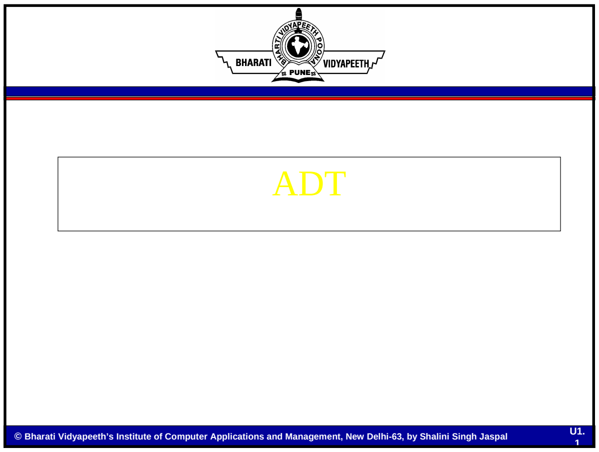
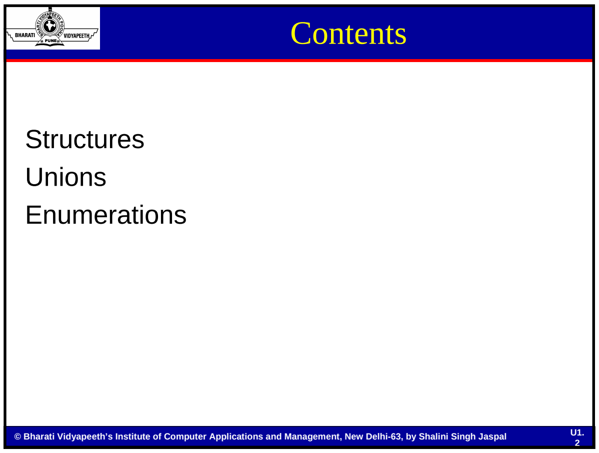


ADT

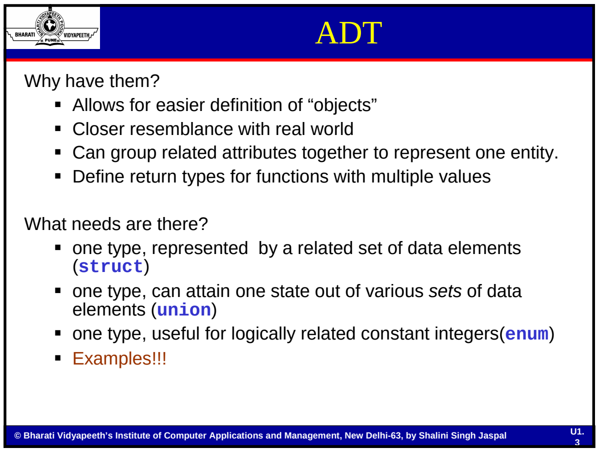


Contents

Structures

Unions

Enumerations



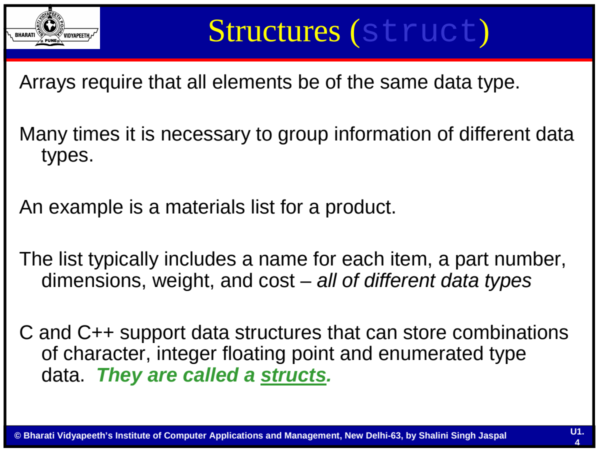
ADT

Why have them?

- Allows for easier definition of “objects”
- Closer resemblance with real world
- Can group related attributes together to represent one entity.
- Define return types for functions with multiple values

What needs are there?

- one type, represented by a related set of data elements (`struct`)
- one type, can attain one state out of various *sets* of data elements (`union`)
- one type, useful for logically related constant integers(`enum`)
- Examples!!!



Structures (`struct`)

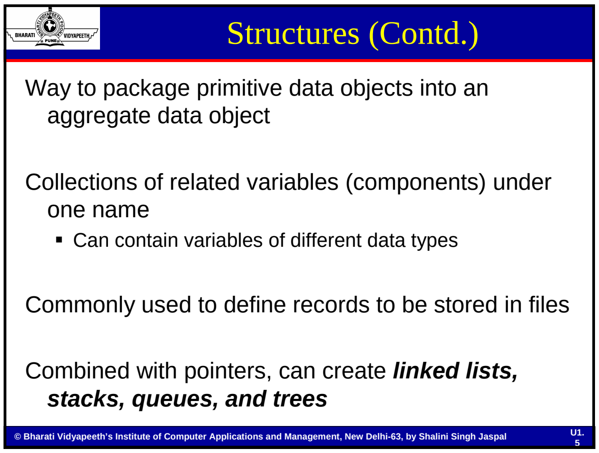
Arrays require that all elements be of the same data type.

Many times it is necessary to group information of different data types.

An example is a materials list for a product.

The list typically includes a name for each item, a part number, dimensions, weight, and cost – *all of different data types*

C and C++ support data structures that can store combinations of character, integer floating point and enumerated type data. ***They are called a structs.***



Structures (Contd.)

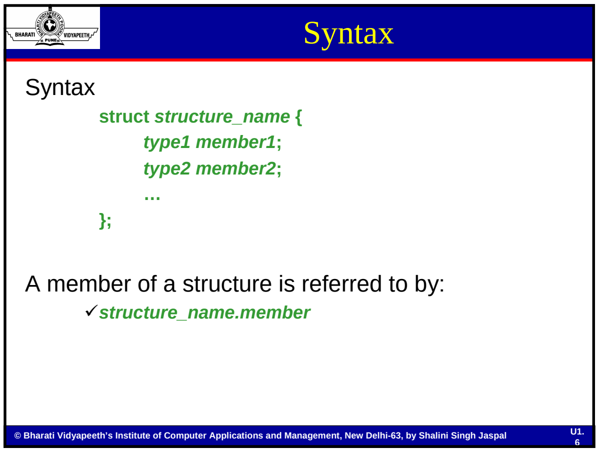
Way to package primitive data objects into an aggregate data object

Collections of related variables (components) under one name

- Can contain variables of different data types

Commonly used to define records to be stored in files

Combined with pointers, can create ***linked lists, stacks, queues, and trees***



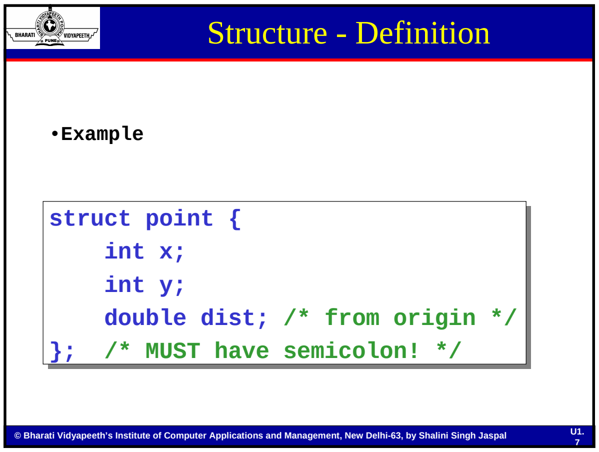
Syntax

Syntax

```
struct structure_name {
    type1 member1;
    type2 member2;
    …
};
```

A member of a structure is referred to by:

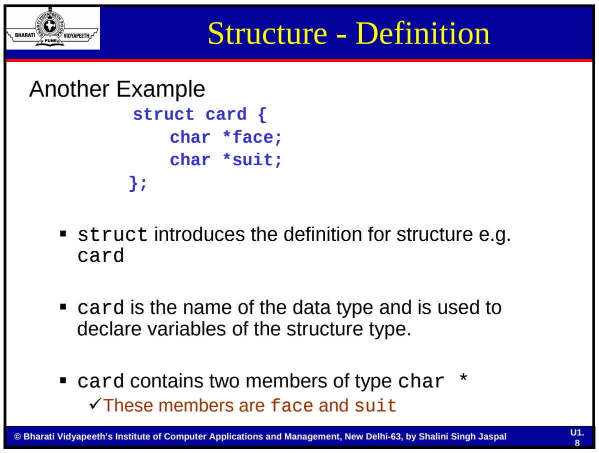
✓*structure_name.member*



Structure - Definition

- **Example**

```
struct point {
    int x;
    int y;
    double dist; /* from origin */
};  /* MUST have semicolon! */
```

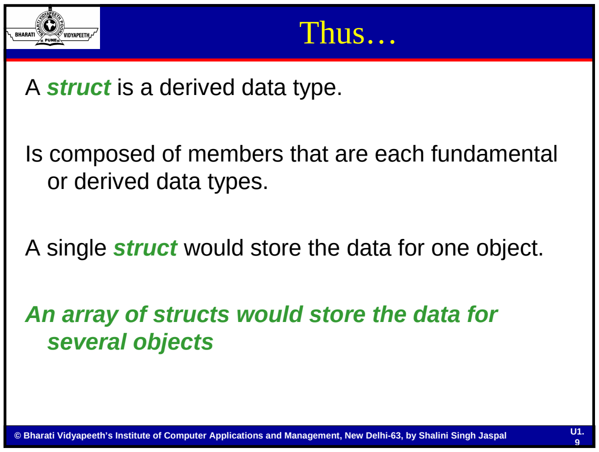


Structure - Definition

Another Example

```
        struct card {
            char *face;
            char *suit;
        };
```

- `struct` introduces the definition for structure e.g. `card`
- `card` is the name of the data type and is used to declare variables of the structure type.
- `card` contains two members of type `char *`
  - ✓These members are `face` and `suit`



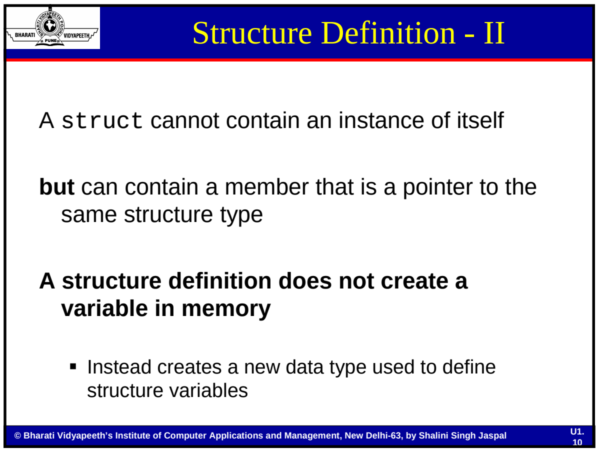
Thus…

A ***struct*** is a derived data type.

Is composed of members that are each fundamental or derived data types.

A single ***struct*** would store the data for one object.

An array of structs would store the data for several objects



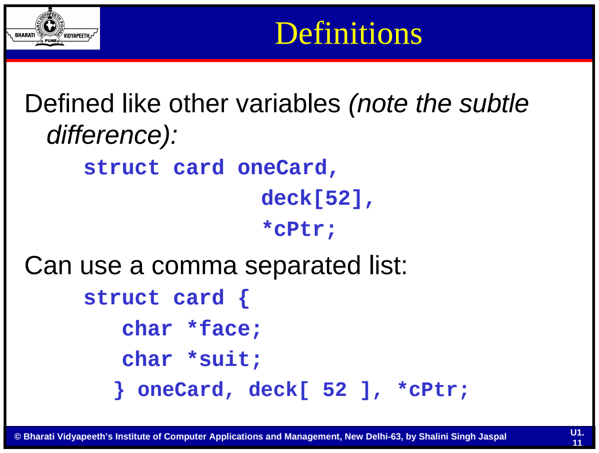
Structure Definition - II

A `struct` cannot contain an instance of itself

but can contain a member that is a pointer to the same structure type

A structure definition does not create a variable in memory

- Instead creates a new data type used to define structure variables



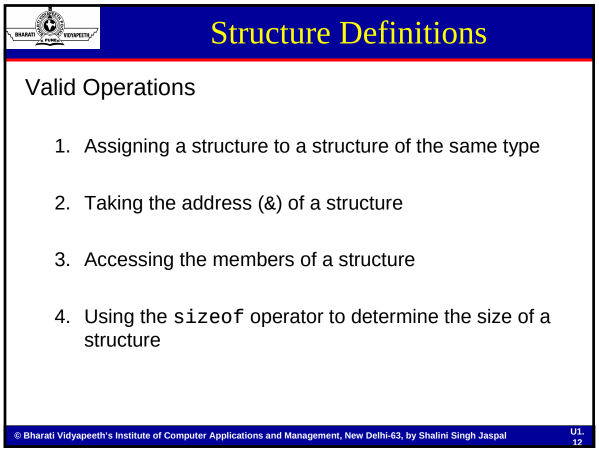
Definitions

Defined like other variables *(note the subtle difference):*

```
struct card oneCard,
            deck[52],
            *cPtr;
```

Can use a comma separated list:

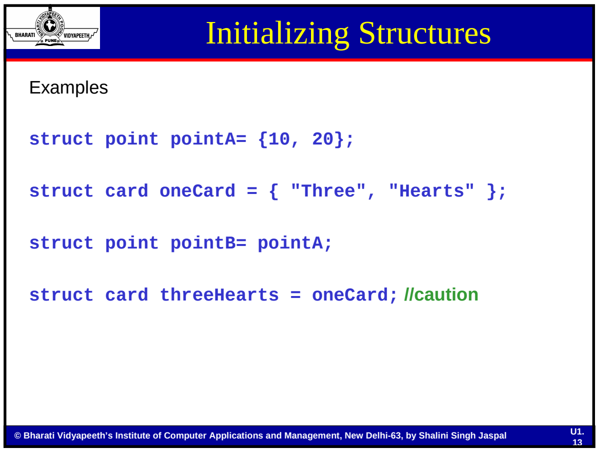
```
struct card {
   char *face;
   char *suit;
  } oneCard, deck[ 52 ], *cPtr;
```



Structure Definitions

Valid Operations

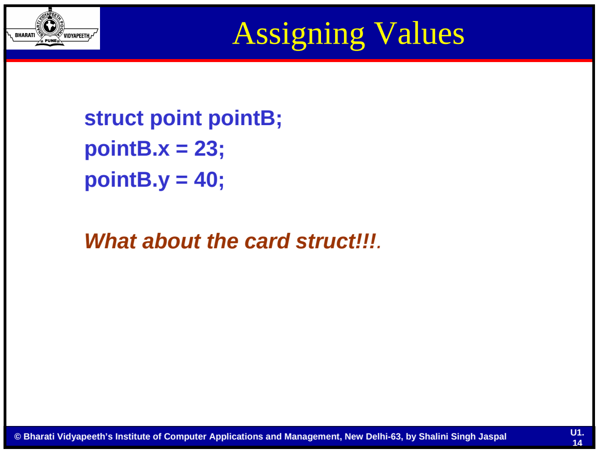
1. Assigning a structure to a structure of the same type
2. Taking the address (&) of a structure
3. Accessing the members of a structure
4. Using the `sizeof` operator to determine the size of a structure



Initializing Structures

Examples

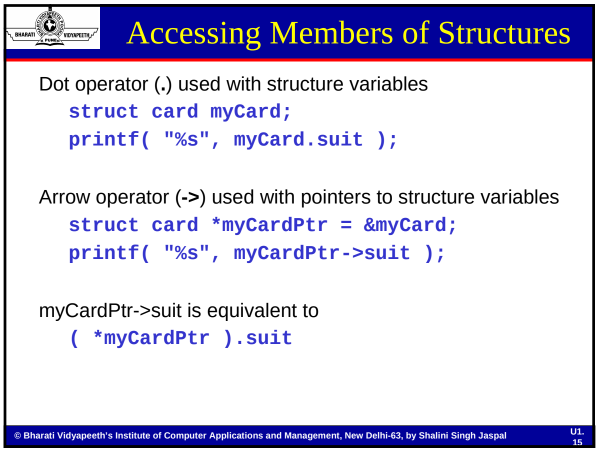
```
struct point pointA= {10, 20};

struct card oneCard = { "Three", "Hearts" };

struct point pointB= pointA;

struct card threeHearts = oneCard; //caution
```



Assigning Values

```
struct point pointB;
pointB.x = 23;
pointB.y = 40;
```

What about the card struct!!!.



Accessing Members of Structures

Dot operator (**.**) used with structure variables

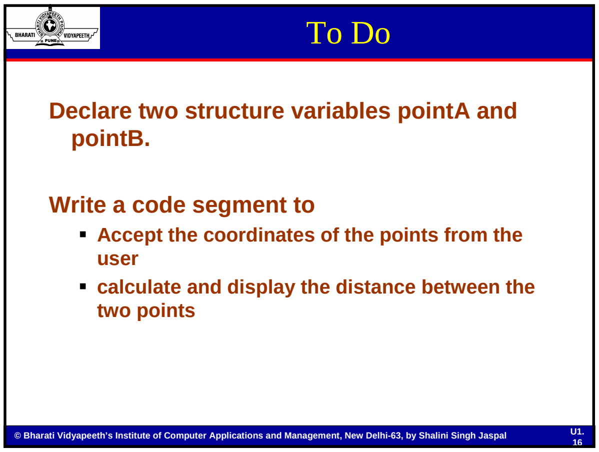
```
struct card myCard;
printf( "%s", myCard.suit );
```

Arrow operator (**->**) used with pointers to structure variables

```
struct card *myCardPtr = &myCard;
printf( "%s", myCardPtr->suit );
```

myCardPtr->suit is equivalent to

```
( *myCardPtr ).suit
```

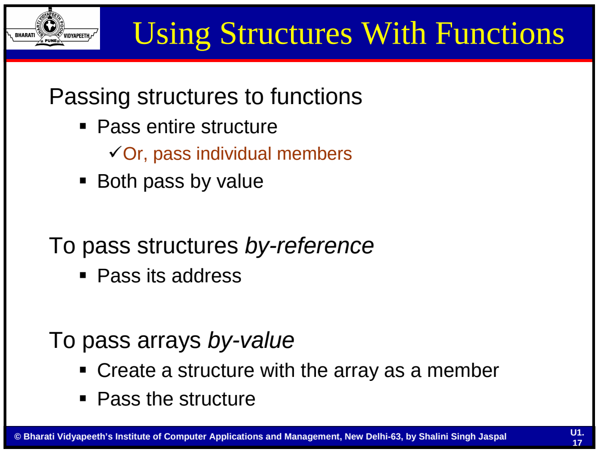


To Do

Declare two structure variables pointA and pointB.

Write a code segment to

- Accept the coordinates of the points from the user
- calculate and display the distance between the two points



Using Structures With Functions

Passing structures to functions

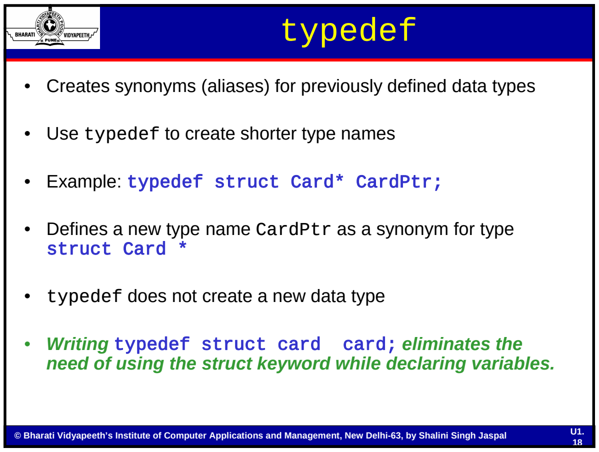
- Pass entire structure
  - ✓Or, pass individual members
- Both pass by value

To pass structures *by-reference*

- Pass its address

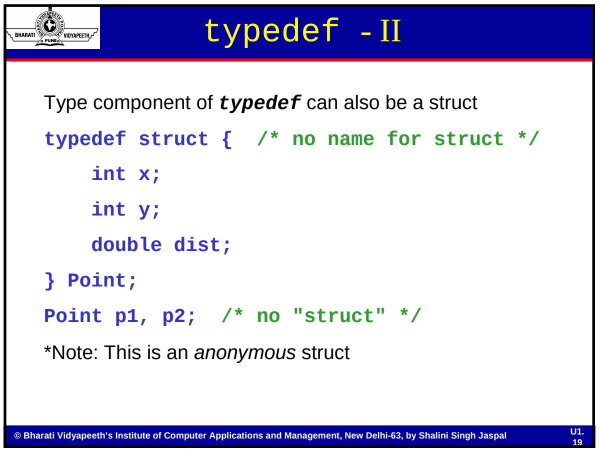
To pass arrays *by-value*

- Create a structure with the array as a member
- Pass the structure



`typedef`

- Creates synonyms (aliases) for previously defined data types
- Use `typedef` to create shorter type names
- Example: `typedef struct Card* CardPtr;`
- Defines a new type name `CardPtr` as a synonym for type `struct Card *`
- `typedef` does not create a new data type
- ***Writing `typedef struct card  card;` eliminates the need of using the struct keyword while declaring variables.***

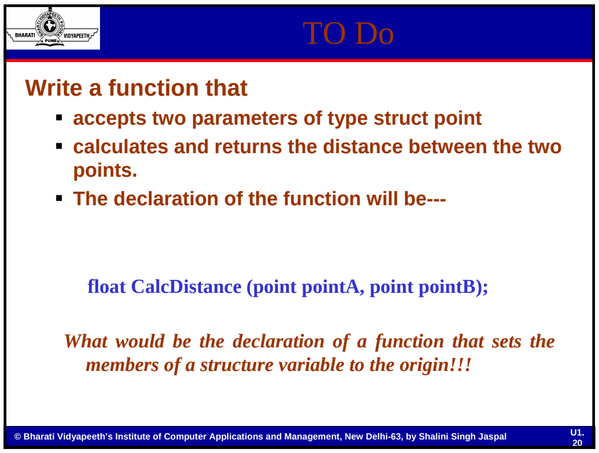


typedef - II

Type component of ***typedef*** can also be a struct

```
typedef struct {  /* no name for struct */
    int x;
    int y;
    double dist;
} Point;
Point p1, p2;  /* no "struct" */
```

*Note: This is an *anonymous* struct



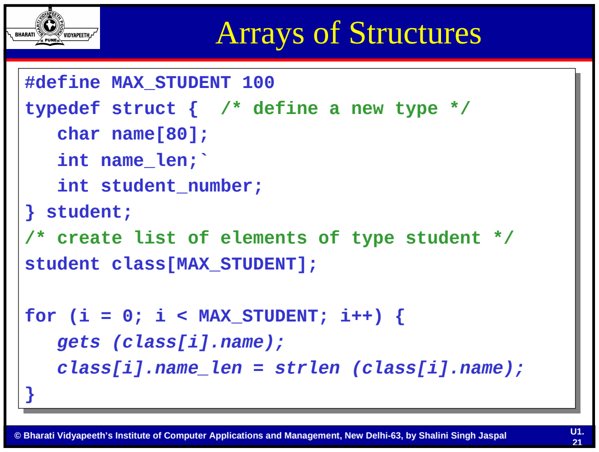
TO Do

Write a function that

- **accepts two parameters of type struct point**
- **calculates and returns the distance between the two points.**
- **The declaration of the function will be---**

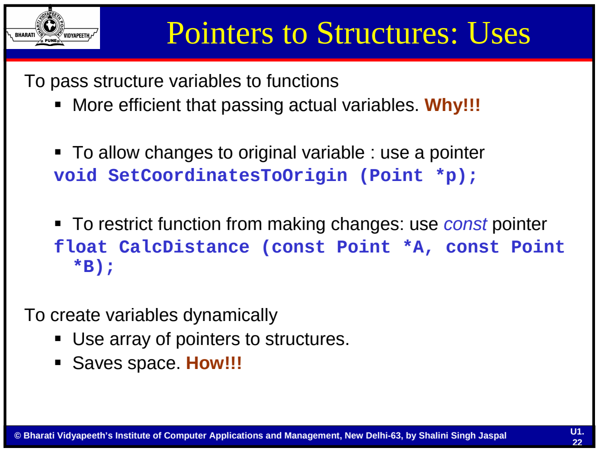
float CalcDistance (point pointA, point pointB);

What would be the declaration of a function that sets the members of a structure variable to the origin!!!



Arrays of Structures

```
#define MAX_STUDENT 100
typedef struct {  /* define a new type */
   char name[80];
   int name_len;`
   int student_number;
} student;
/* create list of elements of type student */
student class[MAX_STUDENT];

for (i = 0; i < MAX_STUDENT; i++) {
   gets (class[i].name);
   class[i].name_len = strlen (class[i].name);
}
```



Pointers to Structures: Uses

To pass structure variables to functions

- More efficient that passing actual variables. **Why!!!**

- To allow changes to original variable : use a pointer

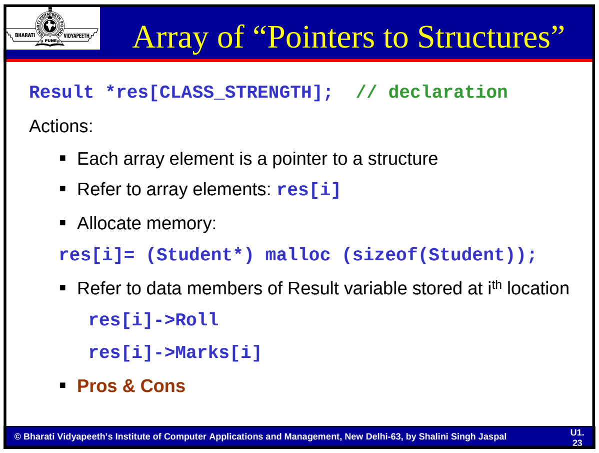
```
void SetCoordinatesToOrigin (Point *p);
```

- To restrict function from making changes: use *const* pointer

```
float CalcDistance (const Point *A, const Point
  *B);
```

To create variables dynamically

- Use array of pointers to structures.
- Saves space. **How!!!**



Array of "Pointers to Structures"

```
Result *res[CLASS_STRENGTH];  // declaration
```

Actions:

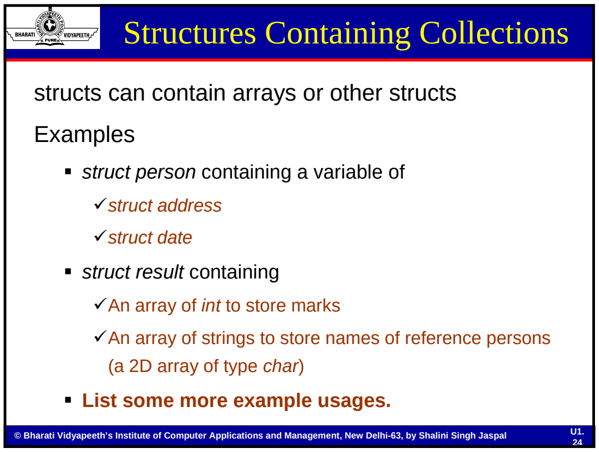
- Each array element is a pointer to a structure
- Refer to array elements: `res[i]`
- Allocate memory:

```
res[i]= (Student*) malloc (sizeof(Student));
```

- Refer to data members of Result variable stored at ith location

```
res[i]->Roll
res[i]->Marks[i]
```

- **Pros & Cons**

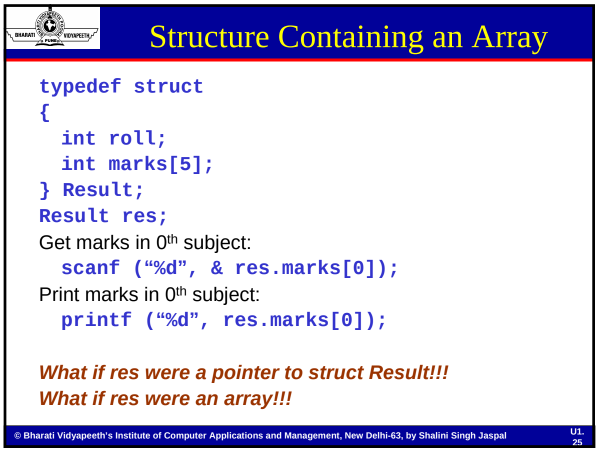


Structures Containing Collections

structs can contain arrays or other structs

Examples

- *struct person* containing a variable of
  - ✓*struct address*
  - ✓*struct date*
- *struct result* containing
  - ✓An array of *int* to store marks
  - ✓An array of strings to store names of reference persons (a 2D array of type *char*)
- **List some more example usages.**



Structure Containing an Array

```
typedef struct
{
  int roll;
  int marks[5];
} Result;
Result res;
```

Get marks in 0th subject:

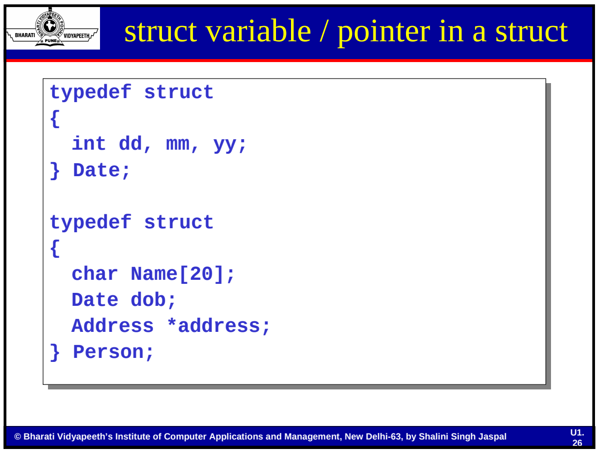
```
  scanf (“%d”, & res.marks[0]);
```

Print marks in 0th subject:

```
  printf (“%d”, res.marks[0]);
```

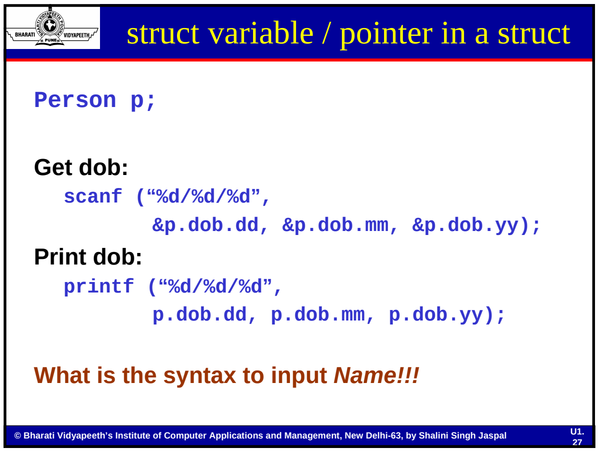
What if res were a pointer to struct Result!!!

What if res were an array!!!



struct variable / pointer in a struct

```
typedef struct
{
  int dd, mm, yy;
} Date;

typedef struct
{
  char Name[20];
  Date dob;
  Address *address;
} Person;
```



struct variable / pointer in a struct

```
Person p;
```

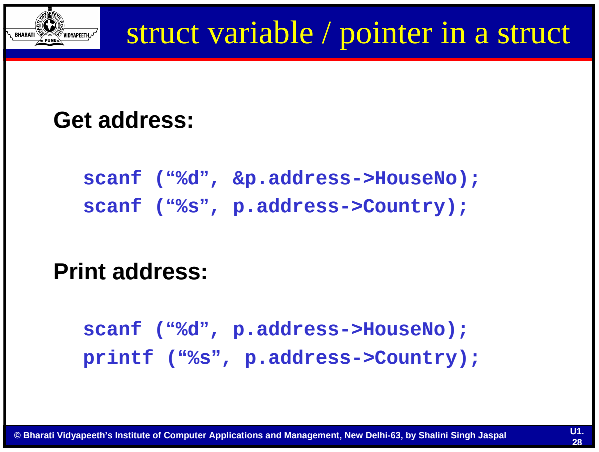
Get dob:

```
   scanf (“%d/%d/%d”,
           &p.dob.dd, &p.dob.mm, &p.dob.yy);
```

Print dob:

```
   printf (“%d/%d/%d”,
           p.dob.dd, p.dob.mm, p.dob.yy);
```

What is the syntax to input *Name!!!*



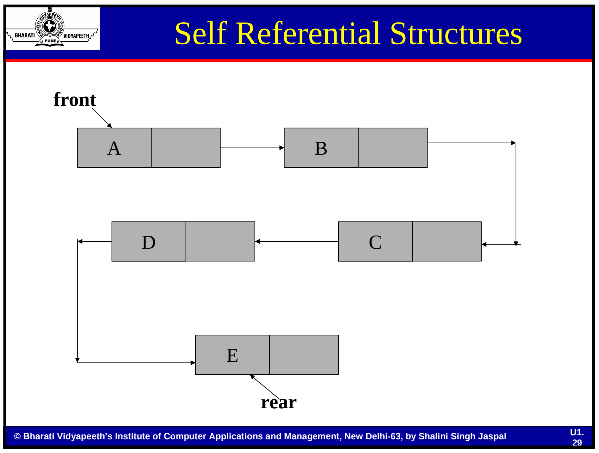
struct variable / pointer in a struct

Get address:

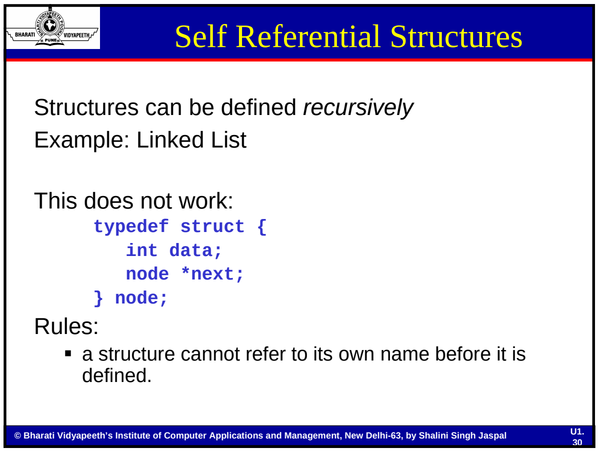
```
scanf (“%d”, &p.address->HouseNo);
scanf (“%s”, p.address->Country);
```

Print address:

```
scanf (“%d”, p.address->HouseNo);
printf (“%s”, p.address->Country);
```



Self Referential Structures



Self Referential Structures

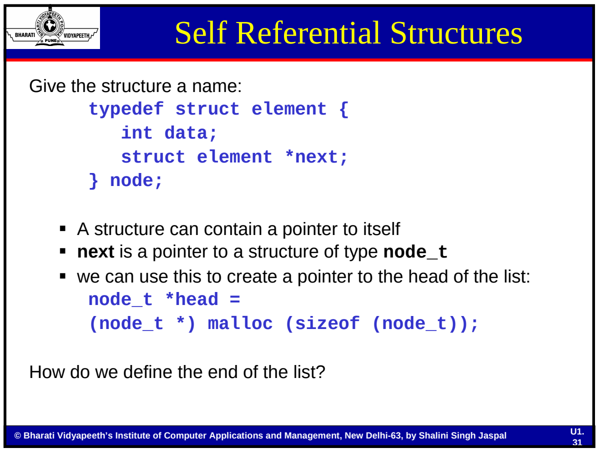
Structures can be defined *recursively*
Example: Linked List

This does not work:

```
typedef struct {
   int data;
   node *next;
} node;
```

Rules:

- a structure cannot refer to its own name before it is defined.



Self Referential Structures

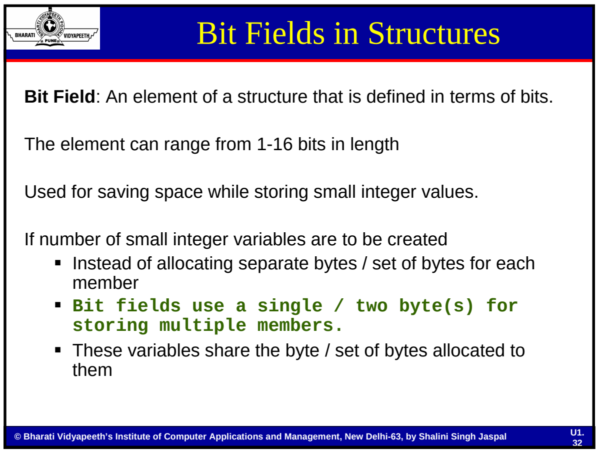
Give the structure a name:

```
typedef struct element {
   int data;
   struct element *next;
} node;
```

- A structure can contain a pointer to itself
- **next** is a pointer to a structure of type **node_t**
- we can use this to create a pointer to the head of the list:

```
node_t *head =
(node_t *) malloc (sizeof (node_t));
```

How do we define the end of the list?



Bit Fields in Structures

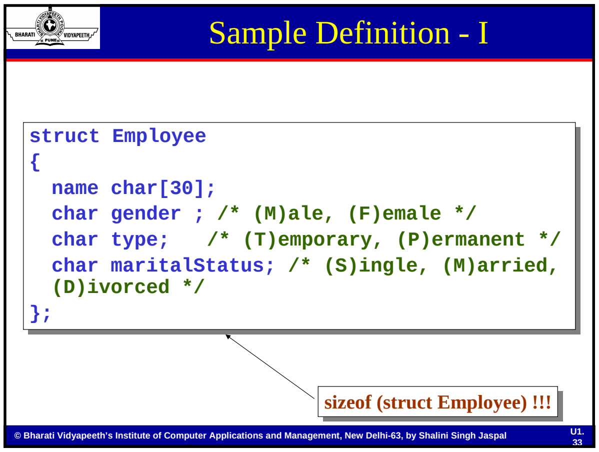
Bit Field: An element of a structure that is defined in terms of bits.

The element can range from 1-16 bits in length

Used for saving space while storing small integer values.

If number of small integer variables are to be created

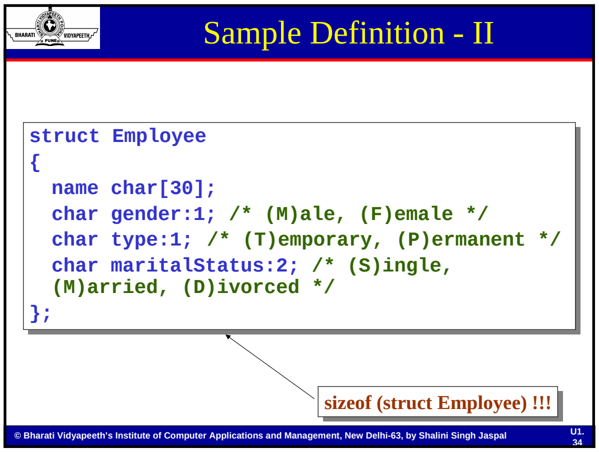
- Instead of allocating separate bytes / set of bytes for each member
- `Bit fields use a single / two byte(s) for storing multiple members.`
- These variables share the byte / set of bytes allocated to them



Sample Definition - I

```
struct Employee
{
  name char[30];
  char gender ; /* (M)ale, (F)emale */
  char type;   /* (T)emporary, (P)ermanent */
  char maritalStatus; /* (S)ingle, (M)arried,
  (D)ivorced */
};
```

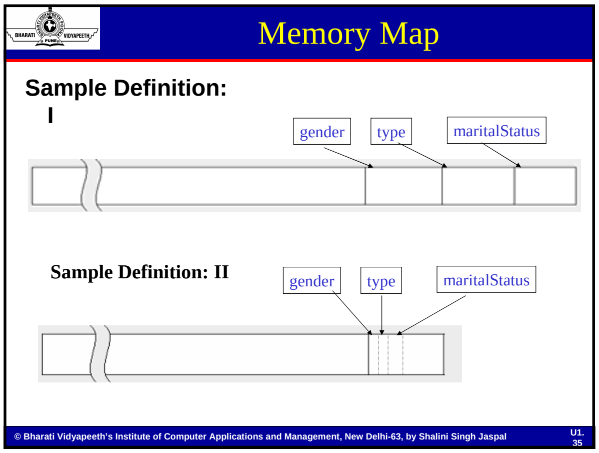
sizeof (struct Employee) !!!



Sample Definition - II

```
struct Employee
{
  name char[30];
  char gender:1; /* (M)ale, (F)emale */
  char type:1; /* (T)emporary, (P)ermanent */
  char maritalStatus:2; /* (S)ingle,
  (M)arried, (D)ivorced */
};
```

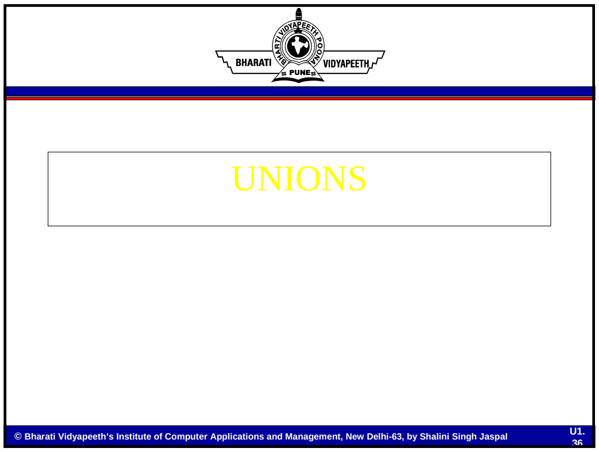
sizeof (struct Employee) !!!



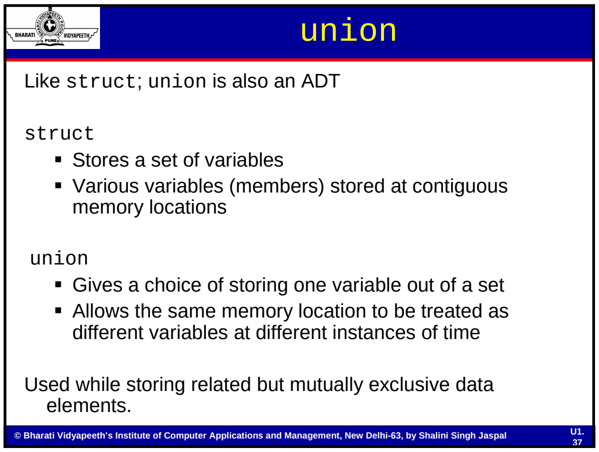
Memory Map

Sample Definition: I

Sample Definition: II



UNIONS



union

Like `struct`; `union` is also an ADT

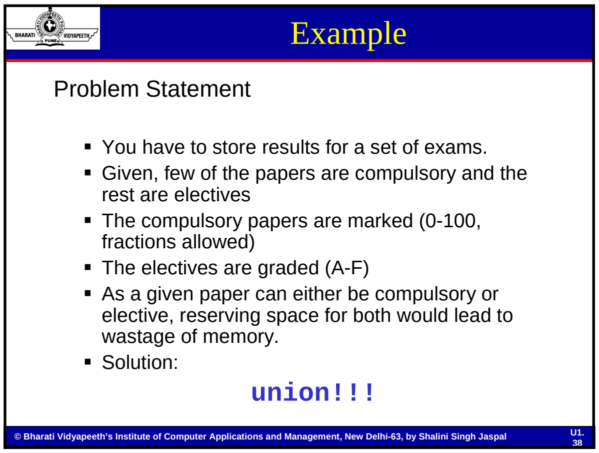
`struct`

- Stores a set of variables
- Various variables (members) stored at contiguous memory locations

`union`

- Gives a choice of storing one variable out of a set
- Allows the same memory location to be treated as different variables at different instances of time

Used while storing related but mutually exclusive data elements.

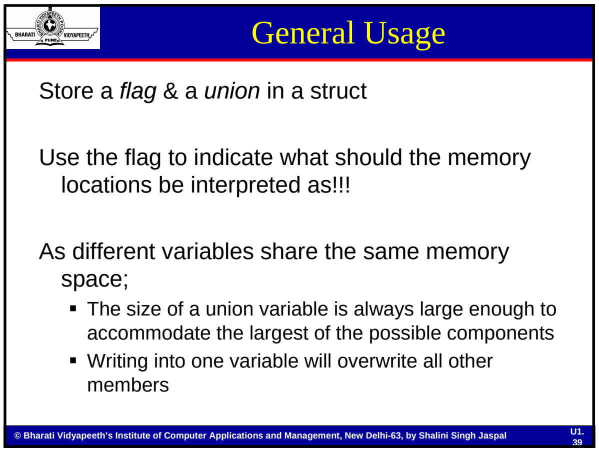


Example

Problem Statement

- You have to store results for a set of exams.
- Given, few of the papers are compulsory and the rest are electives
- The compulsory papers are marked (0-100, fractions allowed)
- The electives are graded (A-F)
- As a given paper can either be compulsory or elective, reserving space for both would lead to wastage of memory.
- Solution:

`union!!!`



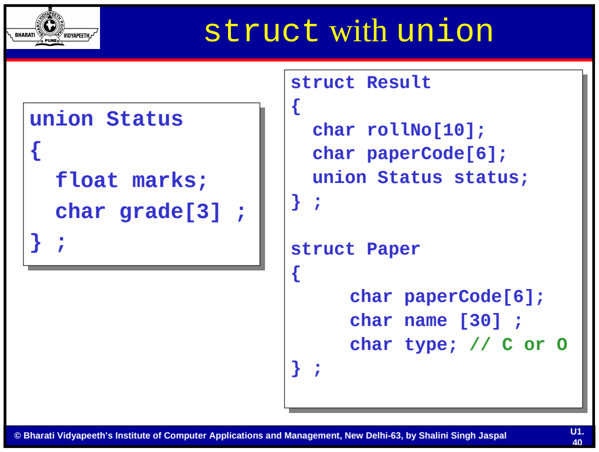
General Usage

Store a *flag* & a *union* in a struct

Use the flag to indicate what should the memory locations be interpreted as!!!

As different variables share the same memory space;

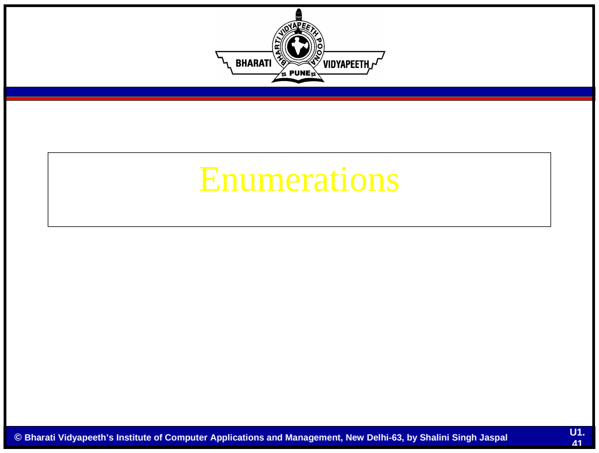
- The size of a union variable is always large enough to accommodate the largest of the possible components
- Writing into one variable will overwrite all other members



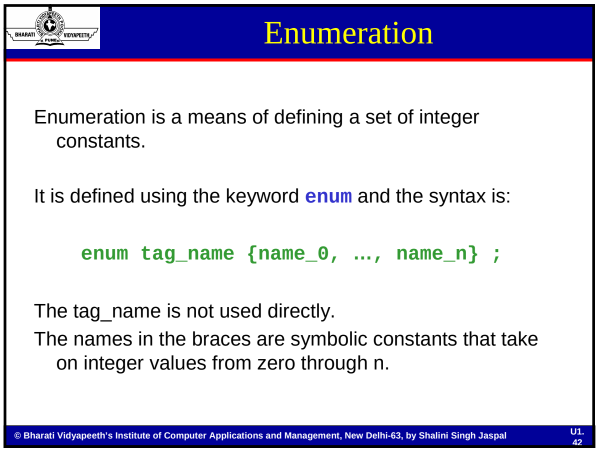
struct with union

```
union Status
{
  float marks;
  char grade[3] ;
} ;
```

```
struct Result
{
  char rollNo[10];
  char paperCode[6];
  union Status status;
} ;

struct Paper
{
     char paperCode[6];
     char name [30] ;
     char type; // C or O
} ;
```



Enumerations



Enumeration

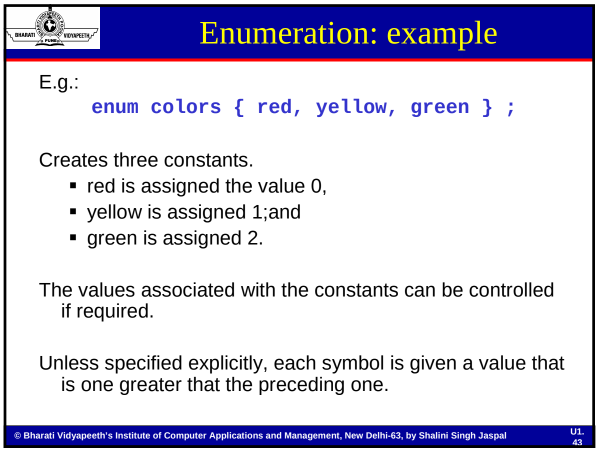
Enumeration is a means of defining a set of integer constants.

It is defined using the keyword **enum** and the syntax is:

```
enum tag_name {name_0, …, name_n} ;
```

The tag_name is not used directly.

The names in the braces are symbolic constants that take on integer values from zero through n.



Enumeration: example

E.g.:

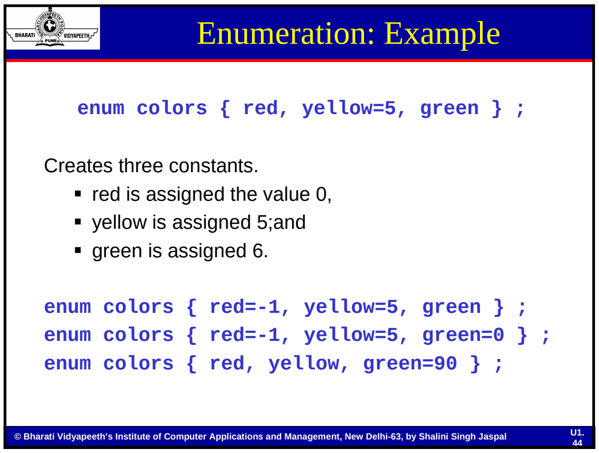
```
enum colors { red, yellow, green } ;
```

Creates three constants.

- red is assigned the value 0,
- yellow is assigned 1;and
- green is assigned 2.

The values associated with the constants can be controlled if required.

Unless specified explicitly, each symbol is given a value that is one greater that the preceding one.



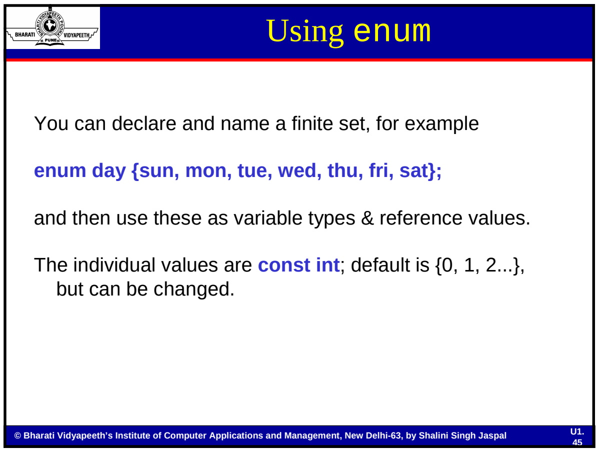
Enumeration: Example

```
enum colors { red, yellow=5, green } ;
```

Creates three constants.

- red is assigned the value 0,
- yellow is assigned 5;and
- green is assigned 6.

```
enum colors { red=-1, yellow=5, green } ;
enum colors { red=-1, yellow=5, green=0 } ;
enum colors { red, yellow, green=90 } ;
```



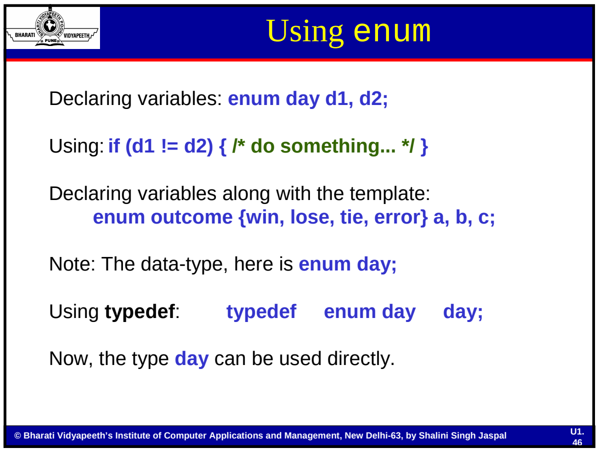
Using enum

You can declare and name a finite set, for example

enum day {sun, mon, tue, wed, thu, fri, sat};

and then use these as variable types & reference values.

The individual values are **const int**; default is {0, 1, 2...}, but can be changed.



Using enum

Declaring variables: **enum day d1, d2;**

Using: **if (d1 != d2) { /* do something... */ }**

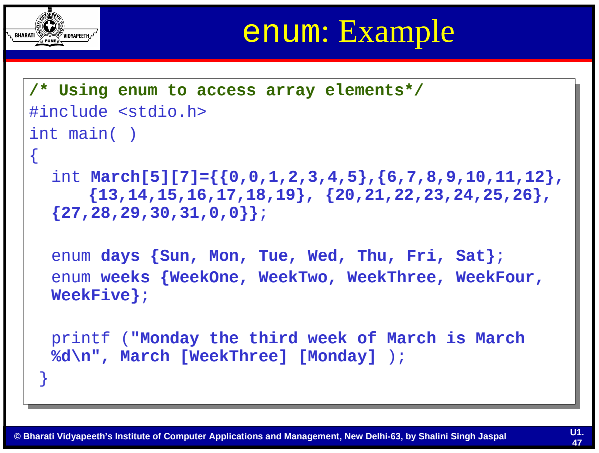
Declaring variables along with the template:

enum outcome {win, lose, tie, error} a, b, c;

Note: The data-type, here is **enum day;**

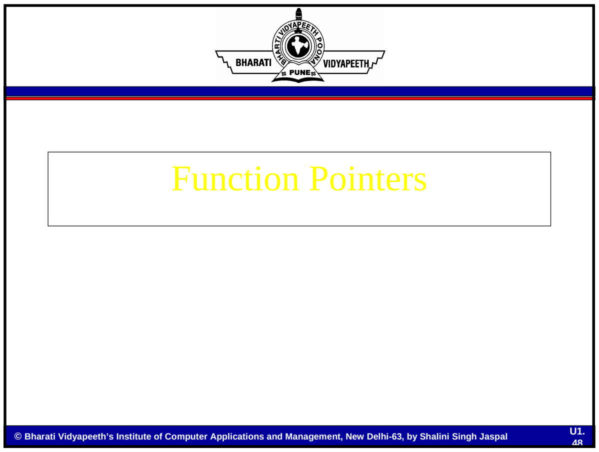
Using **typedef**: **typedef enum day day;**

Now, the type **day** can be used directly.

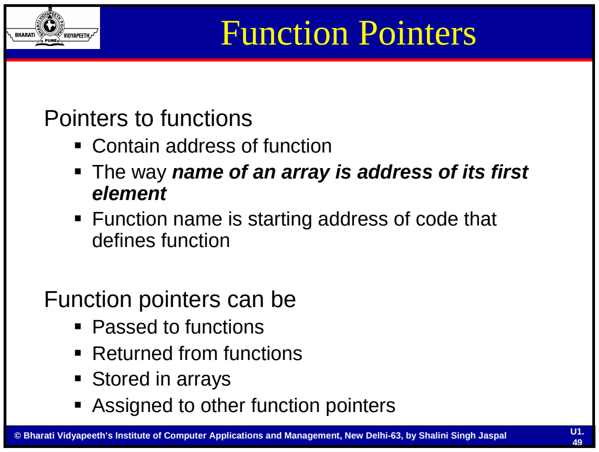


enum: Example

```
/* Using enum to access array elements*/
#include <stdio.h>
int main( )
{
  int March[5][7]={{0,0,1,2,3,4,5},{6,7,8,9,10,11,12},
      {13,14,15,16,17,18,19}, {20,21,22,23,24,25,26},
  {27,28,29,30,31,0,0}};

  enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
  enum weeks {WeekOne, WeekTwo, WeekThree, WeekFour,
  WeekFive};

  printf ("Monday the third week of March is March
  %d\n", March [WeekThree] [Monday] );
 }
```



Function Pointers



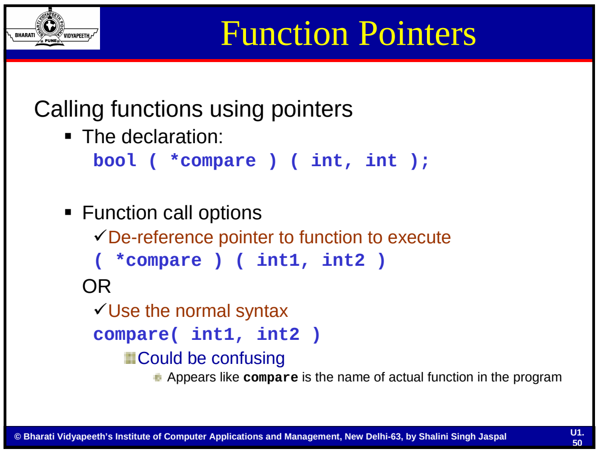
Function Pointers

Pointers to functions

- Contain address of function
- The way ***name of an array is address of its first element***
- Function name is starting address of code that defines function

Function pointers can be

- Passed to functions
- Returned from functions
- Stored in arrays
- Assigned to other function pointers



Function Pointers

Calling functions using pointers

- The declaration:

```
bool ( *compare ) ( int, int );
```

- Function call options
  - ✓De-reference pointer to function to execute

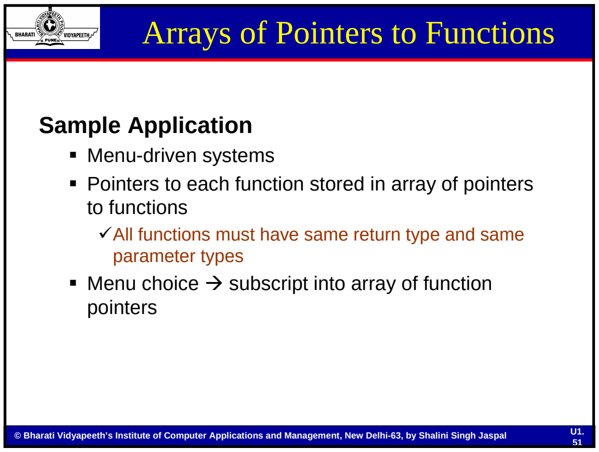
```
( *compare ) ( int1, int2 )
```

OR

  - ✓Use the normal syntax

```
compare( int1, int2 )
```

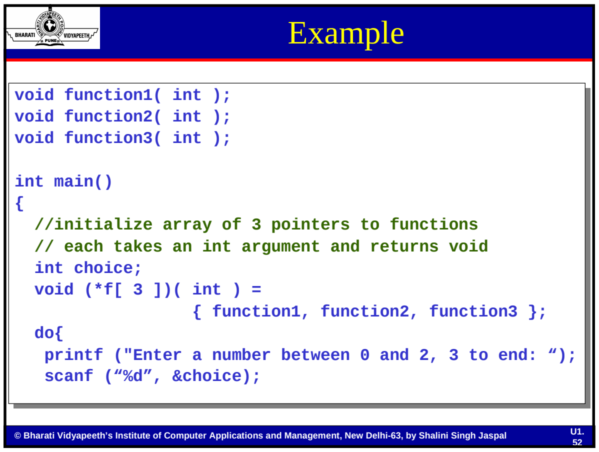
    - Could be confusing
      - Appears like **`compare`** is the name of actual function in the program



Arrays of Pointers to Functions

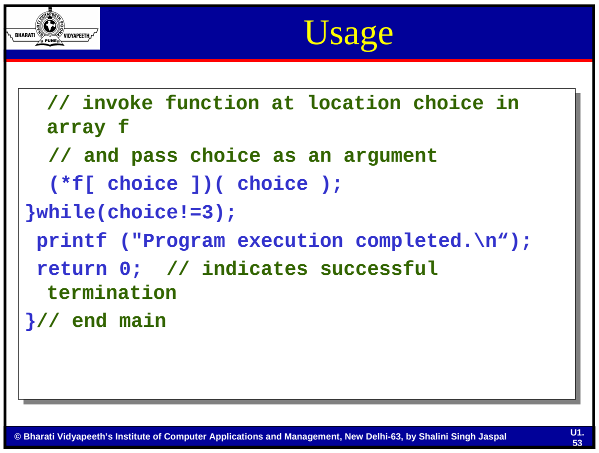
Sample Application

- Menu-driven systems
- Pointers to each function stored in array of pointers to functions
  - ✓All functions must have same return type and same parameter types
- Menu choice → subscript into array of function pointers



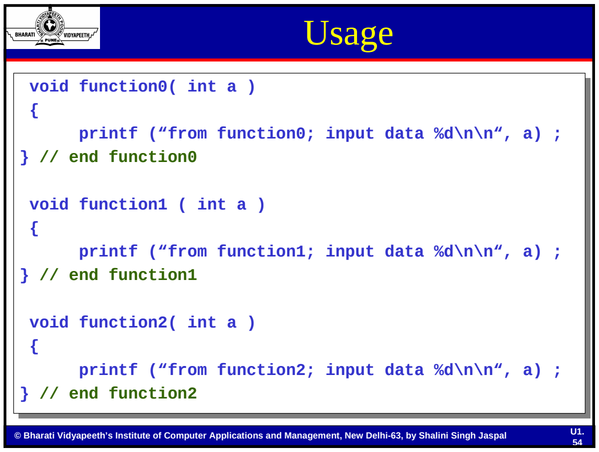
Example

```
void function1( int );
void function2( int );
void function3( int );

int main()
{
  //initialize array of 3 pointers to functions
  // each takes an int argument and returns void
  int choice;
  void (*f[ 3 ])( int ) =
                { function1, function2, function3 };
  do{
   printf ("Enter a number between 0 and 2, 3 to end: “);
   scanf (“%d”, &choice);
```



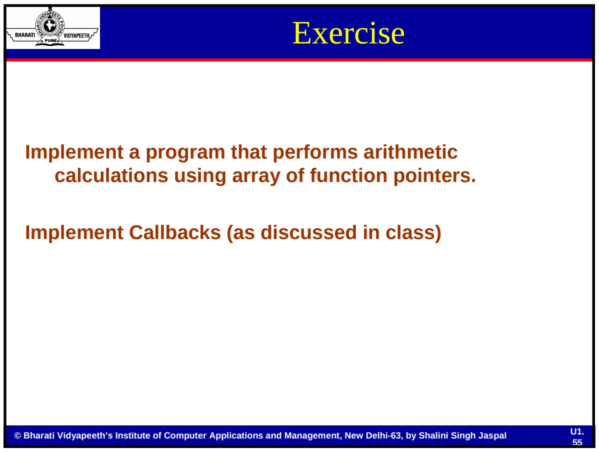
Usage

```
  // invoke function at location choice in
  array f
  // and pass choice as an argument
  (*f[ choice ])( choice );
}while(choice!=3);
 printf ("Program execution completed.\n“);
 return 0;  // indicates successful
  termination
}// end main
```



Usage

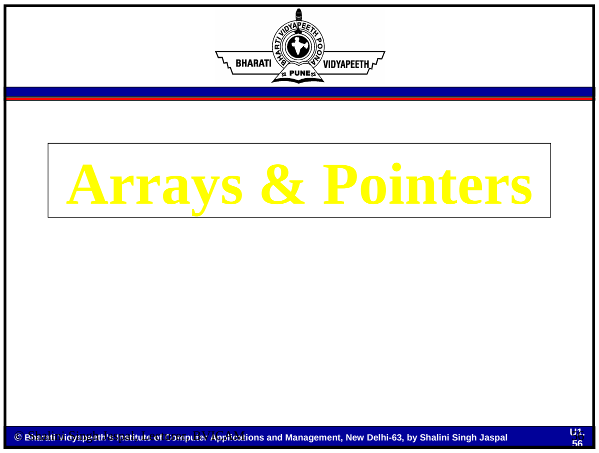
```
 void function0( int a )
 {
      printf (“from function0; input data %d\n\n“, a) ;
} // end function0

 void function1 ( int a )
 {
      printf (“from function1; input data %d\n\n“, a) ;
} // end function1

 void function2( int a )
 {
      printf (“from function2; input data %d\n\n“, a) ;
} // end function2
```



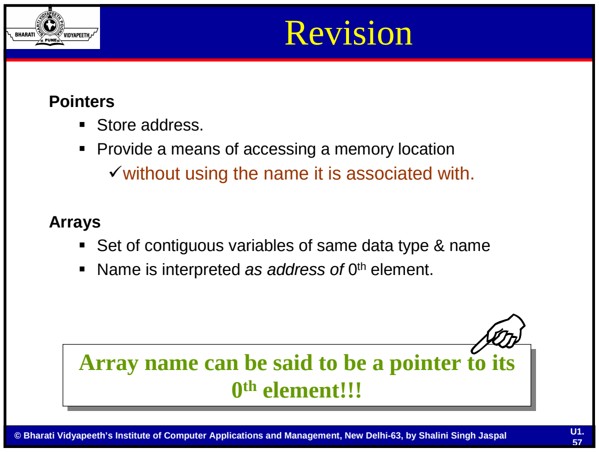
Exercise

Implement a program that performs arithmetic calculations using array of function pointers.

Implement Callbacks (as discussed in class)



Arrays & Pointers



Revision

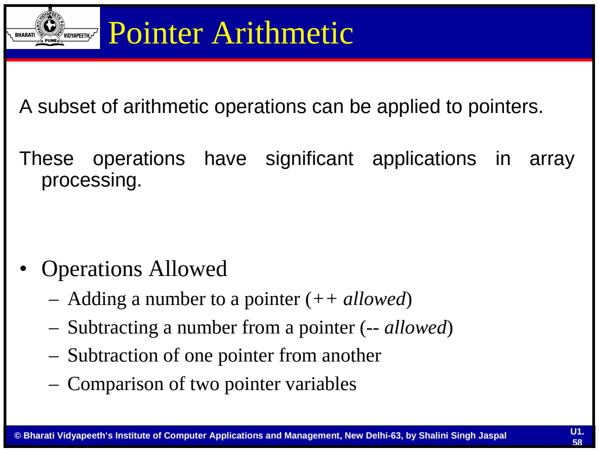
Pointers

- Store address.
- Provide a means of accessing a memory location
  - ✓without using the name it is associated with.

Arrays

- Set of contiguous variables of same data type & name
- Name is interpreted *as address of* 0th element.

Array name can be said to be a pointer to its 0th element!!!

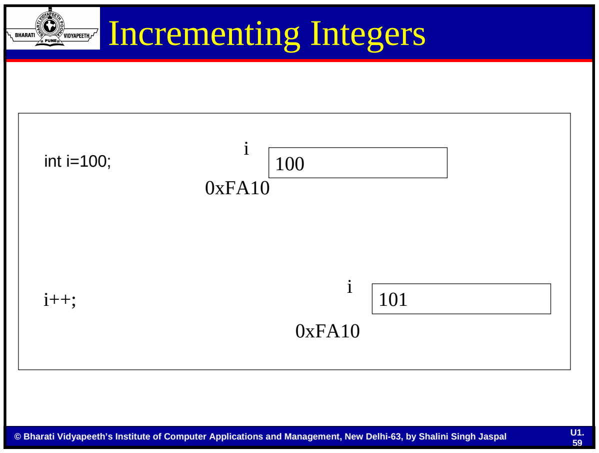


Pointer Arithmetic

A subset of arithmetic operations can be applied to pointers.

These operations have significant applications in array processing.

- Operations Allowed
  - Adding a number to a pointer (++ *allowed*)
  - Subtracting a number from a pointer (-- *allowed*)
  - Subtraction of one pointer from another
  - Comparison of two pointer variables



Incrementing Integers

int i=100;

i

100

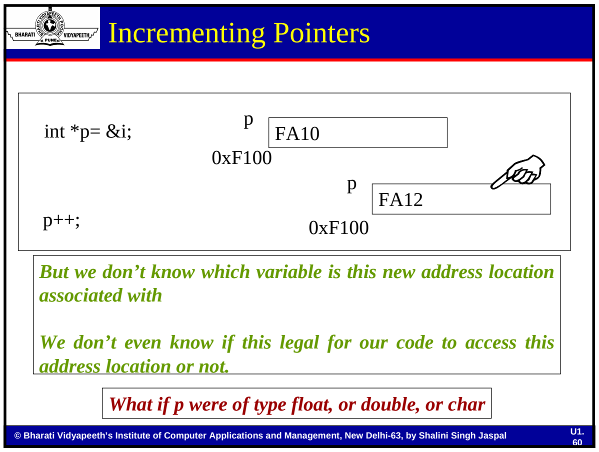
0xFA10

i++;

i

101

0xFA10



Incrementing Pointers

int *p= &i;

p

FA10

0xF100

p

FA12

p++;

0xF100

But we don’t know which variable is this new address location associated with

We don’t even know if this legal for our code to access this address location or not.

What if p were of type float, or double, or char

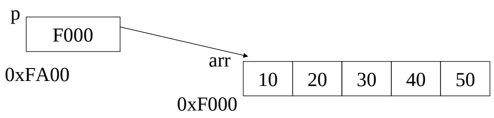
Adding Numbers to Pointers

Adding (Subtracting) a number to (from) a pointer

- Makes the pointer point to a new memory location
- This location lies n bytes ahead (behind) the location where the pointer was initially pointing.
  - ✓ *Here n is the size of a variable having the same data-type as that of the pointer.*
- This operation is of little use when dealing with simple variables
- But can be used to process arrays efficiently. **How!!!**

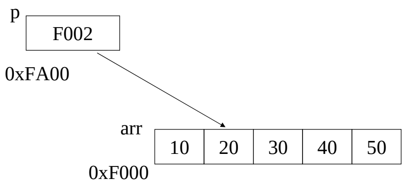
Pointer Arithmetic & Array Processing

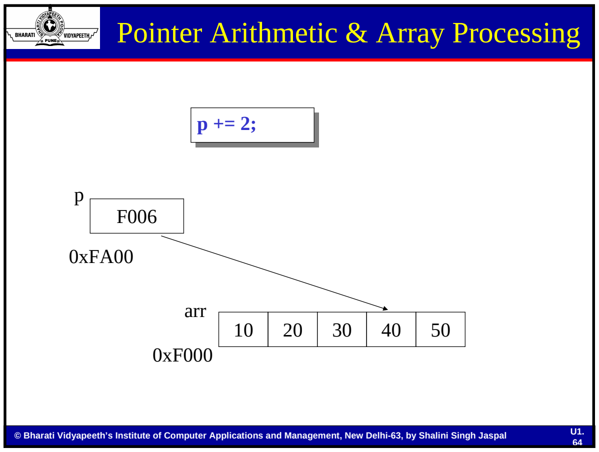
```
int arr[5]= {10, 20, 30, 40, 50}, *p;
p= arr;
```



Pointer Arithmetic & Array Processing

```
p++;
```





Pointer Arithmetic & Array Processing

p += 2;

p

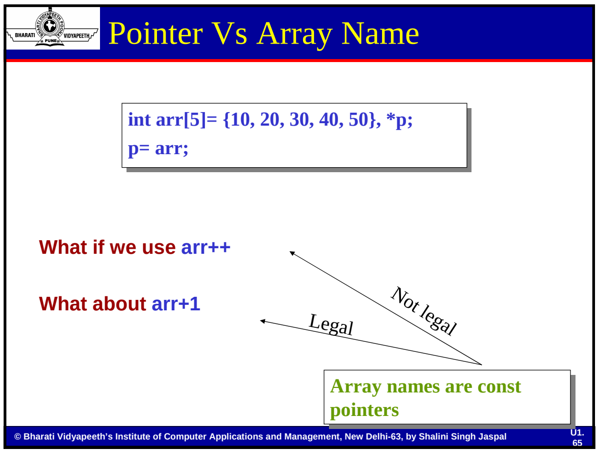
F006

0xFA00

arr

10	20	30	40	50

0xF000



Pointer Vs Array Name

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

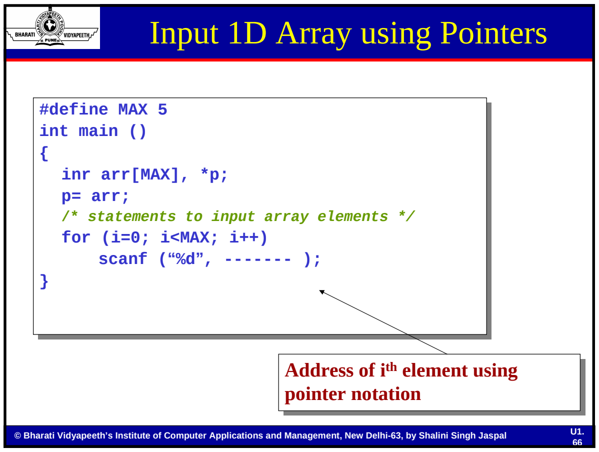
What if we use arr++

What about arr+1

Not legal

Legal

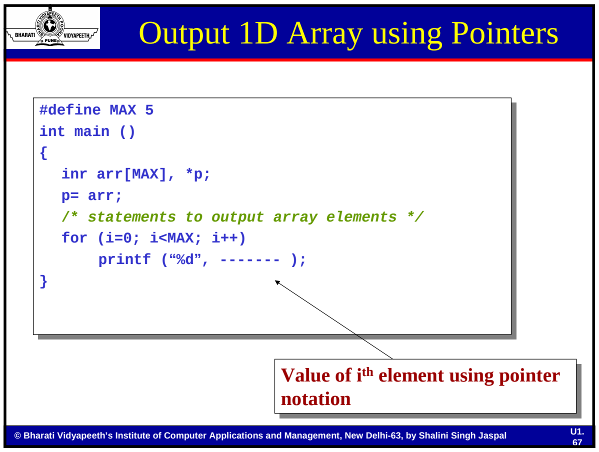
Array names are const pointers



Input 1D Array using Pointers

```
#define MAX 5
int main ()
{
  inr arr[MAX], *p;
  p= arr;
  /* statements to input array elements */
  for (i=0; i<MAX; i++)
      scanf (“%d”, ------- );
}
```

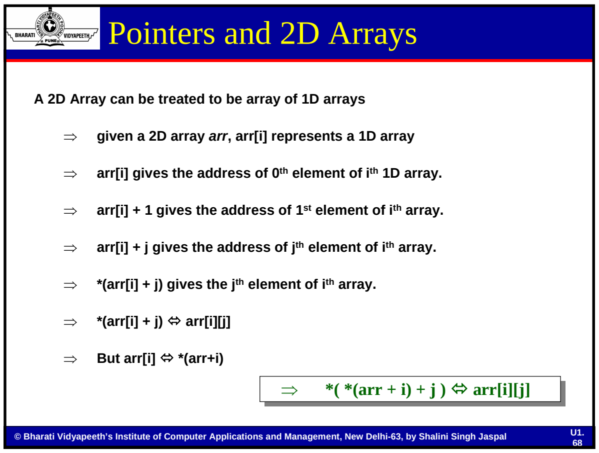
Address of i^{th} element using pointer notation



Output 1D Array using Pointers

```
#define MAX 5
int main ()
{
   inr arr[MAX], *p;
   p= arr;
   /* statements to output array elements */
   for (i=0; i<MAX; i++)
        printf (“%d”, ------- );
}
```

Value of i^{th} element using pointer notation

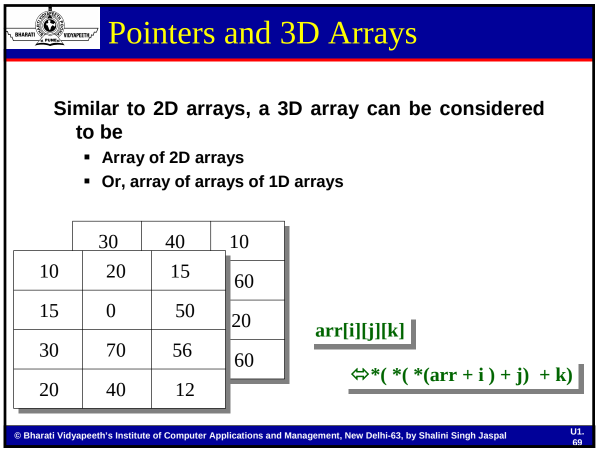


Pointers and 2D Arrays

A 2D Array can be treated to be array of 1D arrays

- ⇒ given a 2D array *arr*, arr[i] represents a 1D array
- ⇒ arr[i] gives the address of 0th element of i^{th} 1D array.
- ⇒ arr[i] + 1 gives the address of 1st element of i^{th} array.
- ⇒ arr[i] + j gives the address of j^{th} element of i^{th} array.
- ⇒ *(arr[i] + j) gives the j^{th} element of i^{th} array.
- ⇒ *(arr[i] + j) ⇔ arr[i][j]
- ⇒ But arr[i] ⇔ *(arr+i)

⇒ *(*(arr + i) + j) ⇔ arr[i][j]



Pointers and 3D Arrays

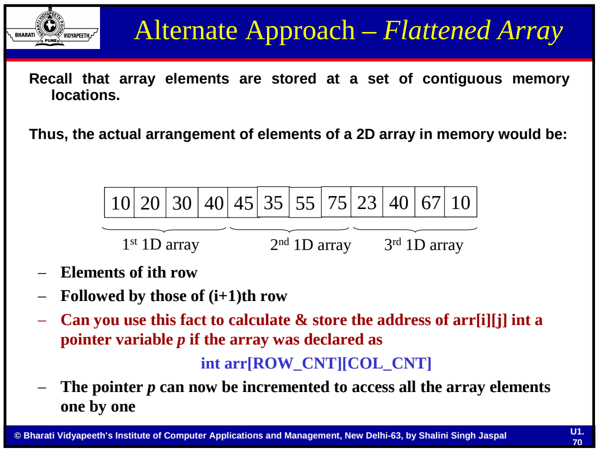
Similar to 2D arrays, a 3D array can be considered to be

- Array of 2D arrays
- Or, array of arrays of 1D arrays

	30	40	10
10	20	15	60
15	0	50	20
30	70	56	60
20	40	12	

arr[i][j][k]

⇔*(*(*(arr + i) + j) + k)



Alternate Approach – *Flattened Array*

Recall that array elements are stored at a set of contiguous memory locations.

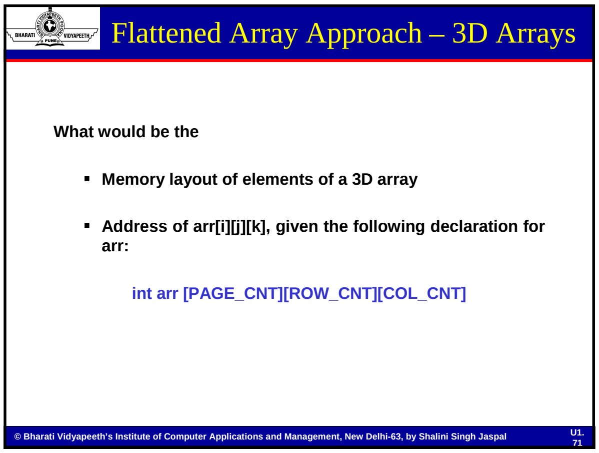
Thus, the actual arrangement of elements of a 2D array in memory would be:

10	20	30	40	45	35	55	75	23	40	67	10

1^{st} 1D array 2^{nd} 1D array 3^{rd} 1D array

- **Elements of ith row**
- **Followed by those of (i+1)th row**
- **Can you use this fact to calculate & store the address of arr[i][j] int a pointer variable *p* if the array was declared as**

  int arr[ROW_CNT][COL_CNT]
- **The pointer *p* can now be incremented to access all the array elements one by one**

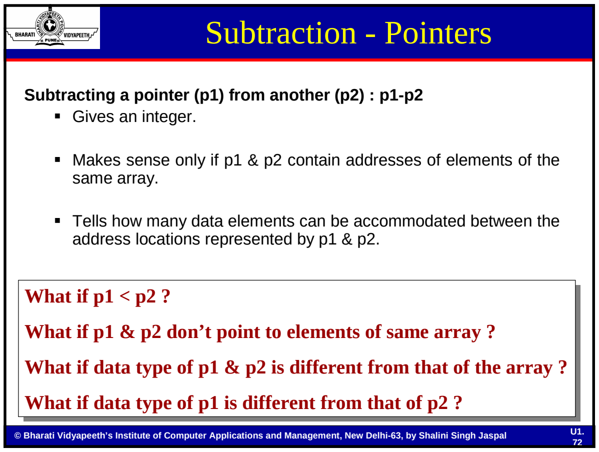


Flattened Array Approach – 3D Arrays

What would be the

- **Memory layout of elements of a 3D array**
- **Address of arr[i][j][k], given the following declaration for arr:**

  int arr [PAGE_CNT][ROW_CNT][COL_CNT]



Subtraction - Pointers

Subtracting a pointer (p1) from another (p2) : p1-p2

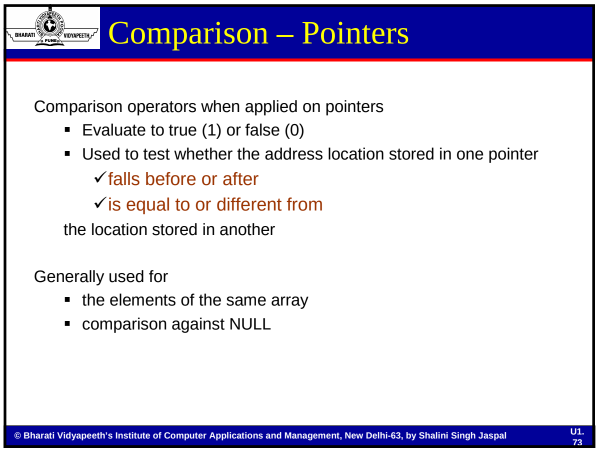
- Gives an integer.
- Makes sense only if p1 & p2 contain addresses of elements of the same array.
- Tells how many data elements can be accommodated between the address locations represented by p1 & p2.

What if p1 < p2 ?

What if p1 & p2 don't point to elements of same array ?

What if data type of p1 & p2 is different from that of the array ?

What if data type of p1 is different from that of p2 ?



Comparison – Pointers

Comparison operators when applied on pointers

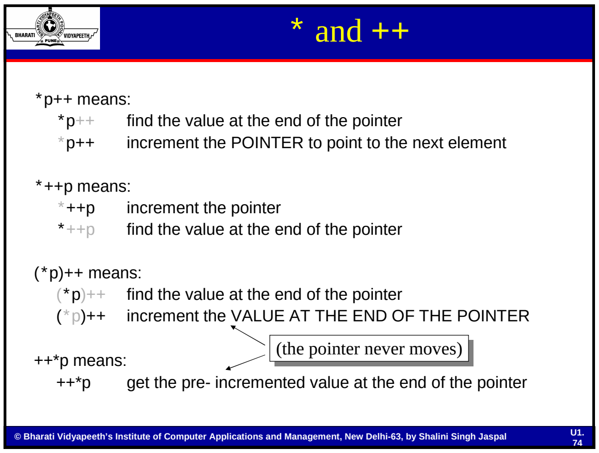
- Evaluate to true (1) or false (0)
- Used to test whether the address location stored in one pointer
  - ✓falls before or after
  - ✓is equal to or different from

the location stored in another

Generally used for

- the elements of the same array
- comparison against NULL

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.73



* and ++

*p++ means:

```
*p++     find the value at the end of the pointer
*p++     increment the POINTER to point to the next element
```

*++p means:

```
*++p     increment the pointer
*++p     find the value at the end of the pointer
```

(*p)++ means:

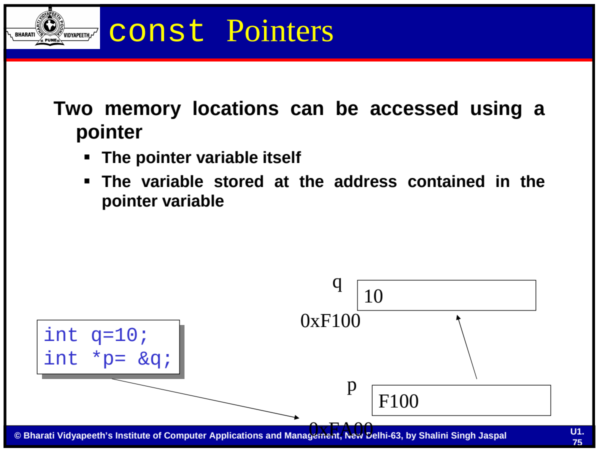
```
(*p)++   find the value at the end of the pointer
(*p)++   increment the VALUE AT THE END OF THE POINTER
```

(the pointer never moves)

++*p means:

```
++*p     get the pre- incremented value at the end of the pointer
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.74



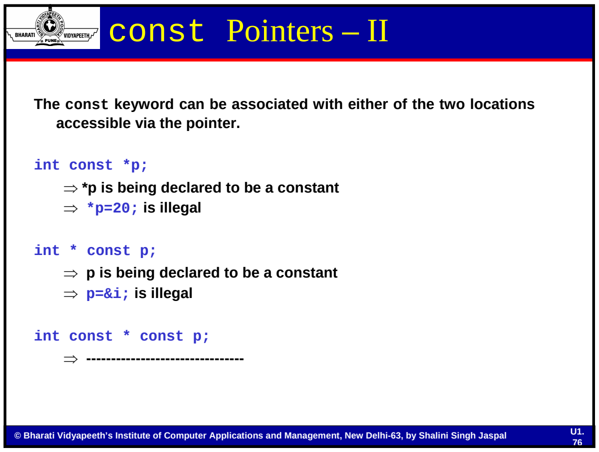
const Pointers

Two memory locations can be accessed using a pointer

- **The pointer variable itself**
- **The variable stored at the address contained in the pointer variable**

```
int q=10;
int *p= &q;
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh Jaspal U1.75



`const` Pointers – II

The `const` keyword can be associated with either of the two locations accessible via the pointer.

```
int const *p;
```

⇒ *p is being declared to be a constant

⇒ `*p=20;` is illegal

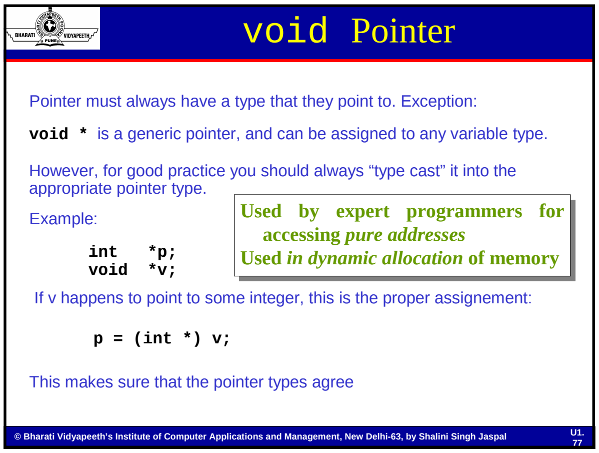
```
int * const p;
```

⇒ p is being declared to be a constant

⇒ `p=&i;` is illegal

```
int const * const p;
```

⇒ --------------------------------



`void` Pointer

Pointer must always have a type that they point to. Exception:

`void *` is a generic pointer, and can be assigned to any variable type.

However, for good practice you should always “type cast” it into the appropriate pointer type.

Example:

```
    int   *p;
    void  *v;
```

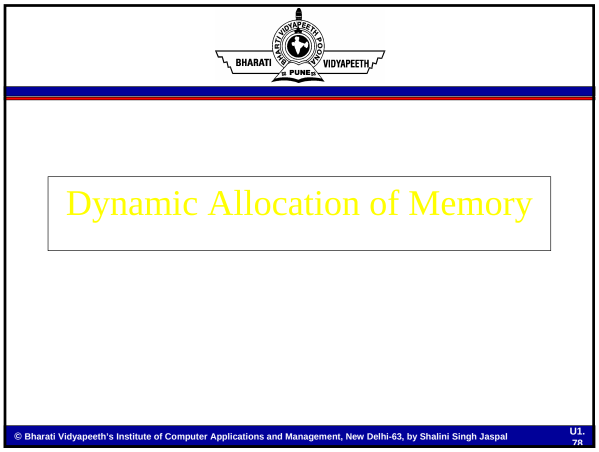
Used by expert programmers for accessing *pure addresses*

Used *in dynamic allocation* of memory

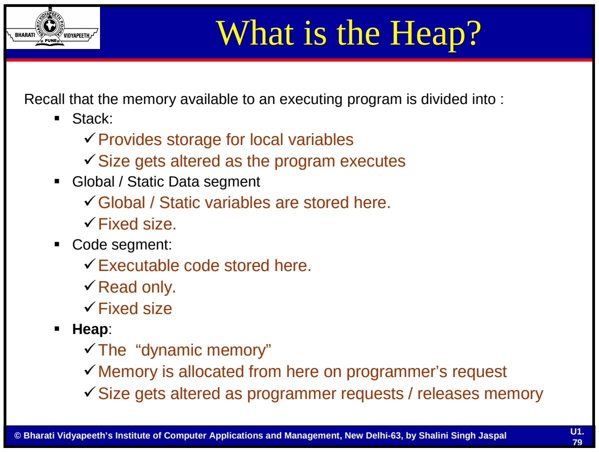
If v happens to point to some integer, this is the proper assignement:

```
    p = (int *) v;
```

This makes sure that the pointer types agree



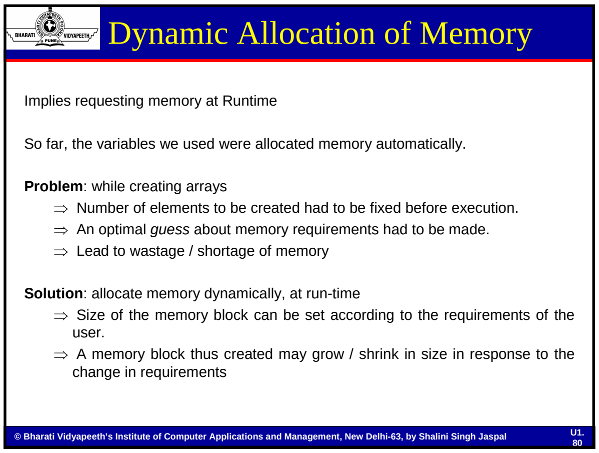
Dynamic Allocation of Memory



What is the Heap?

Recall that the memory available to an executing program is divided into :

- Stack:
  - ✓Provides storage for local variables
  - ✓Size gets altered as the program executes
- Global / Static Data segment
  - ✓Global / Static variables are stored here.
  - ✓Fixed size.
- Code segment:
  - ✓Executable code stored here.
  - ✓Read only.
  - ✓Fixed size
- **Heap**:
  - ✓The "dynamic memory"
  - ✓Memory is allocated from here on programmer's request
  - ✓Size gets altered as programmer requests / releases memory



Dynamic Allocation of Memory

Implies requesting memory at Runtime

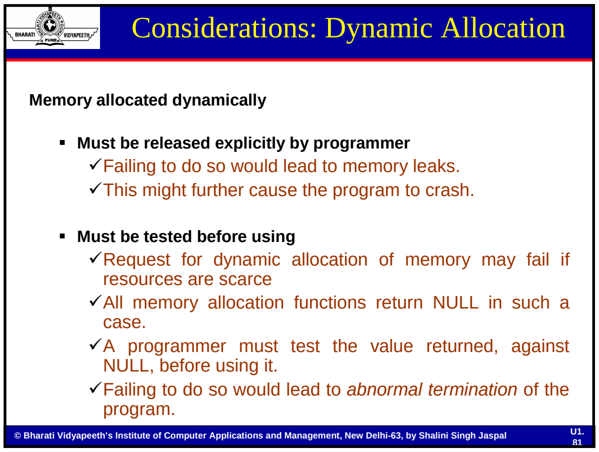
So far, the variables we used were allocated memory automatically.

Problem: while creating arrays

- ⇒ Number of elements to be created had to be fixed before execution.
- ⇒ An optimal *guess* about memory requirements had to be made.
- ⇒ Lead to wastage / shortage of memory

Solution: allocate memory dynamically, at run-time

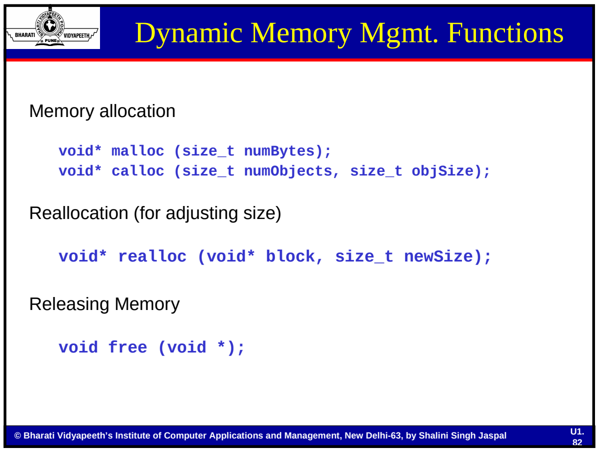
- ⇒ Size of the memory block can be set according to the requirements of the user.
- ⇒ A memory block thus created may grow / shrink in size in response to the change in requirements



Considerations: Dynamic Allocation

Memory allocated dynamically

- **Must be released explicitly by programmer**
  - ✓Failing to do so would lead to memory leaks.
  - ✓This might further cause the program to crash.
- **Must be tested before using**
  - ✓Request for dynamic allocation of memory may fail if resources are scarce
  - ✓All memory allocation functions return NULL in such a case.
  - ✓A programmer must test the value returned, against NULL, before using it.
  - ✓Failing to do so would lead to *abnormal termination* of the program.



Dynamic Memory Mgmt. Functions

Memory allocation

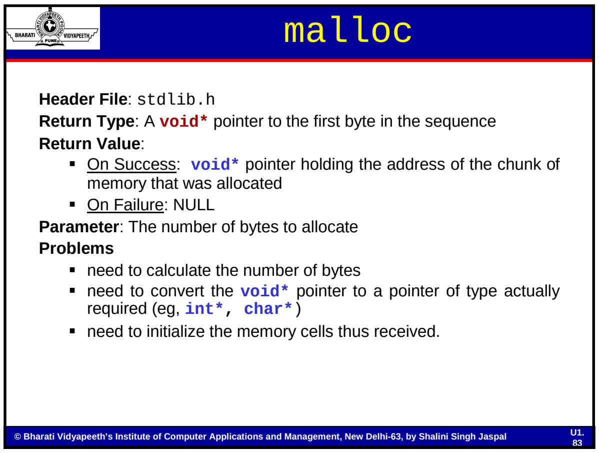
```
void* malloc (size_t numBytes);
void* calloc (size_t numObjects, size_t objSize);
```

Reallocation (for adjusting size)

```
void* realloc (void* block, size_t newSize);
```

Releasing Memory

```
void free (void *);
```



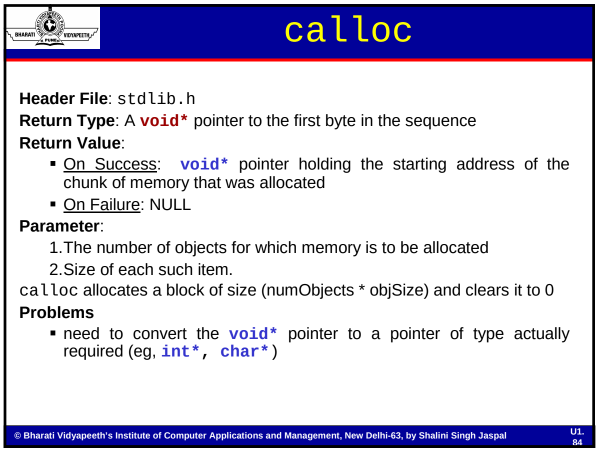
malloc

Header File: `stdlib.h`
Return Type: A `void*` pointer to the first byte in the sequence
Return Value:

- On Success: `void*` pointer holding the address of the chunk of memory that was allocated
- On Failure: NULL

Parameter: The number of bytes to allocate
Problems

- need to calculate the number of bytes
- need to convert the `void*` pointer to a pointer of type actually required (eg, `int*, char*`)
- need to initialize the memory cells thus received.



calloc

Header File: `stdlib.h`
Return Type: A `void*` pointer to the first byte in the sequence
Return Value:

- On Success: `void*` pointer holding the starting address of the chunk of memory that was allocated
- On Failure: NULL

Parameter:

1. The number of objects for which memory is to be allocated
2. Size of each such item.

`calloc` allocates a block of size (numObjects * objSize) and clears it to 0
Problems

- need to convert the `void*` pointer to a pointer of type actually required (eg, `int*, char*`)

realloc

Header File: `stdlib.h`

Return Type: A `void*` pointer to the first byte in the sequence

Return Value:

- On Success: `void*` pointer holding the starting address of the reallocated block
- On Failure: NULL

Parameter:

1. Staring address of the old block.
2. New size to be allocated.

Action

- Adjusts the size of the block to new size
- Block is copied to a new location if necessary.

Problems

- need to convert the `void*` pointer to the required type.

Releasing Memory

Programs which allocate storage space dynamically often do not need it for the entire run

It is important to release memory once it is no longer required

The standard library function *free* provides this facility of releasing the memory pointed to by p

p should point to an area of storage previously allocated by `malloc`, `calloc` or `realloc`

free

Declaration

```
void free( void *p);
```

Header File: stdlib.h

Parameter:

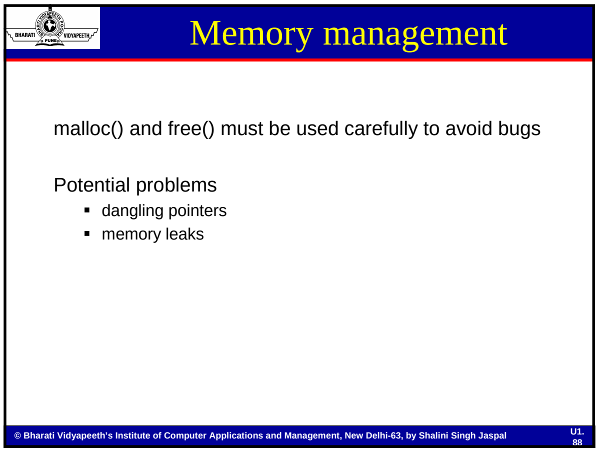
- pointer of type void*
- Any pointer type used will be automatically cast

Action:

- Releases a memory block that was allocate dynamically

Syntax: `free(ptr);`

- Technically, this is: `free((void*) ptr);`

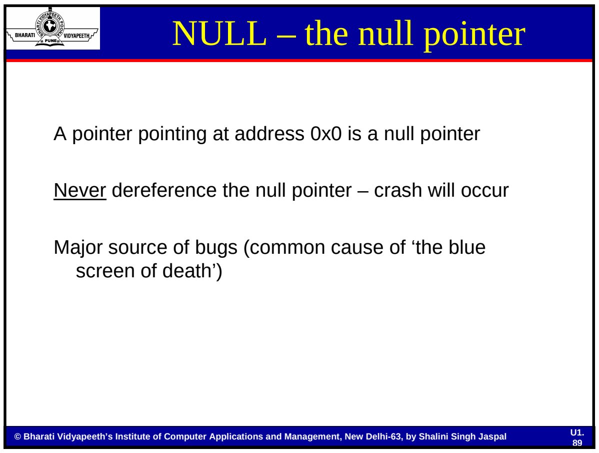


Memory management

malloc() and free() must be used carefully to avoid bugs

Potential problems

- dangling pointers
- memory leaks

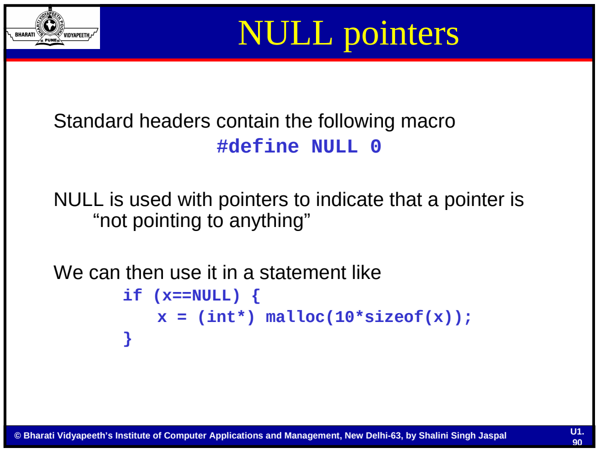


NULL – the null pointer

A pointer pointing at address 0x0 is a null pointer

Never dereference the null pointer – crash will occur

Major source of bugs (common cause of 'the blue screen of death')



NULL pointers

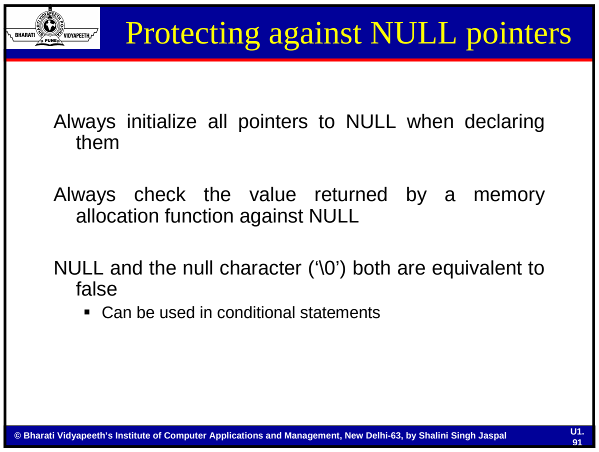
Standard headers contain the following macro

```
#define NULL 0
```

NULL is used with pointers to indicate that a pointer is "not pointing to anything"

We can then use it in a statement like

```
if (x==NULL) {
    x = (int*) malloc(10*sizeof(x));
}
```



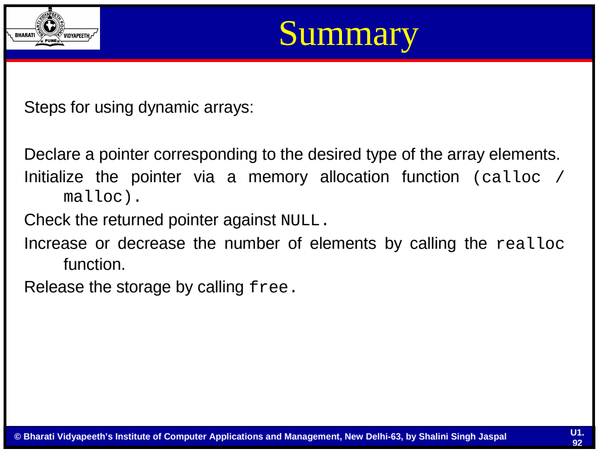
Protecting against NULL pointers

Always initialize all pointers to NULL when declaring them

Always check the value returned by a memory allocation function against NULL

NULL and the null character ('\0') both are equivalent to false

- Can be used in conditional statements



Summary

Steps for using dynamic arrays:

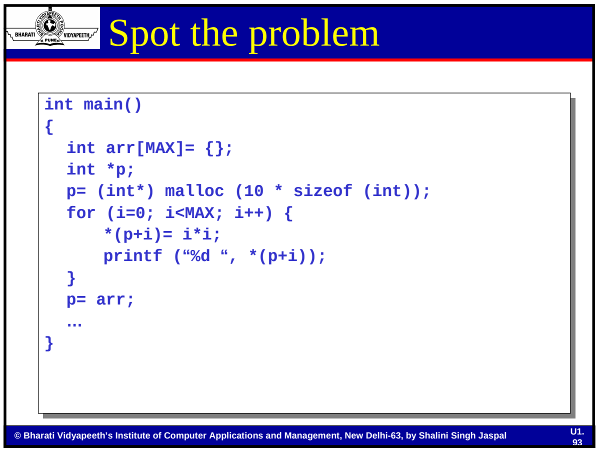
Declare a pointer corresponding to the desired type of the array elements.

Initialize the pointer via a memory allocation function (`calloc / malloc`).

Check the returned pointer against `NULL`.

Increase or decrease the number of elements by calling the `realloc` function.

Release the storage by calling `free`.



Spot the problem

```
int main()
{
  int arr[MAX]= {};
  int *p;
  p= (int*) malloc (10 * sizeof (int));
  for (i=0; i<MAX; i++) {
      *(p+i)= i*i;
      printf (“%d “, *(p+i));
  }
  p= arr;
  …
}
```

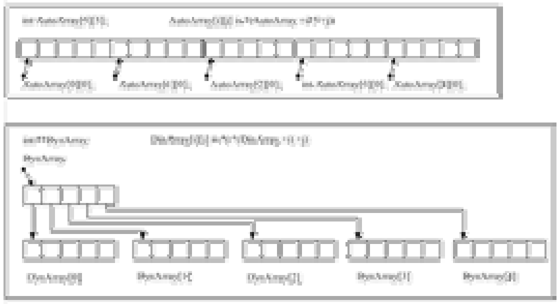
Creating 1D Arrays Dynamically

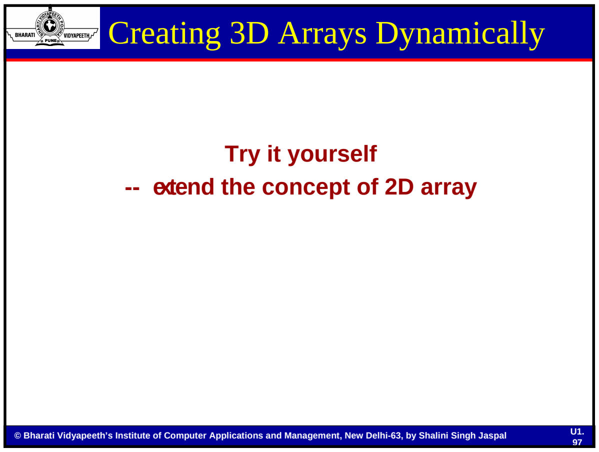
```
#include <stdlib.h>
#include <stdio.h>
int main()
{
   int *dynArray, size;
   printf (“Enter the number of elements required: ”);
   scanf (“%d”, &size);
   dynArray= (int*) malloc (size * sizeof (int));
   if (dynArray==NULL){
        fprintf (strerr, “Error in allocating memory”);
        exit (1);
   }
   //process the array
   free (dynArr);
   return 0; }
```

Creating 2D Arrays Dynamically

```
int **dynArray, rSize, cSize;

//get rSize & cSize

dynArray= (int**) malloc (rSize * sizeof (int*));

for (i=0; i<rSize; i++)
   dynArray [i]= (int*) malloc (cSize * sizeof (int));

//process the array

for (i=0; i<rSize; i++)
   free (dynArray [i]);

free (dynArray);
```

Static Vs. Dynamic 2D Array

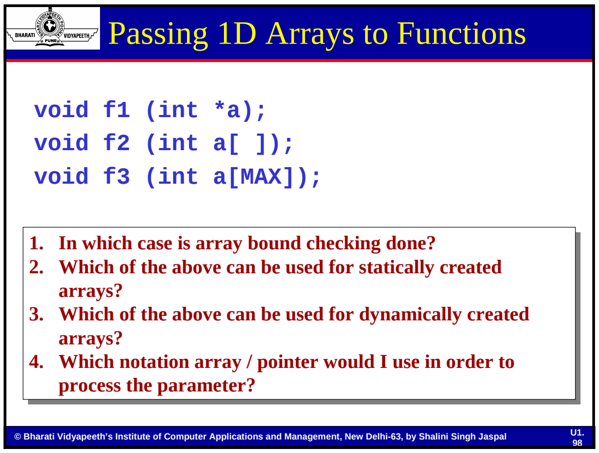




Creating 3D Arrays Dynamically

Try it yourself

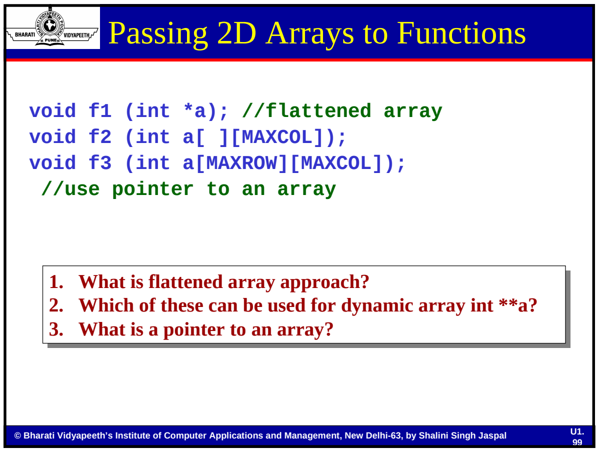
-- extend the concept of 2D array



Passing 1D Arrays to Functions

```
void f1 (int *a);
void f2 (int a[ ]);
void f3 (int a[MAX]);
```

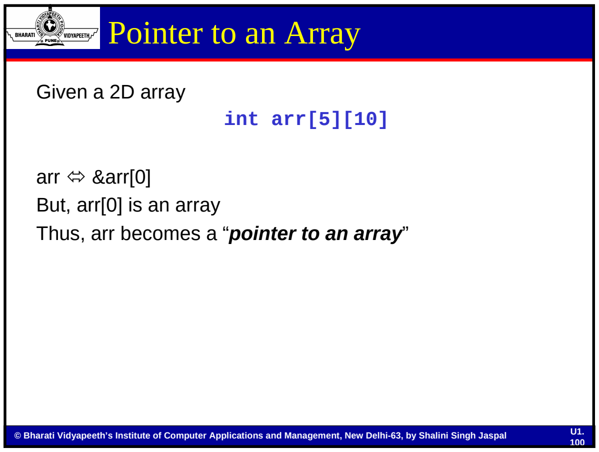
1. In which case is array bound checking done?
2. Which of the above can be used for statically created arrays?
3. Which of the above can be used for dynamically created arrays?
4. Which notation array / pointer would I use in order to process the parameter?



Passing 2D Arrays to Functions

```
void f1 (int *a); //flattened array
void f2 (int a[ ][MAXCOL]);
void f3 (int a[MAXROW][MAXCOL]);
 //use pointer to an array
```

1. What is flattened array approach?
2. Which of these can be used for dynamic array int **a?
3. What is a pointer to an array?



Pointer to an Array

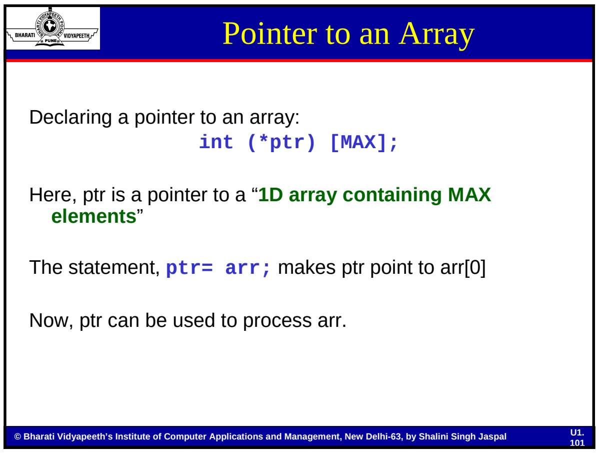
Given a 2D array

```
int arr[5][10]
```

arr ⇔ &arr[0]

But, arr[0] is an array

Thus, arr becomes a “***pointer to an array***”



Pointer to an Array

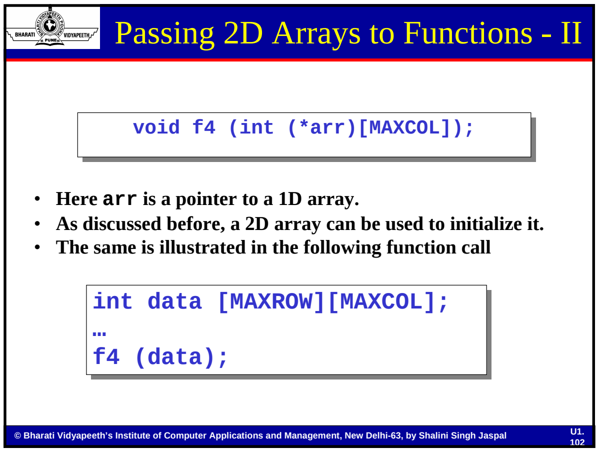
Declaring a pointer to an array:

```
int (*ptr) [MAX];
```

Here, ptr is a pointer to a “**1D array containing MAX elements**”

The statement, `ptr= arr;` makes ptr point to arr[0]

Now, ptr can be used to process arr.

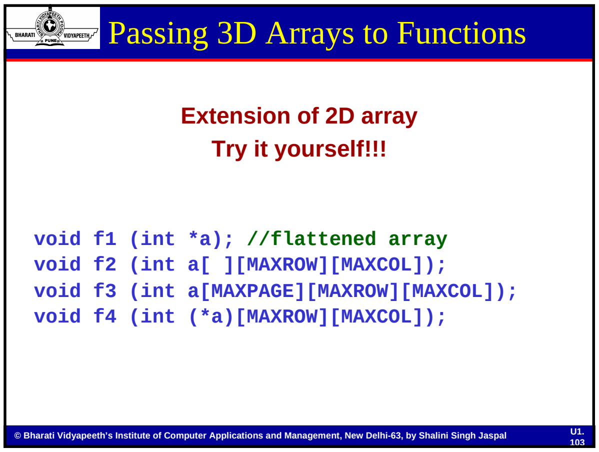


Passing 2D Arrays to Functions - II

```
void f4 (int (*arr)[MAXCOL]);
```

- **Here `arr` is a pointer to a 1D array.**
- **As discussed before, a 2D array can be used to initialize it.**
- **The same is illustrated in the following function call**

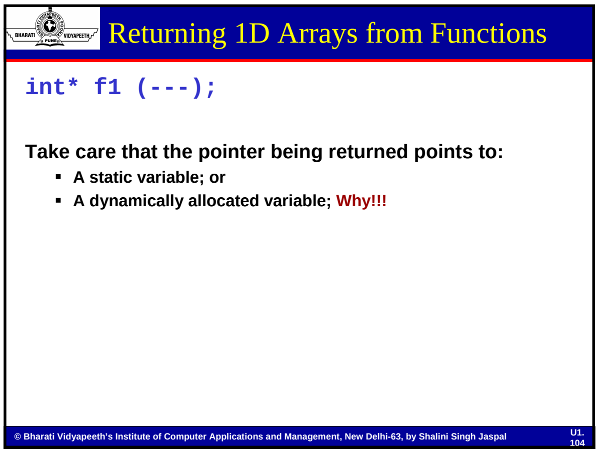
```
int data [MAXROW][MAXCOL];
…
f4 (data);
```



Passing 3D Arrays to Functions

Extension of 2D array
Try it yourself!!!

```
void f1 (int *a); //flattened array
void f2 (int a[ ][MAXROW][MAXCOL]);
void f3 (int a[MAXPAGE][MAXROW][MAXCOL]);
void f4 (int (*a)[MAXROW][MAXCOL]);
```

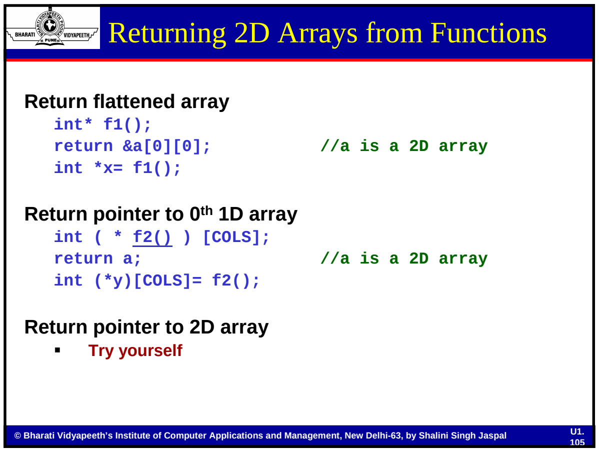


Returning 1D Arrays from Functions

```
int* f1 (---);
```

Take care that the pointer being returned points to:

- **A static variable; or**
- **A dynamically allocated variable; Why!!!**



Returning 2D Arrays from Functions

Return flattened array

```
int* f1();
return &a[0][0];          //a is a 2D array
int *x= f1();
```

Return pointer to 0th 1D array

```
int ( * f2() ) [COLS];
return a;                 //a is a 2D array
int (*y)[COLS]= f2();
```

Return pointer to 2D array

- Try yourself

Returning 3D Arrays from Functions

Extend the 2D array concept.

You can return

- **A pointer to an integer => flattened array**
- **A pointer to 0th 1D array**
- **A pointer to 0th 2D array**
- **A pointer to the 3D array**

`const` Pointers in Function Calls

A `const` pointer when used as a

- Parameter: restricts a function from changing its parameter
- Return value: Restricts the calling program from
  - ✓Changing the return value
  - ✓Passing the returned pointer to a function that accepts non-const pointers.

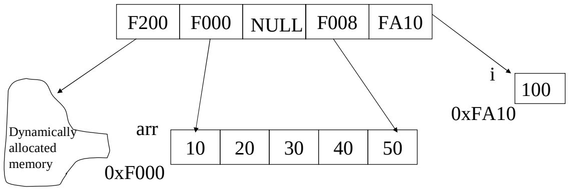
Array of Pointers

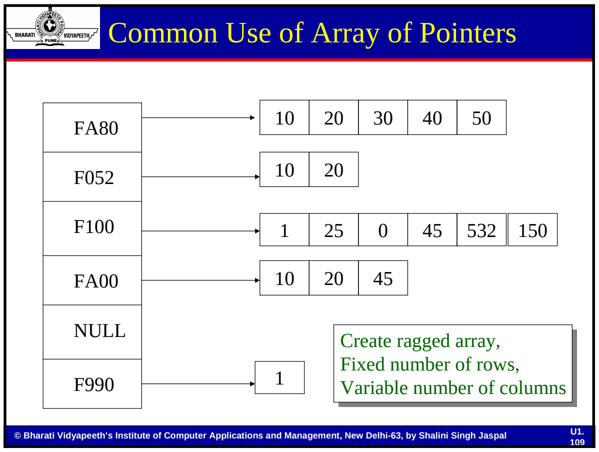
```
int * arr [MAX];
```

Here, `arr` is an array of `MAX` variables.

Each element of the array is an integer pointer.

Thus, each element can be made to point to any integer / set of integers.

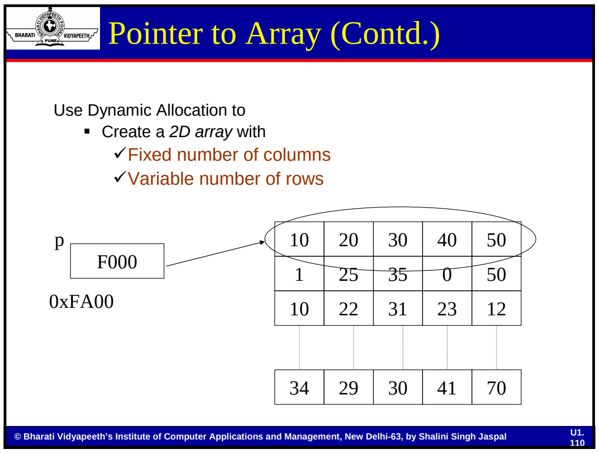




Common Use of Array of Pointers

FA80	10 20 30 40 50
F052	10 20
F100	1 25 0 45 532 150
FA00	10 20 45
NULL	
F990	1

Create ragged array,
Fixed number of rows,
Variable number of columns



Pointer to Array (Contd.)

Use Dynamic Allocation to

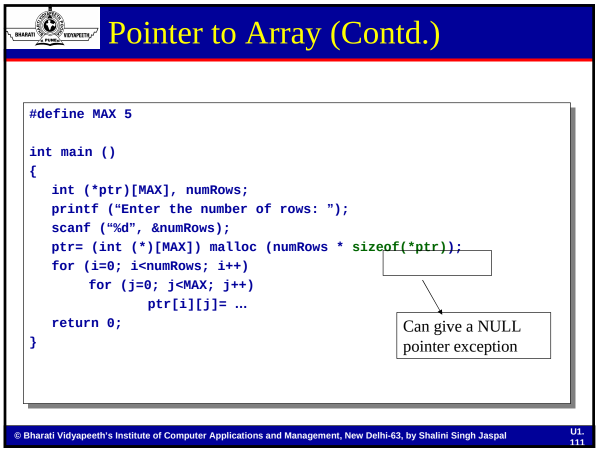
- Create a *2D array* with
  - ✓Fixed number of columns
  - ✓Variable number of rows

p

F000

0xFA00

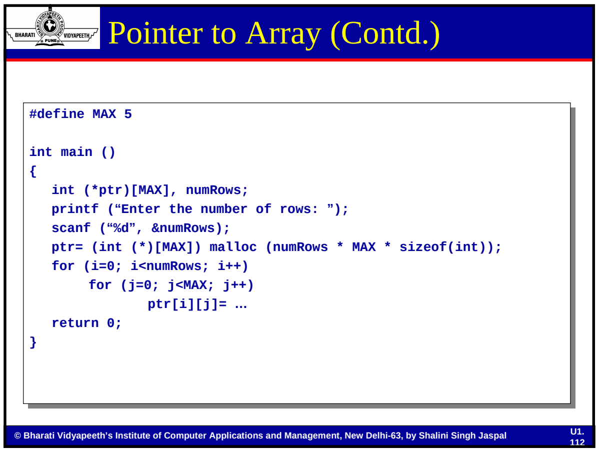
10	20	30	40	50
1	25	35	0	50
10	22	31	23	12
34	29	30	41	70



Pointer to Array (Contd.)

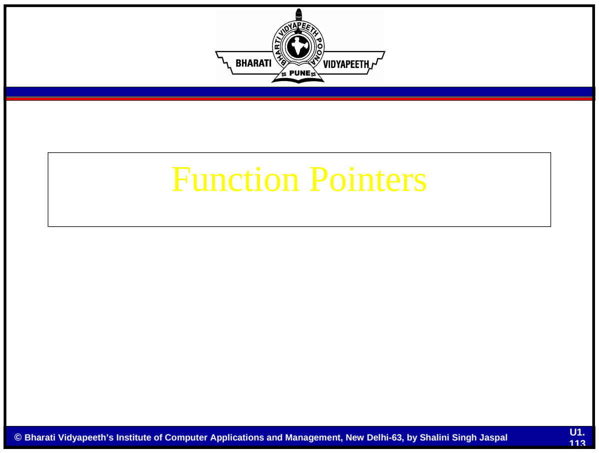
```
#define MAX 5

int main ()
{
   int (*ptr)[MAX], numRows;
   printf (“Enter the number of rows: ");
   scanf (“%d”, &numRows);
   ptr= (int (*)[MAX]) malloc (numRows * sizeof(*ptr));
   for (i=0; i<numRows; i++)
        for (j=0; j<MAX; j++)
                ptr[i][j]= …
   return 0;
}
```

Can give a NULL
pointer exception

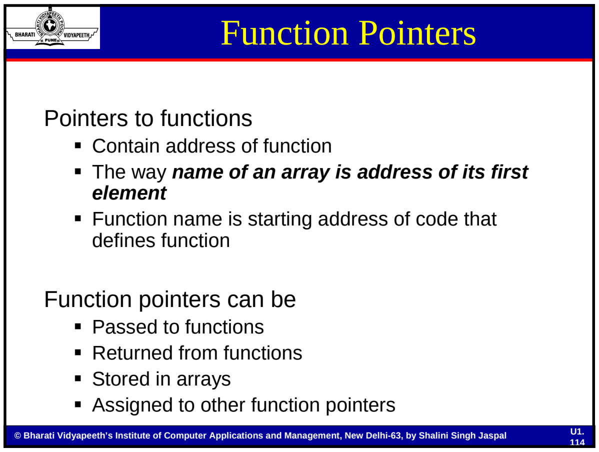


Pointer to Array (Contd.)

```
#define MAX 5

int main ()
{
   int (*ptr)[MAX], numRows;
   printf (“Enter the number of rows: ”);
   scanf (“%d”, &numRows);
   ptr= (int (*)[MAX]) malloc (numRows * MAX * sizeof(int));
   for (i=0; i<numRows; i++)
        for (j=0; j<MAX; j++)
                ptr[i][j]= …
   return 0;
}
```



Function Pointers



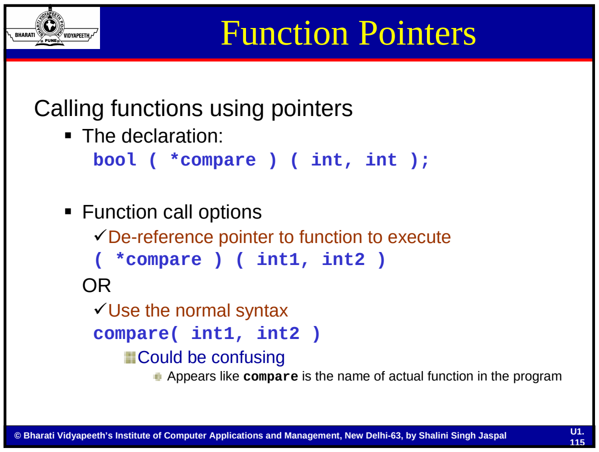
Function Pointers

Pointers to functions

- Contain address of function
- The way ***name of an array is address of its first element***
- Function name is starting address of code that defines function

Function pointers can be

- Passed to functions
- Returned from functions
- Stored in arrays
- Assigned to other function pointers



Function Pointers

Calling functions using pointers

- The declaration:

  ```
  bool ( *compare ) ( int, int );
  ```

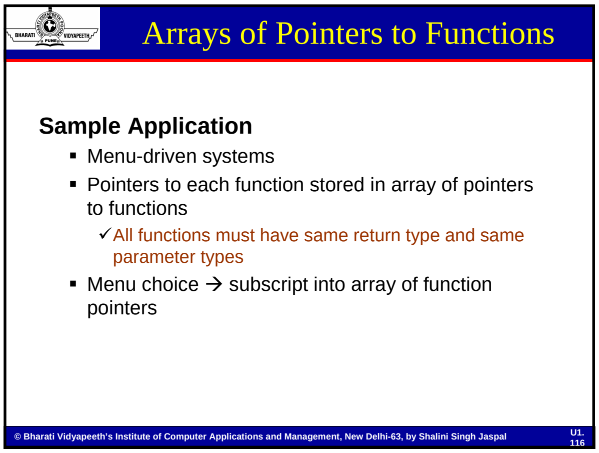
- Function call options
  - ✓De-reference pointer to function to execute

    ```
    ( *compare ) ( int1, int2 )
    ```

  OR
  - ✓Use the normal syntax

    ```
    compare( int1, int2 )
    ```

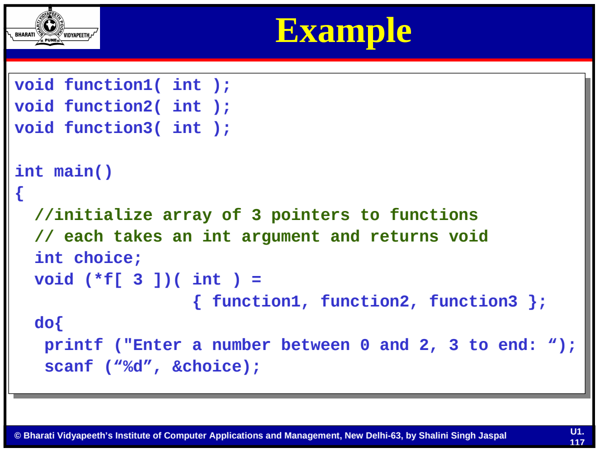
    - Could be confusing
      - Appears like **`compare`** is the name of actual function in the program



Arrays of Pointers to Functions

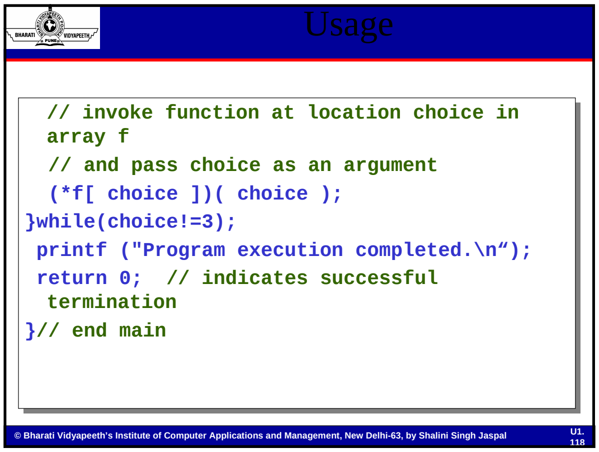
Sample Application

- Menu-driven systems
- Pointers to each function stored in array of pointers to functions
  - ✓All functions must have same return type and same parameter types
- Menu choice → subscript into array of function pointers



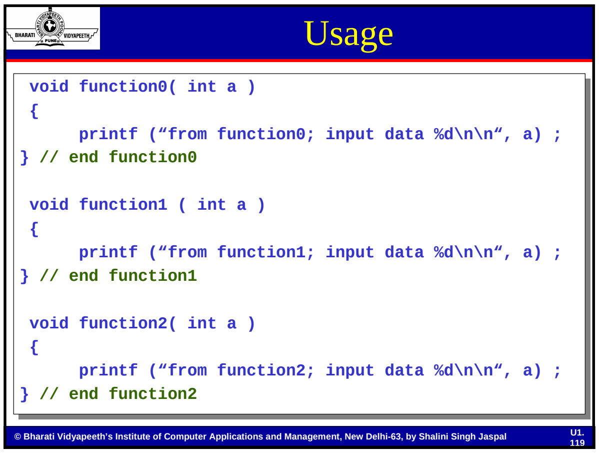
Example

```
void function1( int );
void function2( int );
void function3( int );

int main()
{
  //initialize array of 3 pointers to functions
  // each takes an int argument and returns void
  int choice;
  void (*f[ 3 ])( int ) =
                { function1, function2, function3 };
  do{
   printf ("Enter a number between 0 and 2, 3 to end: “);
   scanf (“%d”, &choice);
```



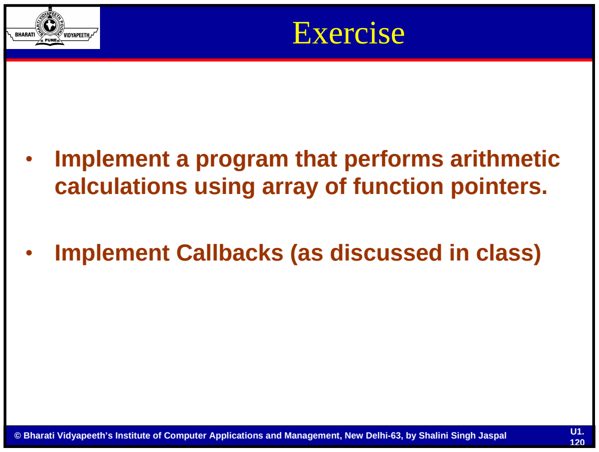
Usage

```
  // invoke function at location choice in
  array f
  // and pass choice as an argument
  (*f[ choice ])( choice );
}while(choice!=3);
 printf ("Program execution completed.\n“);
 return 0;  // indicates successful
  termination
}// end main
```



Usage

```
 void function0( int a )
 {
      printf (“from function0; input data %d\n\n“, a) ;
} // end function0

 void function1 ( int a )
 {
      printf (“from function1; input data %d\n\n“, a) ;
} // end function1

 void function2( int a )
 {
      printf (“from function2; input data %d\n\n“, a) ;
} // end function2
```



Exercise

- Implement a program that performs arithmetic calculations using array of function pointers.
- Implement Callbacks (as discussed in class)