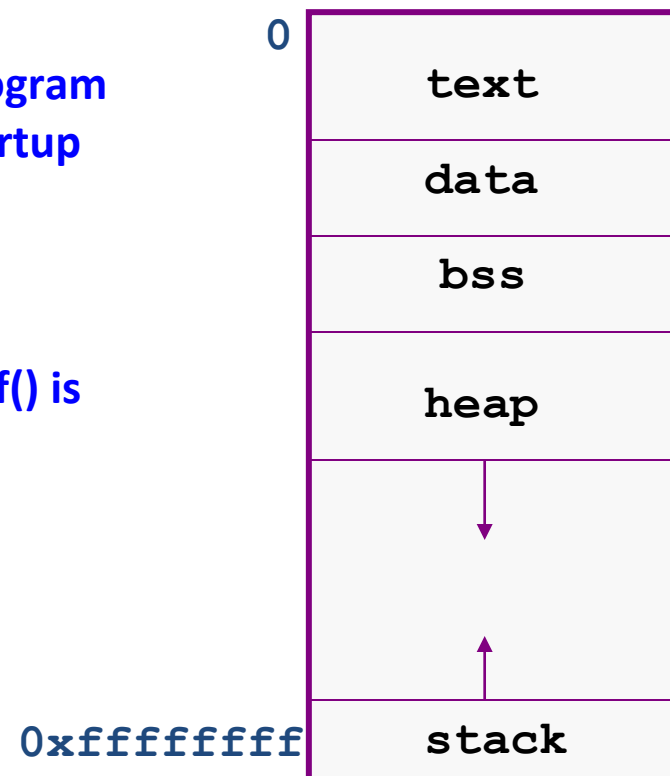
Example

```
char *string = "hello";  
const int iSize=8;  
  
char *f (int x)  
{  
    char *p;  
    p = new char[iSize];  
    return p;  
}
```

program startup

when f() is called

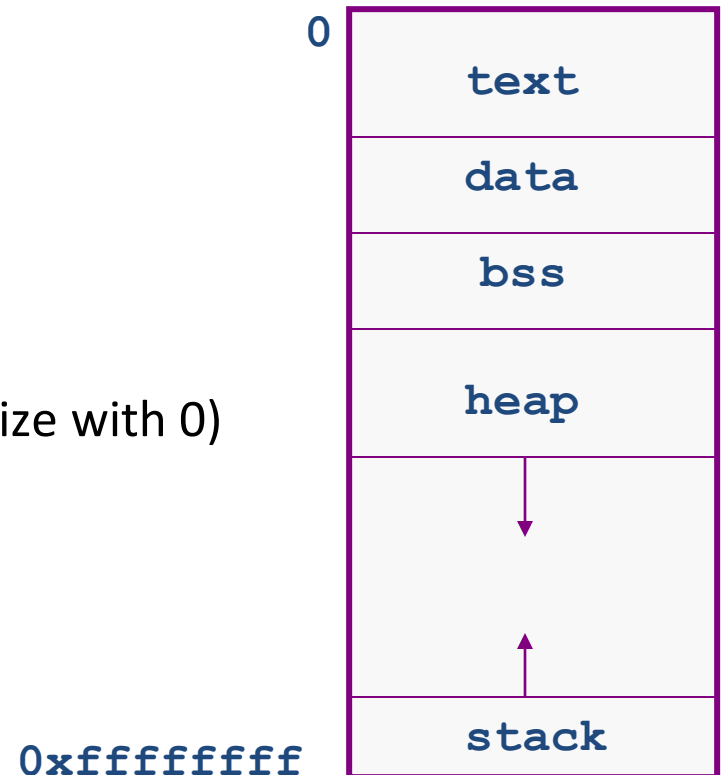
Live after allocation; till delete or program finish





Variables' Initialization

- **text:**
 - Read-only (once; e.g., constants)
- **data**
 - on program startup
- **bss:**
 - **un-initialized** (though some systems initialize with 0)
- **heap:**
 - un-initialized
- **stack:**
 - un-initialized





Pointer Assignments (Aliasing)

- Some **care is required** making assignments to **pointer** variables:

p1 = p2; // changes the location that p1 "points"
to

***p1 = *p2;** // changes the value at location that
// p1 "points" to



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



Swapping variables using Pointers

```
void main() {  
    char a = 'A';  
    char b = 'Z';  
    char *Ptr1= &a;  
    char *Ptr2= &b;  
  
    char temp = *Ptr1;  
    *Ptr1 = *Ptr2;  
    *Ptr2 = temp;  
  
    cout << a << b << endl;  
}
```



Dynamic Memory Allocation

- Used when **space requirements** are **unknown** at **compile time**
- Most of the time **the amount of space required** is **unknown at compile time**
- **Dynamic Memory Allocation (DMA):-**
 - With **Dynamic memory allocation** we can **allocate/deletes memory (elements of an array) at runtime or execution time.**



Static VS. Dynamic Memory Allocation

- **Dynamically allocated memory** is kept on the **memory heap** (also known as the free store)
- **Dynamically allocated memory** cannot have a "name", it must be referred to (using address)
- **Declarations** are used to statically allocate memory
- The ***new* operator** is used to dynamically allocate memory



Dynamic Memory Allocation

Heap management in C++ is explicit:

```
ptr = new data-type;  
//allocte memory for one element
```

```
ptr = new data-type [ size ];  
//allocte memory for fixed number of element
```

```
delete ptr;  
//deallocte memory for one element
```

```
delete[] ptr;  
//deallocte memory for array
```

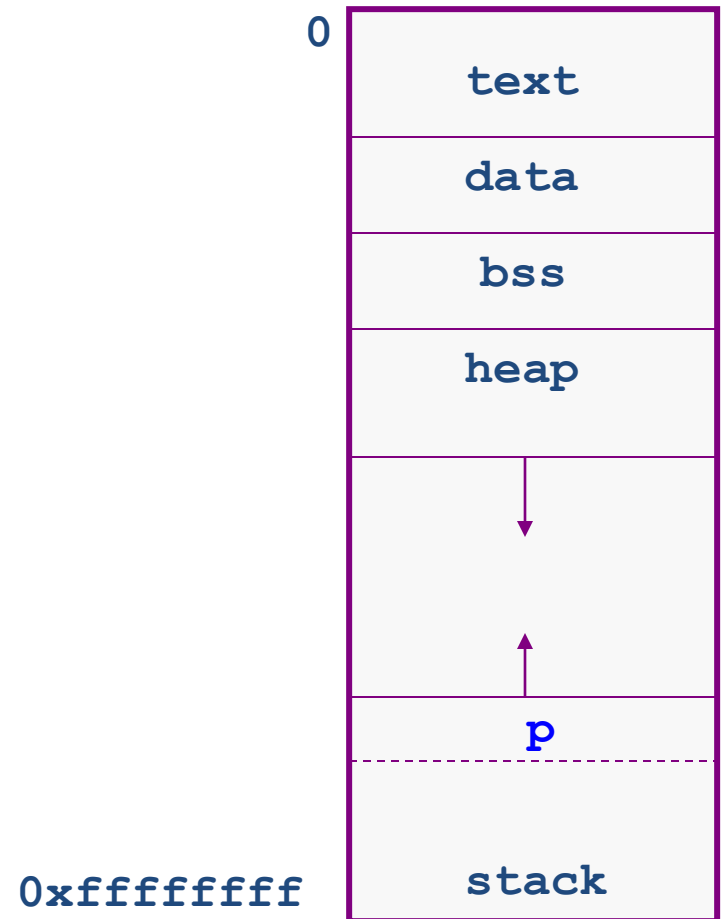


Dynamic Allocation - Example

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

    p = new int;
    *p = 99;

    return 0;
}
```



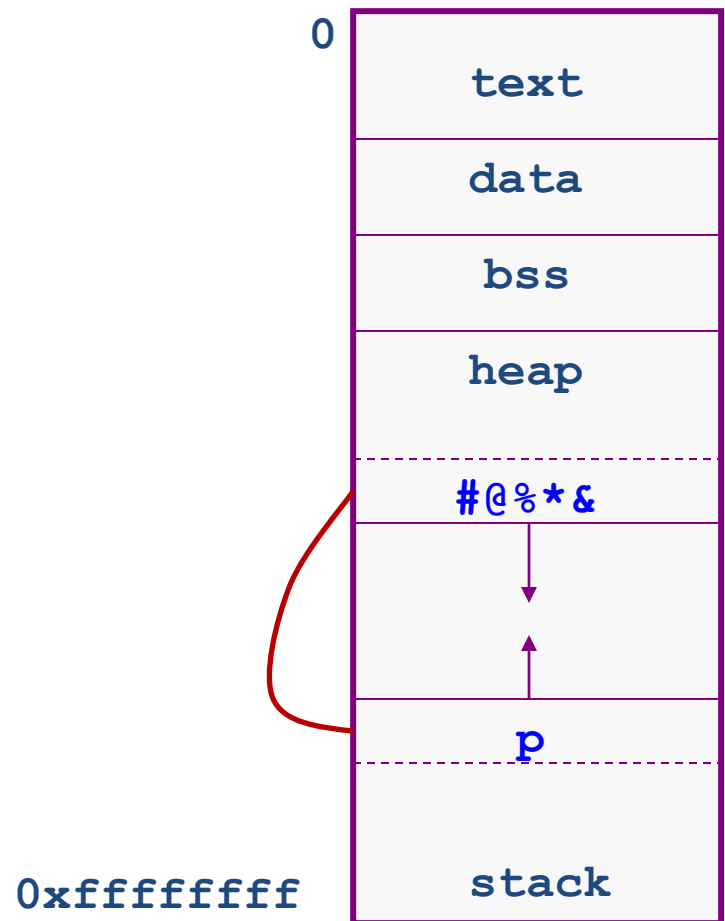


Dynamic Allocation - Example

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

    p = new int;
    *p = 99;

    return 0;
}
```



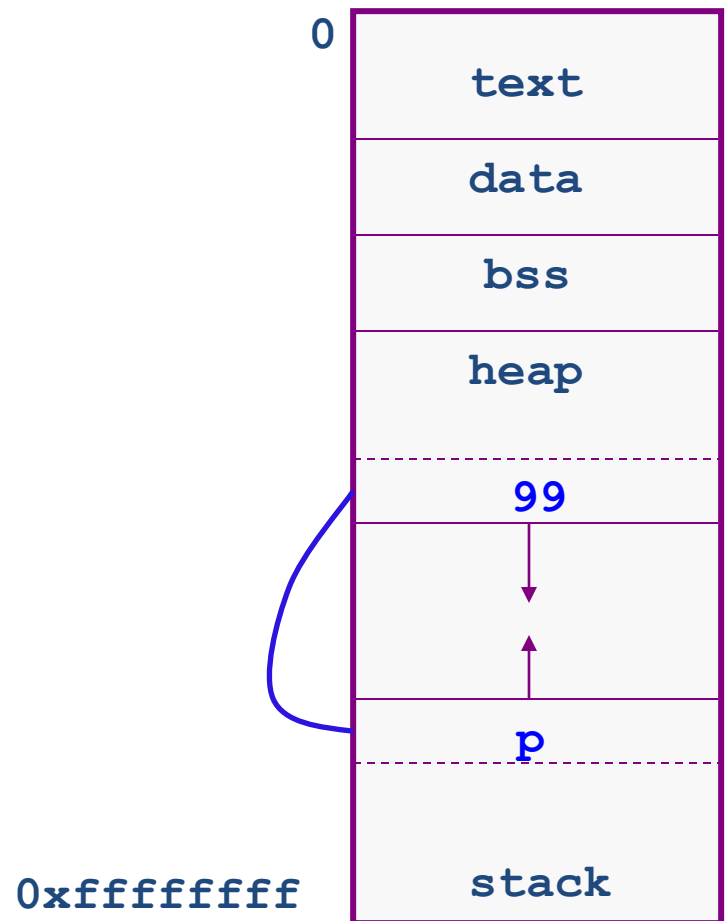


Dynamic Allocation - Example

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

    p = new int;
    *p = 99;

    return 0;
}
```





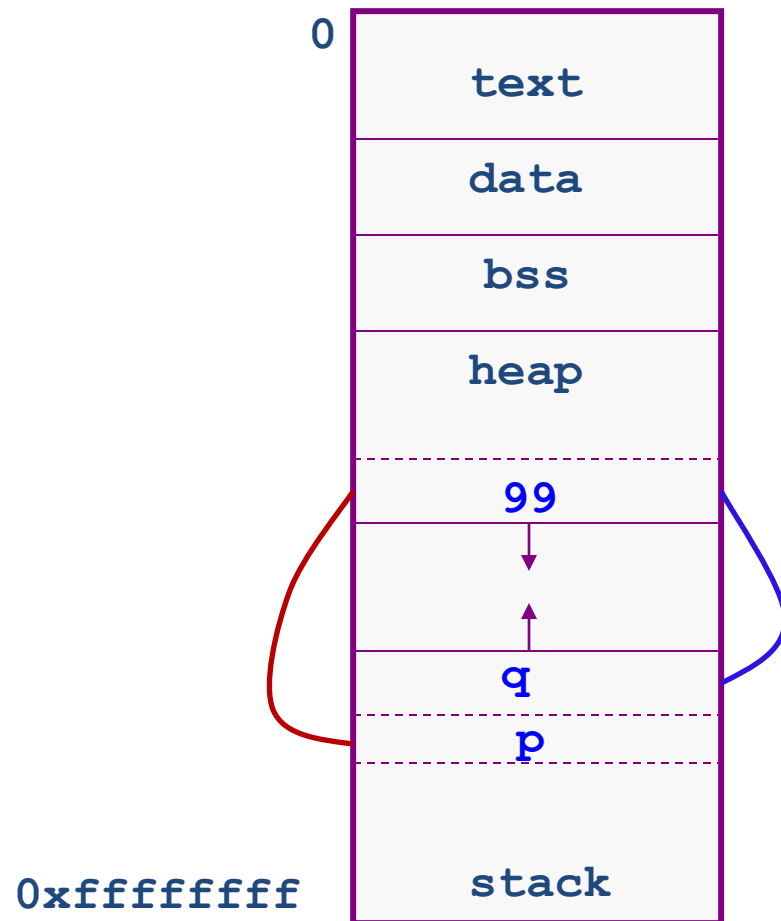
Aliasing

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

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

    return 0;
}
```

Pointer Aliasing: same memory location can be accessed using different names.





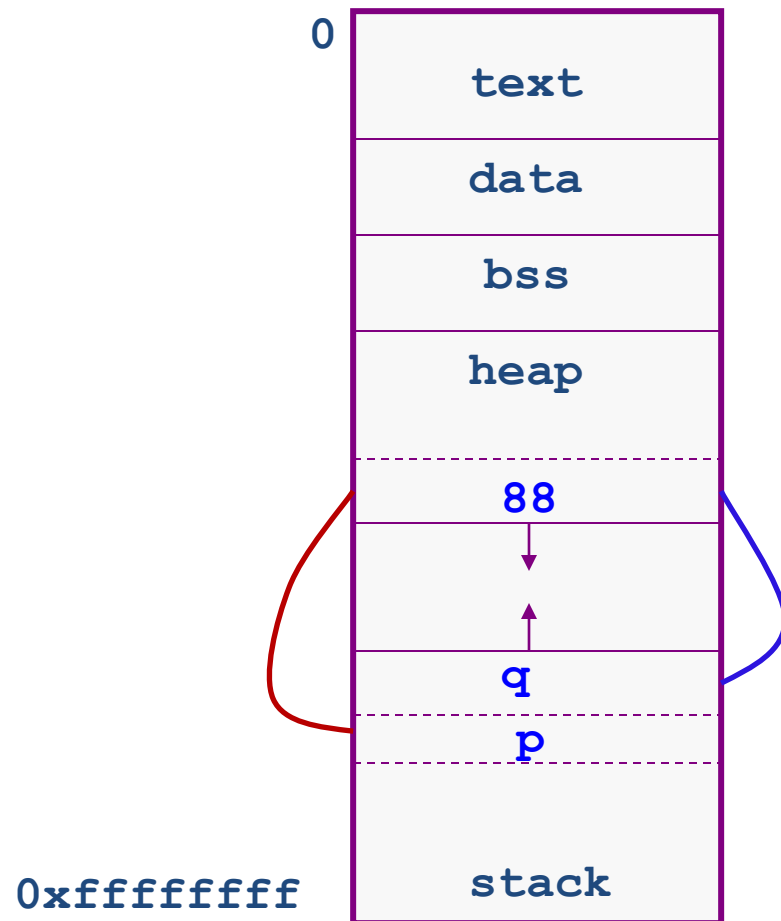
Aliasing

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

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

    *q = 88;

    return 0;
}
```





Aliasing

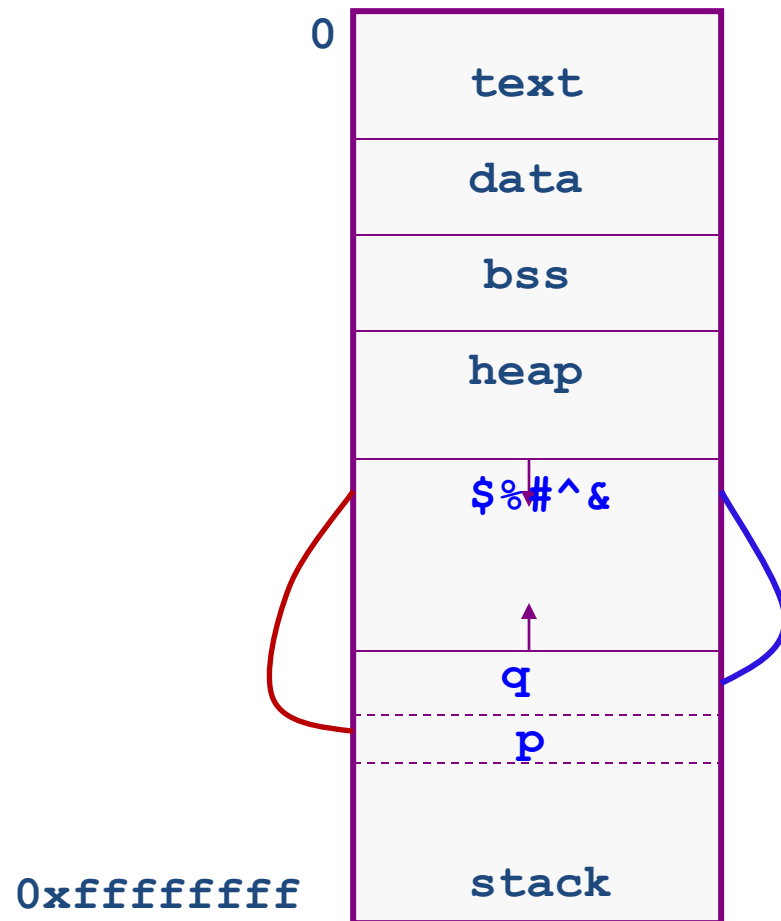
```
int main()
{
    int *p, *q;

    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;

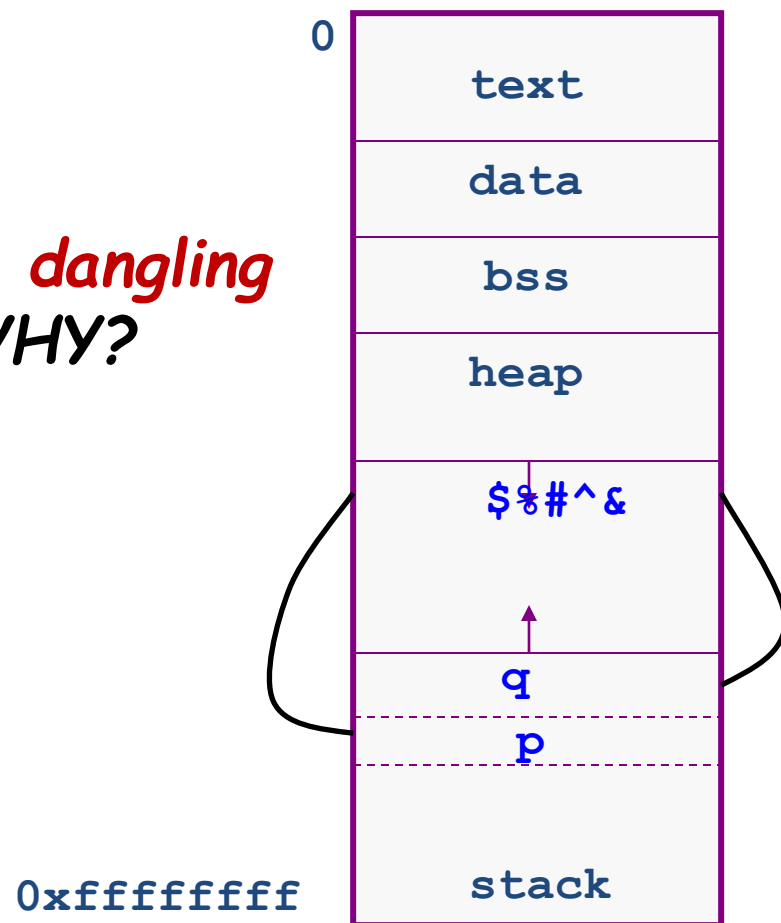
    *q = 88;

    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 code:

```
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



Memory Leaking and Dangling Pointers

- Dangling pointers and memory leaking are evil sources of bugs:
 - hard to debug
 - may appear after a long time of run
 - may far from the bug point
 - hard to prevent

What should be the good programming practices while using Pointers?



Null Address

- Like a local variable, a **pointer** is assigned a **random value** (i.e., address) if not initialized
- **0** is a **pointer constant** that represents the **empty** or **Null address**
- Should be used to **avoid dangling pointers**
 - Cannot Dereference a Pointer whose value is Null:

```
int *ptr = 0; OR int *ptr=NULL;
```

```
cout<< *ptr << endl;    // ERROR: ptr
                        // does not point to
                        // a valid address
```




Pointers Data-Type

- Question:

Why is it important to declare the type of the variable that a pointer points to?

Aren't all memory addresses of the same length?



Pointers Type

- Answer:

- All memory addresses are of the same length,
 - However, with operation “p++” where “p” is a pointer [?] the compiler needs to know the data type of the variable “p” (to jump at next memory location)

Examples:

- If “p” is a character-pointer then “p++” will increment “p” by one byte (next location)
- if “p” is an integer-pointer its value on “p++” would be incremented by 4 bytes (next loc.)



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

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;
```

10

```
    cout<<numbers<<endl;
```

Address e.g., &34234

```
    cout<<*numbers<<endl;
```

10

```
    cout<<*(numbers+1);
```

20

```
}
```



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

Warning: These byte increments are based on 32-bit architecture, adjust according to the machine architecture.



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) ✓



Comparing Pointers

- If one address comes before another address in memory, the *first address* is considered *less than* the *second address*.
- Two pointer variables can be compared using C++ relational operators: $<$, $>$, $<=$, $>=$, $==$
- In an array, elements are stored in consecutive memory locations, E.g., address of **Arr[2]** will be smaller than the address of **Arr[3]** etc.



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;
```

```
int iVar;
```

```
void vp1;
```

```
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

...

...

```
int List [50];
```

```
int *p;
```

```
p = List;
```

```
p = p + 3;
```

```
*p = 293;
```

```
}
```

Address	Data
980	Element 0
984	Element 1
988	Element 2
992	293
996	Element 4
1000	Element 5
1004	Element 6
1008	Element 7
1012	Element 8
...	
...	
1180	Element 49

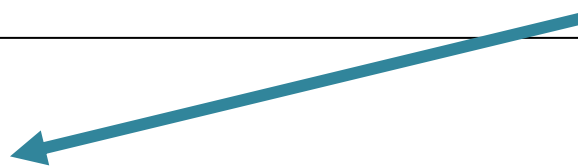


Accessing 1-Demensional Array

...

...

```
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
984	Element 1
988	Element 2
992	293
996	Element 4
1000	Element 5
1004	Element 6
1008	Element 7
1012	Element 8
...	
...	
1180	Element 49



Accessing 2-Dimensional Array

- In 1D array if we use:

```
int *Pointer;
```

```
Pointer = &List[3]; //accessing the  
// address of 4th slot.
```

- In 2-Dimensional array:

```
int main()  
{  
    int A[3][3]={{1,2,3},{4,5,6},{7,8,9}};  
    int *p;  
    p=A[2];  
    cout<<"\nvalue of p= "<<*p<<endl; //outputs 7  
    return 0;  
}
```

Demo Code:
pointersArrays1.cpp

Address	Data
980	Element 0
984	Element 1
988	Element 2
992	293
996	Element 4
1000	Element 5
1004	Element 6
1008	Element 7
1012	Element 8
...	
...	
1180	Element 49



Accessing 2-Dimensional Array

Full array traversal of 2D array:

```
int main()
{
    int A[3][3]={1,2,3},{4,5,6},{7,8,9}};
    int *p;
    p=A; //ERROR
    p=(int*)A;
    for(int i=0;i<9;i++)
    {
        cout<<"\nValue of p= "<<*p<<endl;
        p++; // or just cout<<*(p+i)
    }

    return 0;
}
```

- Pointer access is faster as compared to 2D array notation (one address calculation as compared to [][] row and col addresses)



Casting pointers

- **Pointers** have **types**, so you cannot do

```
int *pi; double *pd;  
pd = pi;
```

- C++ will let you change the type of a pointer with an **explicit cast**:

```
int *pi; double *pd;  
pd = (double*) pi;
```

Warning: Values dereferenced after the cast are undermined (because of possibly difference in memory size)



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

```
#include<iostream>
#include <stdlib.h>
#include<time.h>
using namespace std;
int main()
{
    int M=4; int N=5;
    int* Arr=new int[M*N];
    srand(time(0));

    //set values
    for(int i=0;i<M;i++)
        for(int j=0;j<N;j++)
            *(Arr + i*M+j) = rand()%100;

    //display values
    for(int i=0;i<M;i++) {
        for(int j=0;j<N;j++) {
            cout<<*(Arr + i*M+j)<<" ";
            //cout<<(Arr+i*M)[j]<<" ";
        }
        cout<<endl;
    }
    delete[] Arr;
    return 0; }
```

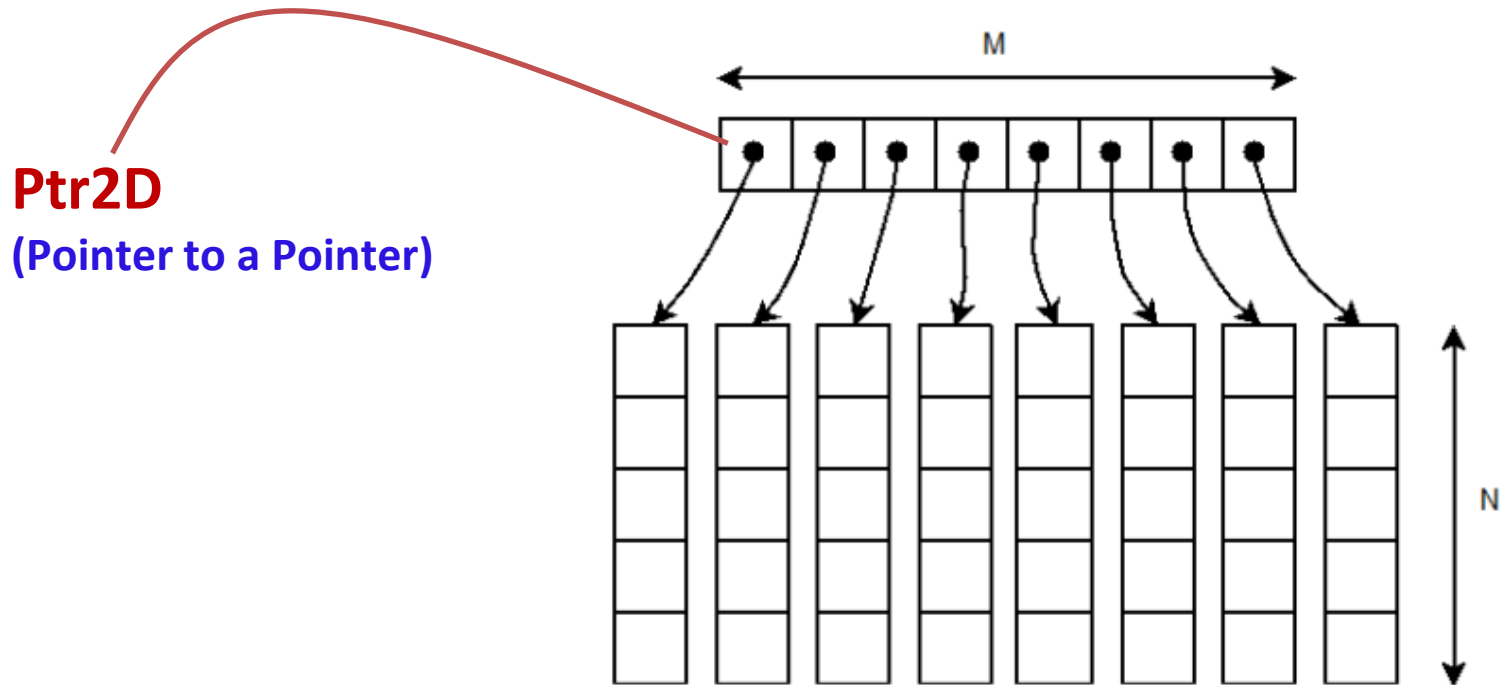
Demo Code:
pointersArrays2.cpp



Dynamic 2D Array – Double Pointer

2. Using a Pointer that points to Array of Pointer

- Total elements in a 2D Array: $M_rows * N_columns$





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 ;
```

Dynamic 2D Arrays

Demo Code:
pointersArrays3.cpp

```
int main() {
    int M=4; int N=6;
    int** A=new int*[M]; srand(time(0));

    for(int i=0;i<M;i++) /*Allocate subarrays */
        A[i]=new int[N];

    for(int i=0;i<M;i++)
        for(int j=0;j<N;j++)
            A[i][j] = rand()%100;

    //display values
    for(int i=0;i<M;i++) {
        for(int j=0;j<N;j++) {
            cout<<A[i][j]<<" ";
        }
        cout<<endl;
    }

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

    delete[] A;
    return 0;
}
```

A ? start of array of pointers

***A** ? First Address pointed by first row (sub-array)

***(A)** ? First value of first array

(A)++ ? Move to next address in the first array

A++ ? Move to Next row (second sub-array)

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

```
// 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;
}
```

Dynamic 2D Array (Varying Row Size)

Output

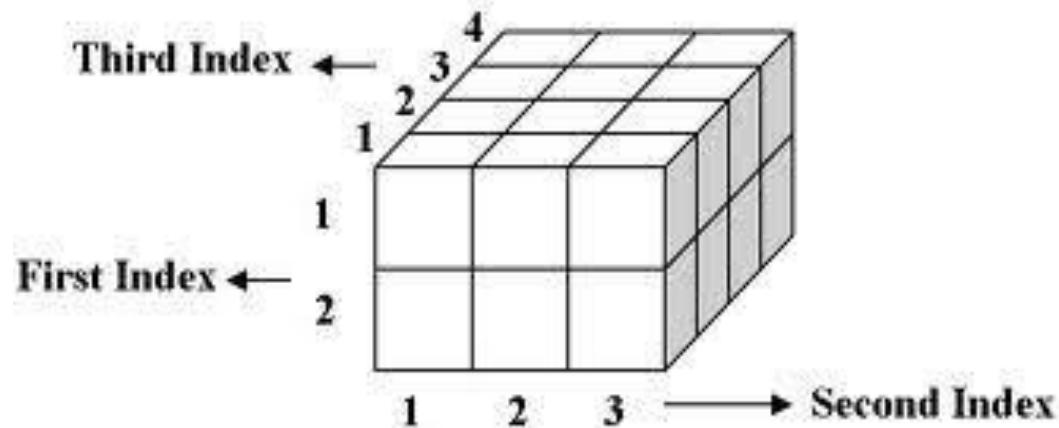
```
83 86 77 15 93
35 86 92 49 21 62
27 90 59 63 26 40 26
72 36 11 68 67 29 82 30
```



Home Work

- Manipulating a 3D Array

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



Three-dimensional array with twenty four elements



Constant Pointer

- A **constant pointer** is a **pointer** that is **constant**, such that we **cannot change** the **location** (address) to which the pointer points to:

```
char c = 'c';
```

```
char d = 'd';
```

```
char* const ptr1 = &c;
```

```
ptr1 = &d; // Not Allowed
```

```
int v1=90;
```

```
int* const ptr2=&v1;
```



Pointer to Constant 1/2

- we **cannot** set a **non-const pointer** to a **const data-item**

```
const int value = 5; // value is const
int *ptr = &value; // compile error: cannot convert
                  // const int* to int*
// To avoid such operations? *ptr = 6;
```

```
const int value = 5;
const int *ptr = &value; // this is okay,
*ptr = 6; // not allowed,
          // we cannot change a const value
```




Pointer to Constant 2/2

- A **pointer** through which we cannot change the **value** of **variable** it **points** is known as a **pointer to constant**.
- These type of pointers **can change** the **address** they point to but **cannot change** the **value** kept at those address.

```
int var1 = 0;  
const int* ptr = &var1;  
*ptr = 1;    // Not Allowed  
cout<<*ptr;
```



Home-Work: char* and const

- **const char *ptr** : This is a pointer to a constant character. You cannot change the value pointed by ptr, but you can change the pointer itself.
- **char* const ptr** : This is a constant pointer to non-constant character. You cannot change the pointer p, but can change the value pointed by ptr.
- **const char* const ptr** : This is a constant pointer to constant character. You can neither change the value pointed by ptr nor the pointer ptr.



C-String and Char Pointer

- A **String**: is simply defined as **an array of characters**
char* s;
// s is the **address** of the **first character** (byte) of the **string**
- A **valid C string ends** with the **null character** **'\0'**
- **Direct initialization** **char* <string Literal>;**

```
char* s="FAST";  
cout<<s<<sizeof(s);  
cout<<++s<<sizeof(s);
```



char [] VS. char *

char A[20]="FAST";

1) A is an Array

1) A++; //invalid

1) sizeof(A) ? 20 Characters or bytes

1) A and &A points to same memory address

5) A="PAKISTAN"; //invalid

A is an address, "PAKISTAN" is the start address where "PAKISTAN" string is stored in memory.

6) A[0]='p'; //Valid

7) A is stored in stack

char* P="FAST";

1) P is a pointer variable

1) P++; //Valid

1) sizeof(P) ? 4 Characters or bytes

1) P points to start address where characters are stored, and &P points to address of pointer variable.

1) P="PAKISTAN" //valid

1) P[0]='p'; //invalid

1) P is stored in Stack, "FAST" is stored in "Text" section (Read-only)



C-String and Char Pointer - Example

```
// Copying string using Pointers
```

```
char* str1 = "Self-conquest is the greatest victory.";
```

```
char str2[80]; //empty string
```

```
char* src = str1;
```

```
char* dest = str2;
```

```
while( *src ) //until null character,
```

```
    *dest++ = *src++; //copy chars from src to dest
```

```
*dest = '\0'; //terminate dest
```

```
cout << str2 << endl; //display str2
```



Functions? Pass by using Reference Pointer

- **Pass-by-reference with pointer** arguments
 - Use pointers as formal parameters and addresses as actual parameters
- Pass address of argument using **&** operator
 - **Arrays not passed with &** because array name already an address
 - Pointers variable are used inside function



Pass by Reference Pointers– Example1

```
void func(int *num)
{
    cout<<"num = "<<*num<<endl;
    *num = 10;
    cout<<"num = "<<*num<<endl;
}

void main()
{
    int n = 5;
    cout<<"Before call: n = "<<n<<endl;
    func(&n);
    cout<<"After call: n = "<<n<<endl;
}
```



Pass by Reference Pointers— Example2

```
void compDouble(int* Ar)
{
    for(int i=0;i<10;i++)
    {
        *Ar=(*Ar)*2;
        Ar++;
    }
}

void main()
{
    int Arr[10]={0,1,2,3,4,5,6,7,8,9};
    compDouble(Arr);
    for(int i=0;i<10;i++)
        cout<<Arr[i]<<endl;
}
```




Any Questions!