Devi Ahilya Vishwavidyalaya

# Programming With C

hmehta.scs@dauniv.ac.in



# Arrays

## A sequential collection

School of Computer Science

hmehta.scs@dauniv.ac.in

# Arrays

- An array is a collection of variables of the same type that are referred to through a common name.
- A specific element in an array is accessed by an index. In C, all arrays consist of contiguous memory locations.
- The lowest address corresponds to the first element and the highest address to the last element.
- Arrays can have from one to several dimensions.

# Single-Dimensional Arrays

- Generic declaration:

  typename variablename[size]

  - typename is any type
  - variablename is any legal variable name
  - size is a constant number

- To define an array of ints with subscripts ranging from 0 to 9, use:

  int a[10];

# Single-Dimensional Arrays

- Array declared using `int a[10];` requires `10*sizeof(int)` bytes of memory
- To access individual array elements, use indexing: `a[0]=10; x=a[2]; a[3]=a[2]; a[i]=a[i+1]+10;` etc.
- To read a value into an array location, use `scanf("%d",&a[3]);`
- Accessing an individual array element is a fast operation

# How Array Indexing Works

- Array elements are stored ***contiguously*** (that is, in adjacent memory locations)
- Address of the $k^{th}$ array element is the start address of the array (that is, the address of the element at location 0) plus k * sizeof(each individual array element)
- *Example*: Suppose we have int a[10]; where a begins at address 6000 in memory. Then a[5] begins at address 6000 + 5*sizeof(int)

# Using Constants to Define Arrays

- It is useful to define arrays using constants:

```
#define  MONTHS 12
 int  array [MONTHS];
```

- However, in ANSI C, you cannot use:

```
int n;
scanf("%d", &n);
int array[n];
```

- GNU C allows variable length arrays – non-standard

# Array-Bounds Checking

- C, unlike many languages, ***DOES NOT*** check array bounds subscripts during:
  - Compilation
  - Runtime
- Programmer must take responsibility for ensuring that array indices are within the declared bounds

# Array-Bounds Checking

- If you access beyond the end of an array:
  - C calculates the address as usual
  - Attempts to treat the location as part of the array
  - Program may continue to run, ***OR*** may crash with a ***memory access violation error*** (segmentation fault, core dump error)
  - It's better if the program crashes right away – easier to debug

# Initializing Arrays

- Initialization of arrays can be done by a comma separated list following its definition.
- *Example*: int array [4] = { 100, 200, 300, 400 };

  is equivalent to:

```
int array [4];
array[0] = 100;
array[1] = 200;
array[2] = 300;
array[3] = 400;
```

- Or, you can let the compiler compute the array size: int array[ ] = { 100, 200, 300, 400};

# Example

```
#include <stdio.h>
int main( ) {
  float expenses[12]={10.3, 9, 7.5, 4.3, 10.5, 7.5, 7.5, 8, 9.9,
    10.2, 11.5, 7.8};
  int count,month;
  float total;
  for (month=0, total=0.0; month < 12; month++)
  {
        total+=expenses[month];
  }
  for (count=0; count < 12; count++)
      printf ("Month %d = %.2f \n", count+1, expenses[count]);
  printf("Total = %.2f,   Average = %.2f\n", total, total/12);
  return 0;
}
```

# Multidimensional Arrays

- Arrays in C can have virtually as many dimensions as you want unlike coordinate geometry.
- Definition is accomplished by adding additional subscripts:

```
int  a [4] [3] ;
```

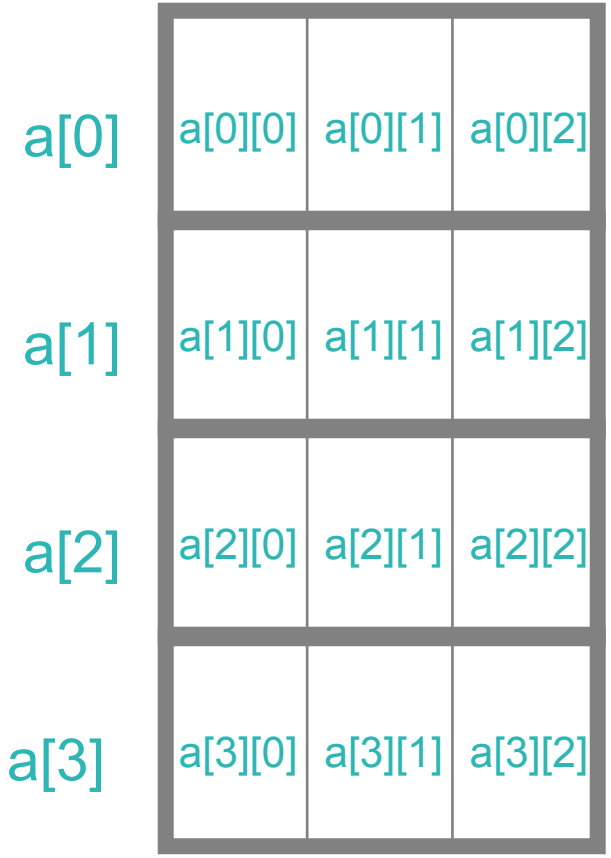
  - defines a two dimensional array with 4 rows and 3 columns
  - a can be thought of as a 1-dimensional array of 4 elements, where each element is of type int[3]

# Multidimensional Arrays

The array declared using

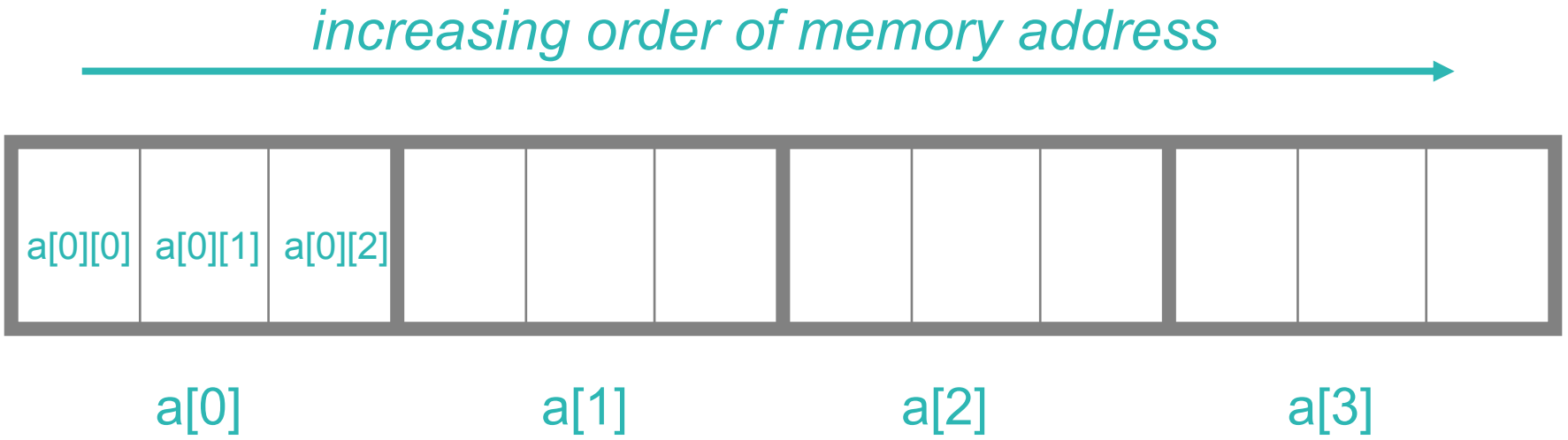
```
int  a [4] [3];
```

is normally thought of as a table.



# Multidimensional Arrays

In memory, which is one-dimensional, the rows of the array are actually stored contiguously.





hmehta.scs@dauniv.ac.in

# Initializing Multidimensional Arrays

- Two ways to initialize a[4][3]:

```
int a[4] [3] = { {1, 2, 3} , { 4, 5, 6} , {7, 8, 9} , {10, 11, 12} };
int a[4] [3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

- These methods are equivalent to:

```
a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 4;
...
a[3][2] = 12;
```

```
#include <stdio.h>
#include <stdlib.h>
int main ()  {
    int random1[8][8];
    int a, b;
    for (a = 0; a < 8; a++)
        for (b = 0; b < 8; b++)
                    random1[a][b] = rand( )%2;
    for (a = 0; a < 8; a++)
    {
        for (b = 0; b < 8; b++)
            printf ("%c " , random1[a][b] ? 'x' : 'o');
        printf("\n");
    }
  return 0;
}
```

## Example

The function

int rand( );

from <stdlib.h> returns a random int between 0 and RAND_MAX a constant defined in the same library.

# The Value of the Array Name

```
#include <stdio.h>
int main(){
    int a[3] = { 1, 2, 3 };
    printf( "%d\n", a[0]);
    scanf( "%d", &a[0] );
    printf( "%d\n", a[0]);
    scanf( "%d", a );
    printf( "%d \n", a[0]);
}
```

- When the array name is used alone, its value is the ***address*** of the array (a ***pointer*** to its first element)
- &a has no meaning if used in this program

# Arrays as Function Parameters

- The array address (i.e., the value of the array name), is passed to the function inc_array( )
- It is ***passed by value***

```
void inc_array(int a[ ], int size)
{
    int i;
    for(i=0;i<size;i++)
    {
        a[i]++;
    }
}
```

```
void inc_array(int a[ ],int size);
main()
{
    int test[3]={1,2,3};
    int ary[4]={1,2,3,4};
    int i;
    inc_array(test,3);
    for(i=0;i<3;i++)
        printf("%d\n",test[i]);
    inc_array(ary,4);
    for(i=0;i<4;i++)
        printf("%d\n",ary[i]);
    return 0;
}
```

# Example

```
void bubbleSort(int a[ ],int size)
{
  int i, j, x;
  for(i=0; i < size; i++)
     for(j=i; j > 0; j--)
        if(a[ j ] < a[ j-1])
        {  /* Switch a[ j ] and a[ j-1] */
           x=a[ j ];  a[ j ]=a[ j-1];  a[j-1]=x;
        }
}
```

Actual parameter corresponding to formal parameter a[ ] can be ***any*** array of int values; its declared size does not matter

Function bubbleSort( ) sorts the first size elements of array a into ascending order


hmehta.scs@dauniv.ac.in

# Pointers

*The likelihood of a program crashing is in direct proportion to the number of pointers used in it.*


hmehta.scs@dauniv.ac.in

# Pointer Variables

- ***Pointers*** are often referred to as ***references***
- The value in a pointer variable is interpreted as a memory address
- Usually, pointer variables hold references to specific kinds of data (e.g.: address of an int, address of a char, etc)

```
int * p;          /*  variable p can hold the address of a
                      memory location that contains an int      */
char * chptr;     /*  chptr can hold the address of a
                       memory location that contains a char     */
```

# Dereferencing Operator

- The expression *p denotes the memory cell to which p points
- Here, * is called the ***dereferencing operator***
- Be careful not to dereference a pointer that has not yet been initialized:

```
int *p;
```

p 

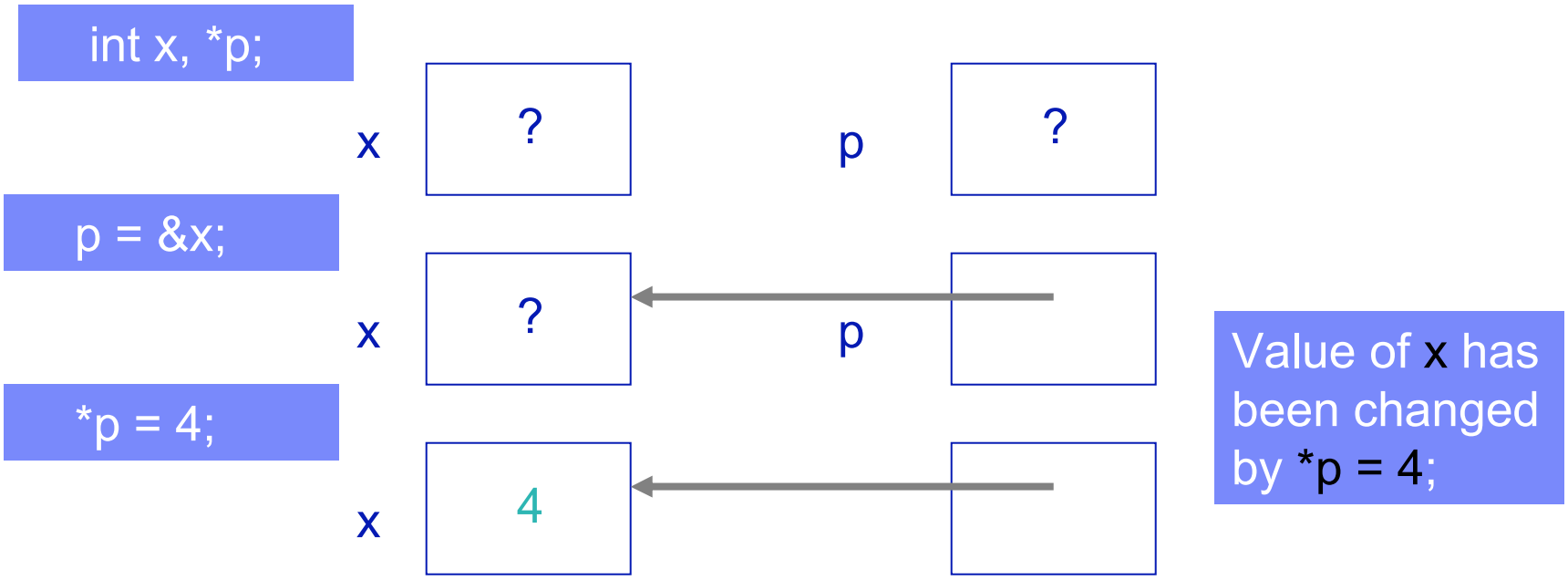

Address in p could be any memory location

```
*p = 7;
```

Attempt to put a value into an unknown memory location will result in a ***run-time error***, or worse, a ***logic error***

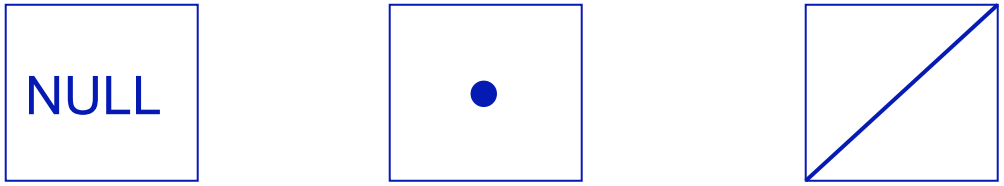
# The Address Operator

- The expression &x denotes the address of a variable x
- Here, & is called the ***address operator*** or the ***reference operator***



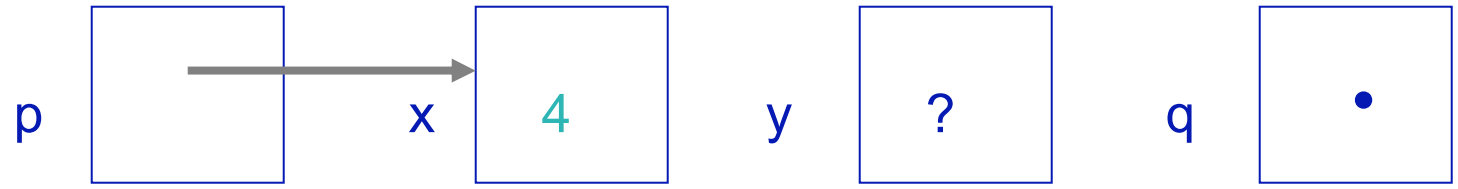
# The Null Pointer

- The ***null pointer*** is a special constant which is used to explicitly indicate that a pointer does not point anywhere
  - **NULL** is defined in the standard library <stdlib.h>
  - In diagrams, indicated as one of:



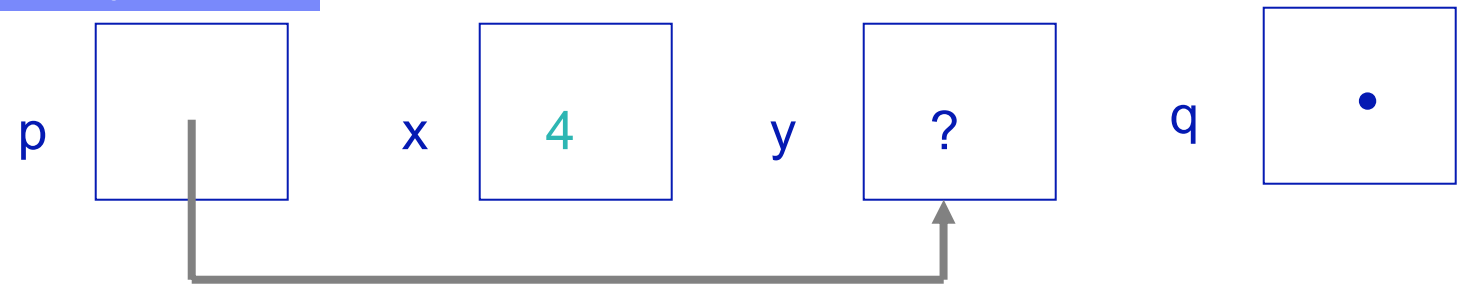
# Pointer Example

```
int *p, x, y, *q = NULL;
p = &x;
*p = 4;
```



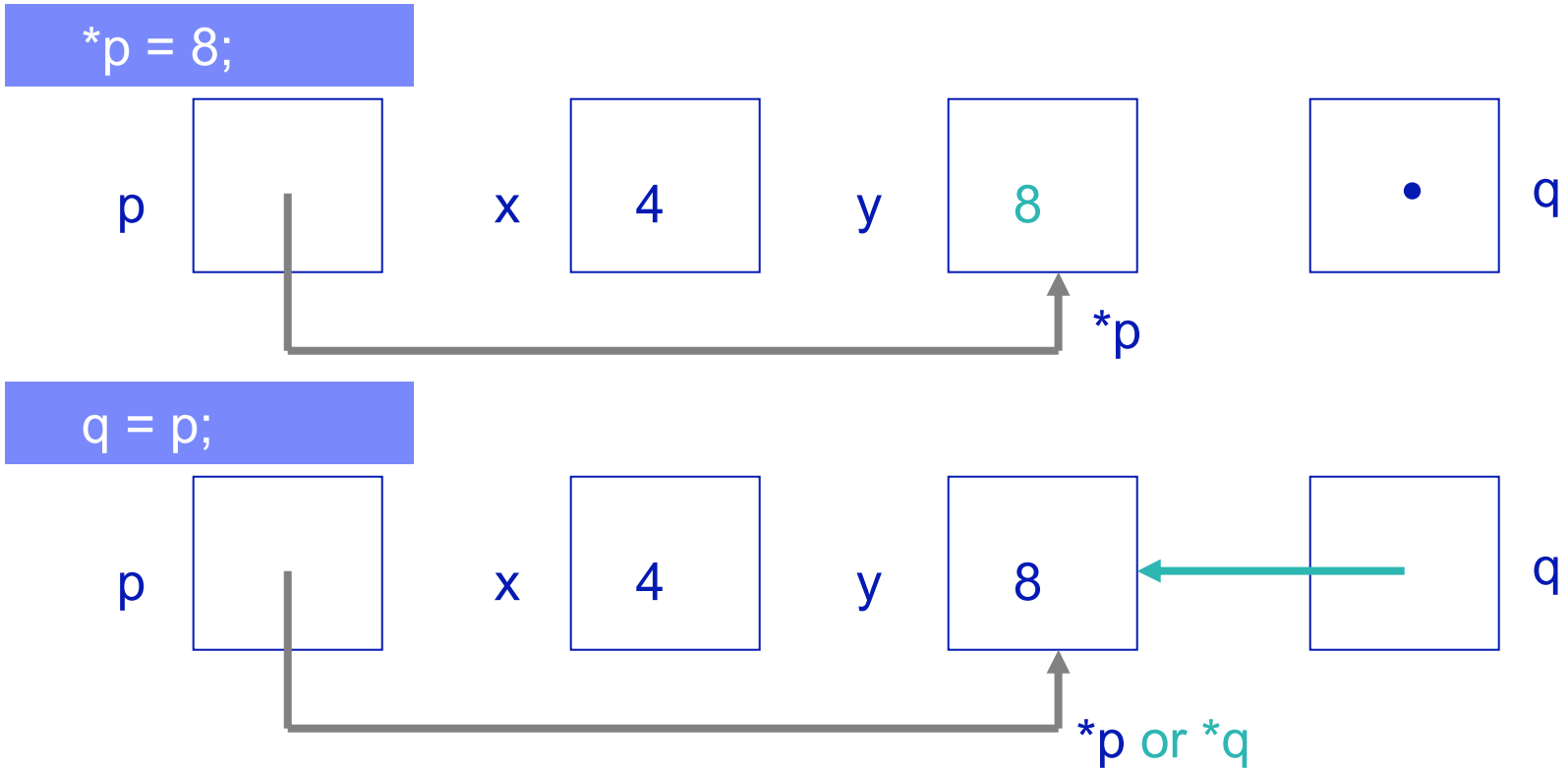

(or *p)

```
p = &y;
```

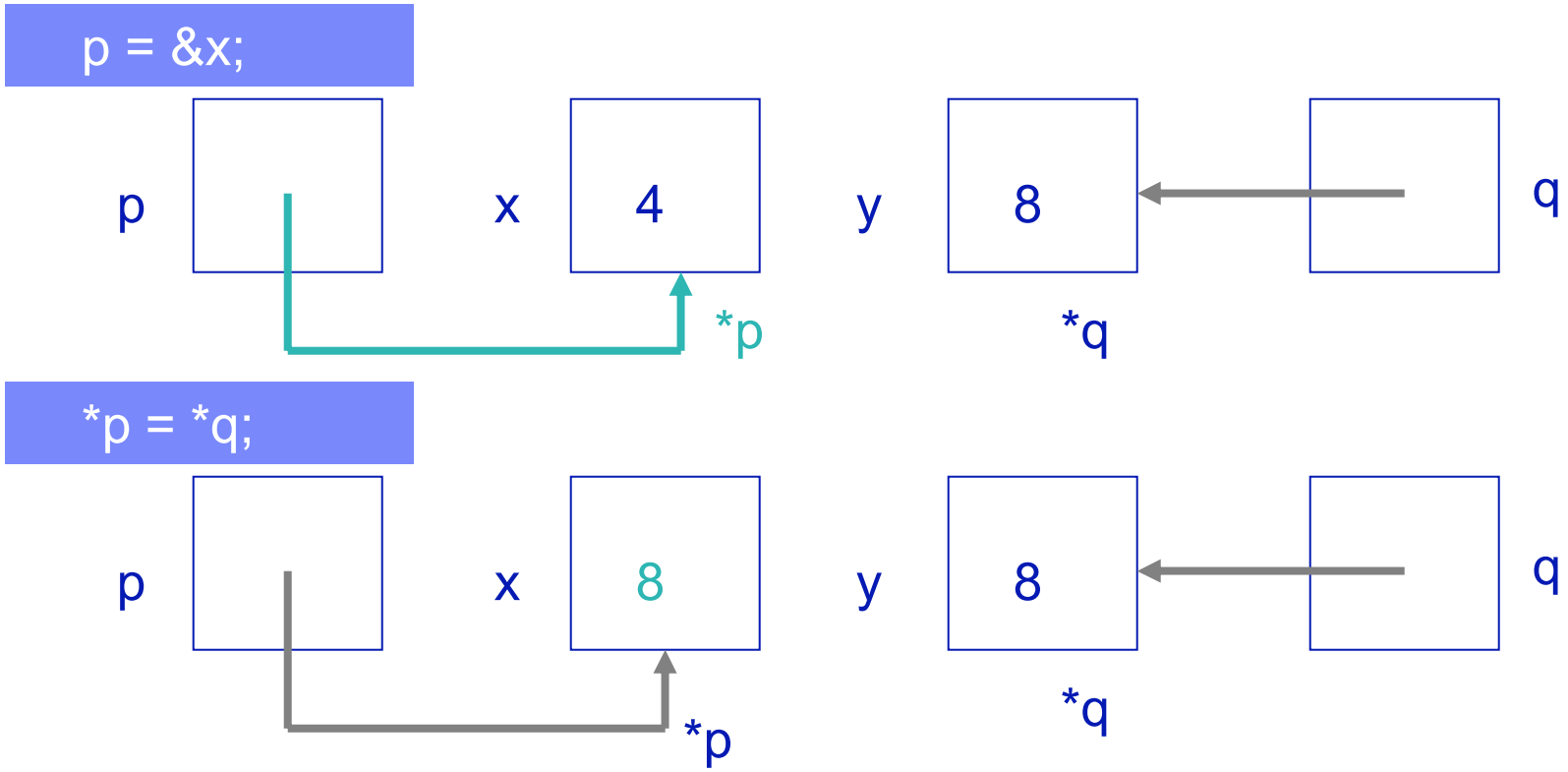



*p is now another name for y

# Pointer Example



# Pointer Example

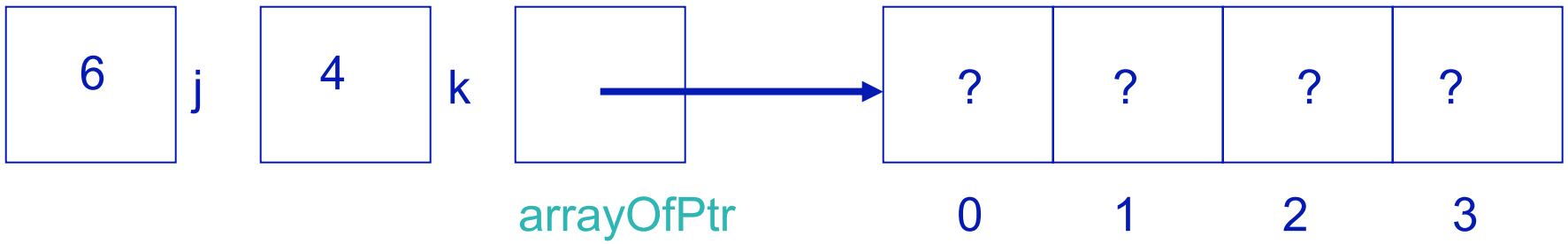


# Arrays of Pointers

- It's possible to have arrays of pointers
- The array name is a pointer to an array of pointers:

```
int * arrayOfPtr[ 4];
int j = 6; k = 4;
```

Pointers in array are not initialized yet

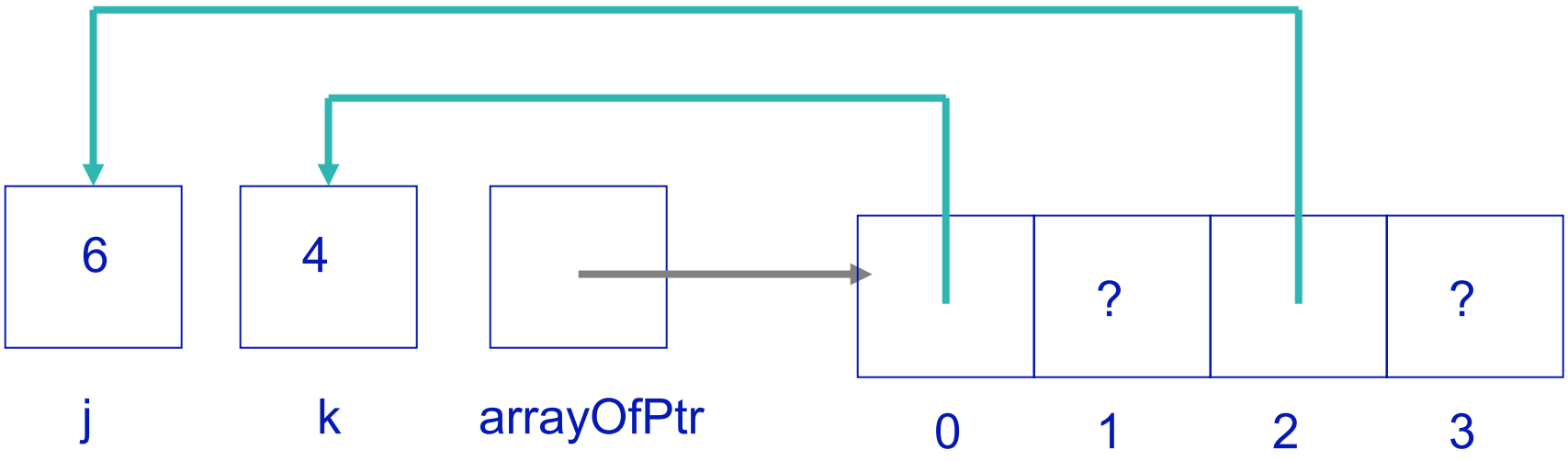




hmehta.scs@dauniv.ac.in

# Arrays of Pointers

```
arrayOfPtr[0] = &k;
arrayOfPtr[2]=&j;
```





hmehta.scs@dauniv.ac.in

# Array Names as Pointers

- Array name is really a pointer to the first element in the array
- Consider the declaration int arr[5];
  - arr has the same meaning as &arr[0]
  - *arr has the same meaning as arr[0]
  - ***Indexing*** into an array is really a ***pointer dereferencing operation***

# Array Names as Pointers

- Array name is really a pointer to the first element in the array
- Consider the declaration int arr[5];
  - a[0] is same as *a or *(a+0)
  - Similarly a[2] is same as *(a+2)
  - Now *(a+2) is same as *(2+a)
  - So 2[a] is same as a[2]

**Therefore a[2], *(a+2), *(2+a) and 2[a].**

# Generic Pointers

- Sometimes we need to use pointer variables that aren't associated with a specific data type
- In C, these ***generic pointers*** simply hold memory addresses, and are referred to as ***pointers to void***:

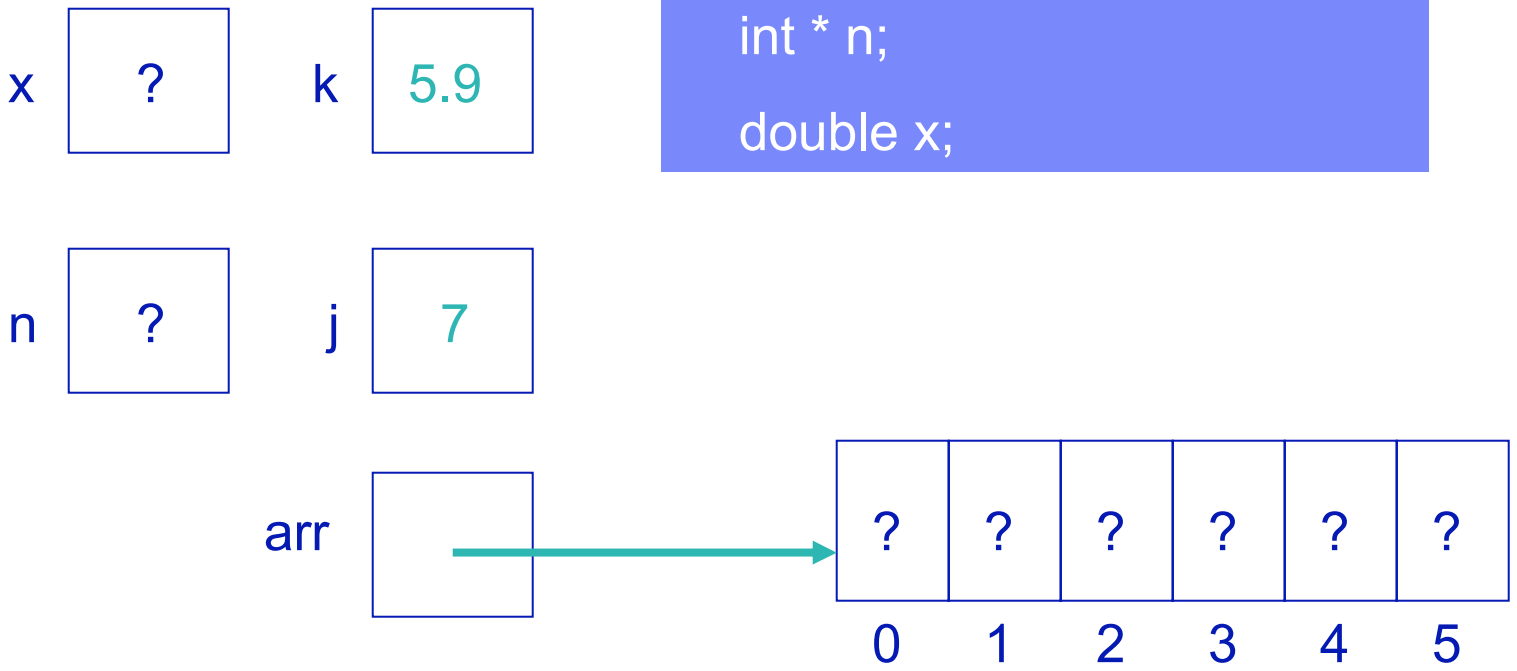
```
void* ptr;
```

## Generic Pointers

- Any kind of pointer can be stored in a variable whose type is void*
- If we know that a value in a pointer p of type void* is really of a specific pointer type x, and we want to treat the value p points to as a value of type x, we have to *cast* the pointer to type x

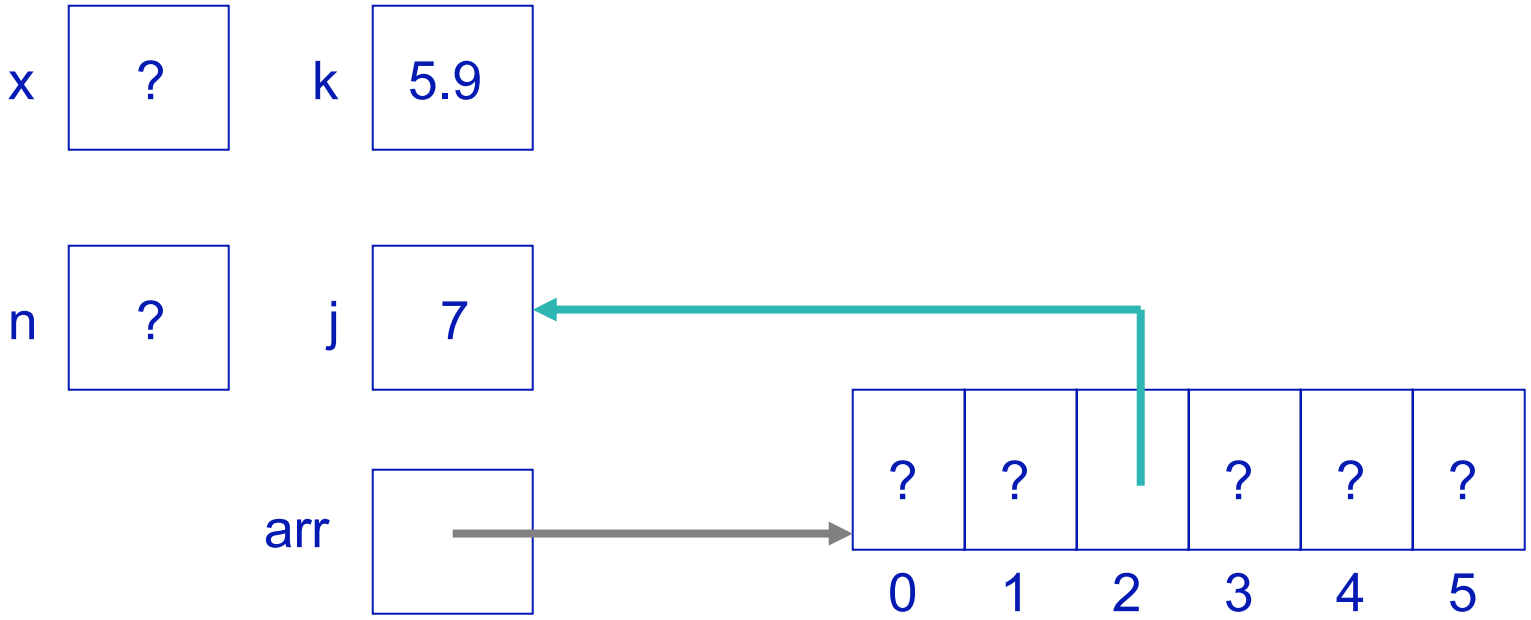
# Generic Pointer Example

```
void * arr[6];
int j=7;
double k = 5.9;
int * n;
double x;
```



## Generic Pointer Example

```
arr[2] = (void*)&j;        // type cast is okay, but not needed here
```

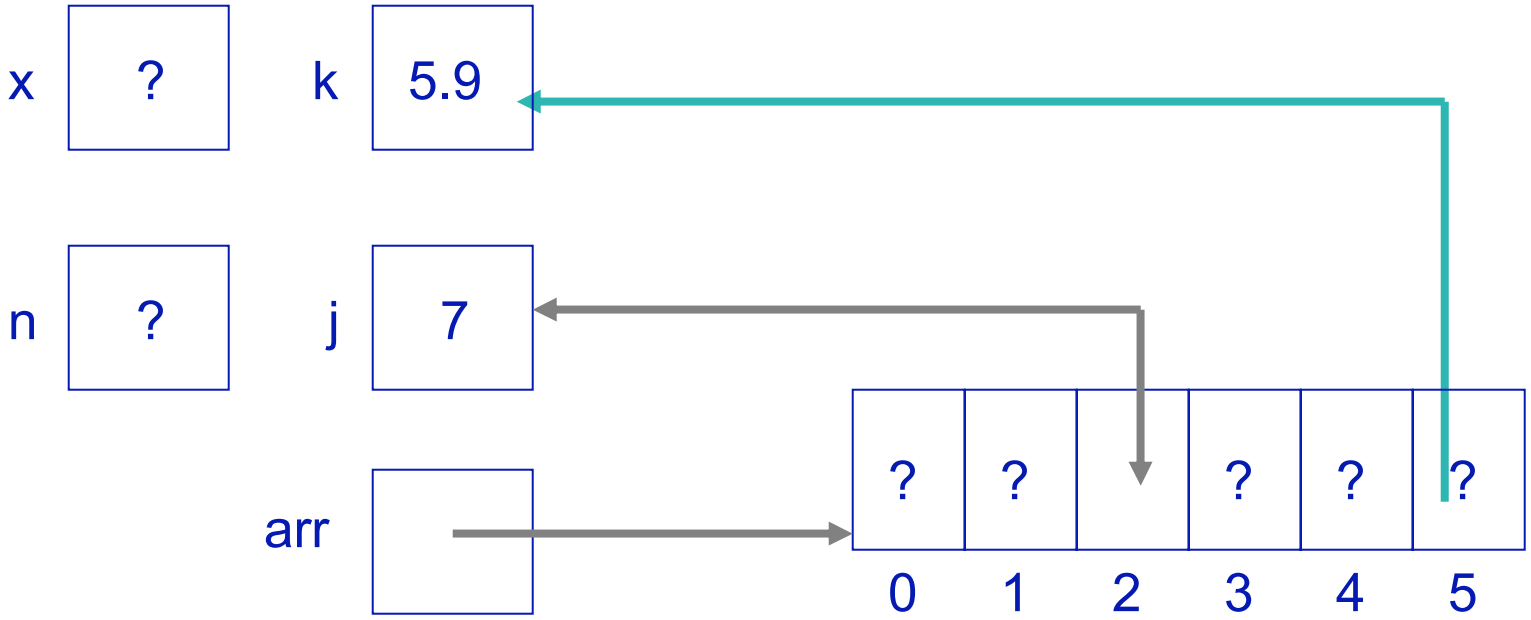




hmehta.scs@dauniv.ac.in

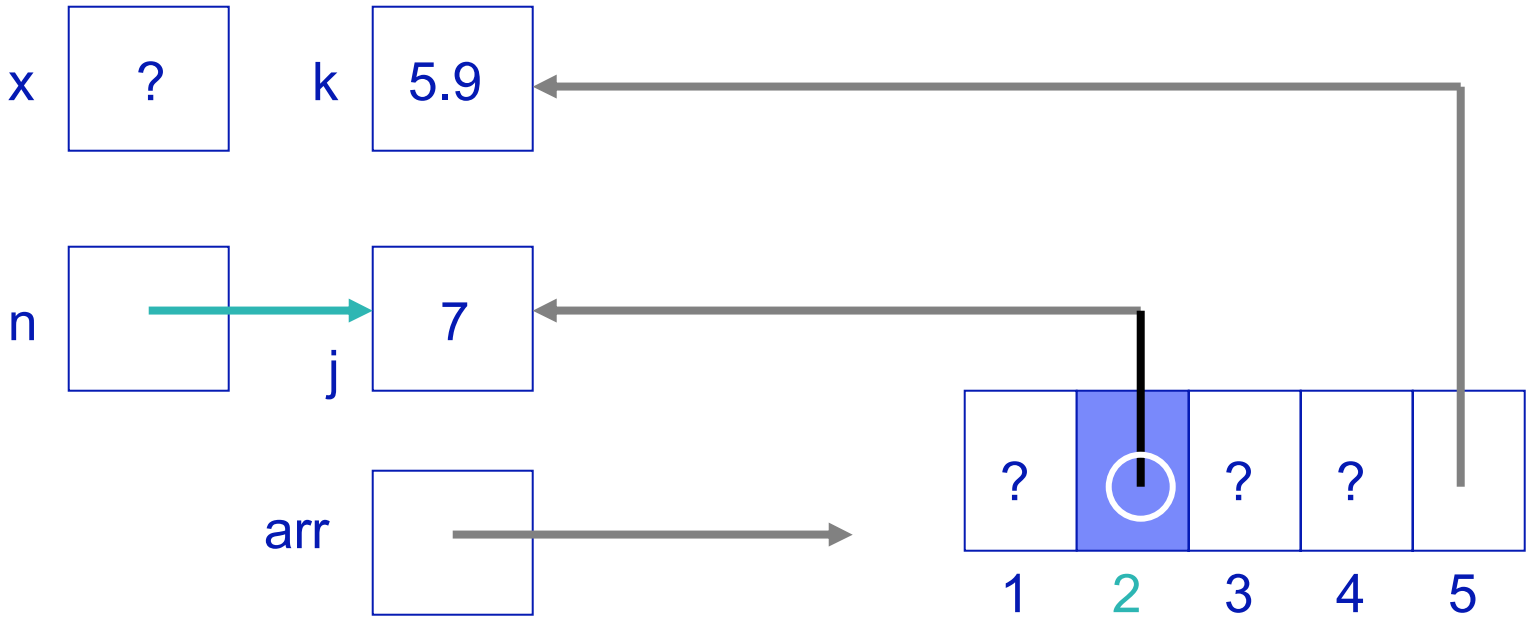
## Generic Pointer Example

```
arr[5] = &k;        // cast not needed, but could be used
```



## Generic Pointer Example

```
n = (int*)arr[2];        // cast is required here
```

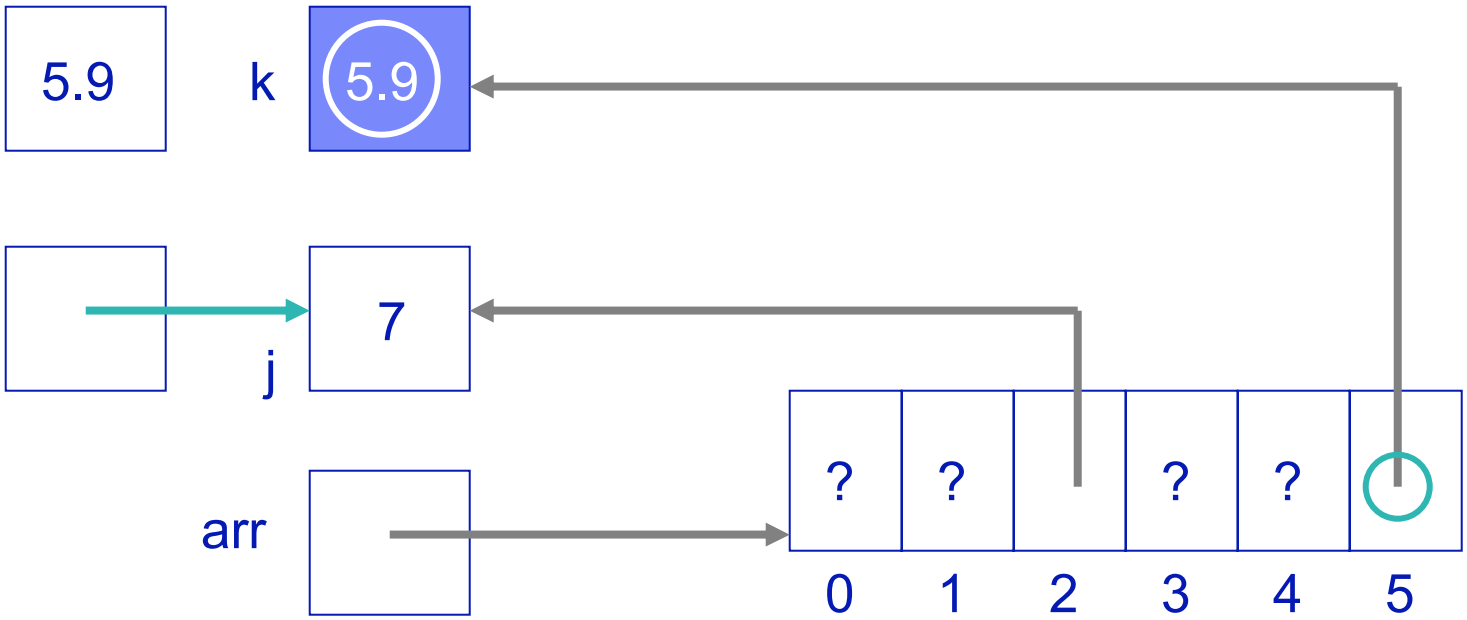




hmehta.scs@dauniv.ac.in

## Generic Pointer Example

```
x = *((double*)arr[5]);  // cast is required here
```





hmehta.scs@dauniv.ac.in



# Dynamic Memory Allocation

## Address allotment on the fly

School of Computer Science

hmehta.scs@dauniv.ac.in

# Dynamic Memory Allocation

- Up to now, any variables, including pointers, that we've created have been ***static***:
  - They exist only while the module in which they've been created is still executing
  - They disappear automatically upon exit from the module

# Dynamic Memory Allocation

- We can create entities such as ints, chars, arrays and complex data structures that will ***persist*** beyond exit from the module that built them
- This method of creating objects is called ***dynamic allocation of memory***

# Dynamic Memory Allocation

- Statically allocated variables are created within a call frame on the ***call stack***
- Dynamically allocated variables are created an area of memory known as the ***heap***

# Dynamic Memory Allocation in C

- Requires the use of pointer variables, and of one of the ***memory allocation functions*** from <stdlib.h>
- ***malloc***: the most commonly used memory allocation function

  void * malloc( size_t size );
  - Locates size consecutive bytes of free memory (memory that is not currently in use) in the heap, and returns a generic pointer to the block of memory
  - Returns NULL instead if size bytes can't be found

# Dynamic Memory Allocation in C

- Value returned by malloc is a generic pointer, and must be ***cast*** to the specific type of pointer the user intended to create

```
double * ptr;

ptr = (double *) (malloc( sizeof( double ) ) );
```

# Dynamic Memory Allocation in C

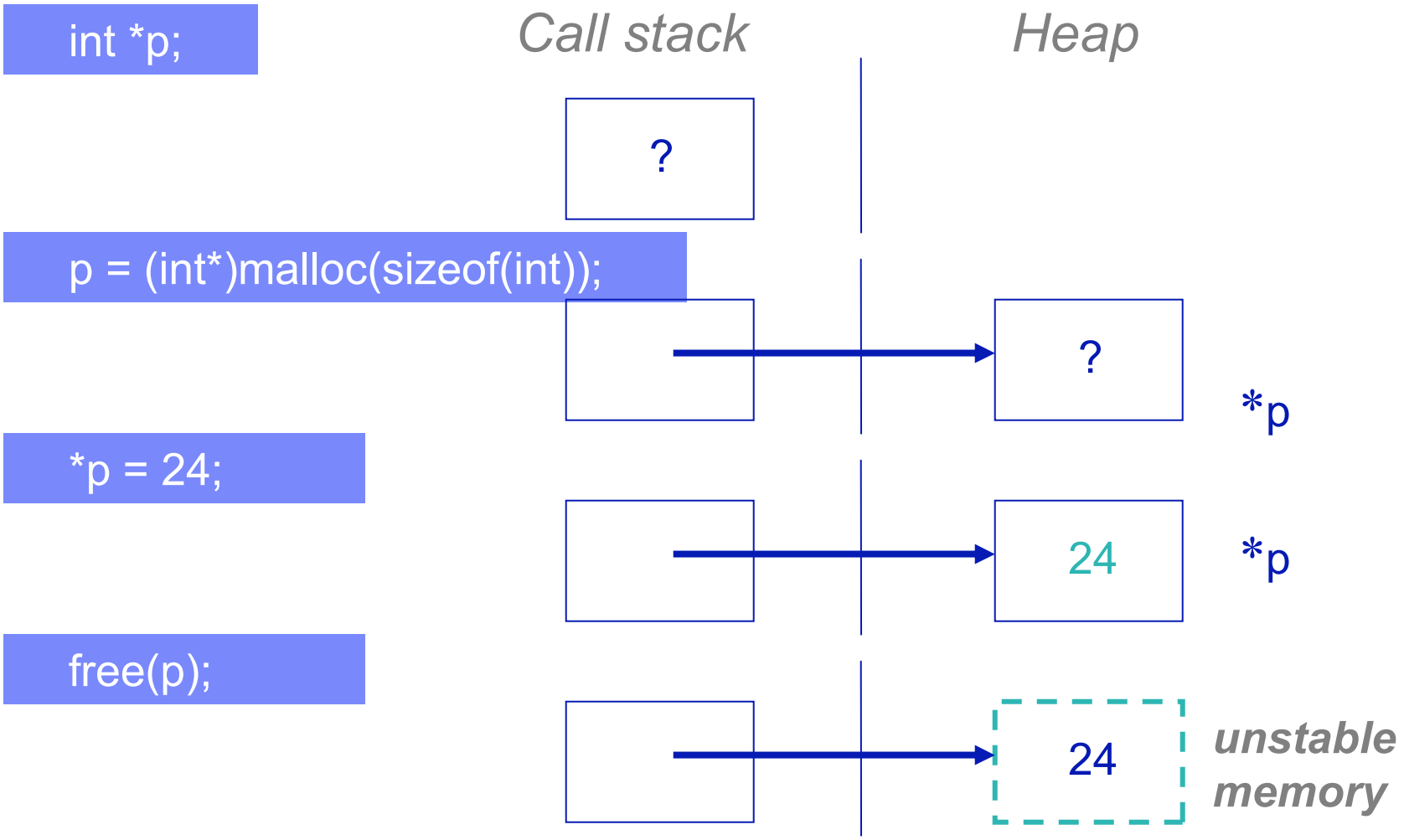
- ***Deallocation*** of dynamically allocated memory is not automatic
- No Java-style garbage collection occurs
- Programmer is responsible for recycling any dynamically allocated memory that is no longer needed
- Must use the free( ) function from <stdlib.h>

# Dynamic Memory Allocation in C

void free(void * ptr);

- Parameter ptr is a pointer to the first byte of an entity that was allocated from the heap
- At run-time, C checks the actual type of the pointer parameter to determine exactly how many bytes to deallocate
- Any attempt to access deallocated memory is unsafe

# Dynamic Allocation – Example 1

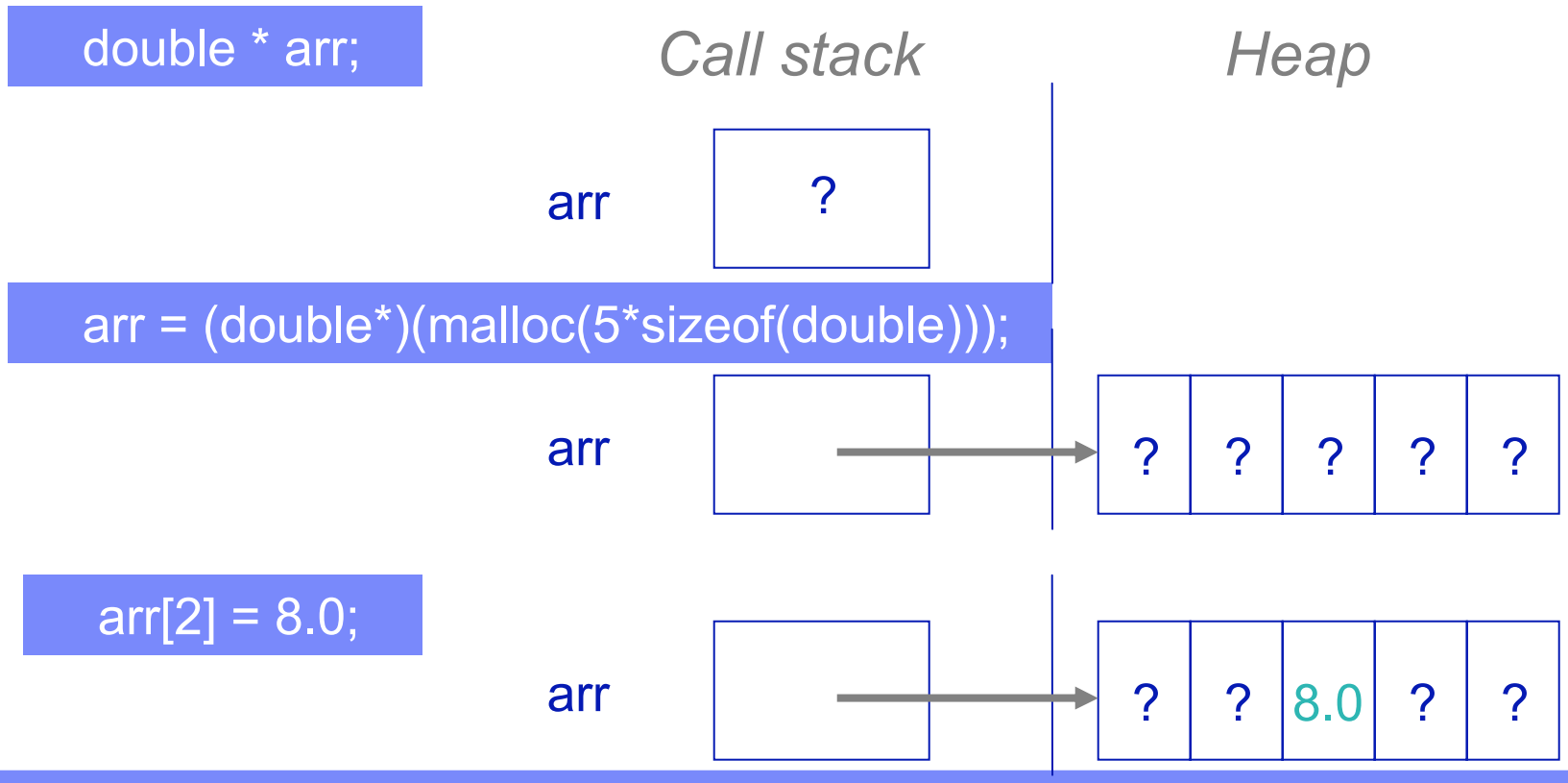




hmehta.scs@dauniv.ac.in

# Dynamic Allocation – Example 2

Pointer to an array:





hmehta.scs@dauniv.ac.in

# Dynamic Allocation – Example 3

The following program is wrong

```
#include <stdio.h>
int main()
{
    int *p;
    scanf("%d",p);
    …
    return 0;
}
```

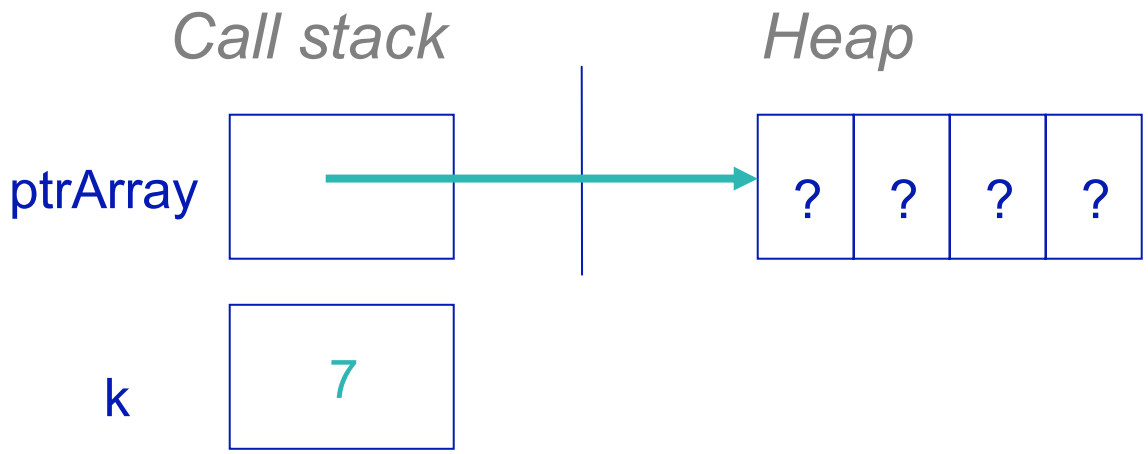
This one is correct:

```
#include <stdio.h>
int main()
{
    int *p;
    p = (int*)
        (malloc(sizeof(int)));
    scanf("%d",p);
    …
    return 0;
}
```

# Dynamic Allocation – Example 4

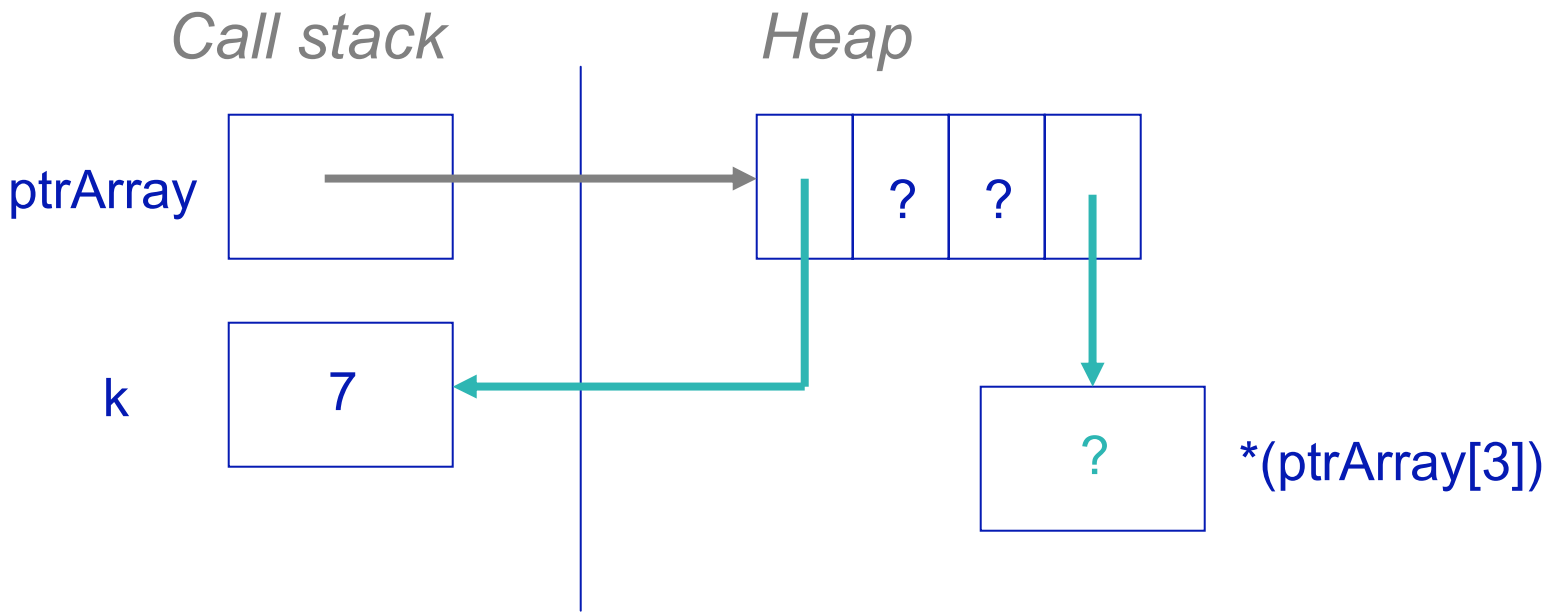
## Allocating an array of pointers:

```
int ** ptrArray;
int k = 7;
ptrArray = (int**)(malloc( 4*sizeof(int*)));
```



# Dynamic Allocation – Example 4

```
ptrArray[0] = &k;
ptrArray[3] = (int*)(malloc(sizeof(int));
```





hmehta.scs@dauniv.ac.in

# Dynamic Memory Allocation in C

- ***calloc***: Another commonly used memory allocation function

  void *calloc(size_t nitems, size_t size);

  - Locates (nitems*size) consecutive bytes of free memory (memory that is not currently in use) in the heap, and returns a generic pointer to the block of memory
  - Returns NULL instead if size bytes can't be found



# Pointers as Parameters in C

School of Computer Science

hmehta.scs@dauniv.ac.in

```
#include <stdio.h>
void add1(int a, int *b) {
    a++; (*b)++;
    printf("%d\n", a);
    printf("%d\n", *b);
    return
}
int main() {
    int j = 4; k = 8;
    add1( j, &k );
    printf("%d\n", j);
    printf("%d\n", k);
    return 0;
}
```

# Example:

add1 accepts as parameters an int a and a pointer to int b

The call in the main is made, a is associated with a copy of main program variable j, and b is associated with a copy of the address of main program variable k

Output from the program is:

5 *(in add1)*

9

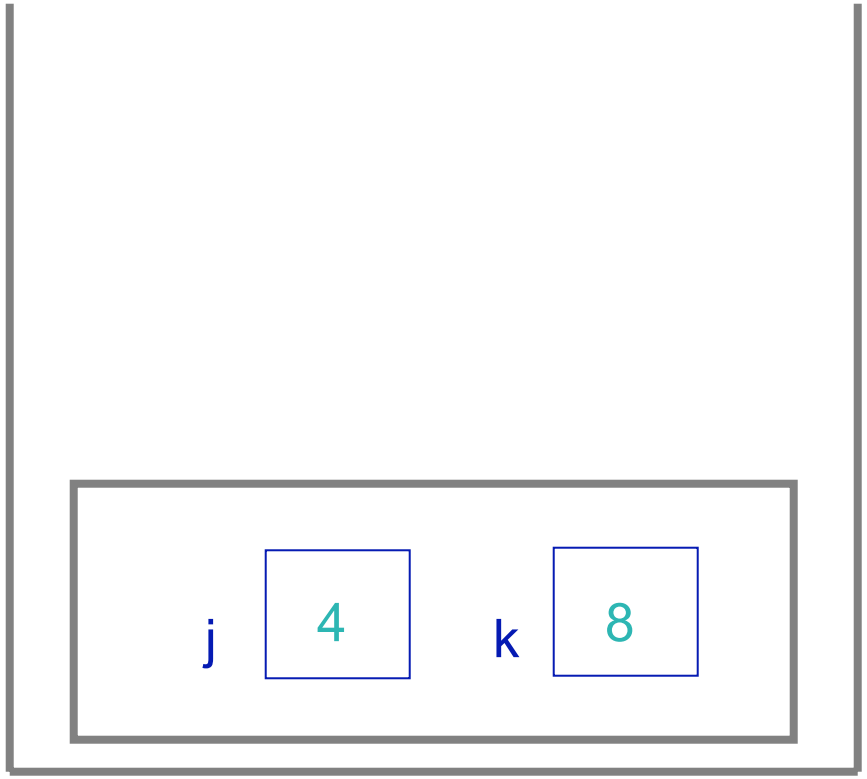
4 *(back in main)*

9

# Example (cont'd):

```
#include <stdio.h>
void add1(int a, int *b) {
    a++; (*b)++;
    printf("%d\n", a);
    printf("%d\n", *b);
    return
}
int main() {
    int j = 4; k = 8;
    add1( j, &k );
    printf("%d\n", j);
    printf("%d\n", k);
    return 0;
}
```

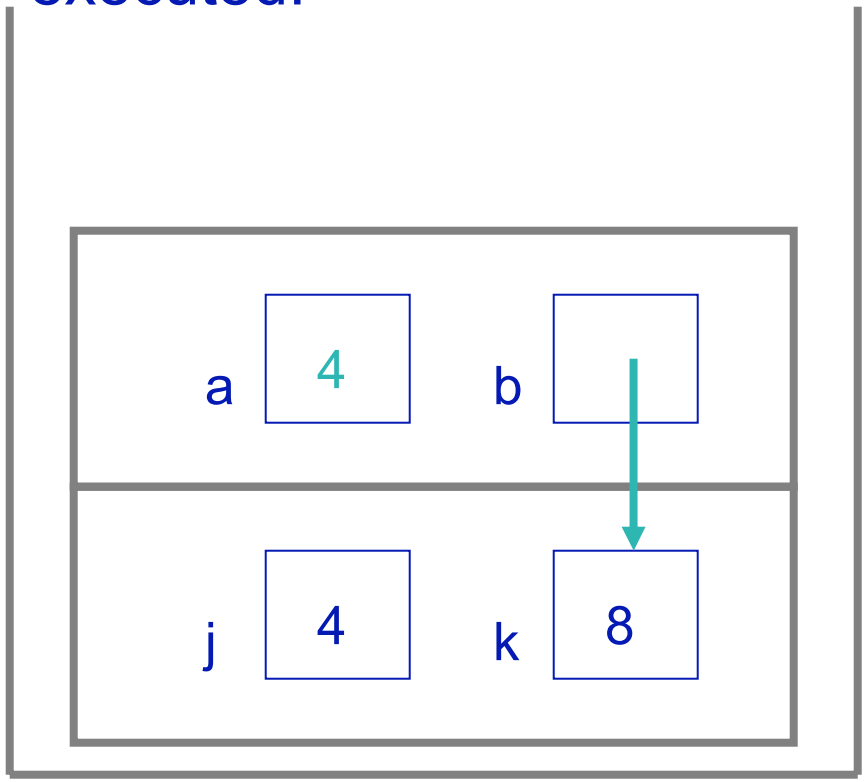
Call stack immediately before the call to add1



```
#include <stdio.h>
void add1(int a, int *b) {
    a++; (*b)++;
    printf("%d\n", a);
    printf("%d\n", *b);
    return
}
int main() {
    int j = 4; k = 8;
    add1( j, &k );
    printf("%d\n", j);
    printf("%d\n", k);
    return 0;
}
```

# Example (cont'd):

Call stack after the call to add1, but before a++; is executed:



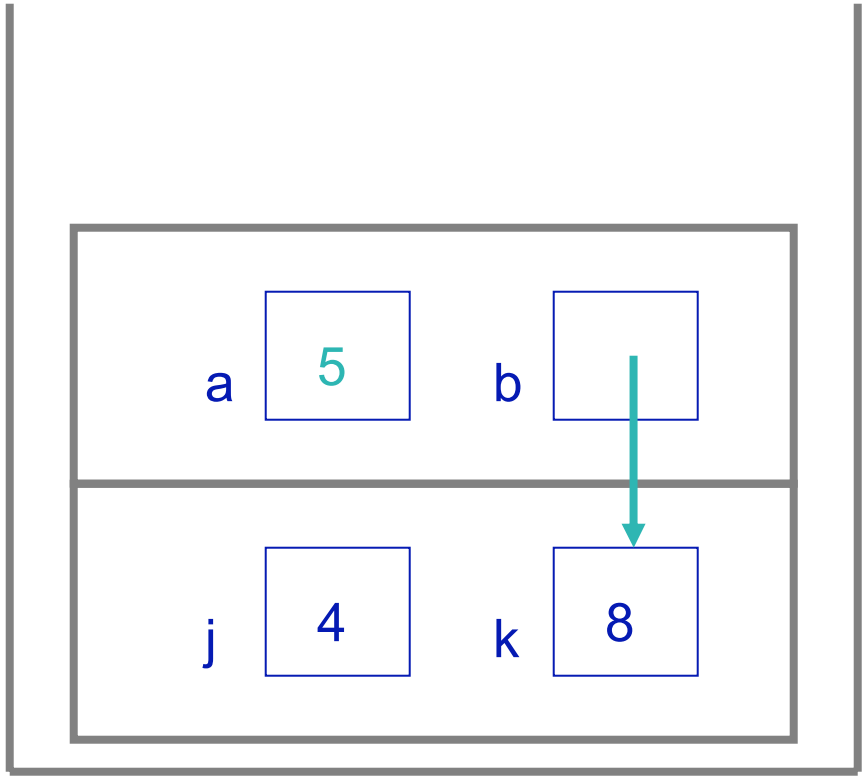


hmehta.scs@dauniv.ac.in

# Example (cont'd):

```
#include <stdio.h>
void add1(int a, int *b) {
    a++; (*b)++;
    printf("%d\n", a);
    printf("%d\n", *b);
    return
}
int main() {
    int j = 4; k = 8;
    add1( j, &k );
    printf("%d\n", j);
    printf("%d\n", k);
    return 0;
}
```

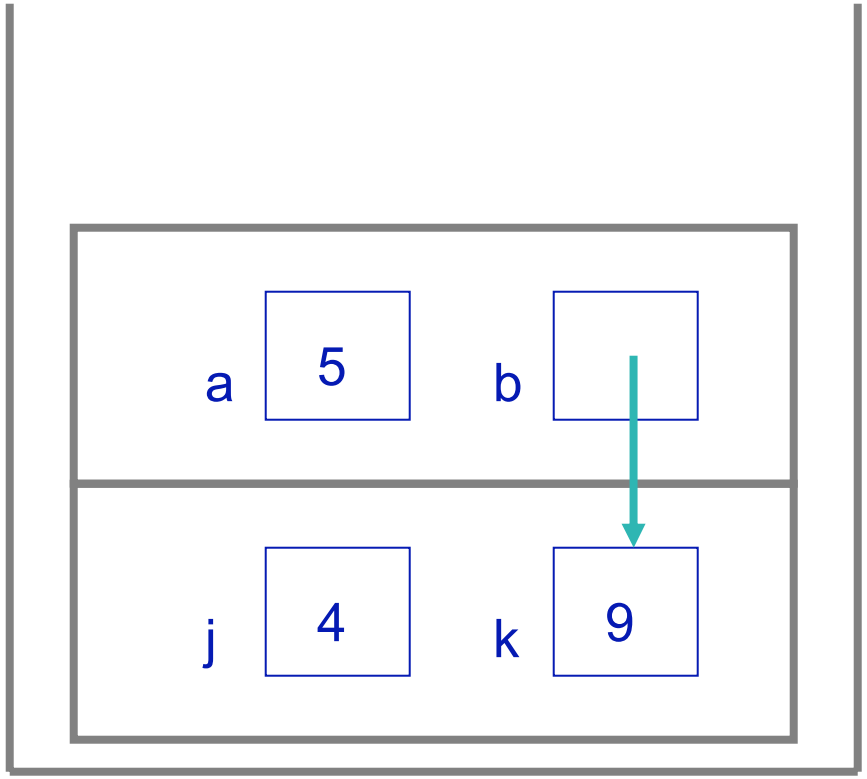
After a++; is executed



# Example (cont'd):

```
#include <stdio.h>
void add1(int a, int *b) {
    a++; (*b)++;
    printf("%d\n", a);
    printf("%d\n", *b);
    return
}
int main() {
    int j = 4; k = 8;
    add1( j, &k );
    printf("%d\n", j);
    printf("%d\n", k);
    return 0;
}
```

After (*b)++; is executed



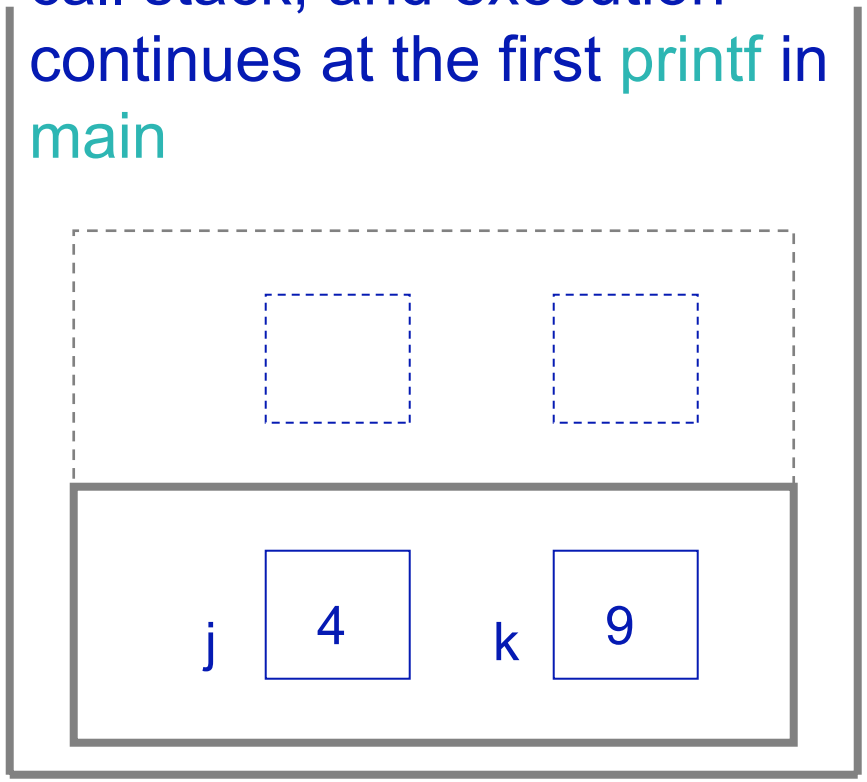


hmehta.scs@dauniv.ac.in

```
#include <stdio.h>
void add1(int a, int *b) {
    a++; (*b)++;
    printf("%d\n", a);
    printf("%d\n", *b);
    return
}
int main() {
    int j = 4; k = 8;
    add1( j, &k );
    printf("%d\n", j);
    printf("%d\n", k);
    return 0;
}
```

# Example (cont'd):

Upon returning from add1, its call frame is popped from the call stack, and execution continues at the first printf in main



## Example: Pointer Parameters

- Suppose that a function has the prototype:

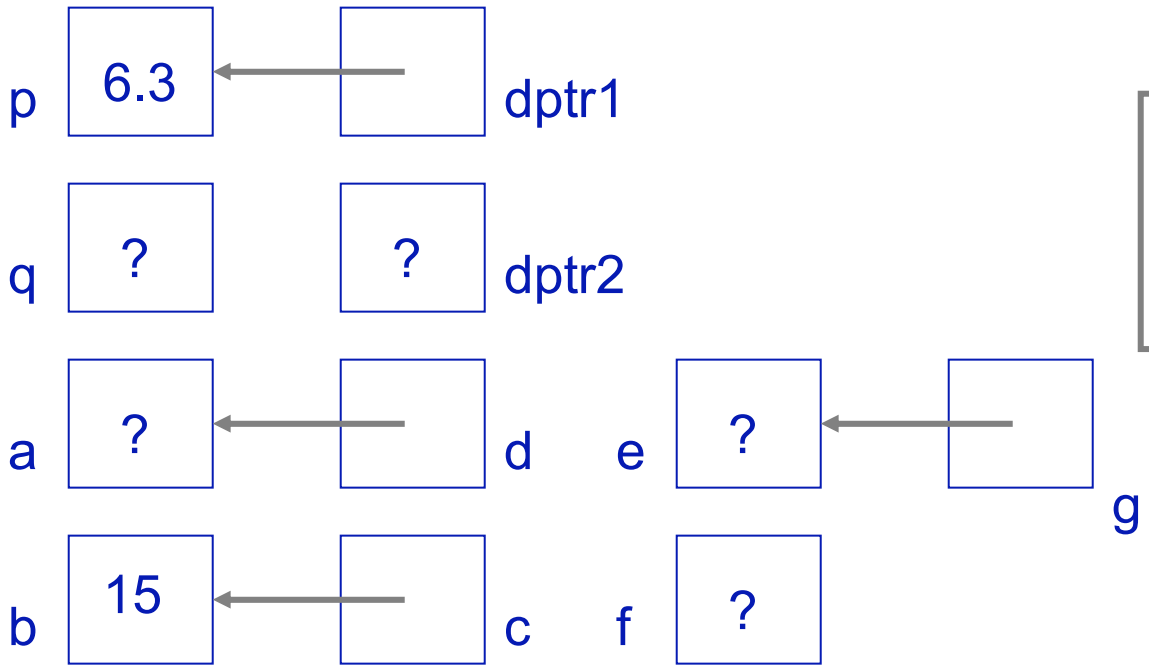
```
void fn( double * x, int ** k)
```

- Inside fn, x is of type double*, and *x is of type double
- k is of type int**, *k is of type int*, and **k is of type int

# Example: Pointer Parameters (cont'd)

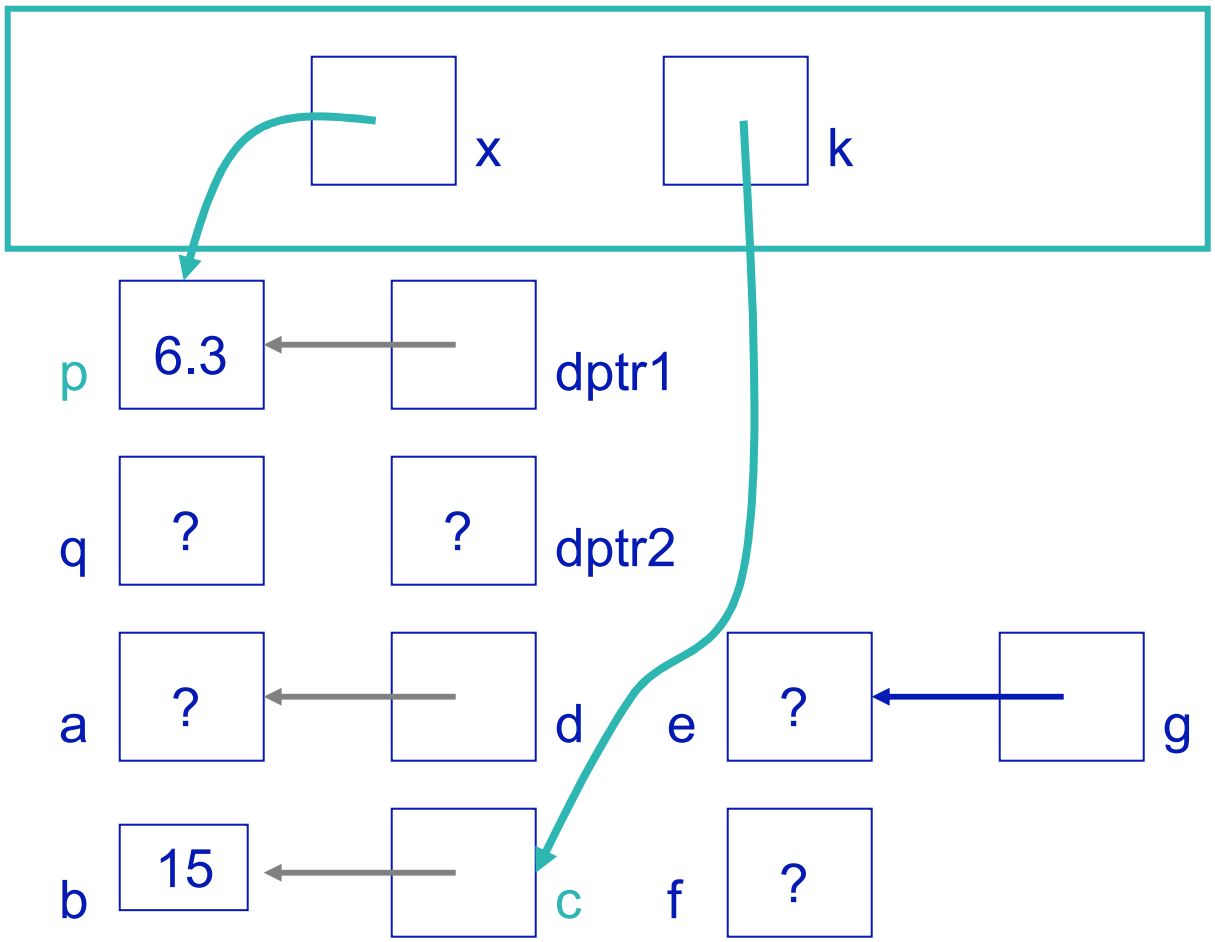
- Assume the main program declarations:

```
double p=6.3, q, *dptr1=&p, *dptr2;
int a, b=15, *c=&b, *d=&a, *e, **f, **g=&e;
```




We will examine a variety of valid calls to fn

# Example: Pointer Parameters (cont'd)




Function call:

```
fn ( &p, &c );
```

***Recall the prototype:***

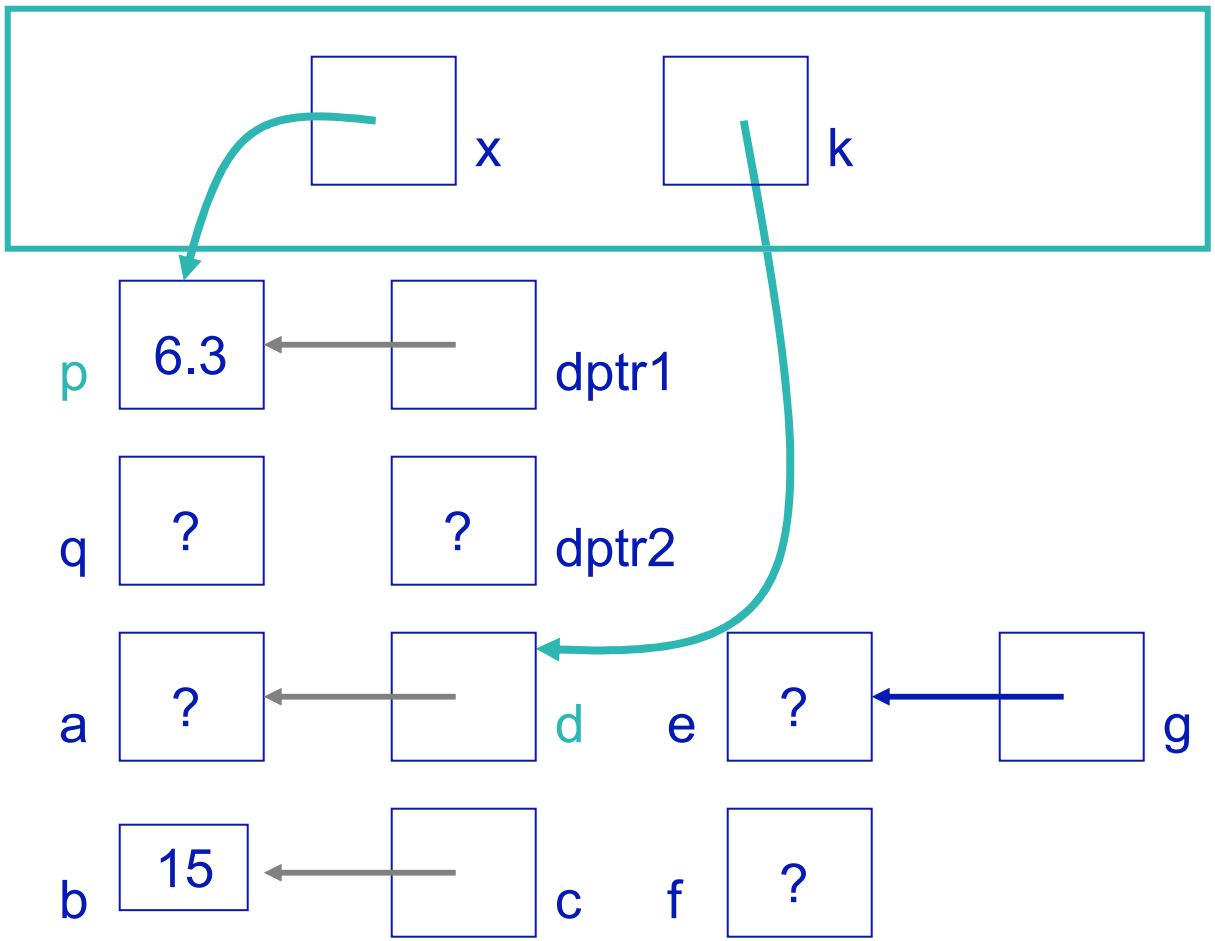
```
void fn( double * x,
         int ** k)
```

***Recall from main:***

```
int b,*c=&b;
double p=6.3;
```


hmehta.scs@dauniv.ac.in

# Example: Pointer Parameters (cont'd)




Function call:

```
fn ( dptr1, &d );
```

***Recall the prototype:***

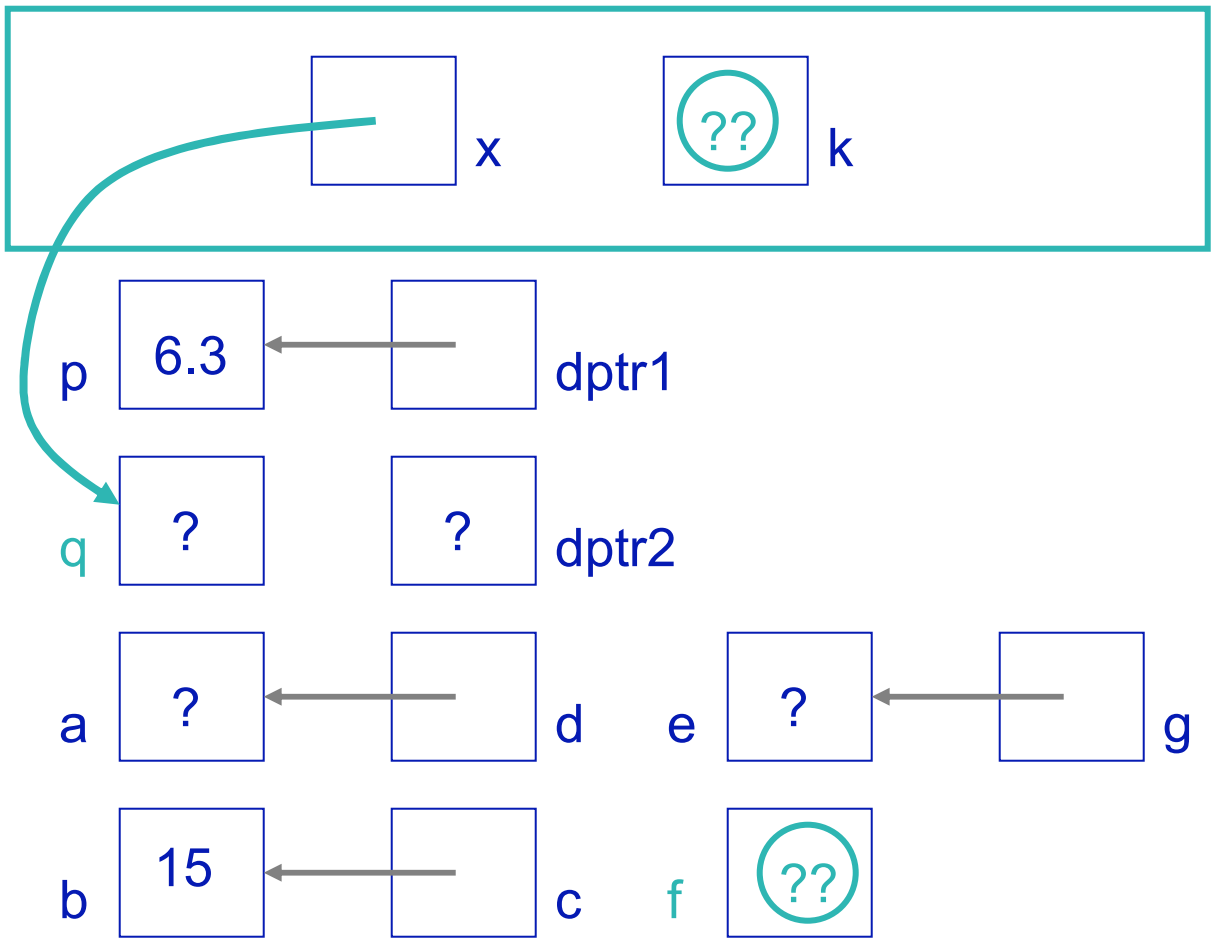
```
void fn( double * x,
         int ** k)
```

***Recall from main:***

```
int a,*d=&a;
double p=6.3,
*dptr1=&p;
```


hmehta.scs@dauniv.ac.in

# Example: Pointer Parameters (cont'd)




Function call:

```
fn ( &q, f );
```

***Danger***: f doesn't point at anything yet

***Recall the prototype:***

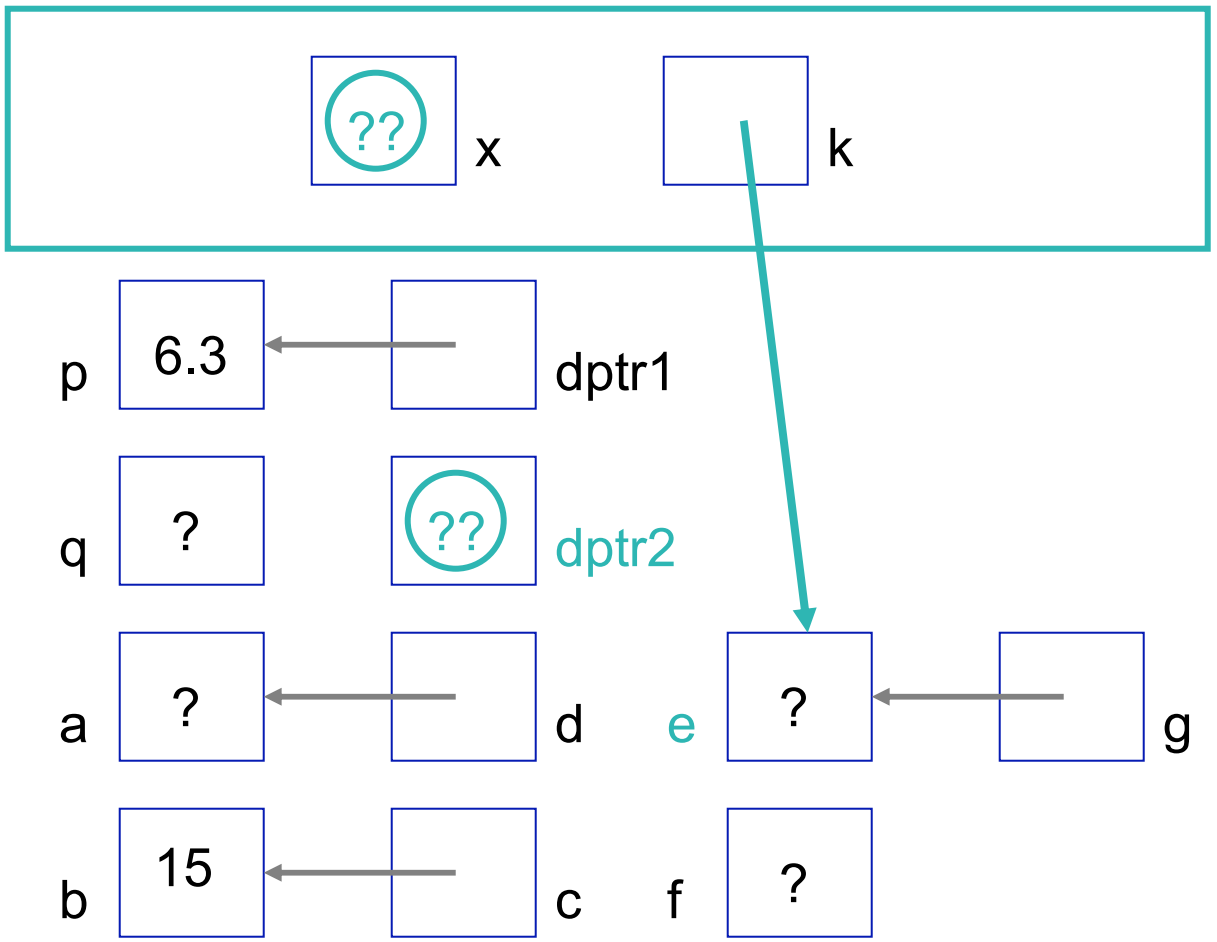
```
void fn( double * x,
         int ** k)
```

***Recall from main:***

```
int ** f;
double q;
```


hmehta.scs@dauniv.ac.in

# Example: Pointer Parameters (cont'd)




Function call:

```
fn ( dptr2, &e);
```

***Danger*** dptr2 is uninitialized

***Recall the prototype:***

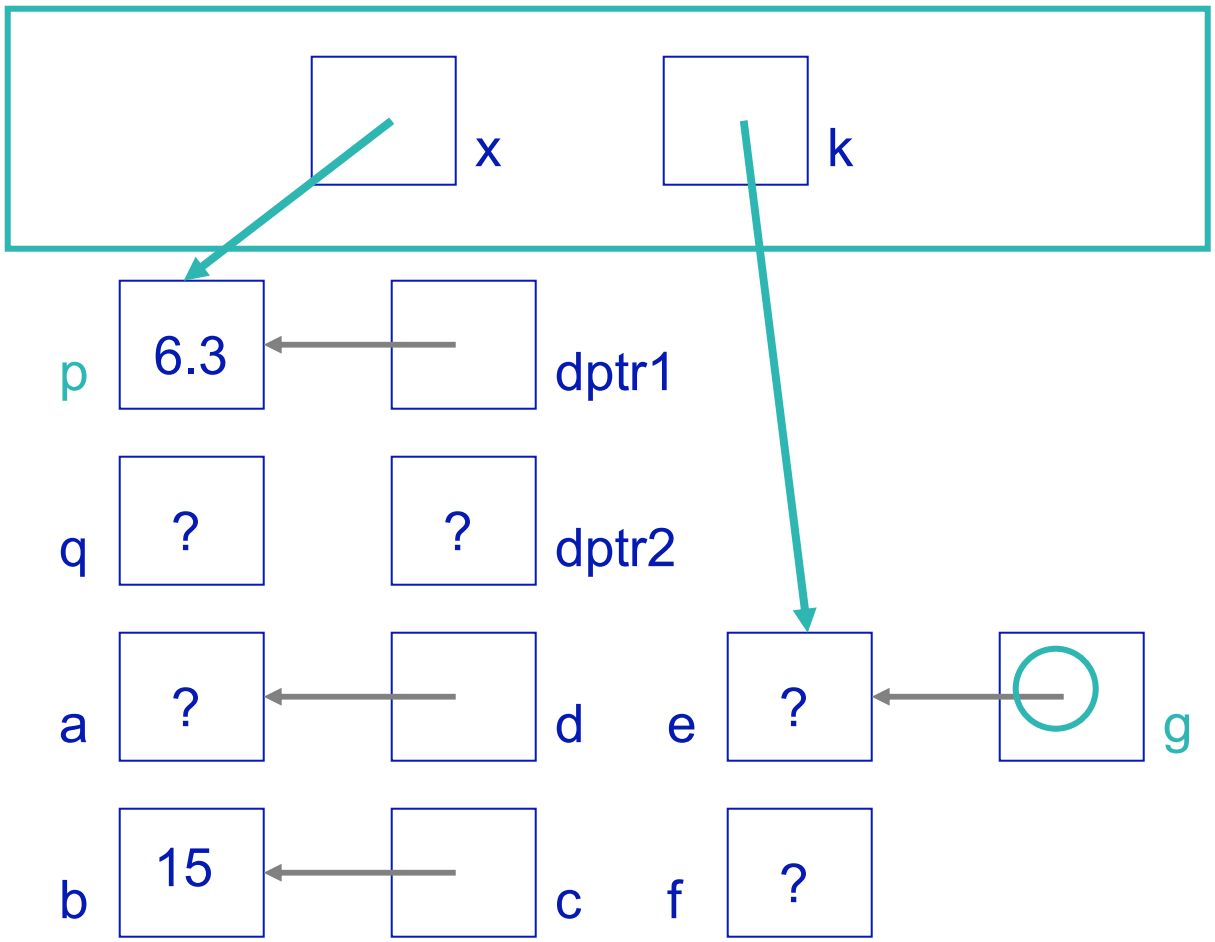
```
void fn( double * x,
         int ** k)
```

***Recall from main:***

```
int * e;
double * dptr2;
```


hmehta.scs@dauniv.ac.in

# Example: Pointer Parameters (cont'd)




Function call:

```
fn ( &p, g );
```

***Recall the prototype:***

***void fn( double * x, int ** k)***

***Recall from main:***

```
int *e, **g=&e;
double p=6.3;
```


hmehta.scs@dauniv.ac.in

# Why Use Pointer Parameters?

- So that the value being referenced *can be modified* by the function
- *Efficiency:* It takes less time and memory to make a copy of a reference (typically a 2 or 4 byte address) than a copy of a complicated structure

# Arrays as Parameters

- C does not allow arrays to be *copied* as parameters
- *Recall*:
  - Array name is really a *pointer* to the first element (location 0)
  - Indexing with square brackets has the effect of *dereferencing* to a particular array location

# Arrays as Parameters

- When we put an array name in a list of actual parameters, we are really passing the pointer to the first element
- Do not use the address operator & when passing an array

# Arrays as Parameters

- Suppose that we wish to pass an array of integers to a function fn, along with the actual number of array locations currently in use
- The following function prototypes are equivalent:

```
void fn( int * arr, int size );

void fn( int [ ] arr, int size);
```

```
#include <stdlib.h>
void init1 ( int [ ] a, int s1, int * b, int s2 ) {
    for (int j = 0; j < s1; j++)
        a[ j ] = j;
    for (int k = 0; k < s2; k++)
        b[ k ] = s2 - k;
}
int main( ) {
    int s[4], *t;
    t = (int*)(malloc(6*sizeof(int));
    t[0] = 6;
    init1( s, 4, t, t[0] );
    …
```

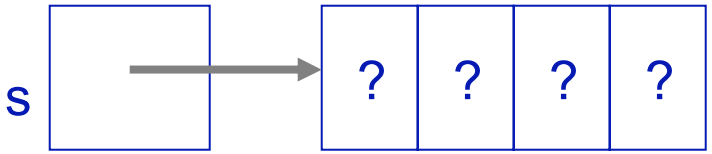
## Example

Could use **int *a** and **int[ ] b** instead

We trace through this code on the next few slides

# Example (cont'd)

Before the call to init1





*Call stack* – **main** variables

*Heap*

***After execution of:***

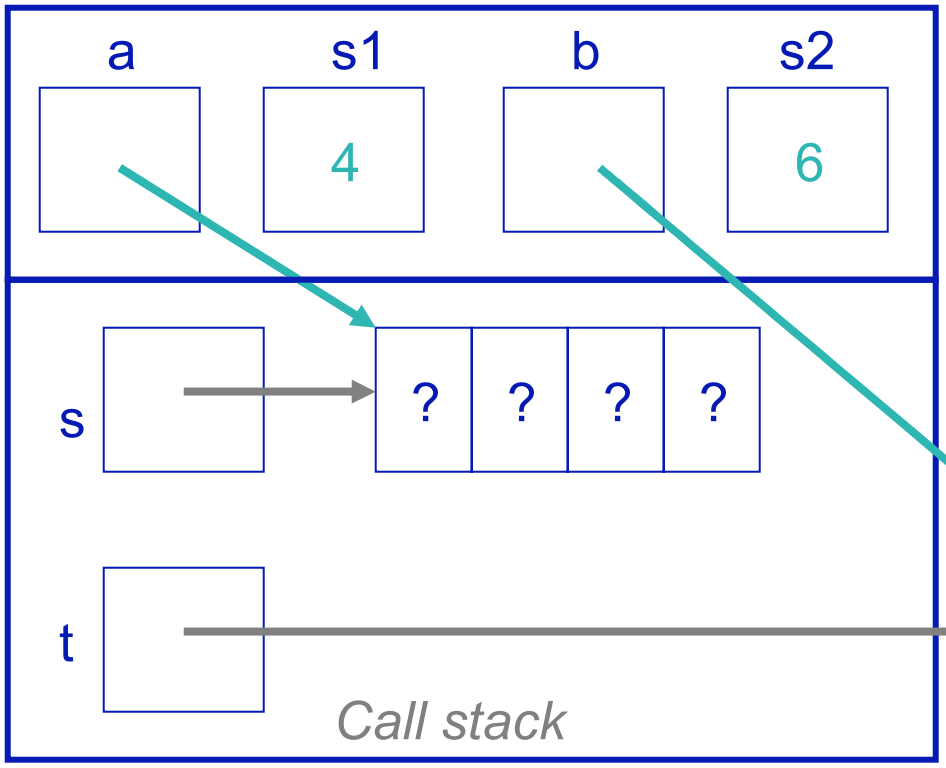
```
int s[4], *t;
t = (int*)(malloc(6*sizeof(int));
t[0] = 6;
```


hmehta.scs@dauniv.ac.in

# Example (cont'd)

***After*** call to init1 but ***before*** its execution begins:

*local vars j and k have been ignored in the call frame for init1*

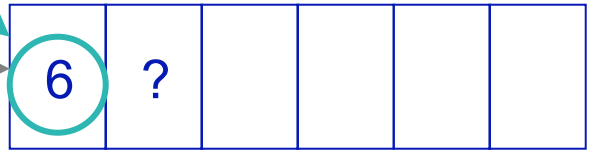



***Function prototype:***

```
void init1 ( int [ ] a, int s1,
             int * b, int s2 );
```

***Function call:***

```
init1( s, 4, t, t[0] );
```



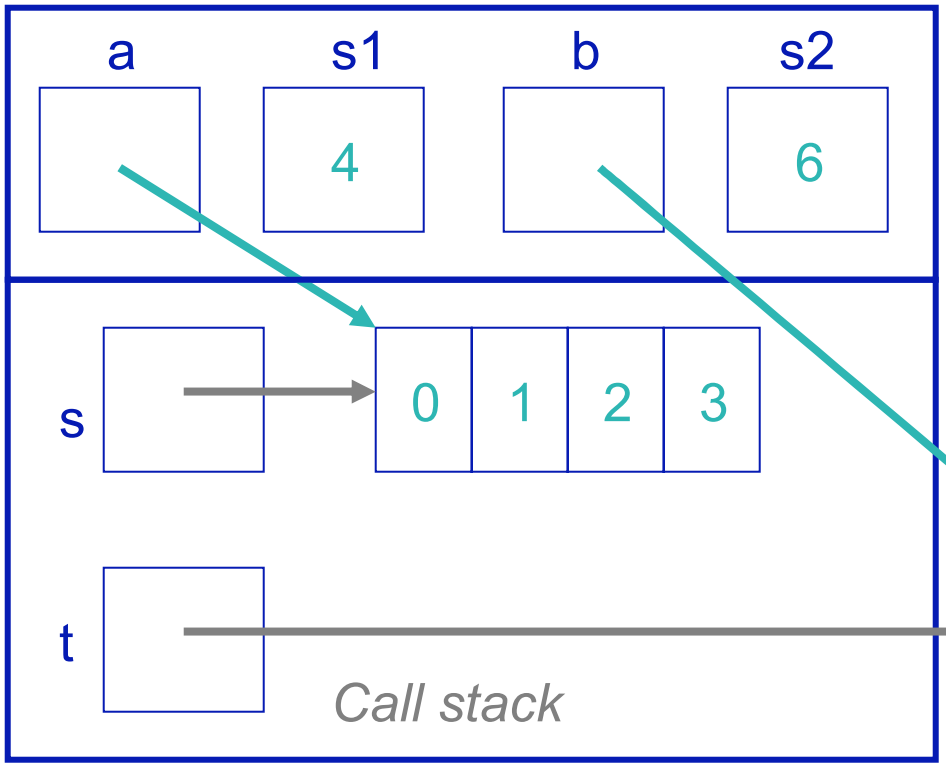

*Heap*


hmehta.scs@dauniv.ac.in

# Example (cont'd)

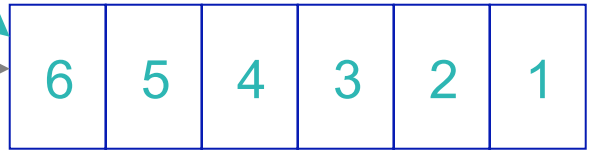
## *After* execution of init1 but *before* return to main

*local vars j and k have been ignored in the call frame for init1*




***After:***

```
for (int j = 0; j < s1; j++)
    a[ j ] = j;
for (int k = 0; k < s2; k++)
    b[ k ] = s2 - k;
```

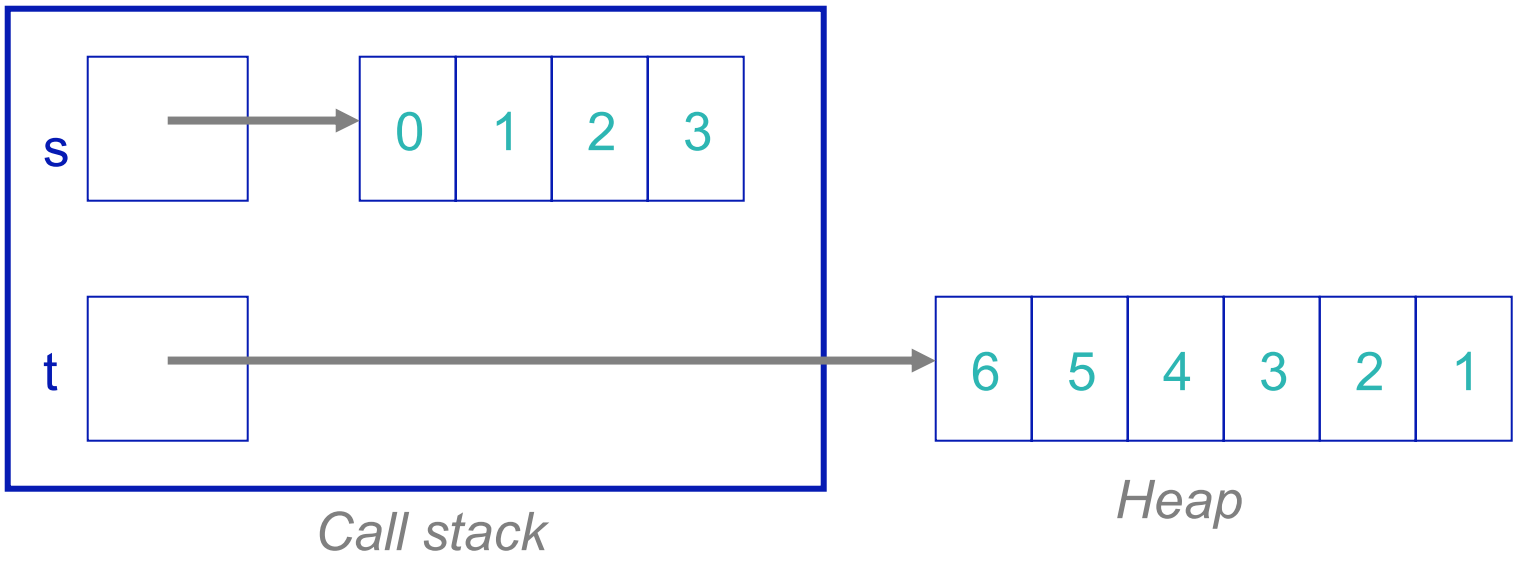



*Heap*


hmehta.scs@dauniv.ac.in

# Example (cont'd)

## *After* returning to main




Call frame for the call to init1 has been recycled

# Why Is '**' Needed?

- We may want a function to initialize or change a pointer
- Notation can also be used when creating or passing an array of pointers

```
#include <stdlib.h>
#include<time.h>
void init2( int *** t, int ** j ) {
    int k = 3 + rand( )%10;
    *t = (int**)(malloc(k*sizeof(int*)));
    *j = (int*)(malloc(sizeof(int)));
    **j = k;
}
int main( ) {
    int **v;    int *q;
    srand((unsigned int) time(NULL));
    init2( &v, &q );
    v[1] = q;
    …
```

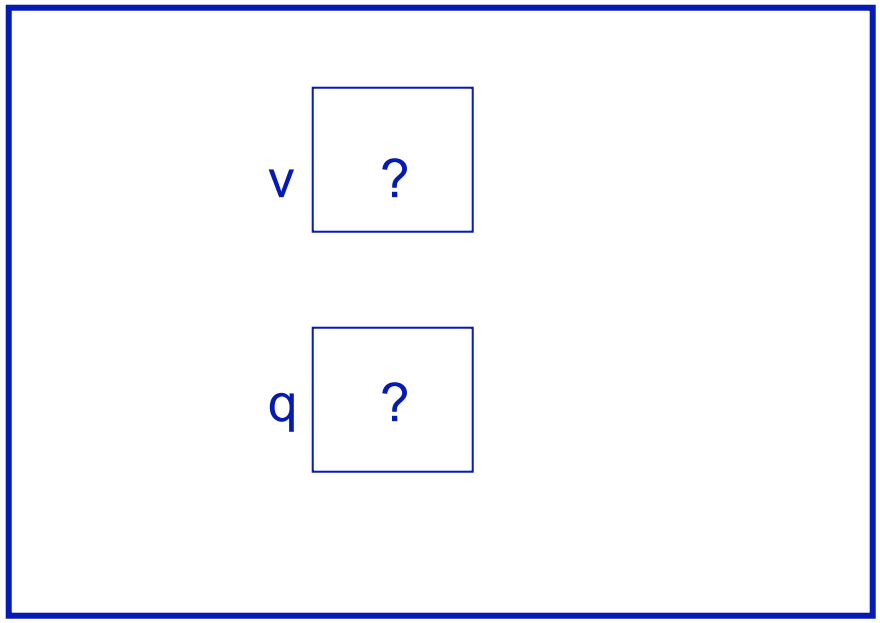
## Example

Dynamic allocation in init2 creates an array of pointers for main program variable v to point at, and an int location for q to point at; the value that q points at is also initialized

Trace follows on next slides

# Example (cont'd)

## Before the call to init2:



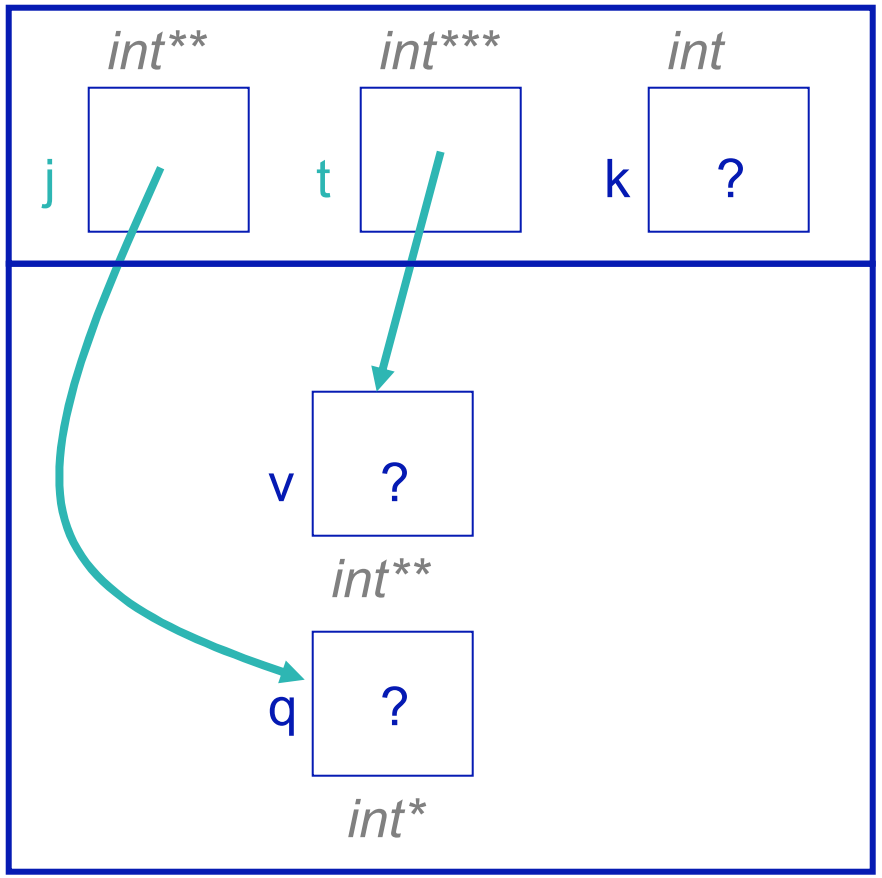

*Call stack*

***After declarations:***

```
int **v;
int *q;
```


hmehta.scs@dauniv.ac.in

# Example (cont'd)




Call stack

*After* the call to init2, but *before* body is executed:

***Function prototype:***

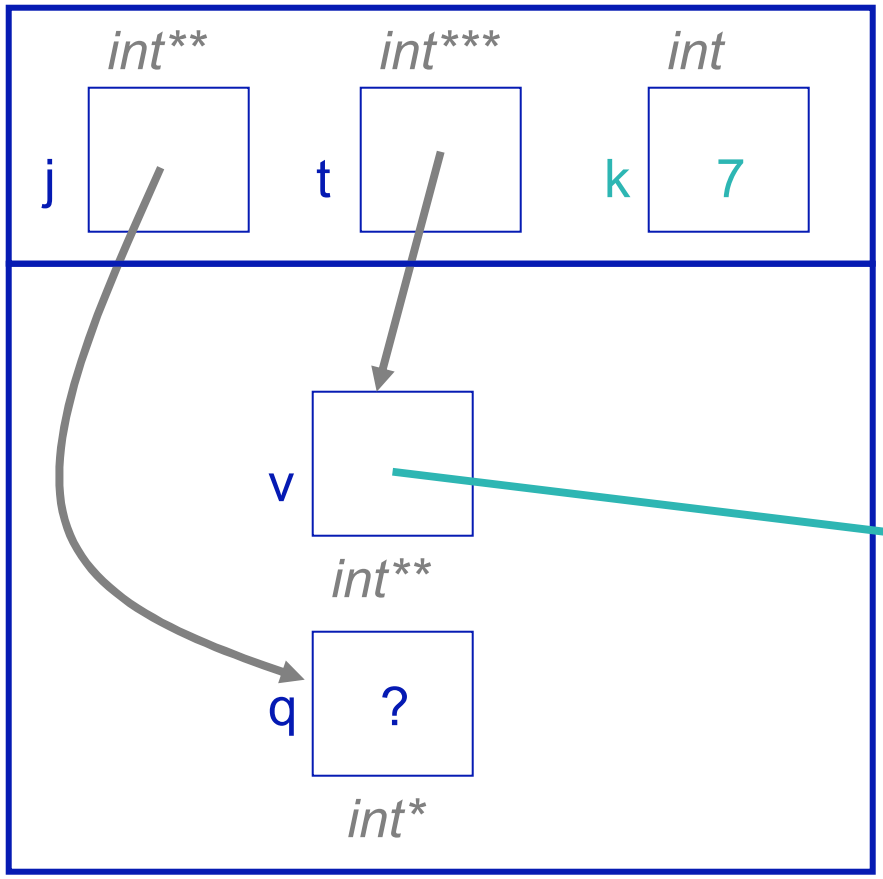
```
void init2( int *** t, int ** j );
```

***Function call:***

```
init2( &v, &q );
```


hmehta.scs@dauniv.ac.in

# Example (cont'd)

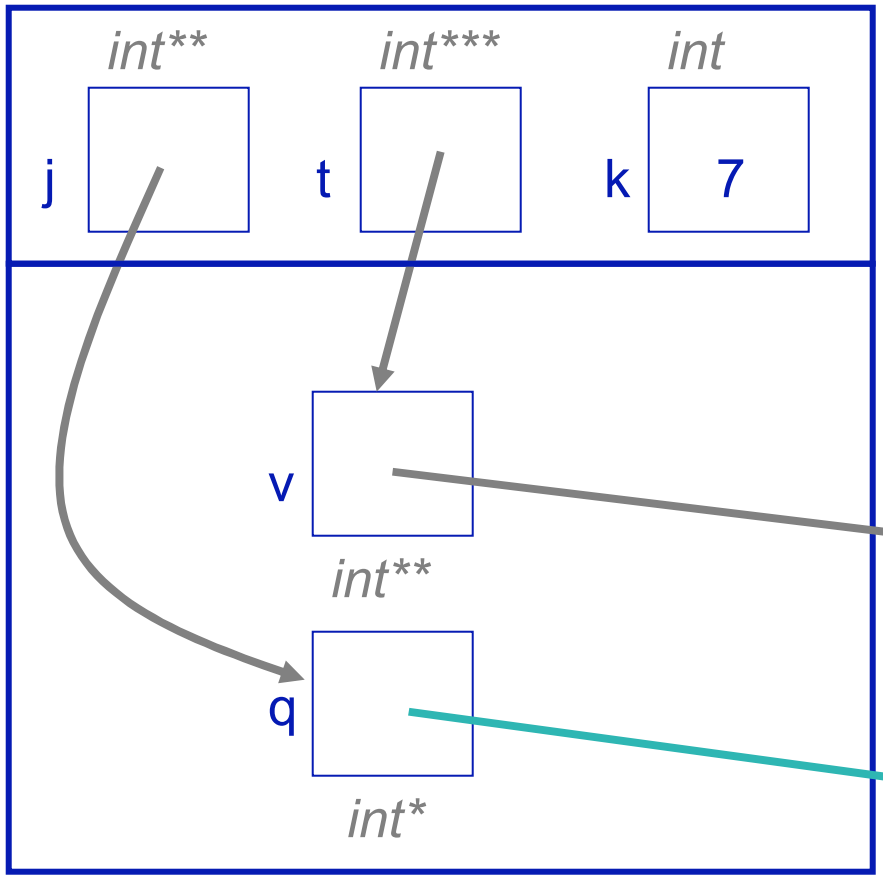



```
int k = 3 + rand( )%10;
// pretend that k is now 7
*t = (int**)(malloc(k*sizeof(int*)));
```


*Heap*

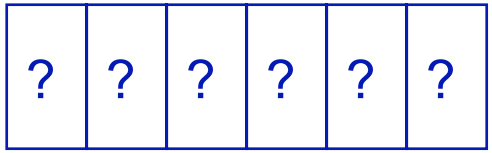

hmehta.scs@dauniv.ac.in

# Example (cont'd)




After execution of:

```
*j = (int*)(malloc(sizeof(int)));
```

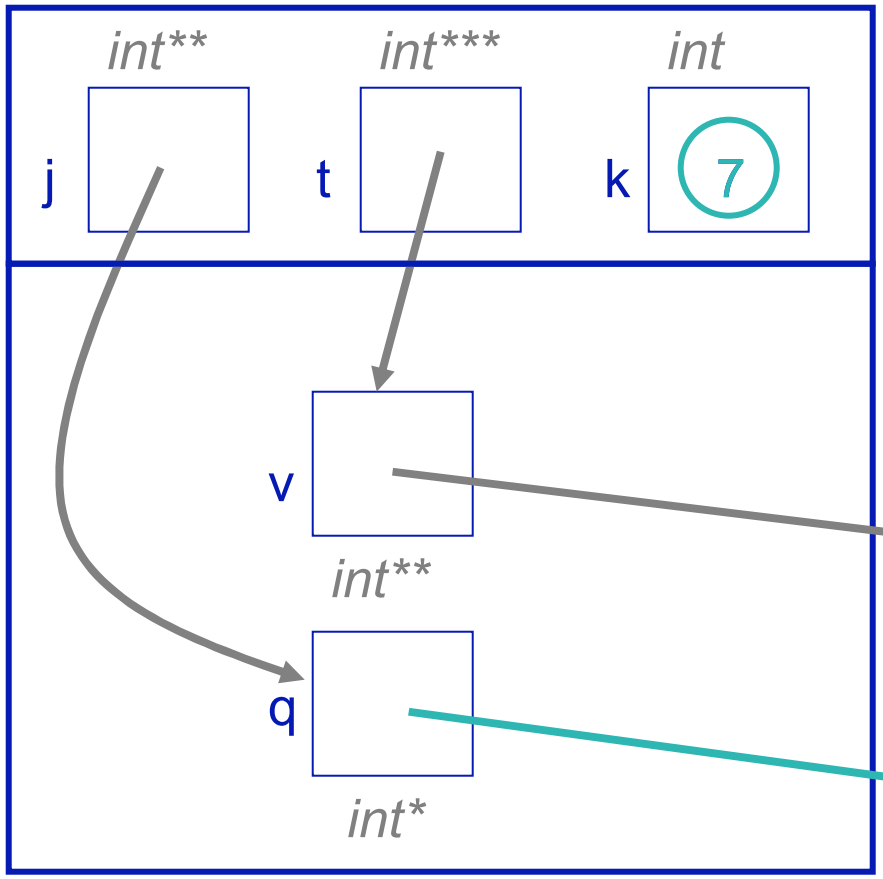




Heap

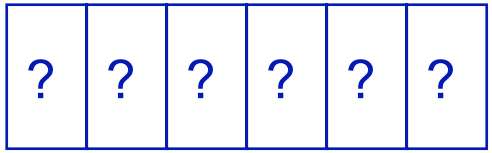

hmehta.scs@dauniv.ac.in

# Example (cont'd)



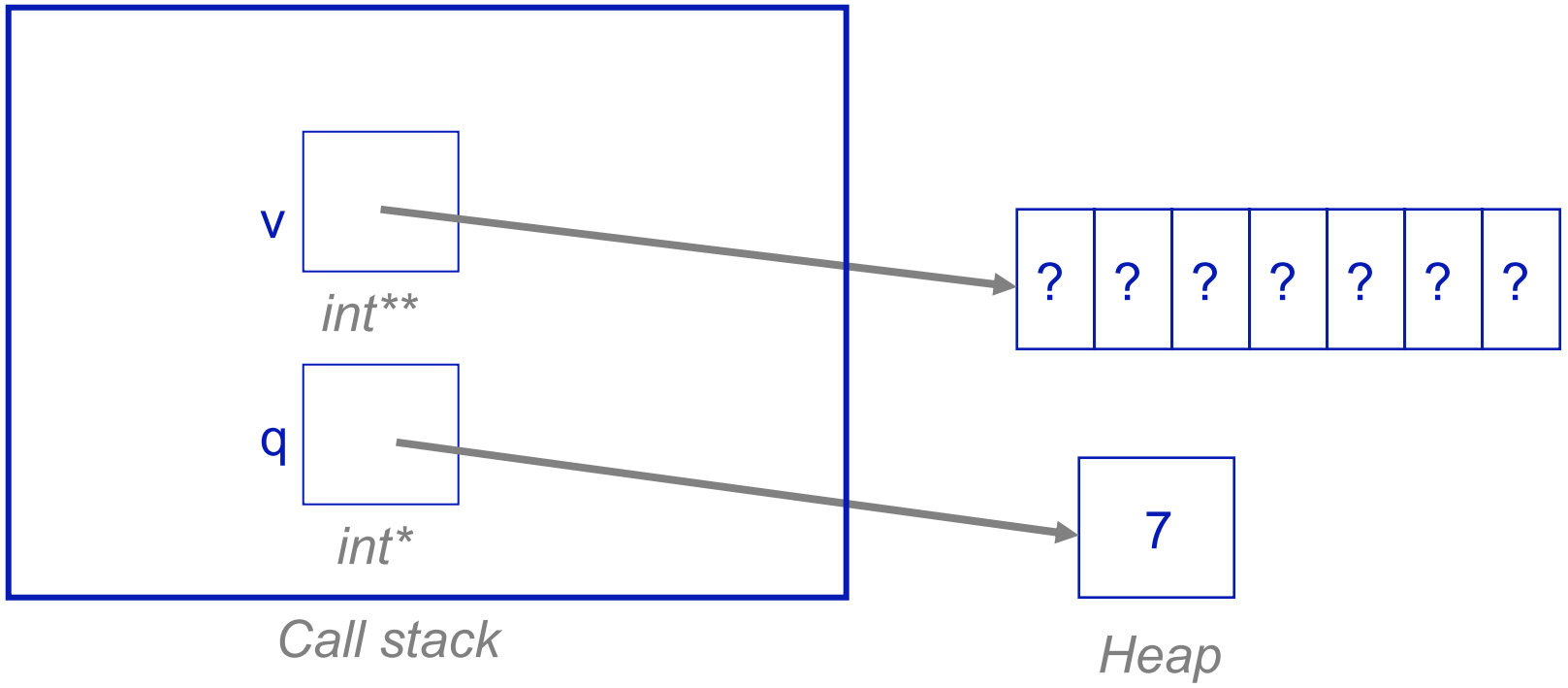

After execution of:

```
**j = k;
```




# Example (cont'd)

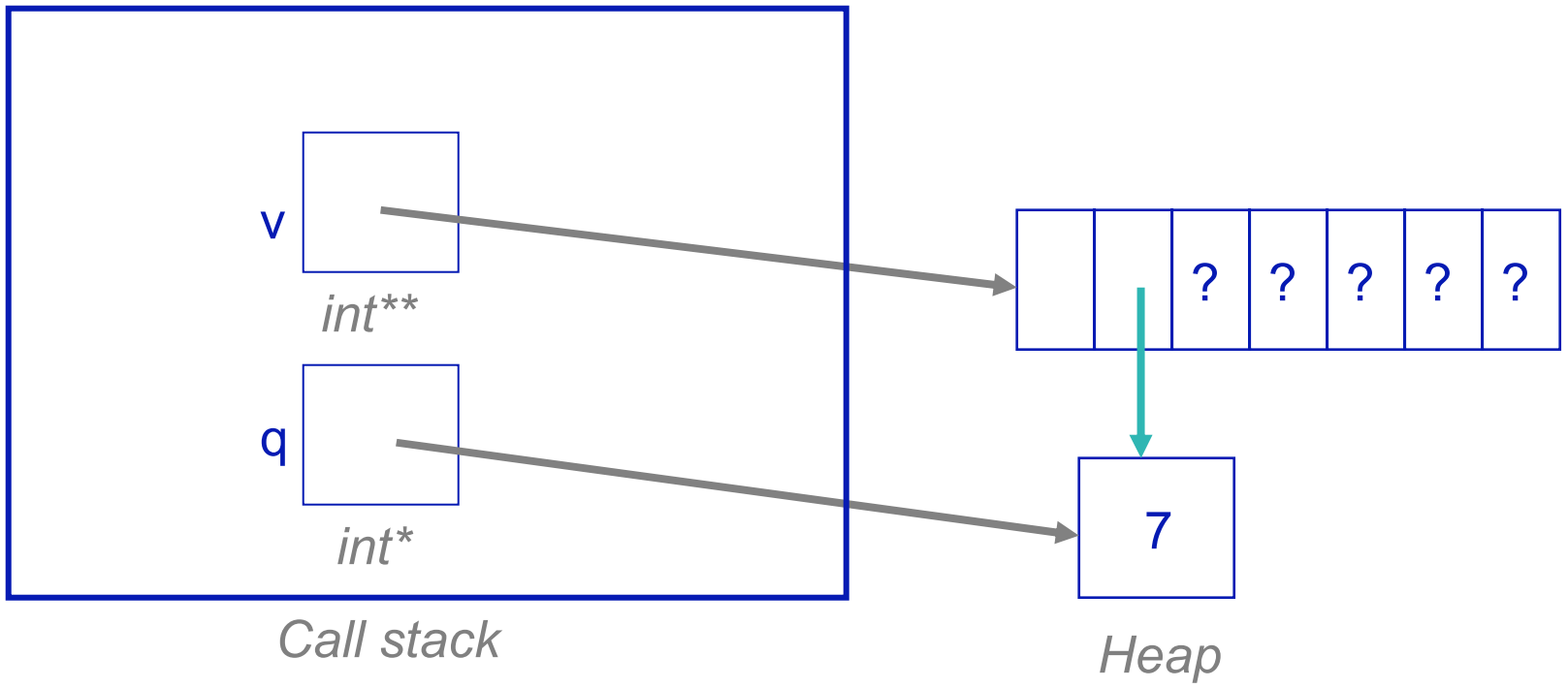
After returning to the main program, the call stack information for init2 has been recycled



# Example (cont'd)

And finally, after the execution of

```
v[1] = q;
```





hmehta.scs@dauniv.ac.in

# Using const with Parameters

- Use const with parameters that aren't pointers to indicate that the formal parameter will not be altered inside the function
- const is used with a pointer parameter to indicate that *the entity being referenced* will not be changed

# Using const with Parameters

```
void tryConst ( const int k, const int *p ) {
    k++;              /* compile time error generated */
    (*p) = 83;        /* compile time error generated – attempt to
                         change value from the calling module
                         that p points at */
    p = (int*)(malloc(sizeof(int)));
}
```

No warning from the final line: the value from the calling module that's being referenced is not modified; however, *the function's copy of the value's address* changes



# Pointers as Function Return Values

School of Computer Science

hmehta.scs@dauniv.ac.in

# Pointers as Return Values

- Functions can return pointers
- We can sometimes use this fact to reduce the complexity surrounding dereferencing operations

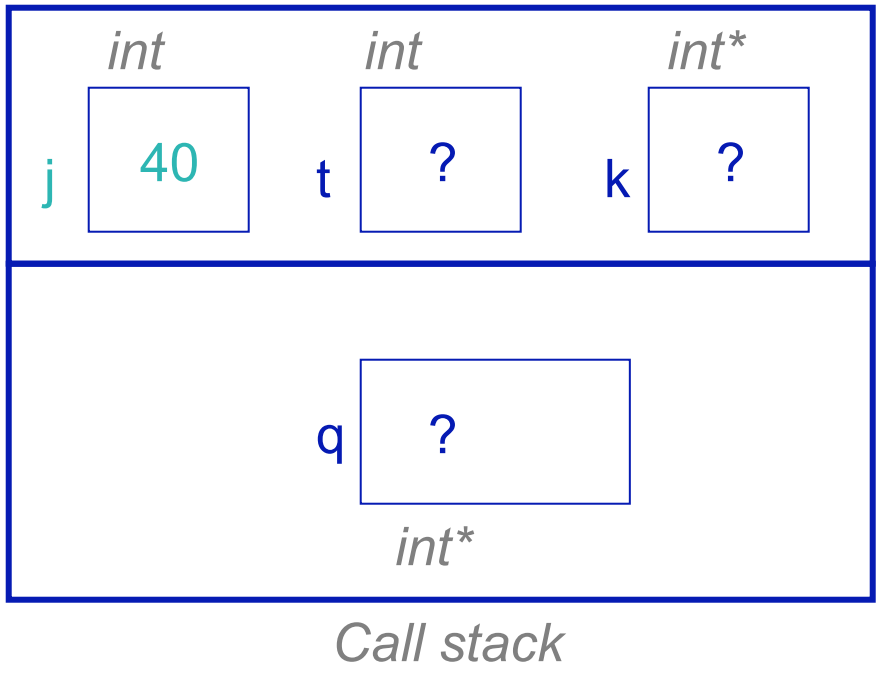
```
#include <stdlib.h>
int * init3( int j ) {
    int *k, t;
    k = (int*)malloc(j*sizeof(int));
    for (t=0; t<j; t++)
        k[t] = t;
    return k;
}
int main() {
    int *q;
    q = init3( 40 );
    …
```

## Example

The function init3 dynamically allocates array of int, initializes it, and returns a reference to it; in this case, the array has the capacity to hold 40 integers

# Example (cont'd)

*After* the call to init3, but *before* body is executed:




***Function prototype:***

```
int * init3( int j );
```

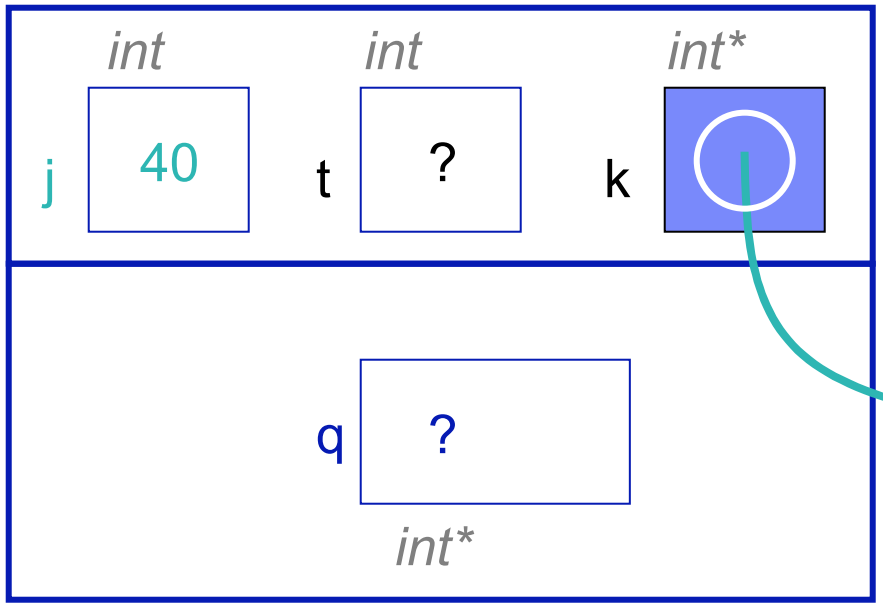
***Function call:***

```
q = init3( 40 );
```

# Example (cont'd)

After execution of:

```
k = (int*)malloc(j*sizeof(int));
for (t=0; t<j; t++)
    k[t] = t;
```




Call stack


Value to be returned is in k

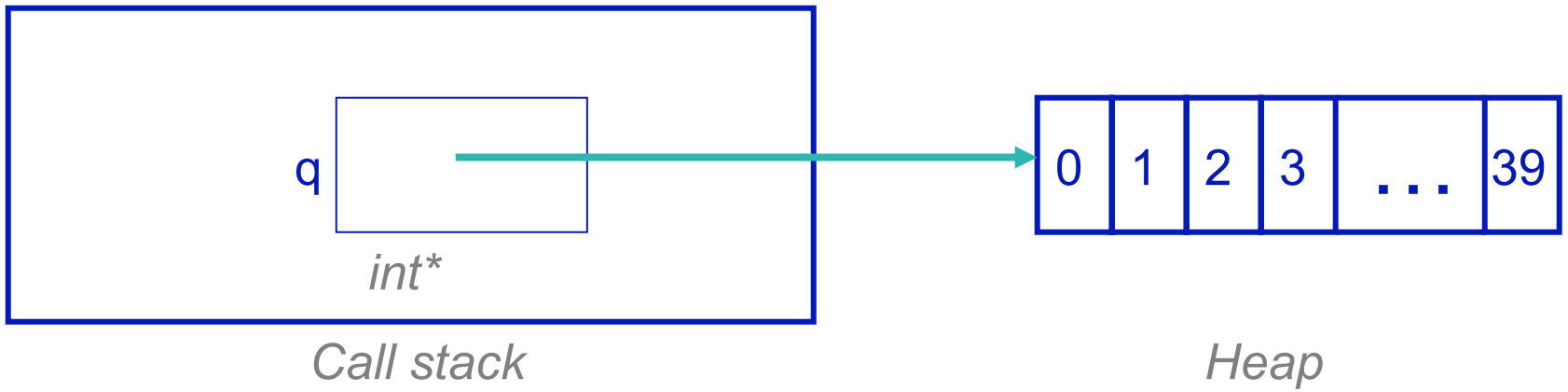
Heap


hmehta.scs@dauniv.ac.in

## Example (cont'd)

After returning from init3, and after execution of the assignment statement:

```
q = init3( 40 );
```




Portion of the call stack for the call to init3 has been recycled

# What ***Not*** to Do with Pointers

- Never let a function return a reference to a local variable within the function
- Never set a pointer parameter to the address of a local variable
- ***Reason***: The temporary space in which local variables are stored is recycled (reused) after the returning from the function, and the value being referenced may be overwritten at the next function call

```
#include <stdlib.h>
#include <time.h>
int ** init4( int ** size ) {
    int k, *arr;
    k = (int) 5 + rand( )%10;
    arr = (int*)malloc(k*sizeof(int));
    *size = &k;   // Wrong
    return &arr;  // Wrong
}
int main() {
    int ** vals, *numvals;
    srand((unsigned int) time(NULL));
     vals = init4( &numvals );
    …
```

## Example

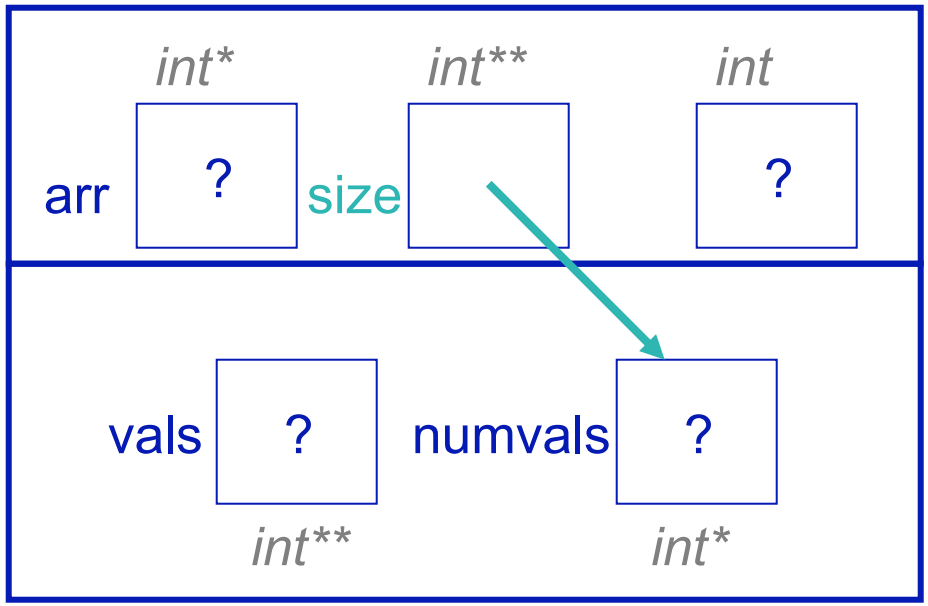
Local variables k and arr exist only while init4 is being executed

Values stored at those locations will likely be overwritten the next time any function is called

Values of vals and numvals (in main) would therefore be destroyed

# Example (cont'd)

*After* the call to init4, but *before* body is executed




Call stack

***Function prototype:***

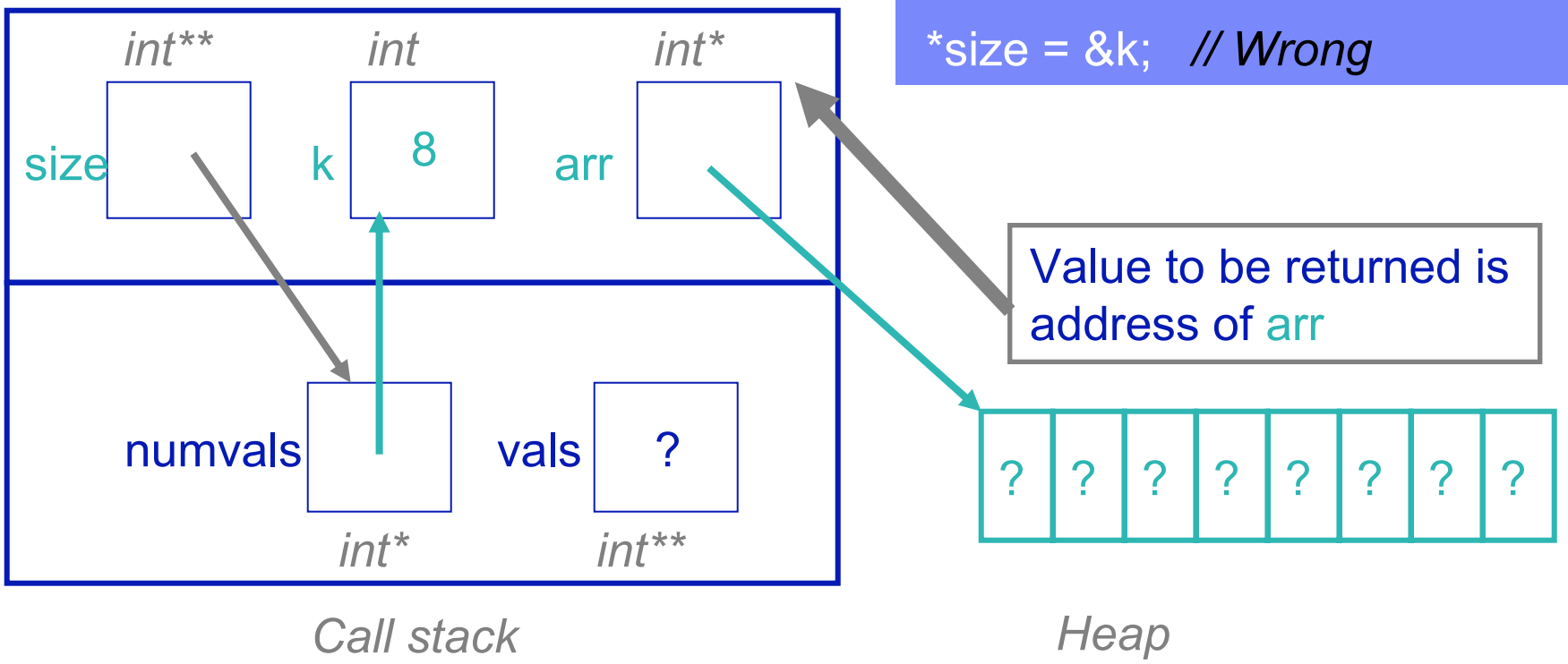
```
int ** init4( int ** size );
```

***Function call:***

```
vals = init4( &numvals );
```

# Example (cont'd)

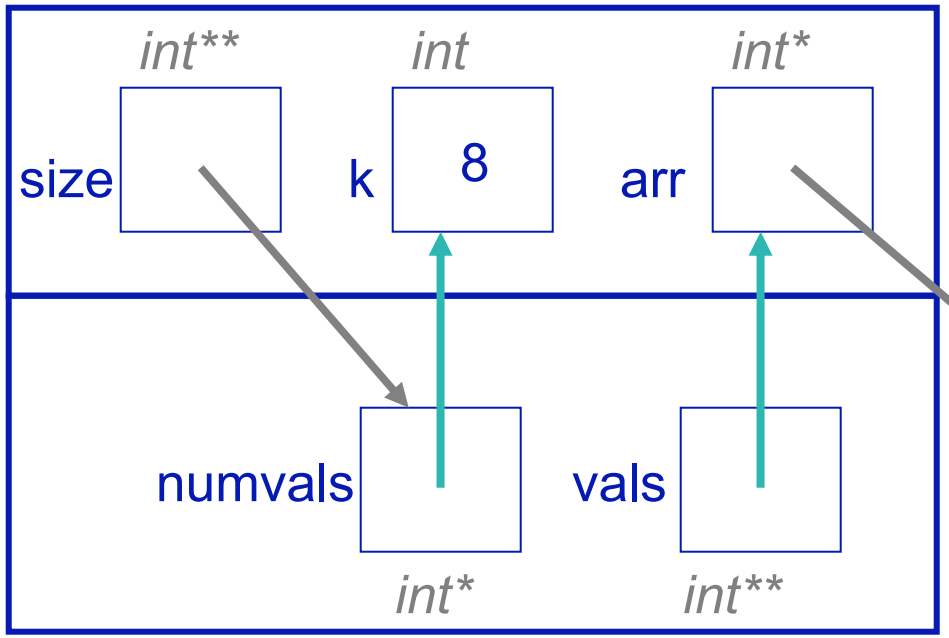
```
k = 5 + rand( )%10;
// pretend k is now 8
arr = (int*)malloc(k*sizeof(int));
*size = &k;   // Wrong
```



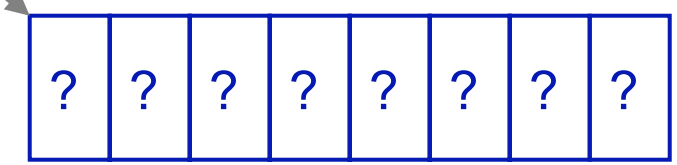
## Example (cont'd)

Here's ***what would happen*** after return, on completion of the assignment statement

```
vals = init4( &numvals );
```




Value to be returned is address of arr

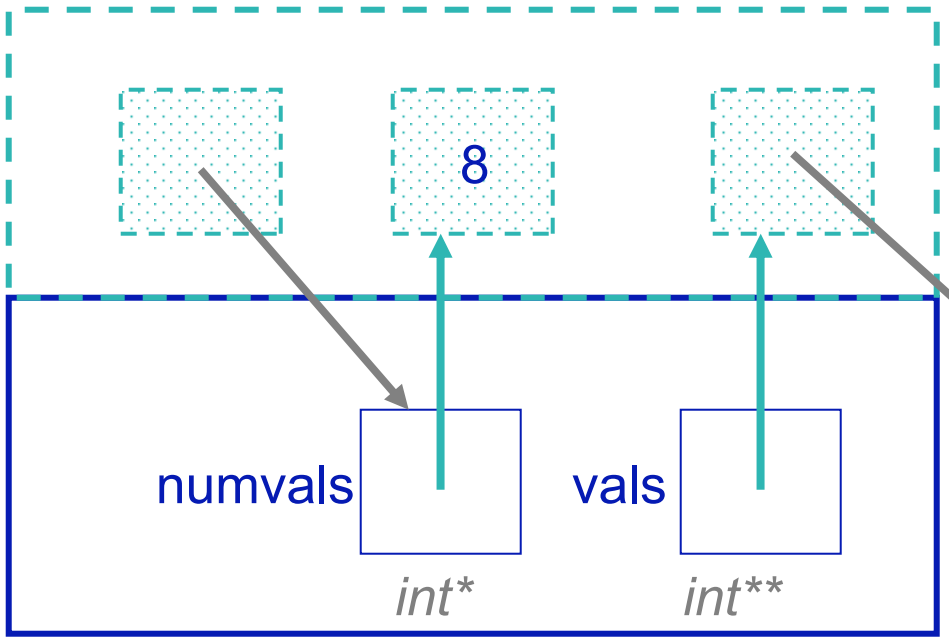


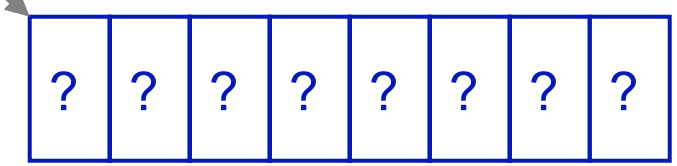

*Call stack*

*Heap*


hmehta.scs@dauniv.ac.in

## Example (cont'd)




*Call stack* *Heap*

***But*** the portion of the call stack for init4 is recycled at this stage, so vals and numvals point at values that are ***unstable*** (*i.e.:* likely to change at any moment); we'd lose the value 8 and the array's address



# Pointer Arithmetic

School of Computer Science

hmehta.scs@dauniv.ac.in

# Pointer Arithmetic

- Pointers are really numeric memory addresses
- C allows programmers to perform arithmetic on pointers
- Most commonly used with arrays and strings, instead of indexing, but can be used with any pointer
- We won't use pointer arithmetic much, but you may need to understand it to read text books and other people's code

# Pointer Arithmetic

- When used with an int array a:
  - a is a pointer to int, and points to a[0]
  - a+1 points to array element a[1]
  - a+2 points to array element a[2] , etc
  - *(a+4) = 10; is equivalent to a[4] = 10;
- Can compare pointers using ==, !=, >, <=, etc

# Pointer Arithmetic

- More examples:

```
int a[10], *p, *q;
p = &a[2];
q = p + 3;          /* q points to a[5] now */
p = q – 1;          /* p points to a[4] now */
p++;                /* p points to a[5] now */
p--;                /* p points to a[4] now */
*p = 123;           /* a[4] = 123 */
*q = *p;            /* a[5] = a[4] */
q = p;              /* q points to a[4] now */
```

# Pointer Arithmetic

- The difference between two pointers of the same type yields an int result

```
int a[10], *p, *q , i;
p = &a[2];
q = &a[5];
i = q - p;        /* i is 3 */
i = p - q;        /* i is –3 */
a[2] = 8; a[5] = 2;
i = *p - *q;      /* i = a[2] – a[5] */
```

# Pointer Arithmetic

- Note that pointer arithmetic and int arithmetic are ***not***, in general, the same
- In our previous examples: on most computers, an int requires 4 bytes (In Turbo C it is 2 Bytes) of storage
- Adding 1 to a pointer to int actually increments the address by 4, so that it points at the next memory location beyond the current int
- Casting a pointer to the wrong type leads to pointer arithmetic errors

# Pointer Arithmetic

```
int a[10], *p, *q , i;
char s[25], *u, *v, k;
p = &a[2]; q = &a[5];
i = q - p;        /* i is 3, but the difference between the two
                    addresses is 12: space for 3 ints */
q++;              /* address in q actually goes up by 4 bytes */
u = &s[6]; v = &s[12];
i = v – u;        /* i is 6, and the difference between the two
                    addresses is 6, because a char requires
                    only 1 byte of storage */
u++;              /* u = &s[7]; address in u goes up 1 byte */
```

## Example

Write a function myStrLen that is equivalent to strlen from <string.h>

```
size_t myStrLen( const char * s ) {
    size_t count = 0;
    while ( *s != '\0' ) {
        count++;
        s++;
    }
    return count;
}
```

We can change the pointer s, but not the string it points at

# Strings in C

- No explicit string type in C; strings are simply arrays of characters that are subject to a few special conventions
- *Example*: char s [10]; declares a 10-element array that can hold a character string of up to 9 characters
- ***Convention***: Use the ***null character '\0'*** to terminate all strings

# Strings in C

- C does not know where an array ends at run time – ***no boundary checking***
- All C library functions that use strings depend on the null character being stored so that the end of the string can be detected
- *Example*: `char str [10] = {'u', 'n', 'i', 'x', '\0'};`
  - Length of `str` is `4` (not 5, and not the declared size of the array)

# Accessing Individual Characters

- Use indexing, just as for any other kind of array:

```
char s[10];
s[0] = 'h';
s[1] = 'i';
s[2] = '!';
s[3] = '\0';
```

- Use single quotes with char literals, and double quotes with string literals

## String Literals

- *Example*: printf("Long long ago.");
- Other ways to initialize a string:

```
char s[10]="unix";  /* s[4] is automatically set to '\0';
                       s can hold up to ten chars,
                       including '\0' */
char s[ ]="unix";   /* s has five elements: enough to
                       hold the 4-character string plus
                       the null character */
```

# Printing Strings with printf ( )

```
char str[ ] = "A message to display";
printf ("%s\n", str);
```

- printf expects to receive a string parameter when it sees %s in the format string
  - Can be from a character array
  - Can be another literal string
  - Can be from a character pointer (more on this later)

# Printing Strings with printf ( )

- When printf encounters the format specifier %s, it prints characters from the corresponding string, up until the null character is reached
- The null character itself is not printed
- If no null character is encountered, printing will continue beyond the array boundary until either one is found, or a memory access violation occurs

## Example

```
char str[11]="unix and c";

 printf("%s\n", str);
 str[6]='\0';
 printf("%s\n", str);
 printf(str); printf("\n");
 str[2]='%';
 str[3]= ‘s';
 printf(str,str);
 printf("\n");
```

What output does this code segment produce?

# Printing with puts( )

- The ***puts*** function is much simpler than printf for printing strings
- Prototype of puts is defined in stdio.h:

```
int puts(const char * str);
```

- More efficient than printf: the program doesn't need to analyze the format string at run-time

Use of **const** in this context indicates that the function will not modify the string parameter

## Printing with puts( )

- *Example*:

```
char sentence[ ] = "The quick brown fox\n";
puts(sentence);
```

- Prints out ***two*** lines:

```
The quick brown fox

```

← *this blank line is part of the output*

- puts adds a newline '\n' character following the string

# Inputting Strings with gets( )

- gets( ) gets a line from standard input
- The prototype is defined in stdio.h:
  char *gets(char *str);
  - str is a ***pointer*** to the space where gets will store the line, or a character array
  - Returns ***NULL*** (the null pointer) upon failure. Otherwise, it returns str
  - Note the additional meaning for operator *

# Inputting Strings with gets( )

- *Example*:

```
char your_line[100];
printf("Enter a line:\n");
gets(your_line);
puts("Your input follows:\n");
puts(your_line);
```

  - Newline character is not stored, but the null character is
  - Make sure the array is big enough to hold the line being read; otherwise, input will overflow into other areas of memory

# Inputting Strings with scanf( )

- To read a string include:
  - %s scans up to but not including the next white space character
  - %***n***s scans the next ***n*** characters ***or*** up to the next white space character, whichever comes first
- *Example*:

```
/* Assume s1, s2 and s3 are char arrays */
scanf ("%s%s%s", s1, s2, s3);
scanf ("%5s%5s%5s", s1, s2, s3);
```

# Inputting Strings with scanf( )

- ***Note***: No address operator (&) is used with the actual parameter when inputting strings into character arrays: the array name is an address already
- Difference between gets( ) and scanf( ):
  - gets( ) read an entire line
  - scanf("%s",…) reads only up to the next whitespace character

## Example

```
#include <stdio.h>
int main ( )
{
    char lname[81], fname[81];
    int  count, age;
    puts ("Enter the last name, first name, and age, separated");
    puts ("by spaces; then press Enter \n");
    count = scanf ("%s%s%d", lname, fname, &age);
    printf ("%d items entered: %s %s %d\n",
                    count, fname, lname, age);
    return 0;
}
```

# The C String Library

- String functions are provided in an ANSI standard string library
- Access this through its header file:

```
#include <string.h>
```

- Includes functions to perform tasks such as:
  - Computing length of string
  - Copying strings
  - Concatenating strings
  - Searching for a sub string or characters
- This library is guaranteed to be there in any ANSI standard implementation of C

# Functions from string.h

- ***strlen*** returns the length of a NULL terminated character string:

  size_t  strlen (const char * str) ;

- size_t: a type defined in string.h that is equivalent to an unsigned int
- char  *str: points to a series of characters or is a character array ending with '\0'
  - What's wrong with:

    char a[5]={'a', 'b', 'c', 'd', 'e'};  strlen(a);

# Functions from string.h

- ***strcpy*** makes a copy of a string:

  char *strcpy (char * destination, const char * source);

- A copy of the string at address source is made at destination
  - String at source should be null-terminated
  - destination should point to enough room
    (at least enough to hold the string at source)
- The return value is the address of the copied string (that is, destination)

# Functions from string.h

- ***strcat*** concatenates strings:

  char * strcat (char * str1, const char * str2);

  - Appends a copy of str2 to the end of str1
  - The result string is null-terminated
  - str2 is not modified
  - A pointer equal to str1 is returned

- Programmer must ensure that str1 has sufficient space to hold the concatenated string

## Example

```
#include <string.h>
#include <stdio.h>
int main( )
{
    char str1[27] = "abc";
    char str2[100];
    printf("%d\n",strlen(str1));
    strcpy(str2,str1);
    puts(str2);
    puts("\n");
    strcat(str2,str1);
    puts(str2);
}
```

Show the output

## Functions from string.h

- ***strcmp*** compares strings for equality or inequality:

  int strcmp (const char *str1, const char *str2);

- Returns an int whose value is interpreted as follows:

  < 0 : str1 is less than str2

  0 : str1 is equal to str2

  > 0 : str1 is greater than str2

## Functions from string.h

- strcmp compares the strings one char at a time until a difference is found; return value is (the ASCII ordinal value of the char from str1) minus (the ASCII ordinal value of the char from str2)
- If both strings reach a '\0' at the same time, they are considered equal, and return value is zero

# Functions from string.h

- Other string comparison functions:

  int ***strncmp*** (const char *str1,
  const char * str2, size_t n);

  - Compares at most n chars of str1 and str2
  - Continues until a difference is found in the first n chars, or n chars have been compared without detecting a difference, or the end of str1 or str2 is encountered

## Functions from string.h

- ***strcasecmp( )*** and ***strncasecmp( )***: same as strcmp( ) and strncmp( ), except that differences between upper and lower case letters are ignored

## Functions from string.h

Show the output

```
#include <string.h>
int main()
{
   char str1[ ] = "The first string.";
   char str2[ ] = "The second string.";
   printf("%d\n", strncmp(str1, str2, 4) );
   printf("%d\n", strncmp(str1, str2, 7) );
}
```

## Functions from string.h

- ***strchr***: Find the first occurrence of a specified character in a string:

  char * strchr (const char * str, int ch) ;

  - Search the string referenced by str from its beginning, until either an occurrence of ch is found or the end of the string ('\0') is reached
  - Return a pointer to the first occurrence of ch in the string; if no occurrence is found, return the NULL pointer instead

## Functions from string.h

- Value returned by strchr can be used to determine the position (index) of the character in the string:
  - Subtract the start address of the string from the value returned by strchr
  - This is ***pointer arithmetic***: the result is offset of the character, in terms of char locations, from the beginning of the string
  - Will work even if sizeof(char) is not 1

## Functions from string.h

```
#include<stdio.h>
#include<string.h>
int main( )
{
    char ch='b', buf[80];
    strcpy(buf, "The quick brown fox");
    if (strchr(buf,ch) == NULL)
        printf ("The character %c was not found.\n",ch);
   else
        printf ("The character %c was found at position %d\n",
                ch, strchr(buf,ch)-buf+1);
}
```

'b' is the 11th character in buf  in fact, it is stored in buf[10]

## Functions from string.h

- ***strstr*** searches for the first occurrence of one string inside another:

```
char * strstr (const char * str,
              char * query) ;
```

  - If found, a pointer to the first occurrence of query inside str is returned; otherwise the NULL pointer is returned

# Functions from stdio.h

- ***sprintf*** behaves like printf, except that, instead of sending a formatted stream of characters to standard output, it stores the characters in a string

  int sprintf( char *s, const char *format, …);
- Result is stored in the string referenced by s
- Useful in formatting a string, and in converting int or float values to strings
- ***sscanf*** works like scanf, but takes its input from a string

# Functions from string.h and stdio.h

```
#include <stdio.h>
#include <string.h>
int main( )
{
        char result[100];
        sprintf(result, "%f", (float)17/37 );
        if (strstr(result, "45") != NULL)
          printf("The digit sequence 45 is in 17 divided by 37. \n");
        return 0;
}
```

# Functions from stdlib.h

- ***atoi***: takes a character string and converts it to an integer

  int atoi (const char *ptr);

  - Ignores leading white space in the string
  - Then expects either + or – or a digit
  - No white space allowed between the sign and the first digit
  - Converts digit by digit until a non-digit (or the end of the string) is encountered

# Functions from stdlib.h

- *Examples* using atoi :

| string s | atoi( s ) |
|---|---|
| "394" | 394 |
| "157  66" | 157 |
| "-1.6" | -1 |
| "  +50x" | 50 |
| "twelve" | 0 |
| "x506" | 0 |
| "  -  409" | 0 |

# Functions from stdlib.h

long atol (const char *ptr) ;

- like atoi, but returns a long

double atof (const char * str);

- Like atoi, but for real values
- Handles a decimal point, and an exponent indicator (e or E) followed by an integer exponent
- 

| string s | atof( s ) |
|---|---|
| “12-6” | 12.000000 |
| “ -0.123.456” | -0.123000 |
| “123E+3” | 123000.000000 |
| “123.1e-5” | 0.001231 |



# Variable Number of Arguments/Parameters

School of Computer Science

hmehta.scs@dauniv.ac.in

# Variable-Length Argument Lists

- Functions with unspecified number of arguments
  - Load `<stdarg.h>`
  - Use ellipsis(`...`) at end of parameter list
  - Need at least one defined parameter
  - Example:

```
double myfunction ( int i, ... );
```

  - The ellipsis is only used in the prototype of a function with a variable length argument list
  - `printf` is an example of a function that can take multiple arguments
  - The prototype of `printf` is defined as

```
int printf( const char* format, ... );
```


hmehta.scs@dauniv.ac.in

# Variable-Length Argument Lists

- Macros and definitions of the variable arguments header (**`stdarg.h`**)
  - **`va_list`**
    - Type specifier, required (va_list arguments;)
  - **`va_start`**`(` *arguments*, *other variables* `)`
    - Intializes parameters, required before use
  - **`va_arg`**`(` *arguments*, *type* `)`
    - Returns a parameter each time **`va_arg`** is called
    - Automatically points to next parameter
  - **`va_end`**`(` *arguments* `)`
    - Helps function have a normal return



# Command Line Arguments/Parameters

School of Computer Science

hmehta.scs@dauniv.ac.in

# Passing Parameters to C

- Often a user wants to pass parameters into the program from the command prompt

  :\tc\bin>progname arg1 arg2 arg3

- This is accomplished in C using argc and argv

Number of arguments, which includes the name of the executable

Array of strings of length argc with one argument per location

```
int main (int argc, char *argv[ ] ) {
  /*  Statements go here */
}
```


hmehta.scs@dauniv.ac.in

## Passing Parameters to C

```
#include <stdio.h>
int main(int argc, char *argv[ ])
{
     int count;
     printf ("Program name: %s\n", argv [0]);
     if (argc > 1)
     {
            for (count=1; count<argc; count++)
              printf ("Argument %d: %s\n",count,argv[count]);
      }
    else
            puts ("No command line arguments entered.");
     return 0;
}
```

# Passing Parameters to C

- Suppose we compiled the previous program to the executable file myargs

```
C:\tc\bin>myargs first "second arg" 3  4  > myargs.out
```

- myargs.out contains the following lines:

**Program name: myargs**
**Argument 1: first**
**Argument 2: second arg**
**Argument 3: 3**
**Argument 4: 4**

# Passing Parameters to C

- C program should check the command line arguments for validity:
  - Are there the right number of arguments?
  - Are numeric arguments in the expected range of acceptable values?
  - If an argument is an input or output file, can it be opened successfully?
  - Issue an error message and abort execution if arguments are incorrect



# Compiler Directives

School of Computer Science

hmehta.scs@dauniv.ac.in

# The C Preprocessor

- The C preprocessor is invoked by the compilation command *before* compiling begins
- Changes your source code based on instructions called ***preprocessor directives*** that are embedded in the source code
- The preprocessor creates a new version of your program and it is this new program that actually gets compiled

## The C Preprocessor

- Normally, you do not see these new versions on the hard disk, as they are deleted after compilation
- You can force the compiler to keep them to see the results
- Each preprocessor directive appears in the source code preceded by a '#' sign

# The #define Directive

- Simple substitution macros

  #define text1 text2

- This tells the compiler to find all occurrences of "text1" in the source code and substitute "text2"
- Usually used for constants:

  #define MAX 1000

  - Generally use upper case letters (by convention)
  - Always separate using white space
  - No trailing semi-colon

## The #define Directive

```
#include <stdio.h>
#define  PI  3.1416
#define  PRINT  printf
int main( )
{
   PRINT("Approximate value of pi: %f", PI);
   return 0;
}
```

# Function Macros

- You can also define ***function macros***:

```
#define MAX(A,B)  ( (a) > (b) ? (a) : (b) )
……
printf("%d", 2 * MAX(3+3, 7));  /* is equivalent to */
printf("%d", 2 * ( (3+3) > (7) ? (3+3) : (7) );
```

- The parentheses are ***important***:

```
#define MAX(A,B)  a>b?a:b
printf("%d", 2 * MAX(3+3, 7));  /*is equivalent to */
printf("%d", 2 * 3+3 > 7 ? 3+3 : 7 );
```

# Function Macros Should be Used with Care

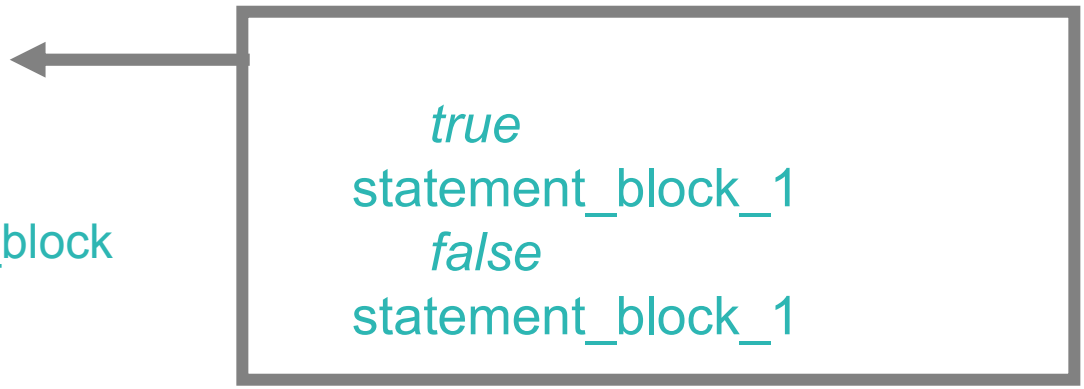
```
#define MAX(X,Y) ((x)>(y)?(x):(y))
……
int n, i=4, j=3;
n= MAX( i++, j);   /* Same as n= (( i++ )>( j )?( i++ ):( j )) */
printf("%d,%d,%d", n, i, j);
```

- The output is: **5, 6, 3**
- If MAX was a function, the output would have been: **4, 5, 3**

# Conditional Compilation

- The pre-processor directives #if, #elif, #else, and #endif tell the compiler if the enclosed source code should be compiled
- Structure:

```
#if condition_1
      statement_block_1
#elif condition_2
      statement_block_2
...
#elif condition_n
      statement_block_n
#else
      default_statement_block
#endif
```



# Conditional Compilation

- For the most part, the only things that can be tested are the things that have been defined by #define statements

```
#define ENGLAND 0
#define FRANCE    1
#define ITALY       0
#if   ENGLAND
      #include "england.h"
#elif  FRANCE
      #include "france.h"
#elif  ITALY
      #include "italy.h"
#else
      #include "canada.h"
#endif
```

# Conditional Compilation

- Conditional compilation can also be very useful for including ***debugging code***
  - When you are debugging your code you may wish to print out some information during the running of your program
  - You may not need want this extra output when you release your program; you'd need to go back through your code to delete the statements

# Conditional Compilation

- Instead, you can use #if … #endif to save time:

```
#define DEBUG 1
……
#if DEBUG
    printf("Debug reporting at function my_sort()!\n");
#endif
……
```

- Change DEBUG to zero and recompile to suppress the debugging output

# Conditional Compilation

- We can use a ***preprocessor function*** as the condition of compilation:

  defined ( NAME )

  - Returns true if NAME has been defined; else false

- *Example*:

```
#define DEBUG
#if defined ( DEBUG )
   printf("debug report at function my_sort() \n");
#endif
```

## Conditional Compilation

- ***Note***: Value of the defined function depends only on whether the name DEBUG has been defined
- It has nothing to do with which value (if any) DEBUG is defined to; in fact, we don't have to provide a value at all
- Can use the notation #ifdef NAME instead
- We also have #ifndef (if not defined)

## Conditional Compilation

- The #undef … directive makes sure that defined( …) evaluates to false.
- *Example*: Suppose that, for the first part of a source file, you want DEBUG to be defined, and for the last part, you want DEBUG to be undefined

## Conditional Compilation

- A directive can also be set on the Unix/DOS command line at compile time:

  tcc/cc –DDEBUG myprog.c

  - Compiles myprog.c with the symbol DEBUG defined as if #define DEBUG was in written at the top of myprog.c

# The #include Directive

- Causes all of the code in the included file to be inserted at the point in the text where #include appears
- Included files can contain other #include directives; usually limited to 10 levels of nesting
- **< >** tells the compiler to look in the standard include directories
- **" "** tells the compiler to first find it in current folder than in include directory

# The #include Directive

- In large programs, some .h files may be #included several times – could lead to multiple definition errors
- To avoid this problem, surround contents of the .h file with

```
#ifndef unique_identifier_name
# define unique_identifier_name
/* contents of .h file belong here */
#endif
```



# Structures and Union

School of Computer Science

hmehta.scs@dauniv.ac.in

# Structures in C

- Structures are used in C to group together related data into a composite variable. This technique has several advantages:
  - It clarifies the code by showing that the data defined in the structure are intimately related.
  - It simplifies passing the data to functions. Instead of passing multiple variables separately, they can be passed as a single unit.

# Structures

- ***Structure***: C's way of grouping a collection of data into a single manageable unit
- Defining a structure type:

```
struct coord {
        int x ;
        int y ;
};
```

  - This defines a new type struct coord; no variable is actually declared or generated

## Structures

- To define struct variables:

```
struct coord {
    int x,y ;
} first, second;
```

Defines the structured type struct coord and statically allocates space for two structures, first and second; the structures are ***not initialized***

- Another approach:

```
struct coord {
    int x,y ;
};

.............
struct coord first, second;
struct coord third;
```

Just defines the structured type

Statically allocated variables are declared here


hmehta.scs@dauniv.ac.in

# Structures

- You can use a typedef if you want to avoid two-word type names such as struct coord:

```
typedef struct coord coordinate;
coordinate first, second;
```

- In some compilers, and all C++ compilers, you can usually simply say just:

```
coord first, second;
```

## Structures

- Type definition can also be written as:

```
typedef struct coord {
    int x,y ;
} coordinate;
```

  - In general, it's best to separate the type definition from the declaration of variables

# Structures

- Access structure ***members*** by the dot (.) operator
- Generic form:

  structure_var.member_name
- For example:

```
coordinate pair1, pair2;
pair1.x = 50 ;
pair2.y = 100;
```

# Structures

- struct_var.member_name can be used anywhere a variable can be used:
  - printf ("%d , %d", second.x , second.y );
  - scanf("%d, %d", &first.x, &first.y);

# Structures

- Can *copy* entire structures using =

```
pair1 = pair2;
```

performs the same task as:

```
pair1.x = pair2.x ;
pair1.y = pair2.y ;
```

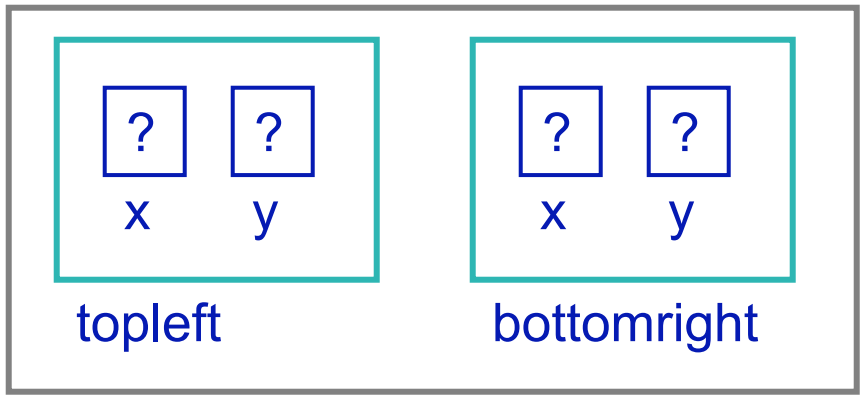
# Memory alignment/ Structure Padding

- Compiler inserts unused bytes into the structure to align the memory on a word boundary.
- Different C compilers may give different offsets to the elements.

## Structures Containing Structures

- Structures can have other structures as members:
- To define rectangle in terms of coordinate:

```
typedef struct rect {
   coordinate topleft;
   coordinate bottomright;
} rectangle;
```



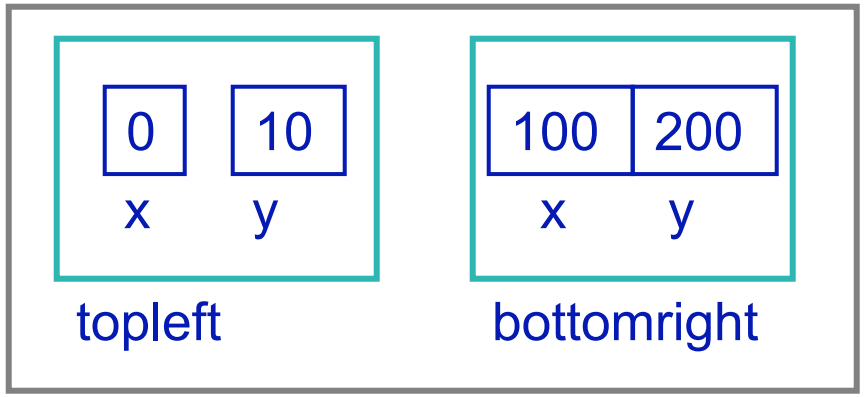

Rectangle blueprint


hmehta.scs@dauniv.ac.in

## Structures Containing Structures

- To initialize the points describing a rectangle:

```
struct rect mybox ;
mybox.topleft.x = 0 ;
mybox.topleft.y = 10 ;
mybox.bottomright.x = 100 ;
mybox.bottomright.y = 200 ;
```



## Example

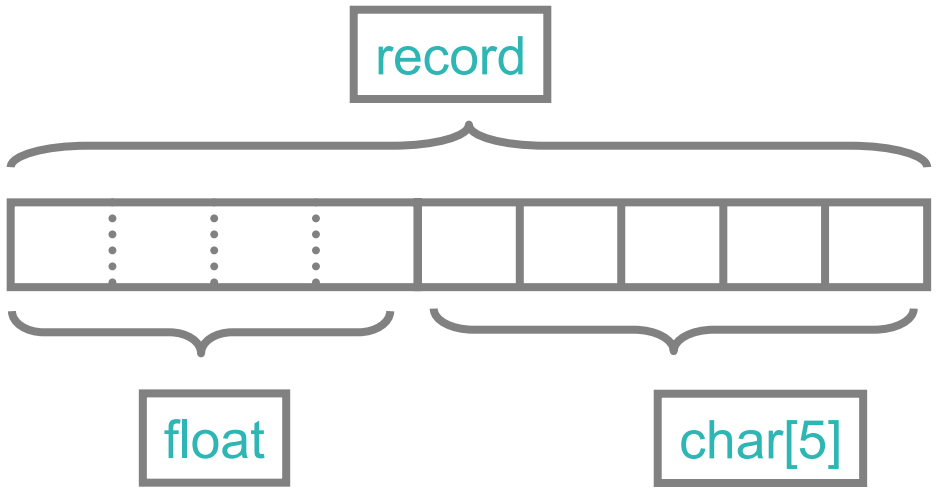
```
#include <stdio.h>
typedef struct coord {
    int x;
    int y;
}coordinate;
typedef struct rect {
    coordinate topleft;
    coordinate bottomright;
}rectangle;
```

```
int main ( )
{
    int length, width;
    long area;
    rectangle mybox;
    mybox.topleft.x = 0;
    mybox.topleft.y = 0;
    mybox.bottomright.x = 100;
    mybox.bottomright.y = 50;
    width = mybox.bottomright.x –
                mybox.topleft.x;
    length = mybox.bottomright.y –
                mybox.topleft.y;
    area = width * length;
    printf (“Area is %ld units.\n", area);
}
```

# Structures Containing Arrays

- Arrays within structures are the same as any other member element
- *Example*:

```
struct record {
        float  x;
        char  y [5] ;
} ;
```



## Example

```
#include <stdio.h>
struct data
{
    float amount;
    char  fname[30];
    char lname[30];
}  rec;
int main () {
      struct data rec;
      printf ("Enter the donor's first and last names, \n");
      printf ("separated by a space: ");
      scanf  ("%s %s", rec.fname, rec.lname);
      printf ("\nEnter the donation amount: ");
      scanf  ("%f", &rec.amount);
      printf ("\nDonor %s %s gave $%.2f.\n",
                  rec.fname,rec.lname,rec.amount);
}
```

## Arrays of Structures

- *Example*:

```
struct  entry {
   char  fname [10] ;
   char  lname [12] ;
   char phone [8] ;
} ;
struct entry list [1000];
```

This creates an array of 1000 structures of type struct entry

- Possible assignments:

```
list [1] = list [6];
strcpy (list[1].phone, list[6].phone);
list[6].phone[1] = list[3].phone[4] ;
```

## Example

```
#include <stdio.h>
struct entry {
    char fname [20];
    char lname [20];
    char phone [10];
} ;
```

```
int main() {
   struct entry list[4];
   int i;
   for (i=0; i < 4; i++) {
       printf ("\nEnter first name: ");
       scanf  ("%s", list[i].fname);
       printf ("Enter last name: ");
       scanf  ("%s", list[i].lname);
       printf ("Enter phone in 123-4567 format: ");
       scanf  ("%s", list[i].phone);
   }
   printf ("\n\n");
   for (i=0; i < 4; i++) {
       printf ("Name: %s %s", list[i].fname,
      list[i].lname);
       printf ("\t\tPhone: %s\n", list[i].phone);
   }
}
```

## Initializing Structures

- *Example:*

```
struct sale {
   char  customer [20] ;
   char  item [20] ;
   int amount ;
};
struct sale mysale = { "Acme Industries",
                       "Zorgle blaster",
                       1000 } ;
```

# Initializing Structures

- *Example*: Structures within structures:

```
struct  customer {
   char firm [20] ;
   char contact [25] ;
};
struct sale {
   struct customer buyer ;
   char item [20] ;
   int amount ;
} mysale =
{ { "Acme Industries", "George Adams"} ,
    "Zorgle Blaster",  1000
} ;
```

## Initializing Structures

- *Example*

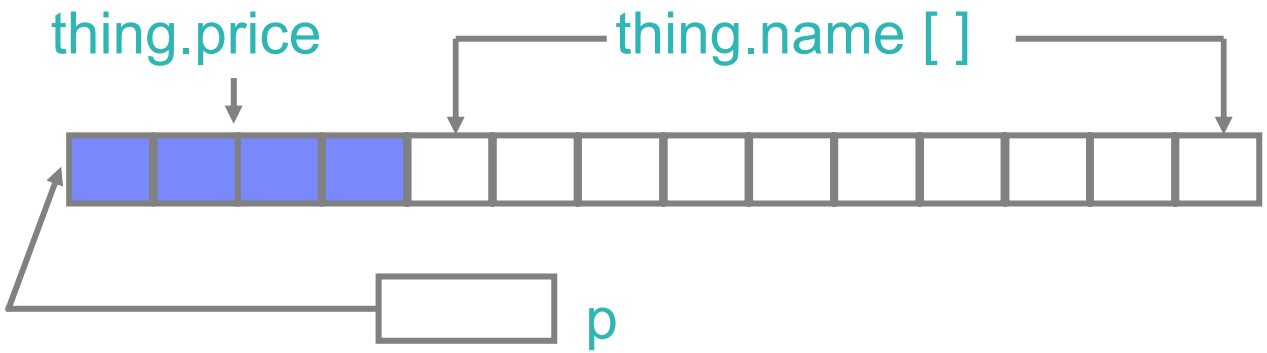
```
struct customer {
    char firm [20] ;
    char contact [25] ;
} ;
struct sale {
    struct customer buyer ;
    char item [20] ;
    int amount ;
} ;
```

```
struct saley1990 [100] = {
    { { "Acme Industries",
    "George Adams"} ,
   “Widget" , 1000
    },
    { { "Wilson & Co.",
        "Ed Wilson"} ,
    "Thingamabob" , 290
    }
} ;
```

## Pointers to Structures

```
struct part {
    float price ;
    char name [10] ;
} ;
struct part *p , thing;
p = &thing;
/* The following two statements are equivalent */
thing.price = 50;
(*p).price = 50;   /* ( ) around *p is needed */
```

# Pointers to Structures




- p is set to point to the first byte of the struct variable

## Pointers to Structures

- When we have a pointer to a structure, we must dereference the pointer ***before*** attempting to access the structure's members
- The membership (dot) operator has a higher precedence than the dereferencing operator

```
struct part *p , thing;
p = &thing;
(*p).price = 50;
```

Parentheses around *p are necessary

## Pointers to Structures

- C provides an operator **->** that combines the dereferencing and membership operations into one, performed in the proper order
- Easier form to read (and to type) when compared with the two separate operators

```
struct part *p , thing;
p = &thing;
p->price = 50; /*equivalent to  (*p).price = 50;  and
                    thing.price = 50;  and  (&thing)->price = 50;*/
```

## Pointers to Structures

```
struct part * p, *q;
p = (struct part *) malloc( sizeof(struct part) );
q = (struct part *) malloc( sizeof(struct part) );
p -> price = 199.99 ;
strcpy( p -> name, "hard disk" );
(*q) = (*p);
q = p;
free(p);
free(q); /* This statement causes a problem.
                Why? */
```

## Pointers to Structures

- You can allocate a structure array as well:

```
    struct part *ptr;
    ptr = (struct part *) malloc(10*sizeof(struct part) );
    for( i=0; i< 10; i++)
    {
          ptr[ i ].price = 10.0 * i;
          sprintf( ptr[ i ].name, "part %d", i );
    }
    ……
    free(ptr);
}
```

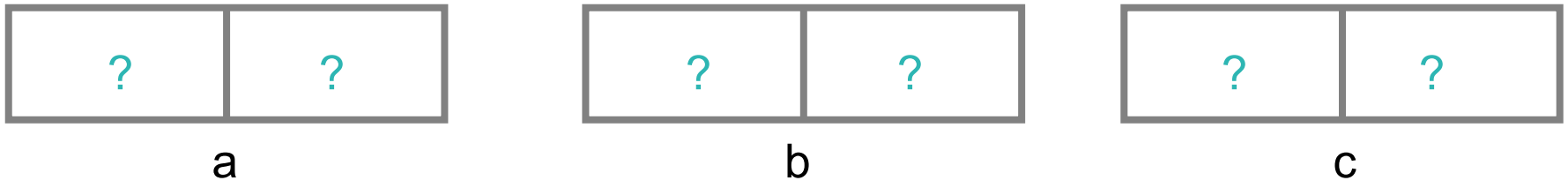
## Pointers to Structures

- You can use pointer arithmetic to access the elements of the array:

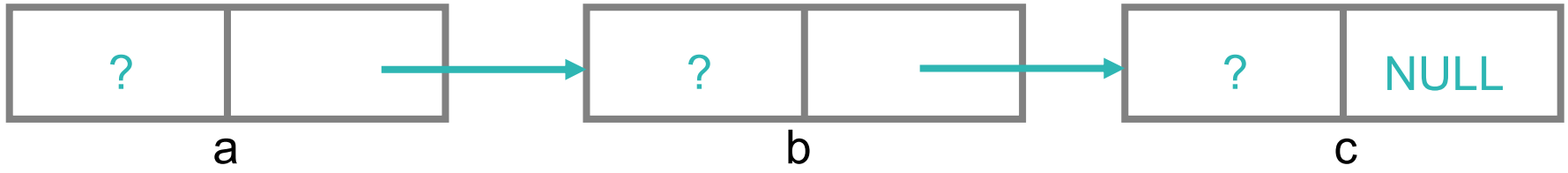
```
{
    struct part *ptr,  *p;
    ptr = (struct part *) malloc(10 * sizeof(struct part) );
    for( i=0, p=ptr; i< 10; i++, p++)
    {
            p -> price = 10.0 * i;
            sprintf( p -> name, "part %d", i );
    }
    ……
    free(ptr);
}
```

# Pointer as Structure Member

```
struct node{
    int data;
    struct node *next;
} a,b,c;
```

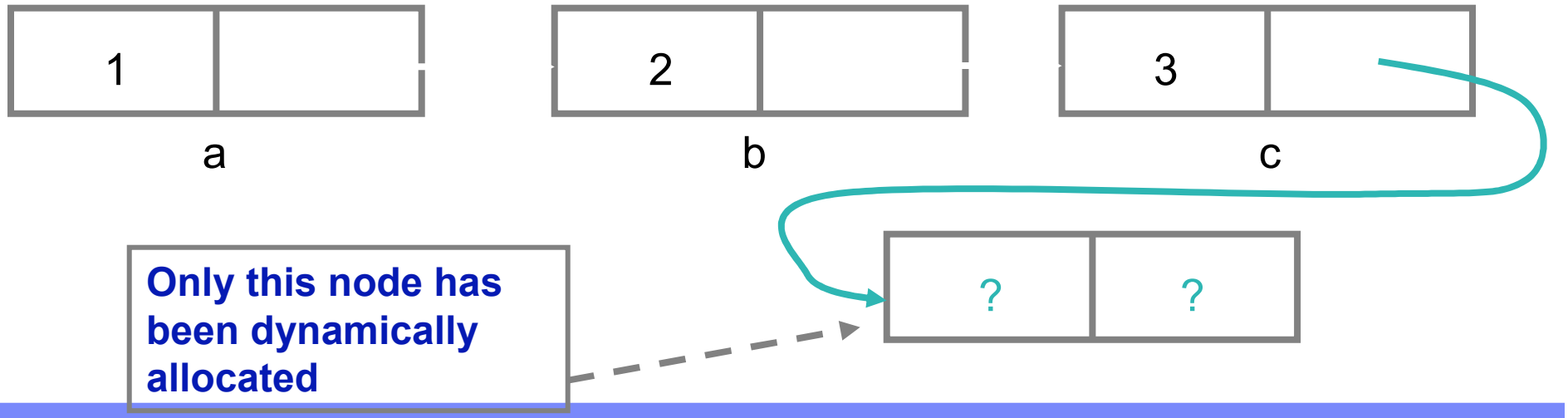



```
struct node a,b,c;
a.next = &b;
b.next = &c;
c.next = NULL;
```



# Pointer as Structure Member

```
a.data = 1;
a.next->data = 2;
/* or  b.data = 2;  or  (*(a.next)).data = 2 */
a.next->next->data = 3;
/* or  c.data = 3; or  (*(a.next)).next->data = 3;
    or  (* (*(a.next)).next).data = 3;  or  (*(a.next->next)).data = 3; */
c.next = (struct node *) malloc(sizeof(struct node));
```



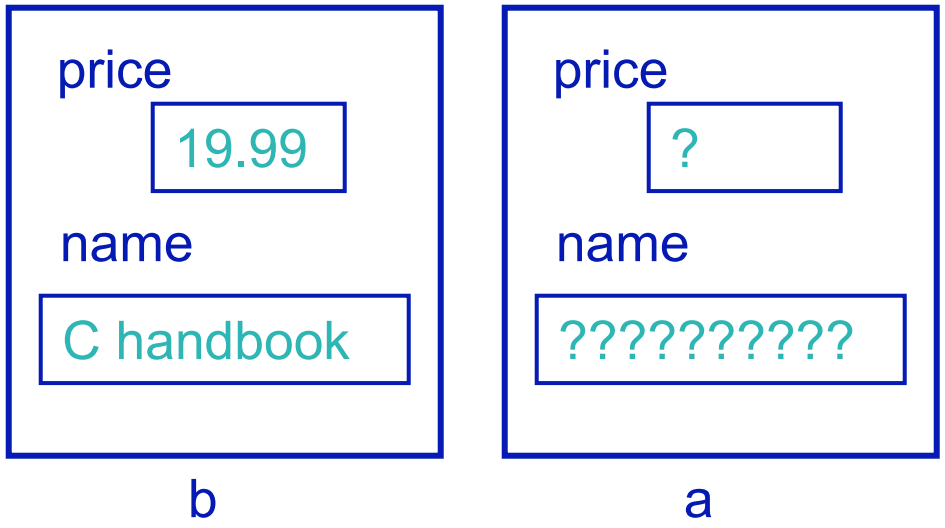
# Assignment Operator vs. *memcpy*

```
/* This copies one struct into
    another */
#include<string.h>
{
    struct part a,b;
    b.price = 39.99;
    strcpy(b.name,"floppy“);
    a = b;
}
```

```
/* Equivalently, you can use memcpy */
#include <string.h>
{
    struct part a,b;
    b.price = 39.99;
    strcpy(b.name,"floppy“);
    memcpy(&a,&b,sizeof(part));
}
```

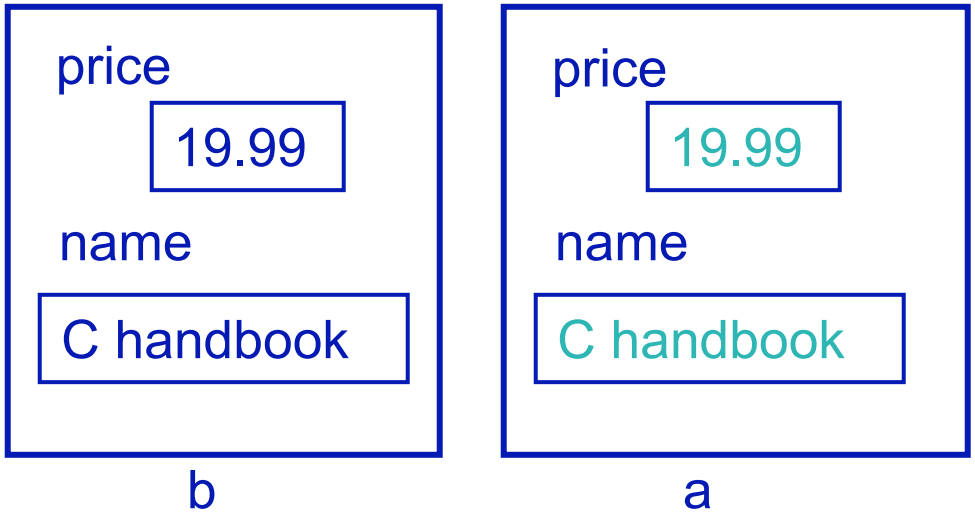
# Array Member vs. Pointer Member

```
struct book {
    float price;
    char name[50];
};
int main( )
{
    struct book a,b;
    b.price = 19.99;
    strcpy(b.name,
           "C handbook");
 /* continued… */
```

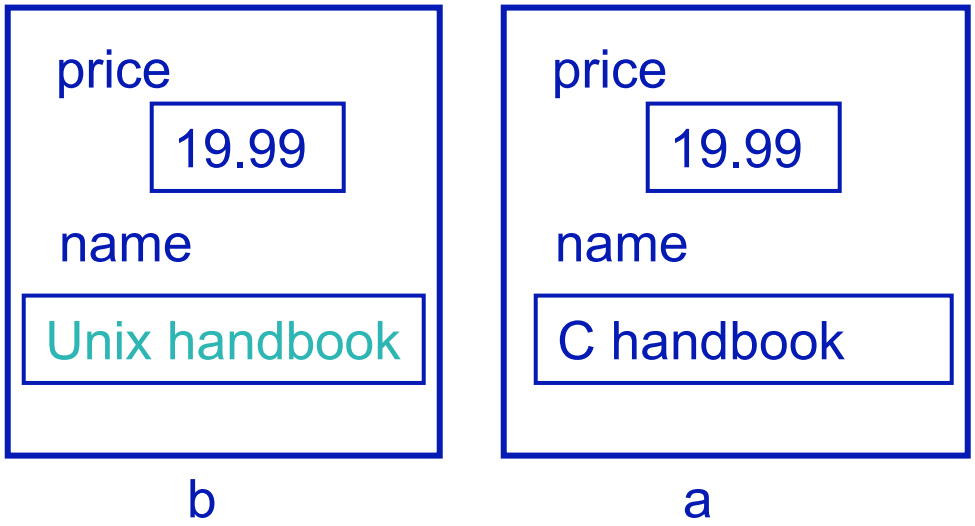


# Array Member vs. Pointer Member

```
/* …continued… */
 a = b;
```

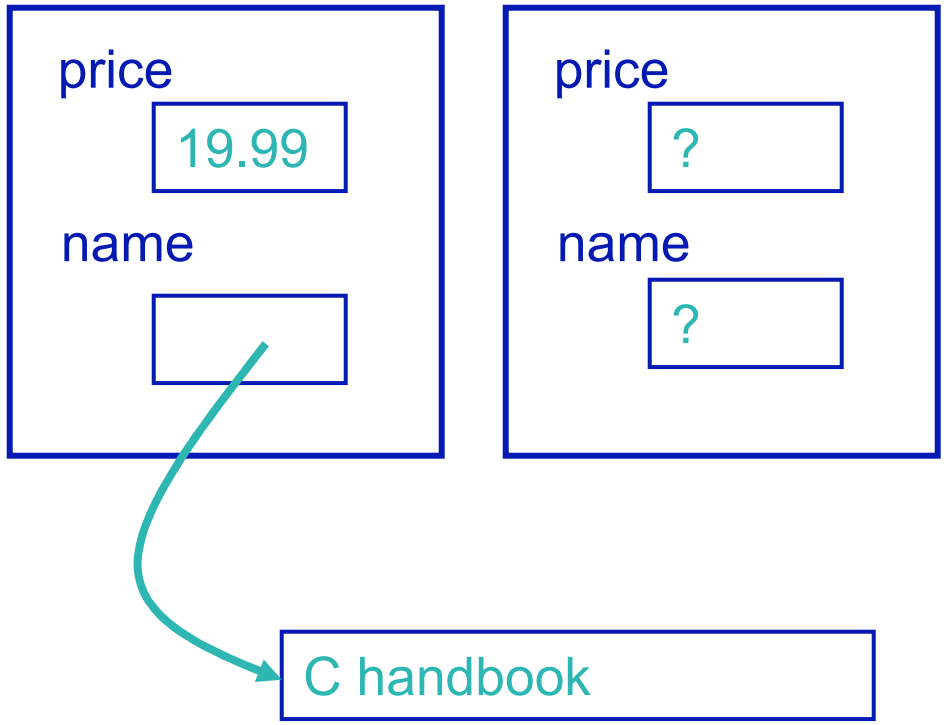



```
/* …continued… */
 strcpy(b.name,
      "Unix handbook");
/*… rest of program */
```



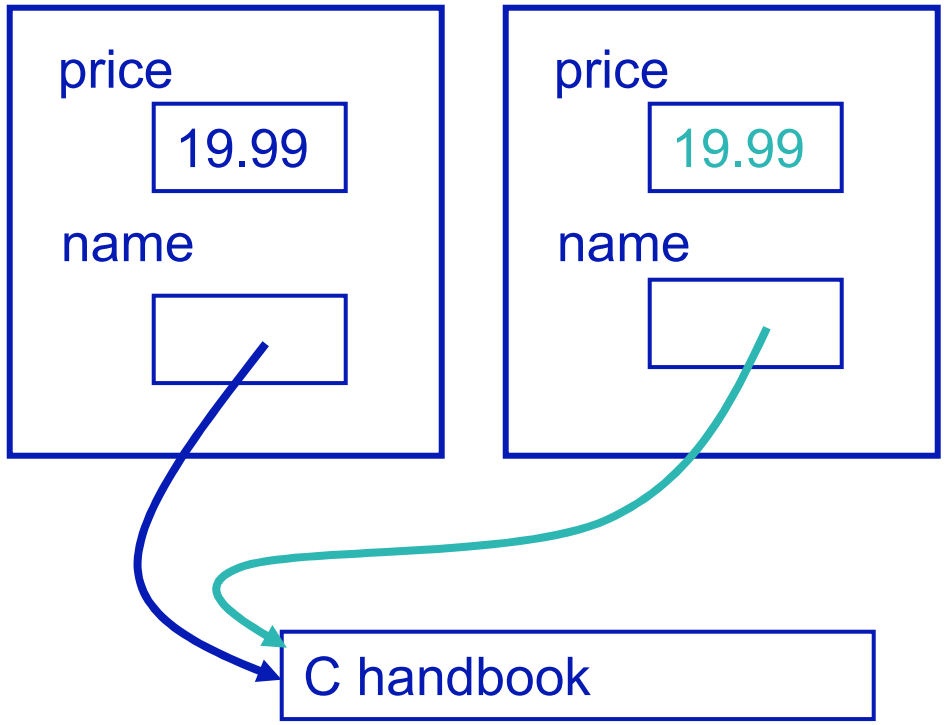
# Array Member vs. Pointer Member

```
struct book {
    float price;
    char *name;
};
int main()
{
    struct book a,b;
    b.price = 19.99;
    b.name = (char *)
    malloc(50);
    strcpy(b.name,
    "C handbook");
/* continued… */
```



# Array Member vs. Pointer Member

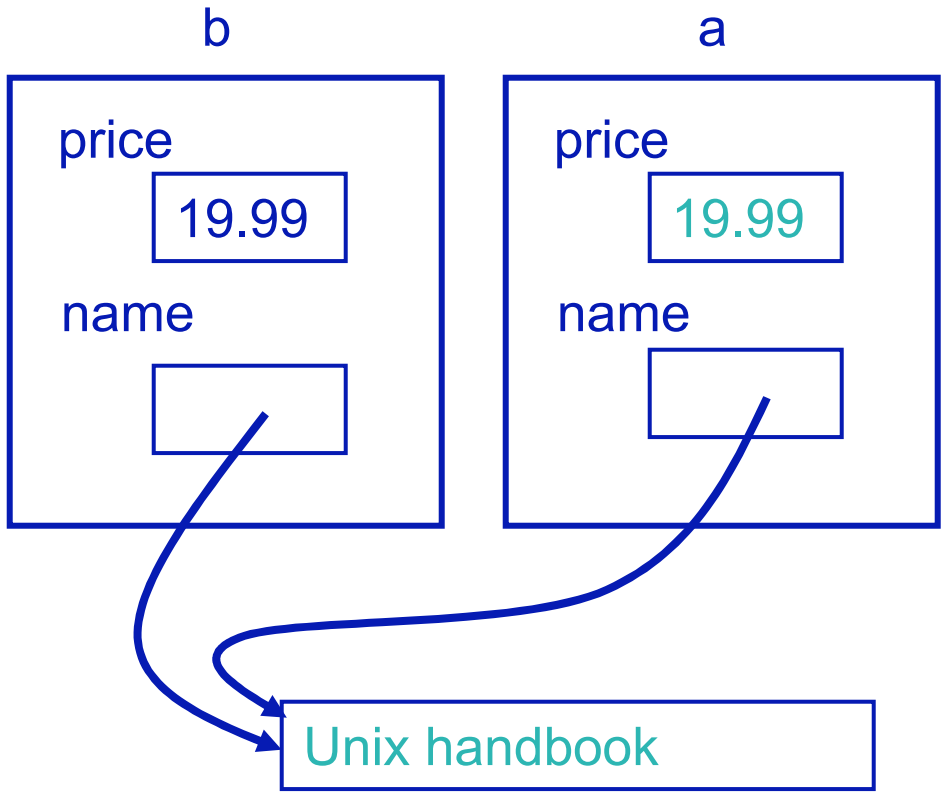
```
/* … continued… */
    a = b;
/*  … continued… */
```




Values of the struct's members were copied here; the string lies outside the structure, and was not copied

# Array Member vs. Pointer Member

```
/* … continued… */
    strcpy(b.name,
     "Unix handbook");
/* …rest of program  */
```




This isn't likely what we wanted to happen here

## strdup() from <string.h>

- General form:

  char * strdup (const char * source);

- strdup( ) makes a dynamically allocated copy of a string at source, and returns a pointer to it; returns NULL instead if a copy can't be made
- Size of the new string is strlen(source)

## strdup() from <string.h>

```
/* from earlier example */
struct book {
    float price;
    char *name;
};
int main()
{
    struct book a,b;
    b.price = 19.99;
    b.name = (char *) malloc(50);
    strcpy(b.name,
            "C handbook");
```

- Instead of the calls to malloc() and strcpy(), we can use

  b.name = strdup("C handbook");

- Only difference: b.name will have the capacity to store only 10 chars plus the null character, rather than the 50 chars it held in the original codeSize of the new string is strlen(source)

# Passing Structures to Functions

- Structures are passed by value to functions
- The formal parameter variable will be a copy of the actual parameter in the call
- Copying can be inefficient if the structure is big
- It is usually more efficient to pass a pointer to the struct

# Structures as Return Values

- A function can have a struct as its return value
- It may in general be more efficient to have a function return a pointer to a struct, but be careful:
  - Don't return a pointer to a local variable
  - It's fine to return a pointer to a dynamically allocated structure

# Passing Structures to Functions

```
#include<stdio.h>
struct pairInt {
      int min, max;
};
struct pairInt min_max(int x,int y)
{
      struct pairInt pair;
      pair.min = (x > y) ? y : x;
      pair.max = (x > y) ? x : y;
      return pairInt;
}
int main(){
      struct pairInt result;
      result = min_max( 3, 5 );
      printf("%d<=%d", result.min, result.max);
}
```

# Passing Structures to Functions

```
#include<stdio.h>
struct book {
    float price;
    char abstract[5000];
};
void print_abstract( struct book
    *p_book)
{
    puts( p_book->abstract );
};
```

## Unions

- **union**
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a **union** share space
  - Conserves storage
  - Only the last data member defined can be accessed
- **union** declarations
  - Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```

# Unions

- Valid **`union`** operations
  - Assignment to **`union`** of same type: **`=`**
  - Taking address: **`&`**
  - Accessing union members: **`.`**
  - Accessing members using pointers: **`->`**

# Big and Little Endian Representations

- Endianness refers to the order that the individual bytes (not bits) of a multibyte data element is stored in memory.
- Big endian is the most straightforward method. It stores the most significant byte first, then the next significant byte and so on.
- Little endian stores the bytes in the opposite order (least significant first).
- The x86 family of processors use little endian representation.

# How to Determine Endianness

```
unsigned short word = 0x1234; /* assumes sizeof ( short) == 2 */
unsigned char  p = (unsigned char ) &word;
if ( p[0] == 0x12 )
        printf (”Big Endian Machine\n”);
else
        printf (” Little Endian Machine\n”);
```

# When to Care About Little and Big Endian

- For typical programming, the endianness of the CPU is not significant.
- The most common time that it is important is when binary data is transferred between different computer systems.
- Since ASCII data is single byte, endianness is not an issue for it.



# Bitwise operator and Bit fields

School of Computer Science

hmehta.scs@dauniv.ac.in

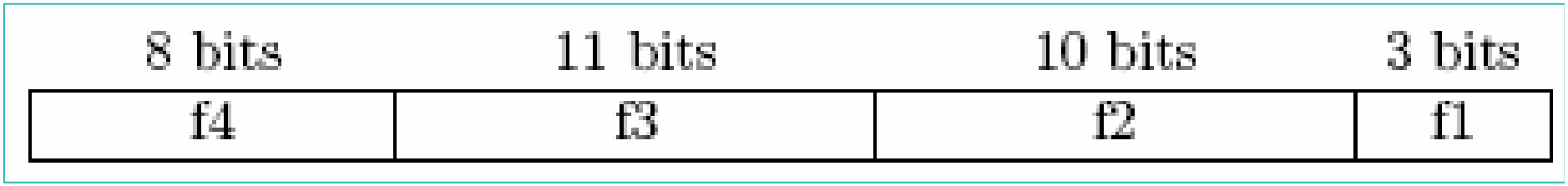
# Bit Fields

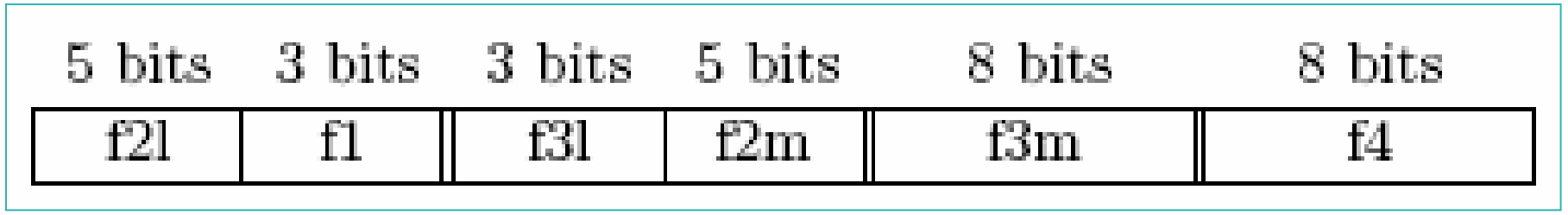
- Bit fields allow one to specify members of a struct that only use a specified number of bits. The size of bits does not have to be a multiple of eight.
- A bit field member is defined like an unsigned int or int member with a colon and bit size appended to it.

## An example of bitfield

```
struct S {
   unsigned f1  : 3;    /* 3-bit field  */
   unsigned f2  : 10;   /* 10-bit field */
   unsigned f3  : 11;   /* 11-bit field */
   unsigned f4  : 8;    /* 8-bit field  */
};
```

**The first bitfield is assigned to the least significant bits of its word**





hmehta.scs@dauniv.ac.in

# Bitwise Operators

- All data represented internally as sequences of bits
  - Each bit can be either `0` or `1`
  - Sequence of 8 bits forms a byte

| Operator | Name | Description |
|---|---|---|
| `&` | bitwise AND | The bits in the result are set to `1` if the corresponding bits in the two operands are both `1`. |
| `\|` | bitwise OR | The bits in the result are set to `1` if at least one of the corresponding bits in the two operands is `1`. |
| `^` | bitwise exclusive OR | The bits in the result are set to `1` if exactly one of the corresponding bits in the two operands is `1`. |
| `<<` | left shift | Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from right with `0` bits. |
| `>>` | right shift | Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent. |
| `~` | One's complement | All `0` bits are set to `1` and all `1` bits are set to `0`. |



# Files in C

School of Computer Science

hmehta.scs@dauniv.ac.in

# FILE *

- C uses the ***FILE**** data type to access files
- FILE is defined in <stdio.h>

```
#include <stdio.h>
int main( )
{
    FILE * fp;
    fp = fopen("tmp.txt", "w");
    fprintf(fp,"This is a test\n");
    fclose(fp);
    return 0;
}
```

# Opening a File

- Must include <stdio.h>
- Prototype form:

  FILE * fopen (const char * filename, const char * mode)
- FILE is a ***structure type*** declared in stdio.h.
  - Keeps track of the file mode (read, write, etc), position in the file that we're accessing currently, and other details
  - May vary from system to system

# Opening a File

- ***fopen*** returns a pointer to a FILE structure
- Must declare a pointer of type FILE to receive that value when it is returned
- Use the returned pointer in all subsequent references to that file
- If fopen fails, *NULL* is returned.
- The argument filename is the name of the file to be opened

# Opening a File

- Enclose the ***mode*** in double quotes or pass as a string variable
- Modes are:
  - ***r***: open the file for reading; fopen returns NULL if the file doesn't exist or can't be opened
  - ***w***: create file for writing; destroy old if file exists
  - ***a***: open for writing; create if not there; start at the end-of-file (append mode)
  - ***r+***: open for update (r/w); create if not there; start at the beginning
  - ***w+***: create for r/w; destroy old if there
  - ***a+***: open for r/w;create if not there; start at the end-of-file

# Four Ways to Read and Write Files

- Formatted file I/O
- Get and put a character
- Get and put a line
- Block read and write

# Formatted File I/O

- Formatted file input is done through fscanf:
  - int fscanf (FILE * fp, const char * fmt, ...) ;
- Formatted file output is done through fprintf:
  - int fprintf(FILE *fp, const char *fmt, …);
- fscanf and fprintf work just like scanf and printf, except that a file pointer is required

# Formatted File I/O

```
{   …
    FILE *fp1, *fp2;
    int n;
    fp1 = fopen("file1", "r");
    fp2 = fopen("file2", "w");
    fscanf(fp1, "%d", &n);
    fprintf(fp2, "%d", n);
    fclose(fp1);
    fclose(fp2);
}
```

## Get and Put a Character

```
#include <stdio.h>
int fgetc(FILE * fp);
int fputc(int c, FILE * fp);
```

- These two functions read or write a single byte from or to a file
- ***fgetc*** returns the character that was read, converted to an integer
- ***fputc*** returns the value of parameter c if it succeeds; otherwise, returns ***EOF***

# Get and Put a Line

```
#include <stdio.h>
char *fgets(char *s, int n, FILE * fp);
int fputs(char *s, FILE * fp);
```

- ***fgets*** reads an entire line into s, up to n-1 chars in length; includes the newline char in the string, unless line is longer than n-1
- fgets returns the pointer s on success, or NULL if an error or end-of-file is reached
- ***fputs*** returns the number of chars written if successful; otherwise, returns ***EOF***

# Closing and Flushing Files

int fclose (FILE * fp) ;

- Call to fclose closes fp -- returns 0 if it works, or1 if it fails

- Can clear a buffer *without closing* it

int fflush (FILE * fp) ;

- Essentially this is a force to disk
- Very useful when debugging

- Without fclose or fflush, updates to a file may not be written to the file on disk. (Operating systems like Unix usually use “write caching” disk access.)

# Detecting End of an Input File

- When using fgetc:

  while ( (c = fgetc (fp) ) != EOF ) { … }

  - Reads characters until it encounters the EOF char
  - The problem is that the byte of data read may actually be indistinguishable from EOF
- When using fgets:

  while ( fgets( buf, bufsize, fp ) != NULL ) { … }

  - Reads strings into buf until end of file is reached

# Detecting End of an Input File

- When using fscanf:
  - Tricky to detect end of file: value of fscanf call will be less than the expected value, but this condition can occur for a number of other reasons as well
- In all these situations, end of file is detected only when we attempt to read past it
- Function to detect end of file:

  int feof (FILE * fp) ;
  - ***Note***: the feof function realizes the end of file only ***after*** a read attempt has failed (fread, fscanf, fgetc)

# Example

```
#include<stdio.h>
#define BUFSIZE 100
int main ( ) {
    char buf[BUFSIZE];
    if ( (fp=fopen("file1", "r"))==NULL) {
       fprintf (stderr,"Error opening file.");
       exit (1);
    }
    while (!feof(fp)) {
       fgets  (buf,BUFSIZE,fp);
       printf ("%s",buf);
    }
    fclose (fp);
    return 0;
}
```

This program echoes the contents of file1 to standard output with one flaw: the last line is echoed ***twice***; it would be better to use:

```
while (fgets(buf, BUFSIZE,
                    fp) !=  NULL)
   printf("%s",buf);
```



# Advanced File Features

School of Computer Science

hmehta.scs@dauniv.ac.in

# Block Reading and Writing

- ***fread*** and ***fwrite*** are ***binary file*** reading and writing functions
- Prototypes are found in stdio.h
- ***Advantages*** of using binary files:
  - Reading and writing are *quick*, since I/O is not being converted from/to ASCII characters
  - *Large amounts* of data can be read/written with a single function call (***block reading and writing***)
- ***Disadvantage*** of using binary files:
  - Contents are not easily read by humans

# Block Reading and Writing

- Generic form:

  int fwrite (void *buf, int size, int count, FILE *fp) ;

  int fread  (void *buf, int size, int count, FILE *fp) ;

- buf: is a pointer to the region in memory to be written/read; it can be a pointer to anything (a simple variable, an array, a structure, etc)
- size: the size in bytes of each individual data item
- count: the number of data items to be written/read

# Block Reading and Writing

- *Example*: We can write all 100 elements from an array of integers to a ***binary file*** with a single statement
  - fwrite( buf, sizeof(int), 100, fp);
  - Bit patterns for 100*sizeof(int) bytes at address buf are copied directly to the output file, without any type conversion
- The fwrite (fread) returns the number of items actually written (read)

# Block Reading and Writing

- Testing for errors:

```
if ((frwrite(buf,size,count,fp)) != count)
   fprintf(stderr, "Error writing to file.");
```

- Writing value of a double variable x to a file:

```
fwrite (&x, sizeof(double), 1, fp) ;
```

  - This writes the double x to the file in raw binary format (I.e.: its internal machine format)

# Block Reading and Writing

- Writing an array text[50] of 50 characters can be done by:
  - fwrite (text, sizeof(char), 50, fp) ;
    - or
  - fwrite (text, sizeof(text), 1, fp);  /* text must be a local array name */
- fread and fwrite are more efficient than fscanf and fprintf: no type conversions required

## Sequential and Random Access

- A FILE structure contains a long that indicates the position (*disk address*) of the next read or write
- When a read or write occurs, this position changes
- You can ***rewind*** and start reading from the beginning of the file again:

```
void rewind (FILE * fp) ;
```

  - A call to rewind resets the position indicator to the beginning of the file

## Sequential and Random Access

- To determine where the position indicator is, use:

  long ftell (FILE * fp) ;

  - Returns a long giving the current position in bytes
  - The first byte of the file is byte zero
  - If an error occurs, ftell () returns -1

# Random Access

- If we're aware of the structure of a file, we can move the file's position indicator anywhere we want within the file (random access):

  int fseek (FILE * fp, long offset, int origin) ;

  - offset is the number of bytes to move the position indicator
  - origin says where to move from

## Random Access

- Three options/constants are defined for origin:
  - SEEK_SET: move the indicator offset bytes from the beginning
  - SEEK_CUR: move the indicator offset bytes from its current position
  - SEEK_END: move the indicator offset bytes from the end

# Random Access

- Random access is most often used with binary input files, when *speed* of execution matters: we want to avoid having to read in data sequentially to get to a known location
- Writing to a location in a file other than the end *does not insert* content: it *overwrites*

# Example: End of File

```
…
fseek(fp,0,SEEK_END);  /* position indicator is 0 bytes from
                          the end-of-file marker */
printf("%d\n", feof(fp))      /* Value printed is zero */
fgetc(fp);                        /* fgetc returns -1 (EOF) */
printf("%d\n",feof(fp));      /* Nonzero value, now that an attempt
                          has been made to read at the end
                          of the file  */
```

# File Management Functions

- Erasing a file:

  int remove (const char * filename);

  - This is a character string naming the file
  - Returns 0 if deleted, and -1otherwise
  - If no pathname is provided, attempts to delete the file from the current working directory
  - Can fail for several reasons: file not found, user does not have write privileges, file is in use by another process, etc

# File Management Functions

- Renaming a file:

  int rename (const char * oldname,
  const char * newname);

  - Returns 0 if successful, or -1 if an error occurs
  - error: file oldname does not exist
  - error: file newname already exists
  - error: try to rename to another disk

# Using Temporary Files

- ***Temporary files:*** exist only during the execution of the program
- To generate a filename, use:

  char *tmpnam (char *s) ;
  - Creates a valid filename that does not conflict with any other existing files
- You then open it and write to it
- The file will continue to exist after the program executes unless you delete it using remove()

## Example

```
#include <stdio.h>
int main () {
    char buffer[25];
    tmpnam(buffer);
    printf ("Temporary name is: %s", buffer);
    return 0;
}
```

***Output***:

Temporary name is: c:\tc\bin\aaaceaywB

# Implicitly Opened Files: stdin, stdout, and stderr

- Every C program has three files opened for them at start-up: ***stdin***, ***stdout***, and ***stderr***
- stdin (standard input) is opened for reading, while stdout (standard output) and stderr (standard error) are opened for writing
- They can be used wherever a FILE * can be used
- Writing to stderr is a good practice when reporting error messages: it causes all output buffers to be flushed (written), and aids debugging

## stdin, stdout, and stderr

- *Examples*:
  - fprintf(stdout, "Hello there\n");
    - This is the same as printf("Hello there\n");
  - fscanf(stdin, "%d", &int_var);
    - This is the same as scanf("%d", &int_var);
  - fprintf(stderr, "An error has occurred\n");
    - Will force anything in the stdout buffer or in the buffer for an output file to be printed as well

# The exit () Function

- Used to abort the program at anytime from anywhere before the normal exit location
- Syntax:

```
exit (status);
```

- *Example*:

```
#include <stdlib.h>
……
if( (fp=fopen("a.txt","r")) == NULL){
        fprintf(stderr, "Cannot open file a.txt!\n");
        exit(1);
}
```

Devi Ahilya Vishwavidyalaya

# I/O Redirection, Unconditional Branching, Enumerated Data Type, Little Endian and Big Endian

School of Computer Science

hmehta.scs@dauniv.ac.in

# Redirecting Input/Output on UNIX and DOS Systems

- Standard I/O - keyboard and screen
  - Redirect input and output
- Redirect symbol(<)
  - Operating system feature, not a C feature
  - UNIX and DOS
  - `$` or `%` represents command line
  - Example:

```
$ myProgram < input
```

  - Rather than inputting values by hand, read them from a file
- Pipe command( | )
  - Output of one program becomes input of another

```
$ firstProgram | secondProgram
```

  - Output of **`firstProgram`** goes to **`secondProgram`**


hmehta.scs@dauniv.ac.in

# Redirecting Input/Output on UNIX and DOS Systems

- Redirect output (>)
  - Determines where output of a program goes
  - Example:

```
$ myProgram > myFile
```

    - Output goes into **myFile** (erases previous contents)
- Append output (>>)
  - Add output to end of file (preserve previous contents)
  - Example:

```
$ myOtherProgram >> myFile
```

    - Output is added onto the end of **myFile**

# The Unconditional Branch: goto

- Unstructured programming
  - Use when performance crucial
  - **`break`** to exit loop instead of waiting until condition becomes **`false`**
- **`goto`** statement
  - Changes flow control to first statement after specified label
  - A label is an identifier followed by a colon (i.e. **`start:`**)
  - Quick escape from deeply nested loop

```
goto start;
```

# Enumeration Constants

- Enumeration
  - Set of integer constants represented by identifiers
  - Enumeration constants are like symbolic constants whose values are automatically set
    - Values start at `0` and are incremented by `1`
    - Values can be set explicitly with `=`
    - Need unique constant names
  - Example:

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL,
  AUG, SEP, OCT, NOV, DEC};
```

    - Creates a new type enum Months in which the identifiers are set to the integers `1 to 12`
  - Enumeration variables can only assume their enumeration constant values (not the integer representations)

Devi Ahilya Vishwavidyalaya

# Inline function, Type Qualifiers and Storage Classes

School of Computer Science

hmehta.scs@dauniv.ac.in

# Inline Functions

- Recall the two different ways to compute the larger of two integers:
  - #define max(a,b)  ((a)>(b)? (a):(b))
  - int max(int a, int b) { return a>b?a:b; }
- To execute a function call, computer must:
  - Save current registers
  - Allocate memory on the call stack for the local variables, etc, of the function being called
  - Initialize function parameters
  - Jump to the area of memory where the function code is, and jump back when done

# Inline Functions

- The macro approach is more efficient since it does not have function call overhead, ***but***, this approach can be dangerous, as we saw earlier
- Modern C compilers provide ***inline functions*** to solve the problem:
  - Put the inline keyword before the function header

```
inline int max(int a, int b) {
   return a>b?a:b;
}
```

# Inline Functions

- You then use the inline function just like a normal function in your source code

```
printf( "%d", max( x, y) );
```

- When the compiler compiles your program, it will not compile it as a function; rather, it integrates the necessary code in the line where max( ) is called, and avoids an actual function call
- The above printf(…) is compiled to be something like:

```
printf("%d", x>y?x:y);
```

# Inline Functions

- Writing the small but often-used functions as inline functions can improve the speed of your program
- *A small problem*: You must include the inline function definition (not just its prototype) before using it in a file
- Therefore, inline functions are often defined in header (.h) files

# Inline Functions

- Once you include a header file, you can use:
  - Inline functions whose definitions are in the header file
  - Normal functions whose prototypes are in the header file
- *Another minor problem*: Some debuggers get confused when handling inline functions – it may be best to turn functions into inline functions only after debugging is finished

# Two advantages and the main disadvantage to inlining

- The inline function is faster. No parameters are pushed on the stack, no stack frame is created and then destroyed, no branch is made.
- Secondly, the inline function call uses less code!
- The main disadvantage of inlining is that inline code is not linked and so the code of an inline function must be available to all files that use it.

# Type Qualifiers

Type qualifiers that control how variables may be accessed or modified

## const

Variables of type const may not be changed by your program. The compiler is free to place variables of this type into read-only memory (ROM).

```
const int a=10;
```

creates an integer variable called a with an initial value of 10 that your program may not modify.

The const qualifier can be used to prevent the object pointed to by an argument to a function from being modified by that function. That is, when a pointer is passed to a function, that function can modify the actual object pointed to by the pointer.

# Volatile

The modifier volatile tells the compiler that a variable's value may be changed in ways not explicitly specified by the program. For example, a global variable's address may be passed to the operating system's clock routine and used to hold the system time. In this situation, the contents of the variable are altered without any explicit assignment statements in the program. This is important because most C compilers automatically optimize certain expressions by assuming that a variable's content is unchanging if it does not occur on the left side of an assignment statement; thus, it might not be reexamined each time it is referenced.

## Const + Volatile

You can use const and volatile together. For example, if 0x30 is assumed to be the value of a port that is changed by external conditions only, the following declaration would prevent any possibility of accidental side effects:

```
const volatile char *port = (const volatile char *) 0x30;
```

# Storage Class Specifiers

C supports four storage class specifiers:
extern
static
register
Auto

These specifiers tell the compiler how to store the subsequent variable. The general form of a variable declaration that uses one is shown here:

storage_specifier type var_name;

# Global Variables

Global variables are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program's execution. You create global variables by declaring them outside of any function. Any expression may access them, regardless of what block of code that expression is in. In the following program, the variable count has been declared outside of all functions. Although its declaration occurs before the main( ) function, you could have placed it anywhere before its first use as long as it was not in a function. However, it is usually best to declare global variables at the top of the program.

# Global Variables …..

Storage for global variables is in a fixed region of memory set aside for this purpose by the compiler. Global variables are helpful when many functions in your program use the same data. You should avoid using unnecessary global variables, however. They take up memory the entire time your program is executing, not just when they are needed.

# Linkage

C defines three categories of linkage: external, internal, and none. In general, functions and global variables have external linkage. This means they are available to all files that constitute a program. File scope objects declared as static (described in the next section) have internal linkage. These are known only within the file in which they are declared. Local variables have no linkage and are therefore known only within their own block.

## Extern

The principal use of extern is to specify that an object is declared with external linkage elsewhere in the program.
A declaration declares the name and type of an object. A definition causes storage to be allocated for the object. The same object may have many declarations, but there can be only one definition.

In most cases, variable declarations are also definitions. However, by preceding a variable name with the extern specifier, you can declare a variable without defining it. Thus, when you need to refer to a variable that is defined in another part of your program, you can declare that variable using extern.

## Extern

```
#include <stdio.h>
int main(void)
{
    extern int first, last; /* use global vars */
    printf("%d %d", first, last);
    return 0;
}
/* global definition of first and last */
int first = 10, last = 20;
```

## Multiple-File Programs

An important use of extern relates to multiple-file programs. C allows a program to be spread across two or more files, compiled separately, and then linked together. When this is the case, there must be some way of telling all the files about the global variables required by the program. The best (and most portable) way to do this is to declare all of your global variables in one file and use extern declarations in the other.

# Multiple-File Programs

```
int x, y;
char ch;
int main(void)
{
    /* . . . */
}
void func1(void)
{
    x = 123;
}
```

```
extern int x, y;
extern char ch;
void func22(void)
{
    x = y / 10;
}
void func23(void)
{
    y = 10;
}
```

# Static Variables

Variables declared as static are permanent variables within their own function or file. Unlike global variables, they are not known outside their function or file, but they maintain their values between calls. This feature makes them useful when you write generalized functions and function libraries that other programmers may use. The static modifier has different effects upon local variables and global variables.

# Static Local Variables

When you apply the static modifier to a local variable, the compiler creates permanent storage for it a static local variable is a local variable that retains its value Between function calls.

An example of a function that benefits from a static local variable is a number – series generator that produces a new value based on the previous one.

# Static Global Variables

Applying the specifier static to a global variable instructs the compiler to create a global variable known only to the file in which it is declared. Thus, a static global variable has internal linkage (as described under the extern statement). This means that even though the variable is global, routines in other files have no knowledge of it and cannot alter its contents directly, keeping it free from side effects.

# Register Variables

The register specifier requested that the compiler keep the value of a variable in a register of the CPU rather than in memory, where normal variables are stored. This meant that operations on a register variable could occur much faster than on a normal variable because the register variable was actually held in the CPU and did not require a memory access to determine or modify its value.

The register storage specifier originally applied only to variables of type int, char, or pointer types. type of variable.

# Any Questions

# Thank You



hmehta.scs@dauniv.ac.in