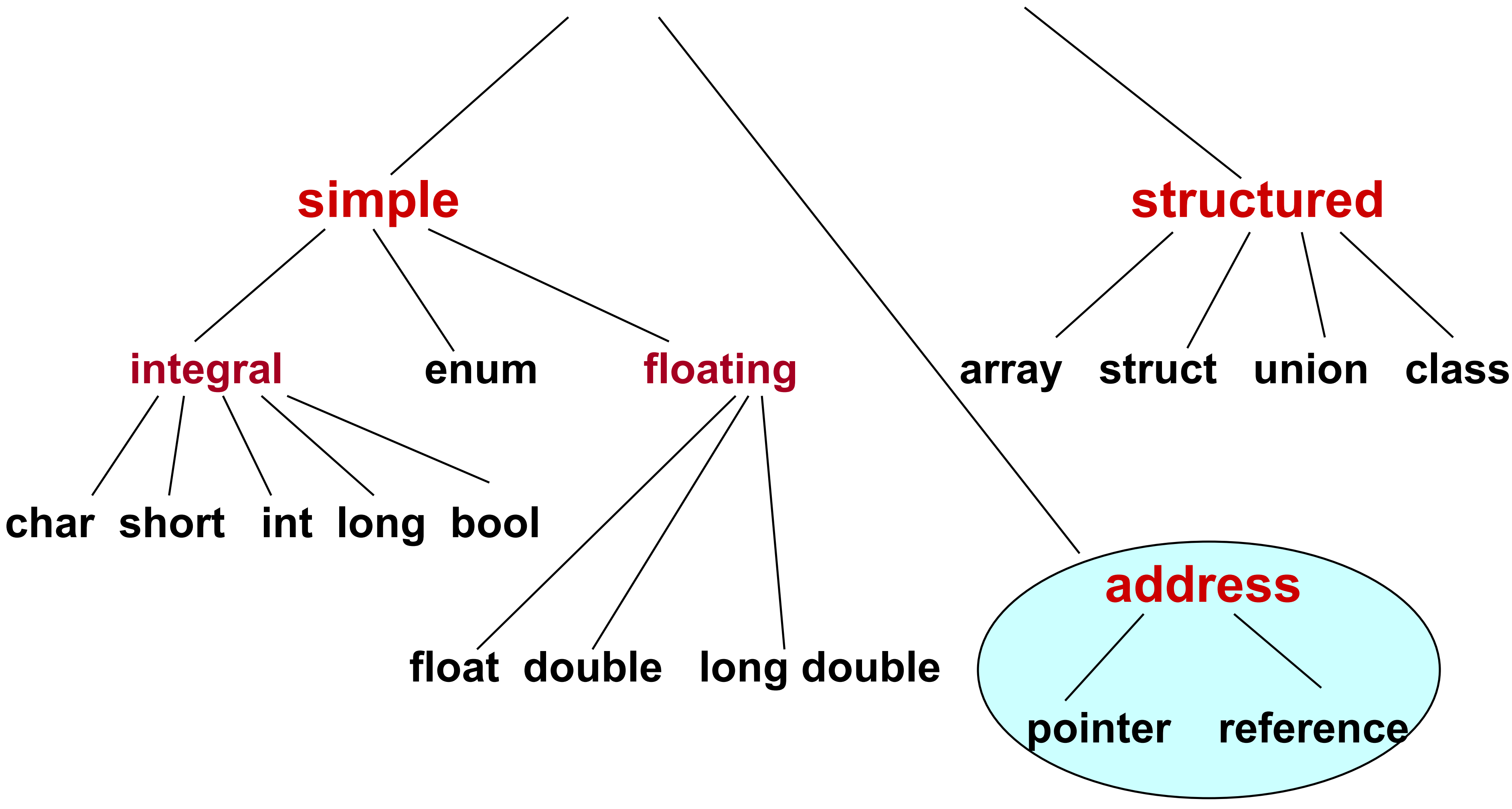
Introduction to Object Oriented Programming

Lecture 1

Pointers, Dynamic Data, Reference Types

Dr. Shamal AL-Dohuki
saldohuk@uod.ac

C++ Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
address
float
double
long double
pointer
reference

Recall that . . .

char str [8];

`str` is the base address of the array. We say `str` is a pointer because its value is an address. It is a pointer constant because the value of `str` itself cannot be changed by assignment. It "points" to the memory location of a `char`.

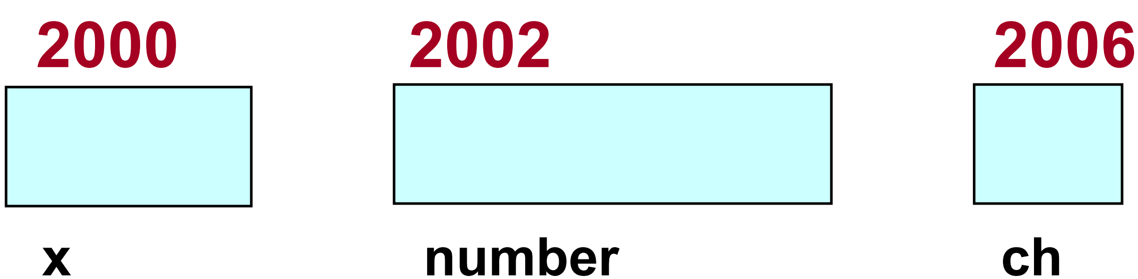
6000

'H'	'e'	'l'	'l'	'o'	'\0'		
`str` [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Addresses in Memory

- **when a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable**

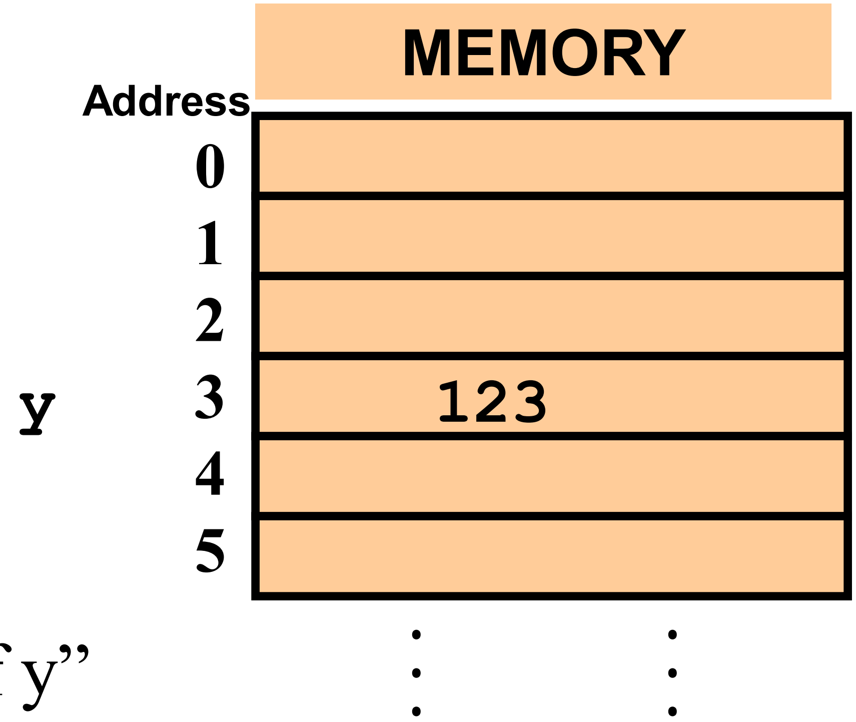
```
int     x;
float   number;
char    ch;
```



&y

In C++ you can get the *address* of a variable with the “&” operator.

```
int y;

y = 123;
cout<< &y;
```



&y means “the address of y”

Obtaining Memory Addresses

- **the address of a non-array variable can be obtained by using the address-of operator &**

```
int      x;
float    number;
char     ch;

cout << "Address of x is " << &x << endl;

cout << "Address of number is " << &number << endl;

cout << "Address of ch is " << &ch << endl;
```

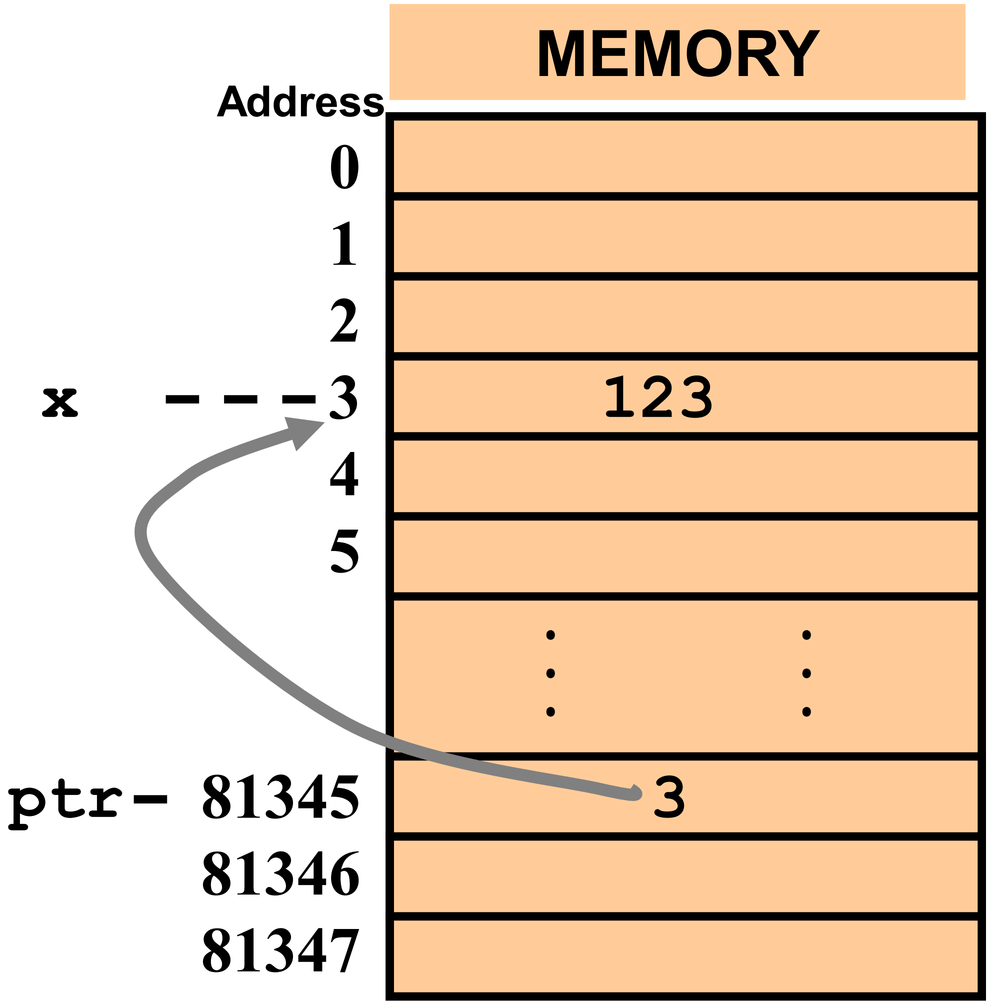
What is a pointer variable?

- 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,

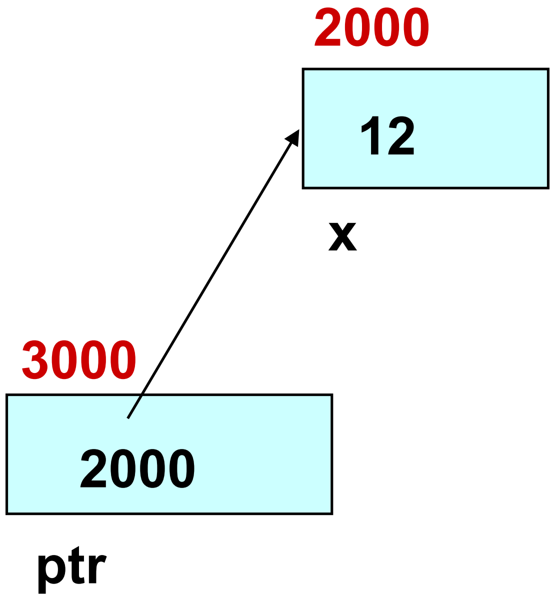
```
int*    ptr; // ptr will hold the address of an int

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

```
int x;
int *ptr;

x = 123;
ptr = &x;
```



Using a Pointer Variable

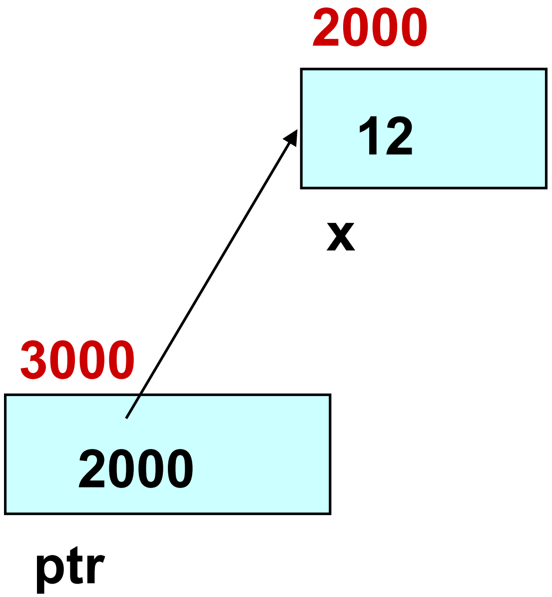
```
int  x;
x = 12;


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



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

Unary operator * is the indirection (deference) operator

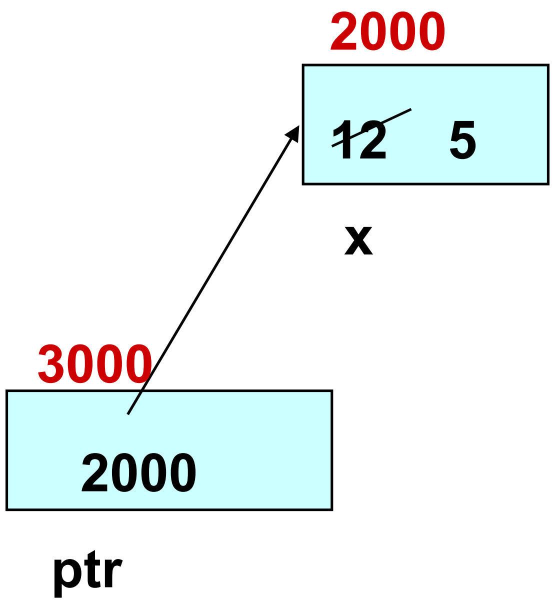
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

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 address ptr to 5
```

Assigning a value to a *dereferenced* pointer

A pointer must have a value before you can *dereference* it (follow the pointer).

```
int *x;
*x=3;
```

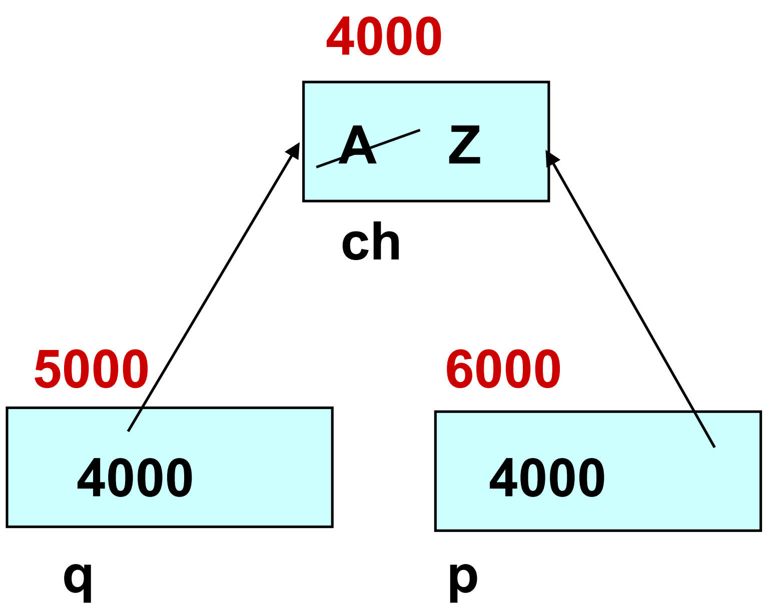
ERROR!!!
x doesn't point to anything!!!

```
int foo;
int *x;
x = &foo;
*x=3;
```

this is fine
x points to foo

Another Example

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

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;
```




```
// the rhs has value 4000
// now p and q both point to ch
```

Pointers and Arrays

An array name is basically a *const* pointer.

You can use the [] operator with a pointer:

```
int *x;
int a[10];
x = &a[2];
for (int  i=0;i<3;i++)
    x[i]++;
```

`x` is “the address of **`a[2]`** ”

`x[i]` is the same as **`a[i+2]`**

Arrays and Pointers

An array name is actually a pointer to the 0th element of the array

*x = 4;   // assigns 0th element

4	15	8		10
x[0]	x[1]	x[2]	x[3]	x[4]

Arrays and Pointers

Adding the integer value 3 to the base address references the 3rd element of the array

```
*(x+3) = 5;        // these statements
  x[3] = 5;        // do the same thing
```

4	15	8	5	10
x[0]	x[1]	x[2]	x[3]	x[4]

Arrays and Pointers

Assigns the entire array to 0’s:

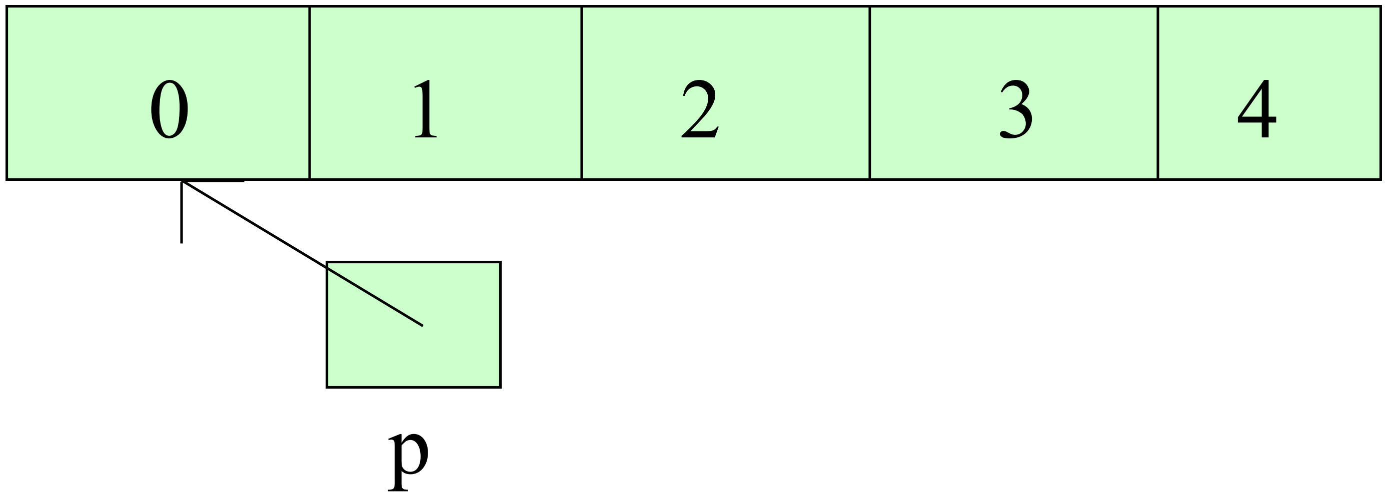
```
for(int *p=x, int cnt=0; cnt<5; cnt++)
   *p++=0;
```

4	15	12	5	10
x[0]	x[1]	x[2]	x[3]	x[4]

Arrays and Pointers

You can dynamically allocate an entire array:

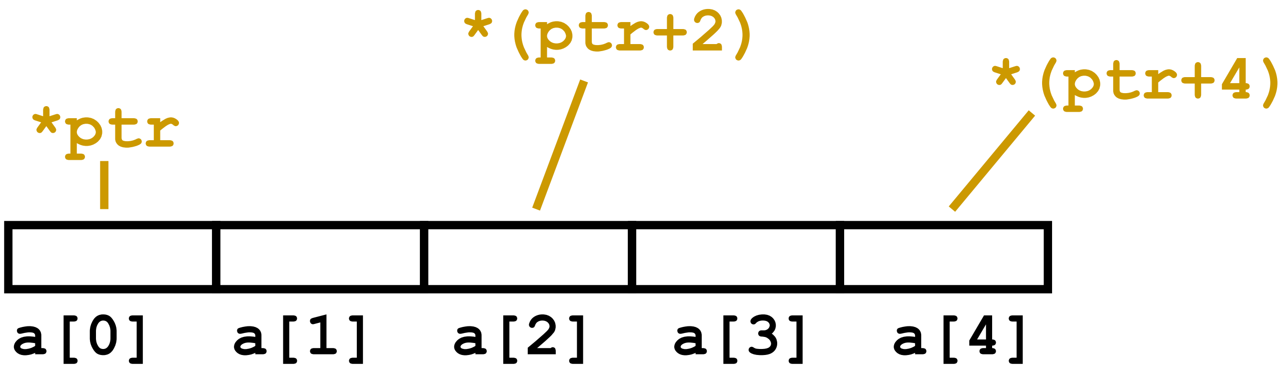
```
int *p = new int[5];
for(int cnt=0; cnt<5; cnt++)
   *(p+cnt)=cnt;
```



Pointer arithmetic

- Integer math operations can be used with pointers.
- If you increment a pointer, it will be increased by the size of whatever it points to.

```
int *ptr = a;
```




```
int a[5];
```

printing an array

```
void print_array(int a[], int len) {
   for (int i=0;i<len;i++)
        cout << "[" << i << "] = "
               << a[i] << endl;
}
```

array version

```
void print_array(int *a, int len) {
   for (int i=0;i<len;i++)
        cout << "[" << i << "] = "
               << *a++ << endl;
}
```

pointer version

Program Data

- **STATIC DATA**: memory allocation exists throughout execution of program

```
static long currentSeed;
```

- **AUTOMATIC DATA**: automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function

- **DYNAMIC DATA**: explicitly allocated and deallocated during program execution by C++ instructions written by programmer using operators `new` and `delete`

Allocation of Memory

STATIC ALLOCATION	DYNAMIC ALLOCATION
Static allocation is the allocation of memory space at compile time.	**Dynamic allocation is the allocation of memory space at run time by using operator new.**

Some C++ Pointer Operations

Precedence

Precedence		Operator	Description
Higher		->	**Select member of class pointed to**
	Unary:	++ -- ! * new delete	**Increment, Decrement, NOT, Dereference, Allocate, Deallocate**
		+ -	Add Subtract
		< <= > >=	Relational operators
		== !=	Tests for equality, inequality
Lower		=	Assignment

Operator new Syntax

```
new   DataType
```

```
new   DataType  [IntExpression]
```

If memory is available, in an area called the heap (or free store) new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.

Otherwise, program terminates with error message.

The dynamically allocated object exists until the delete operator destroys it.

The NULL Pointer

There is a pointer constant 0 called the "null pointer" denoted by NULL in header file cstddef.

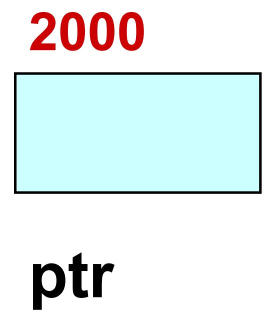
But NULL is not memory address 0.

NOTE: It is an error to dereference a pointer whose value is NULL. Such an error may cause your program to crash, or behave erratically. It is the programmer's job to check for this.

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

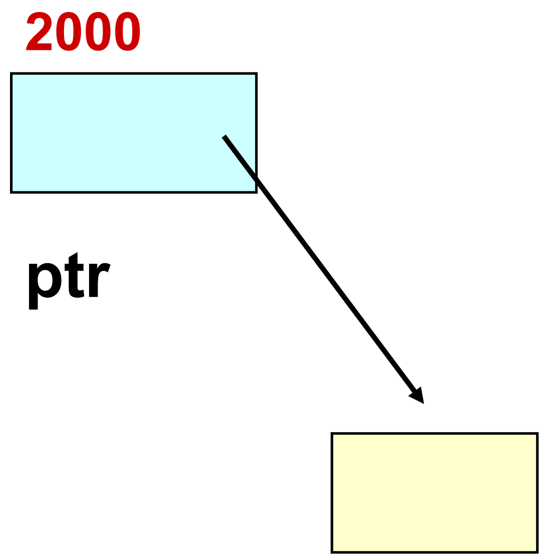
Dynamically Allocated Data

```
char*  ptr;

ptr = new char;

*ptr = 'B';

cout  <<  *ptr;
```



Dynamically Allocated Data

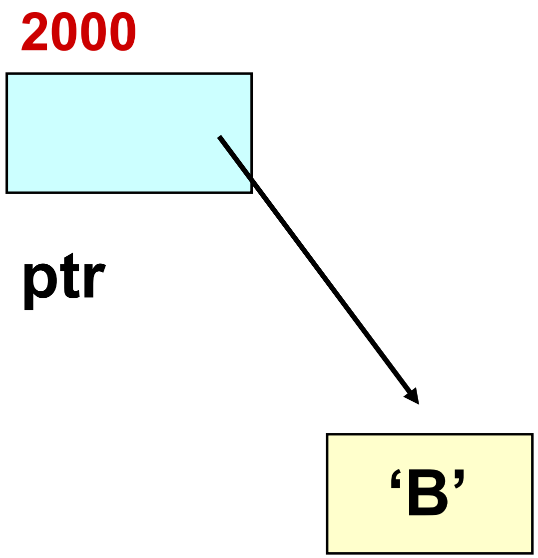
```
char*  ptr;

ptr = new char;
*ptr = ‘B’;

cout  <<  *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

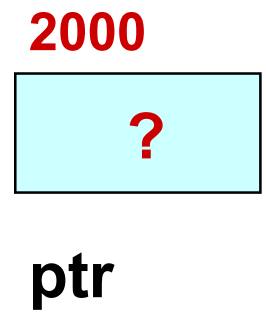
```
char*  ptr;

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



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;

ptr = new char;
*ptr = 'B';
cout  <<  *ptr;
delete  ptr;
```



NOTE: delete deallocates the memory pointed to by ptr

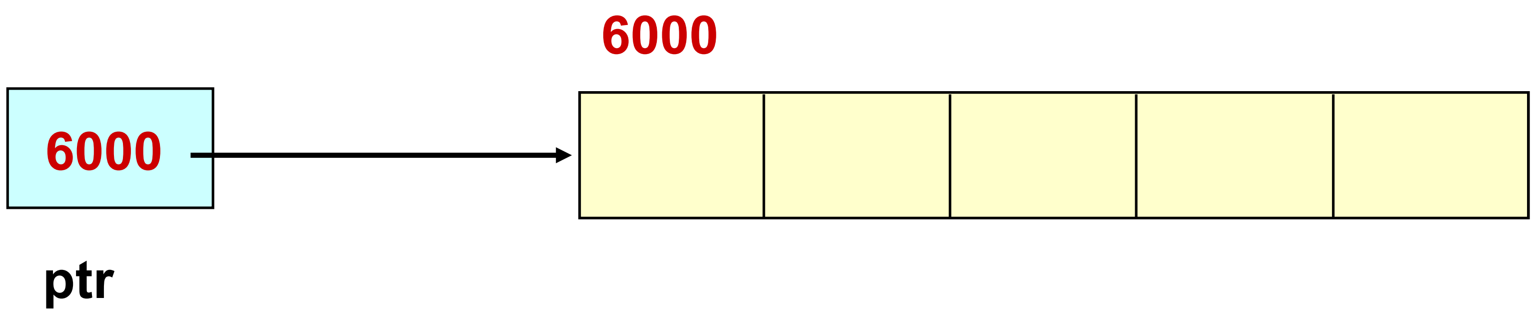
Using Operator `delete`

Operator delete returns to the free store memory which was previously allocated at run-time by operator new.

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

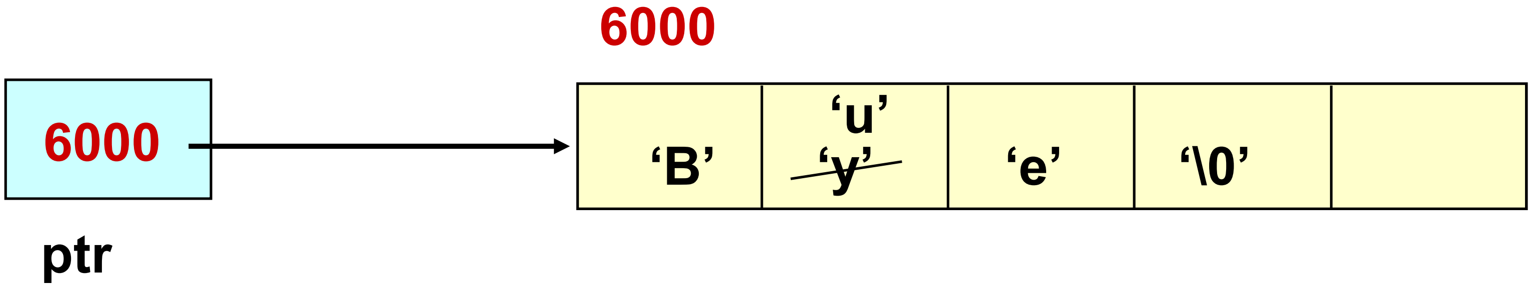
Dynamic Array Allocation

```
char  *ptr;          //  ptr is a pointer variable that
                     //  can hold the address of a char

ptr  =  new  char[ 5 ];

        // dynamically, during run time, allocates
        // memory for a 5 character array
        // and stores the base address into ptr
```



Dynamic Array Allocation

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, “Bye” );

ptr[ 1 ] = ‘u’;              // a pointer can be subscripted

cout  << ptr[ 2] ;
```



Operator `delete` Syntax

```
delete    Pointer
```

```
delete  [ ]    Pointer
```

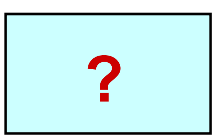
If the value of the pointer is 0 there is no effect.

Otherwise, the object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined. The memory is returned to the free store.

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

Dynamic Array Deallocation

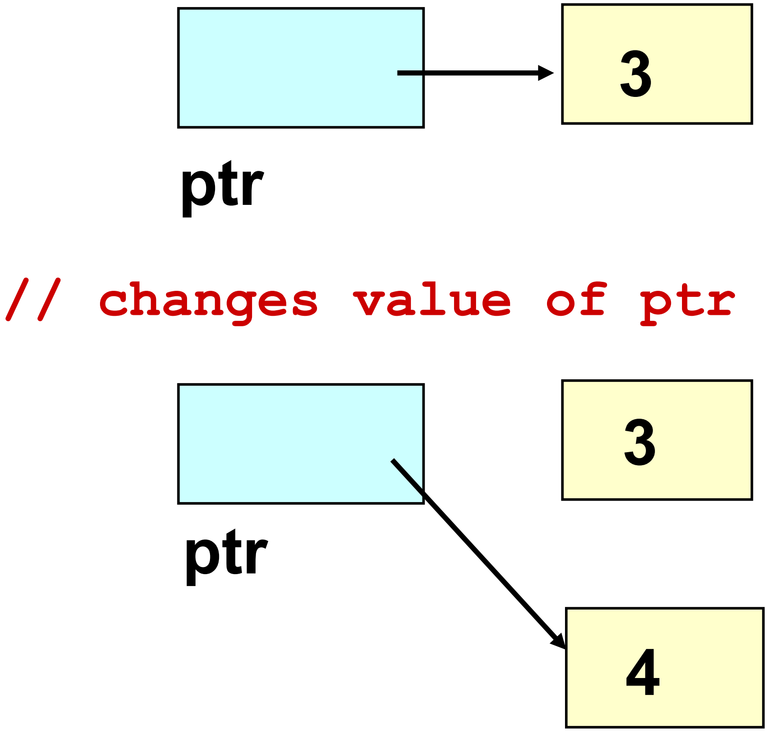
```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, “Bye” );

ptr[ 1 ] = ‘u’;

delete  ptr;    // deallocates array pointed to by ptr
                // ptr itself is not deallocated
                // the value of ptr is undefined.
```



ptr

What happens here?

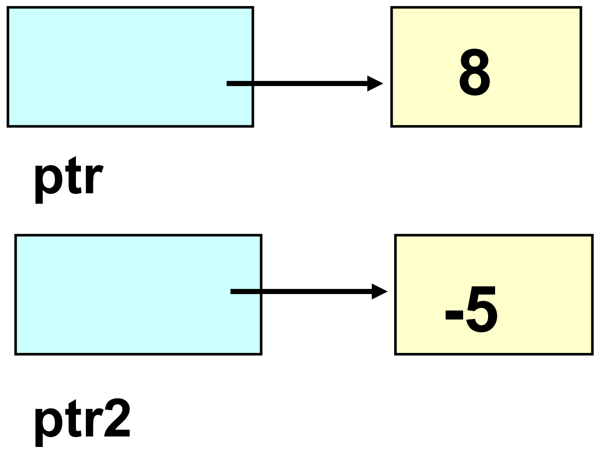
```
int* ptr = new int;
*ptr = 3;

ptr = new int;
*ptr = 4;
```



Inaccessible Object

An inaccessible object is an unnamed object that was created by operator `new` and which a programmer has left without a pointer to it.

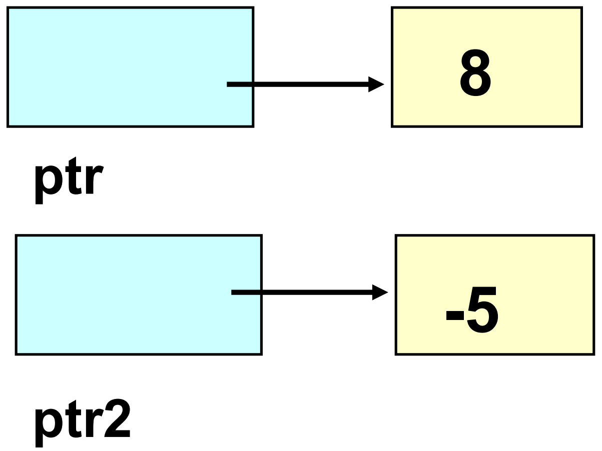
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



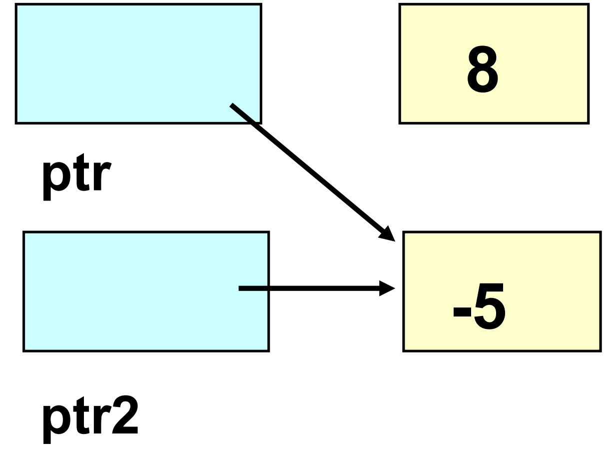
How else can an object become inaccessible?

Making an Object Inaccessible

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

ptr = ptr2;
```



// here the 8 becomes inaccessible



Memory Leak

A memory leak is the loss of available memory space that occurs when dynamic data is allocated but never deallocated.

Memory Leak

```
typedef int* intptr;
void main ()
{
intptr P, Q;
P = new int;*P = 1
Q = new int;*Q = 2;
cout << *P << ' ' << *Q << endl;
*P = *Q + 3;
cout << *P << ' ' << *Q << endl;
P = Q;
cout << *P << ' ' << *Q << endl;
}
```

Memory leak!

Memory Leak

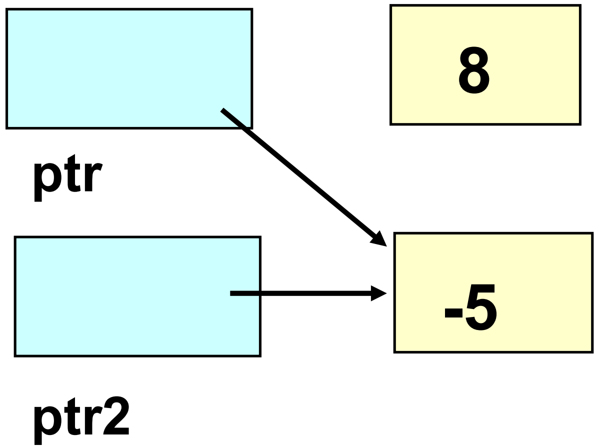
```
#include <iostream.h>
typedef int* intptr;
void main () {
intptr ptr1;
ptr1 = new int;
*ptr1 = 12345;
delete ptr1;
ptr1 = NULL;
ptr1 = new int;
cout << *ptr1 << endl;
}
```

```
#include <iostream.h>
typedef int* intptr;
void main () {
intptr ptr1, ptr2;
ptr1 = new int;
*ptr1 = 12345;
delete ptr1;
ptr1 = NULL;
ptr2 = new int;
ptr1 = new int;
cout << *ptr1 << endl;
}
```

A Dangling Pointer

- is a pointer that points to dynamic memory that has been deallocated

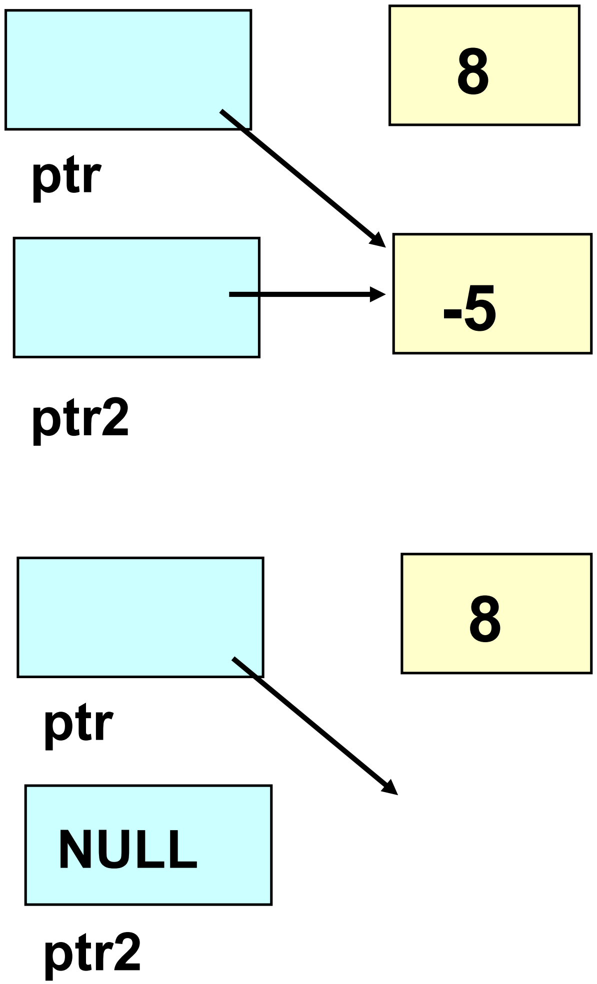
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```



FOR EXAMPLE,

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;
```



const Pointers

You can use the keyword const for pointers before the type, after the type, or in both places. For example, all of the following are legal declarations:

```
const int * pOne;
int * const pTwo;
const int * const pThree;
```

pOne is a pointer to a constant integer. The value that is pointed to can't be changed.

pTwo is a constant pointer to an integer. The integer can be changed, but pTwo can't point to anything else.

pThree is a constant pointer to a constant integer. The value that is pointed to can't be changed, and pThree can't be changed to point to anything else.

The trick to keeping this straight is to look to the right of the keyword const to find out what is being declared constant. If the type is to the right of the keyword, it is the value that is constant. If the variable is to the right of the keyword const, it is the pointer variable itself that is constant.

```
const int * p1;  // the int pointed to is constant
int * const p2;  // p2 is constant, it can't point to anything else
```

The End