CSE225: Data Structure and Algorithm

Array, Pointer & Searching

Array

- Declaration
- Initialization
- Accessing Array Elements
- Inserting and Deleting in a Unsorted Array
- Inserting and Deleting in a sorted Array

Linear Arrays

- A linear array is a list of a finite number of **n** homogeneous data elements (that is data elements of the same type) such that.
  - The elements are of the arrays are referenced respectively by an index set consisting of **n** consecutive numbers.
  - The elements of the arrays are stored respectively in **successive memory locations.**

Linear Arrays

- The number **n** of elements is called the length or size of the array.
- The index set consists of the integer **0**, **1, 2, …n-1**
- **Length** or the number of data elements of the array can be obtained from the index set by

Length = UB – LB + 1

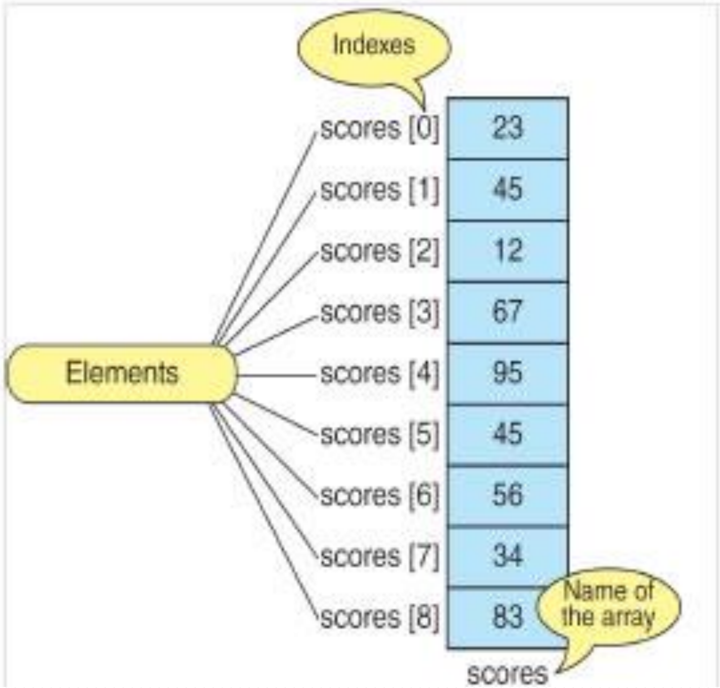
where

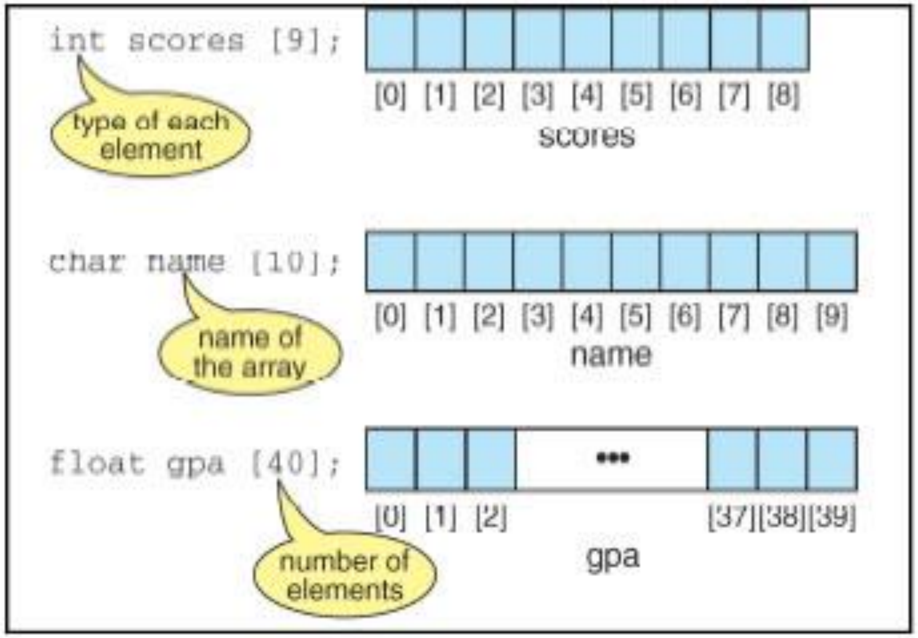
UB is the **largest index** called the **upper bound** and

LB is the **smallest index** called the **lower bound** of the arrays

Linear Arrays

- Element of an array **A** may be denoted by
  - Subscript notation **A_1, A_2, , …. , A_n**
  - Parenthesis notation **A(1), A(2), …. , A(n)**
  - Bracket notation **A[1], A[2], ….. , A[n]**
- The number **K** in A[K] is called subscript or an index and A[K] is called a **subscripted variable.**





Declaring and Initializing Arrays

To declare an array:

data_type array_name[SIZE];

int ar_name[10]

data_type is a valid data type that must be common to all elements.

array_name is name given to array

SIZE is a constant value that defines array maximum capacity.

Initializing Arrays

Initialization of an array either one by one or using a single statement as follows −

int ar[5];

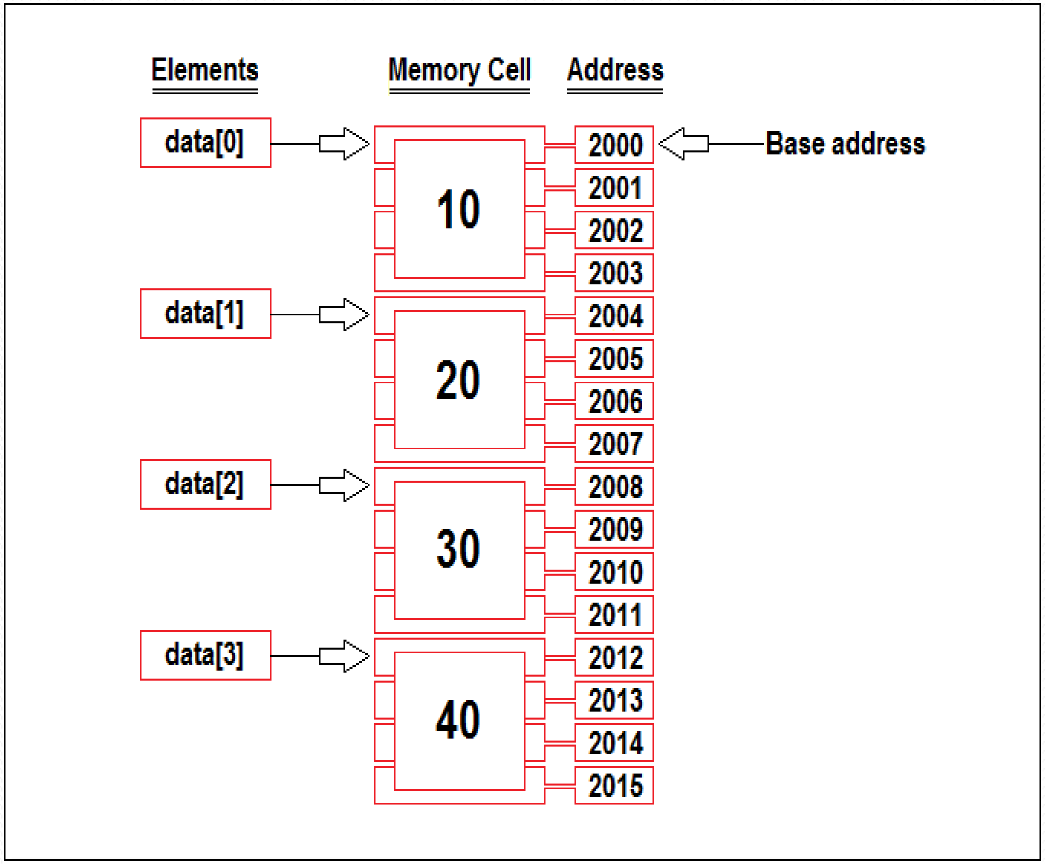
ar[0]=10; ar[1]=20;

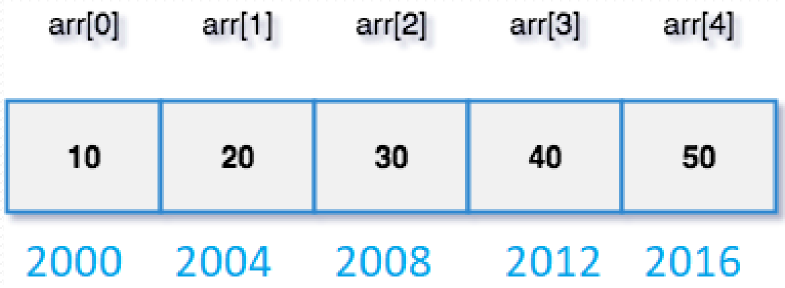
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

Representation of Linear Array in Memory

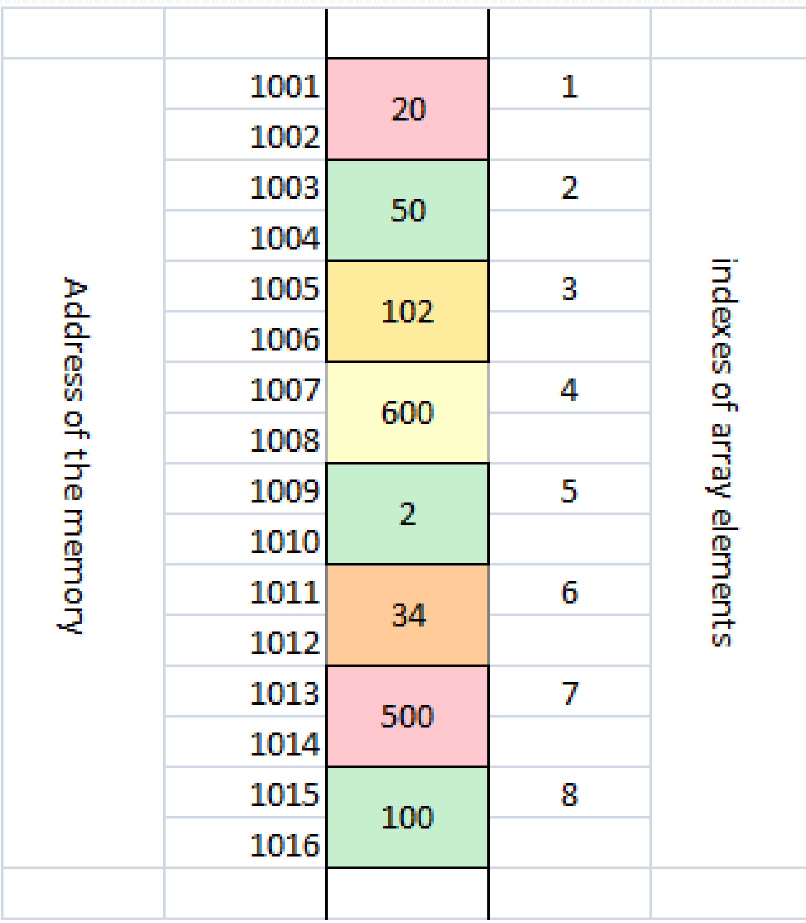
The picture can't be displayed.

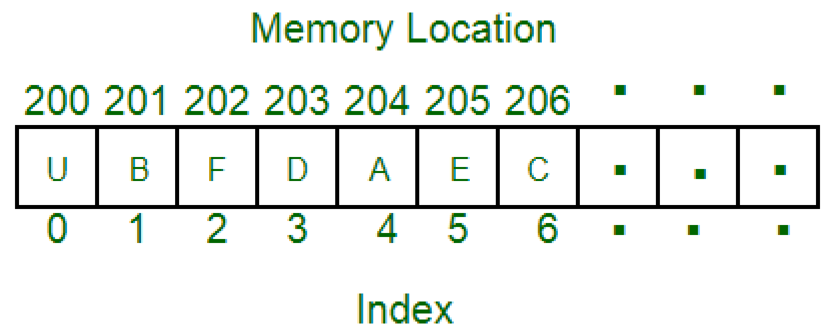
Representation of Linear Array in Memory

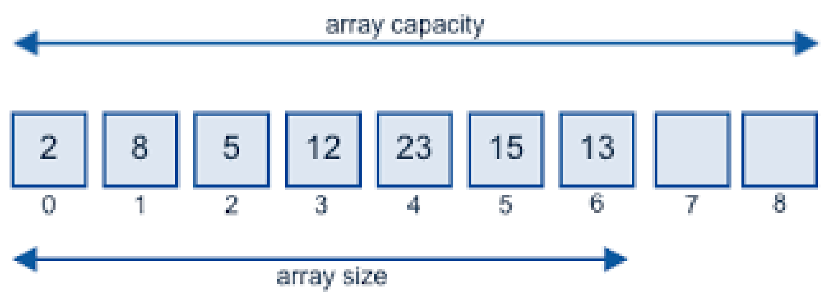




Representation of Linear Array in Memory







Representation of Linear Array in Memory

int LA[10]

Let

- **LA** be a linear array in the memory of the computer.
- **LOC(LA[K])= address of the element LA[K]**

- Computer does not keep track of the address of every element of the array
- Keep track address of the first element of array
- called the **base address** of LA and denoted by **Base(LA)**

- **LOC(LA[K]) = Base(LA) + w(K– lower bound)**

where **w** is the number of words per memory cell

Example 1

Find the address for LA[6] Each element of the array occupy 1 byte

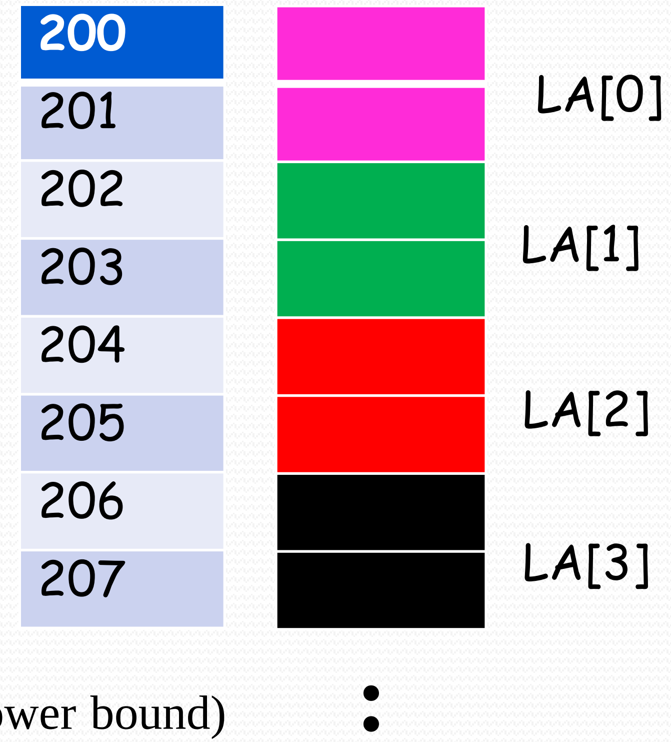
200		LA[1]
201		LA[2]
202		LA[3]
203		LA[4]
204		LA[5]
205		LA[6]
206		LA[7]
207		LA[8]
	⋮	

$$LOC(LA[K]) = Base(LA) + w(K - \text{lower bound})$$

$$LOC(LA[6]) = 200 + 1(6 - 1) = 205$$

Example 2

Find the address for LA[16] Each element of the array occupy 2 byte



LOC(LA[K]) = Base(LA) + w(K – lower bound)

LOC(LA[16]) = 200 + 2(16 – 0) = 232

Searching Arrays

- **Linear search:** Compare each element of array with key value
- Search an array for a *key value*
  - Simple
  - Useful for small and unsorted arrays

- Suppose you want to find a number in an unordered sequence
- You have no choice – look through all elements until you have found a match
- This is called linear or sequential search

Linear Search(LA,N, ITEM)

Linear Search (LA, N, ITEM)
Here LA is a linear array with N elements. This algorithm find an element ITEM into the LA.

1. i=0
2. Repeat steps 3 and 4 while i<=n or LA[i]==ITEM
3. IF LA[i]==ITEM print item found at index i and exit
4. i=i+1
5. Print item not found and exit.

```
int search(int data[],int n, int
item)
{
for(int i = 0; i<n; i++)
     {
     if(data[i] == item)
     return i;
         }
    return -1;
}
```

Searching Arrays: Binary Search

- Binary search
  - For sorted arrays
  - Compares **middle** element with **key**
    - If equal, match found
    - If **key < middle**, looks in first half of array
    - If **key > middle**, looks in last half
    - Repeat
  - Very fast; at most n steps, where 2^n > number of elements
    - 30 element array takes at most 5 steps
      - $2^5 > 30$ so at most 5 steps

Searching Arrays: Binary Search

```
int bsearch(int data[], int n, int value )
      {
      int first, middle, last;
       first = 0;
       last = n - 1;
      while (true) {
          middle = (first + last) / 2;
          if (data[middle] == value)
               return middle;
          else if (first >= last)
               return -1;
          else if (value < data[middle])
            last = middle - 1;
          else
            first = middle + 1;
     }
}
```

Inserting in Unsorted Array

INSERT (LA, N, ITEM) Here LA is a linear array with N elements. This algorithm inserts an element ITEM into the LA.

1. If MAX==N, print overflow
2. Set LA[N] :=ITEM
3. [Reset N.] Set N := N + 1.
4. Exit.

Delete in Unsorted Array

DELETE (LA, N, k) Here LA is a linear array with N elements. This algorithm Delete the element ITEM from the LA.

1. Set LA[k] :=LA[N-1]
2. [Reset N.] Set N := N - 1.
3. Exit.

INSERT_SORTL (LA, N, K, ITEM)

Here LA is a sorted array with N elements and K is a positive integer such that K<N. This algorithm insert an element ITEM from the Kth position in LA.

1. j=N
2. Repeat step while LA[j-1]>ITEM;
3. Set LA[j]=LA[j-1]
4. j = j – 1;
5. Set LA[j]=ITEM
6. [Reset N.] Set N := N+1;
7. Exit.

0	10
1	20
2	25
3	30
4	40
5	45
6	50
7	60
8	70
9	80

DELETE_SORTL (LA, N, K, ITEM)

Here LA is a sorted array with N elements and K is a positive integer such that K<N. This algorithm Delete an element ITEM from the Kth position in LA.

0	10
1	20
2	30
3	50
4	60
5	70
6	80
7	90

1. SEARCH(LA, N, K, ITEM)
   Set j : = k.
2. Repeat Steps 3 and 4 while j < (N-1).
3. Move j^th^ element upward.
   Set LA[j] := LA[j+1].
4. [Increase counter.] Set j:= j+1. [End of Step 2 loop.]
5. [Reset N.] Set N := N-1;
6. Exit.

Multidimensional Array

- One-Dimensional Array
- Two-Dimensional Array
- Three-Dimensional Array
- Some programming Language allows as many as 7 dimension

Two-Dimensional Array

- A Two-Dimensional **$m \times n$** array **A** is a collection of **m.n** data elements
- with property that **$1 \le J \le m$** and **$1 \le K \le n$**

The element of **A** with first subscript **J** and second subscript **K** will be denoted by **A[J][K]**

2D Arrays

The elements of a 2-dimensional array a is shown as below

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Rows Of A 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]	row 0
a[1][0]	a[1][1]	a[1][2]	a[1][3]	row 1
a[2][0]	a[2][1]	a[2][2]	a[2][3]	row 2

Columns Of A 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
column 0	column 1	column 2	column 3

2D Array

- Let **A** be a two-dimensional array **m x n**
- The array **A** will be represented in the memory by a block of **m x n** sequential memory location
- Programming language will store array **A** either
  - **Column by Column**
  - (Called Column-Major Order) Ex: Fortran, MATLAB
  - **Row by Row**

  (Called Row-Major Order) Ex: C, C++ , Java

2D Array in Memory

A	Subscript	
	(1,1)	
	(2,1)	Column 1
	(3,1)	
	(1,2)	
	(2,2)	Colum n 2
	(3,2)	
	(1,3)	
	(2,3)	Colu mn 3
	(3,3)	
	(1,4)	
	(2,4)	Colum n 4
	(3,4)	

Column-Major Order

A	Subscript	
	(1,1)	
	(1,2)	Row 1
	(1,3)	
	(1,4)	
	(2,1)	
	(2,2)	
	(2,3)	Row2
	(2,4)	
	(3,1)	
	(3,2)	Row3
	(3,3)	
	(3,4)	

Row-Major Order

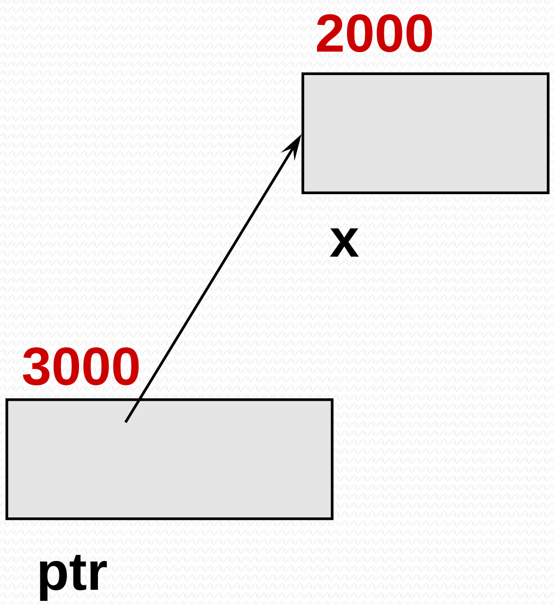
What is a pointer variable?

- **A pointer variable is a variable whose value is the address of a location in memory.**
- **To declare a pointer variable, you must specify the type of value that the pointer will point to.**
- **For example,**

```
int*   ptr; // ptr will hold the address of an int
char*  q;   // q will hold the address of a char
```

Using a pointer variable

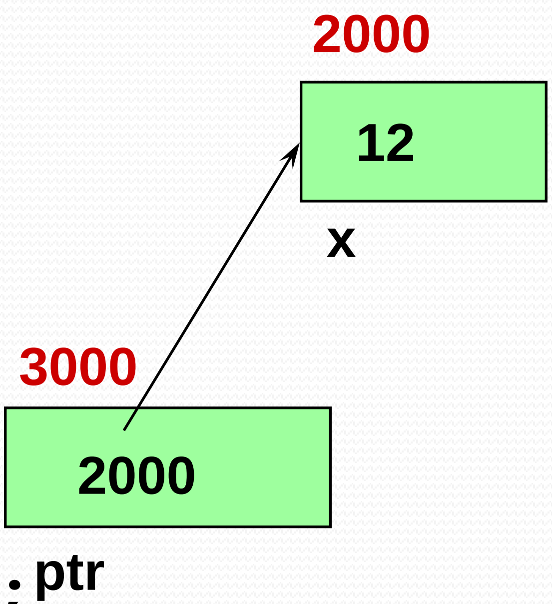
```
int  x;
x = 12;

int*  ptr;
ptr = &x;
```



NOTE: Because `ptr` holds the address of `x`, we say that `ptr` "points to" `x`

Unary **operator *** is the deference (indirection) operator

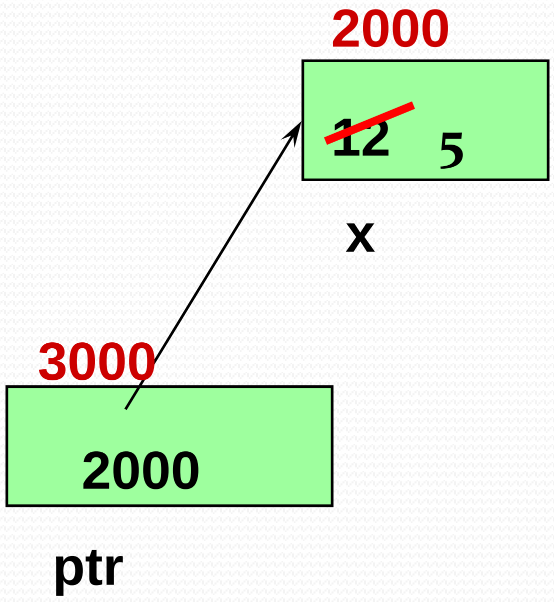
```
int  x;
x = 12;

int*  ptr;
ptr = &x;

std::cout  <<  *ptr;
```



NOTE: The value pointed to by `ptr` is denoted by `*ptr`

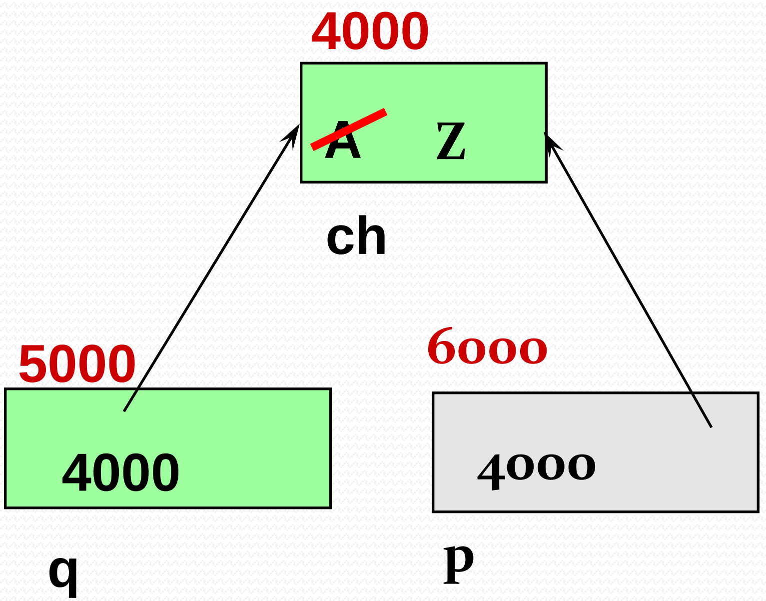
Using the dereference operator

```
int  x;
x = 12;


int*  ptr;
ptr = &x;

*ptr = 5;
// changes the value
// at adddress ptr to 5
```



Another Example

```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;    // the right side has value 4000
          // now p and q both point to ch
```



Dynamically Allocated Data

```
char*  ptr;

ptr = new char;
*ptr = 'B';
std::cout << *ptr;
```

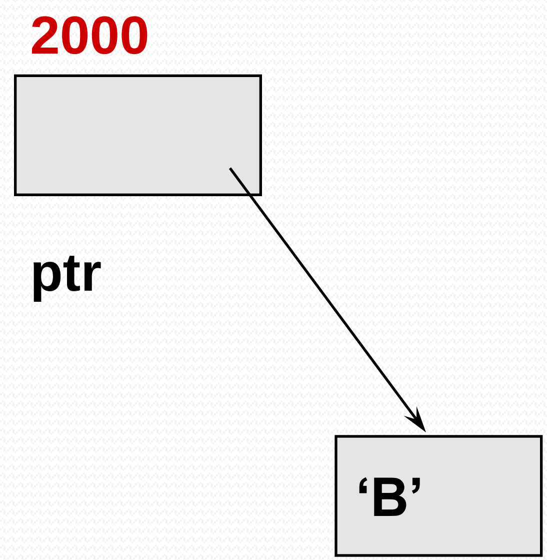
2000

ptr

New is an operator

Dynamically Allocated Data

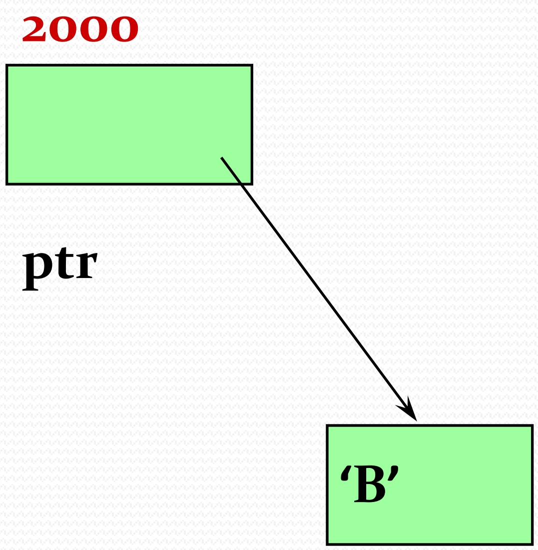
```
char*  ptr;

ptr = new char;
*ptr = 'B';
std::cout << *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;


ptr = new char;
*ptr = 'B';
std::cout << *ptr;
```



Dynamically Allocated Data

```
char*  ptr;


ptr = new char;

*ptr = 'B';

std::cout << *ptr;

delete  ptr;
```

2000

?

ptr

NOTE:
Delete deallocates the memory pointed to by ptr.

```
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

int main() {
        int*  ptr;
         ptr = new int;

    *ptr = 100;
      cout << *ptr<<endl;

       delete  ptr;
       cout << *ptr;
    return 0;
}
```

what does `new` do?

- takes a pointer variable,
- allocates memory for it to point, and
- leaves the address of the assigned memory in the pointer variable.
- If there is no more memory, the pointer variable is set to `NULL`.

The NULL Pointer

There is a pointer constant called **`NULL`** available in cstddef.

`NULL` is not a memory address;

it means that the pointer variable points to nothing.

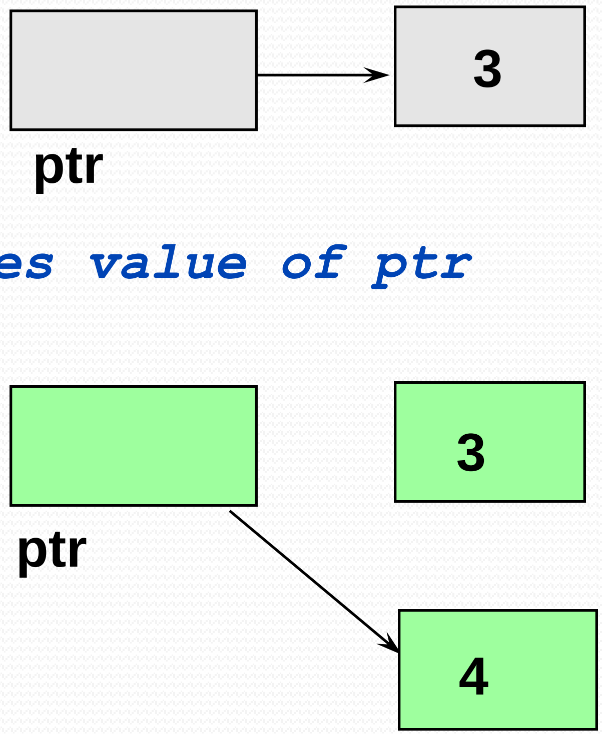
It is an error to dereference a pointer whose value is NULL.

It is the programmer's job to check for this.

```
while (ptr != NULL)
{
   . . .        // ok to use *ptr here
}
```

What happens here?

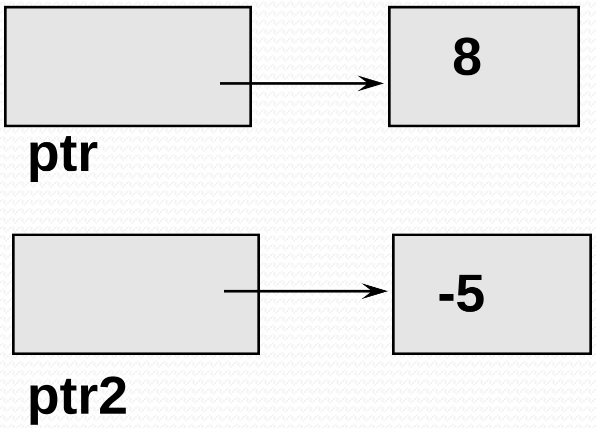
```
int* ptr = new int;
*ptr = 3;

ptr = new int;      // changes value of ptr
*ptr = 4;
```



Memory Leak

A memory leak occurs when dynamic memory (that was created using operator `new`) has been left without a pointer to it by the programmer, and so is inaccessible.

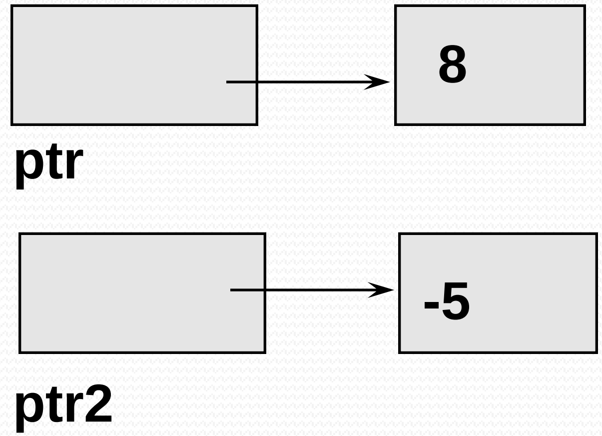
```
int* ptr = new int;
*ptr = 8;

int* ptr2 = new int;
*ptr2 = -5;
```



How else can an object become inaccessible?

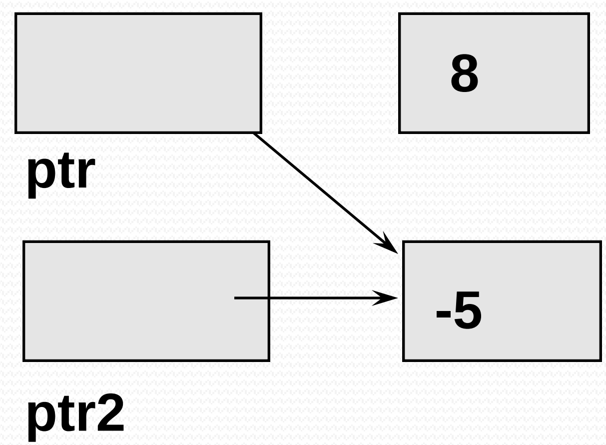
Causing a Memory Leak

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```




```
ptr = ptr2;
```

// here the 8 becomes inaccessible



Using operator `delete`

The **object or array currently pointed to by the pointer is deallocated, and the pointer is considered unassigned.**

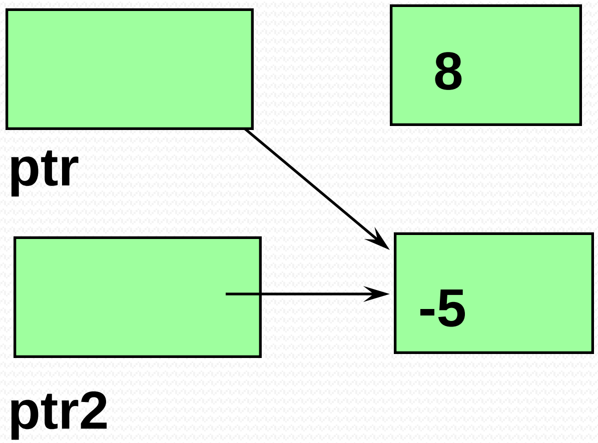
The memory is returned to the free store.

Square brackets are used with delete to deallocate a dynamically allocated array of classes.

A Dangling Pointer

- **occurs when two pointers point to the same object and delete is applied to one of them.**

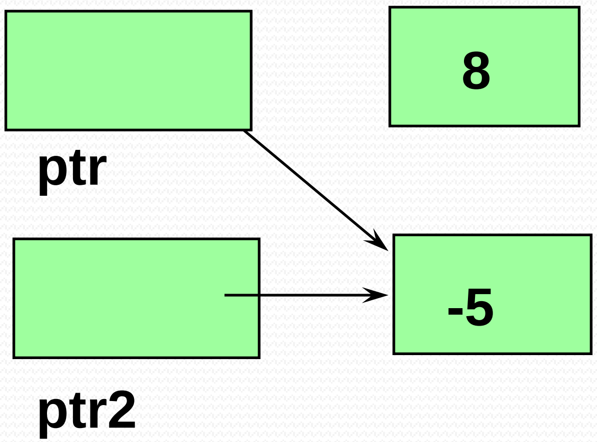
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;

ptr = ptr2;
```



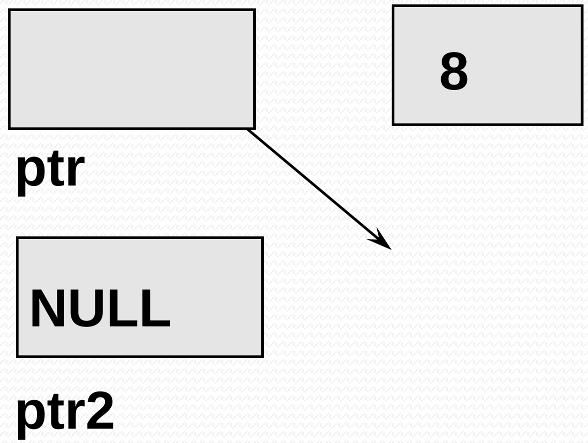
A dangling pointer is a pointer which points to some non-existing memory location.

Leaving a Dangling Pointer

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```




```
delete ptr2;        // ptr is left dangling
ptr2 = NULL;
```



Dangling Pointer

```
#include <iostream>
using namespace std;

int main() {
     int* ptr = new int;
 *ptr = 8;
 cout<<"ptr = "<<*ptr<<endl;

 int* ptr2 = new int;

 *ptr2 = -5;
 ptr = ptr2;

 cout<<"ptr = "<<*ptr<<endl;
 cout<<"ptr2 = "<<*ptr2<<endl;

 delete ptr2;
 cout<<"ptr2 = "<<*ptr2<<endl;

 ptr2 = NULL;
 cout<<"ptr = "<<*ptr<<endl;

    return 0;
}
```

Using Pointer and Reference in the parameter of a Function

Using Pointer

```
void change(int* i){
    *i = 30;
}

int main(void){
    int a = 40;
    change(&a);
    cout << a;
    return 0;
}
```

The parameter is a pointer;
Need to pass pointer in the argument by using the & sign

Using Reference

```
void change(int& i){
    i = 30;
}

int main(void){
    int a = 40;
    change(a);
    cout << a;
    return 0;
}
```

The parameter is a reference;
Can use the variable name directly

Remember?

- A list is a homogeneous collection of elements, with a linear relationship between elements.
- Each list element (except the first) has a unique predecessor, and
- each element (except the last) has a unique successor.