C++ Review

CS595

Fall, 2010

Software Engineering

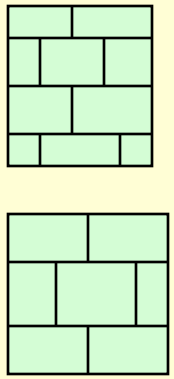
- A disciplined approach to the design, production, and maintenance of computer programs
- that are developed on time and within cost estimates,
- using tools that help to manage the size and complexity of the resulting software products.

Software from 3 different levels

- ***Application (or user) level:*** **modeling real-life data in a specific context.**
- ***Logical (or ADT) level:*** **abstract view of the domain and operations.** **WHAT**
- ***Implementation level:*** **specific representation of the structure to hold the data items, and the coding for operations.** **HOW**

Interface:

Common boundary between two distinct entities



client

implementation

interface

C++: client.cpp header file

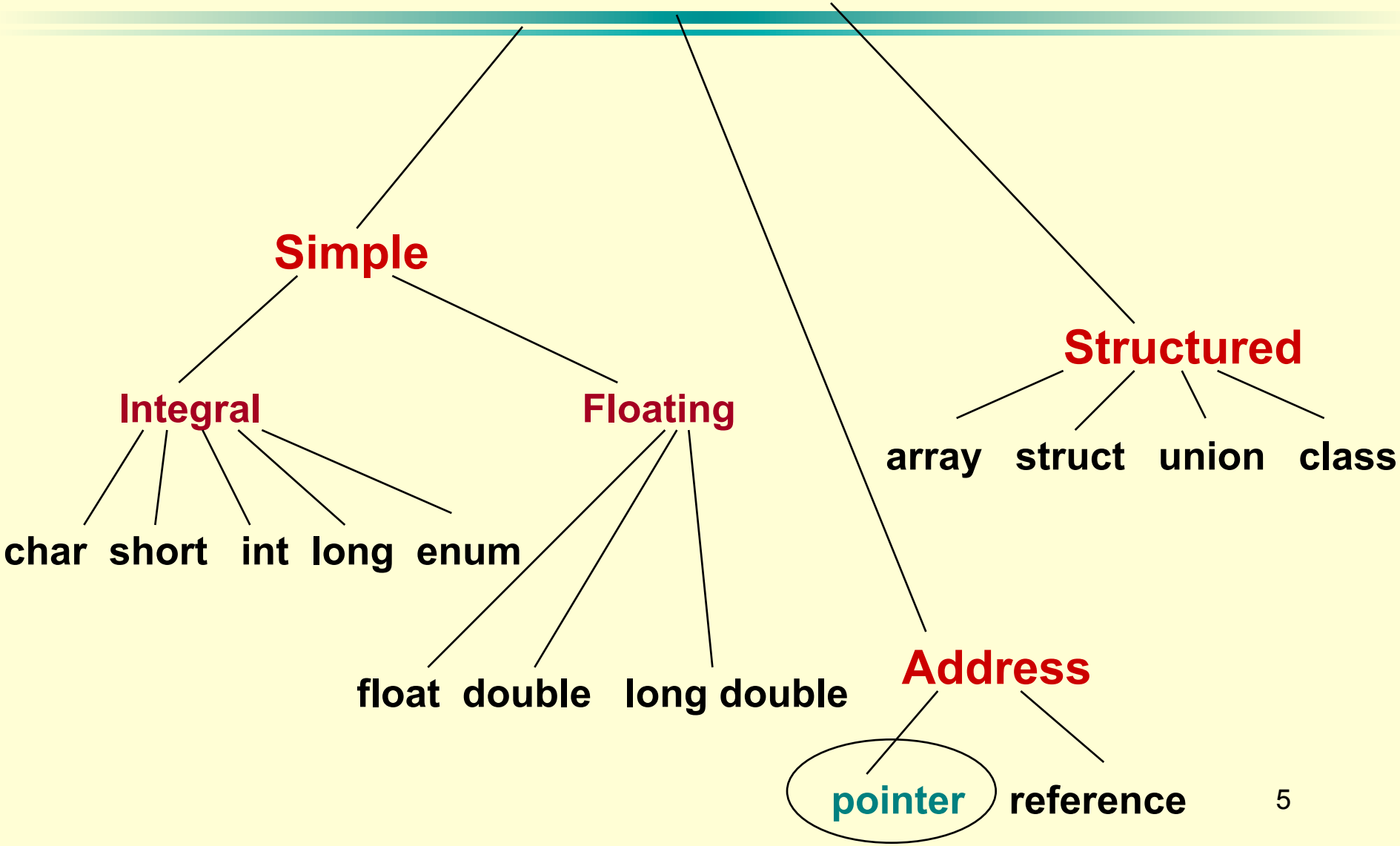
*.h *.cpp

~petsc-dev/src/ksp/ksp/ksp/examples/tutorials/ex2.c

petscksp.h

impls/*

C++ Data Types

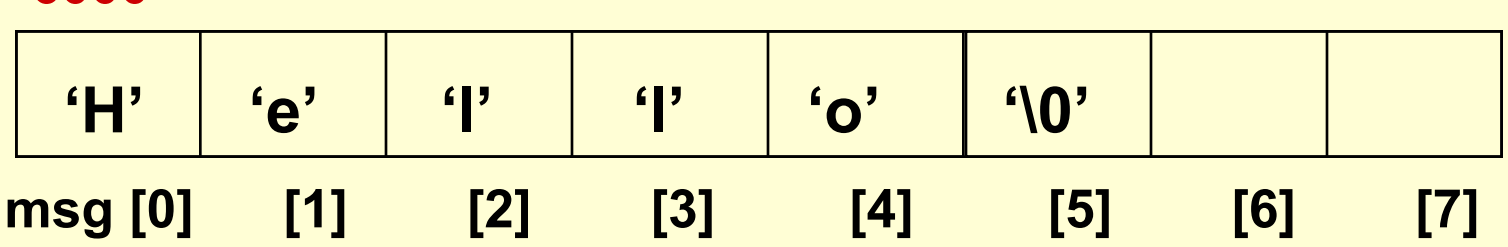

Simple
Integral
Floating
char
short
int
long
enum
float
double
long double
Structured
array
struct
union
class
Address
pointer
reference

Pointer Types

Recall that …

char msg [8];

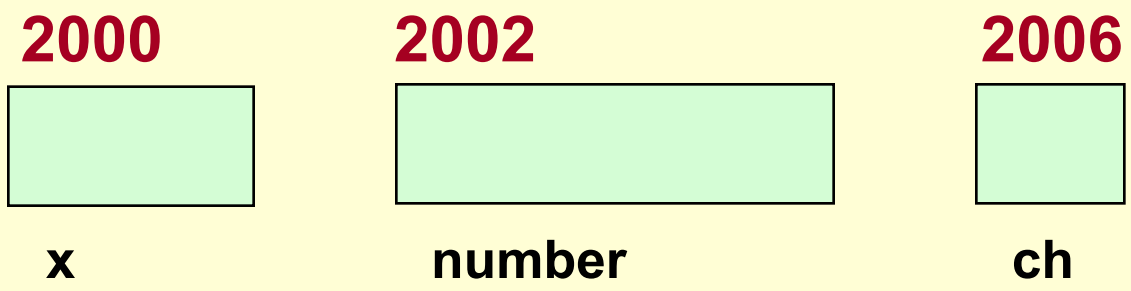
msg is the base address of the array. We say msg is a pointer because its value is an address. It is a pointer constant because the value of msg itself cannot be changed by assignment. It "points" to the memory location of a `char`.



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;
```



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;
```

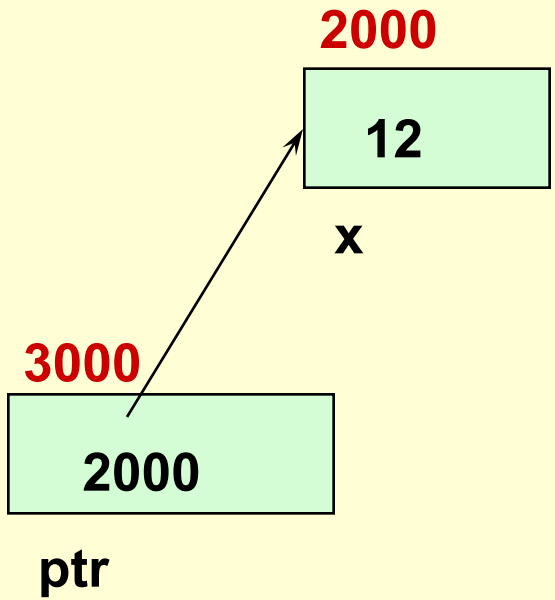
What is a pointer variable?

- 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
```

Using a pointer variable

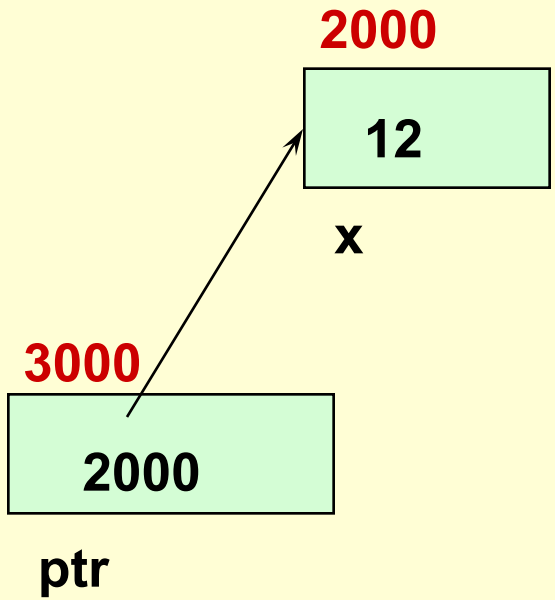
```
int  x;
x = 12;


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



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

Unary operator * is the dereference operator

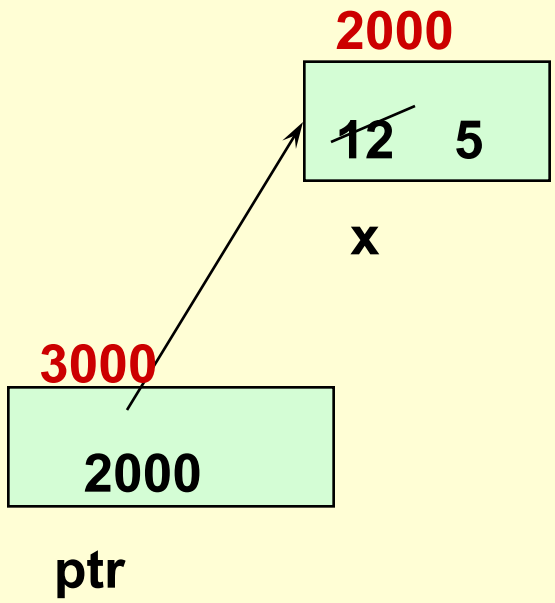
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

cout  <<  *ptr;
```



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

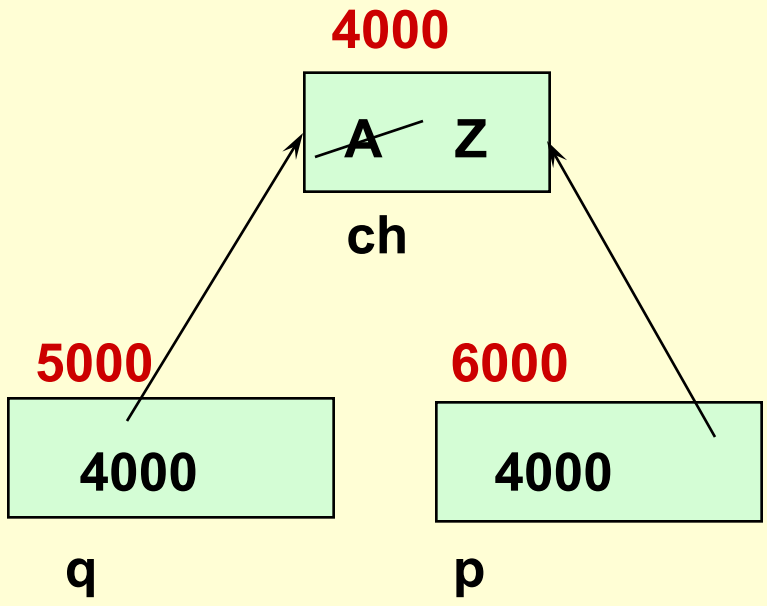
Using the dereference operator

```
int  x;
x = 12;


int*  ptr;
ptr = &x;

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



The NULL Pointer

There is a pointer constant 0 called the "null pointer" denoted by NULL in stddef.h

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
}
```

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.

3 Kinds of Program Data

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

```
static long SeedValue;
```

- **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 unary operators `new` and `delete`**

Using operator new

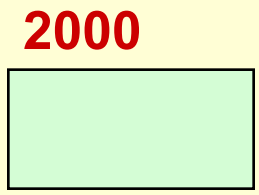
If memory is available in an area called the free store (or heap), operator new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.

Otherwise, the null pointer 0 is returned.

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

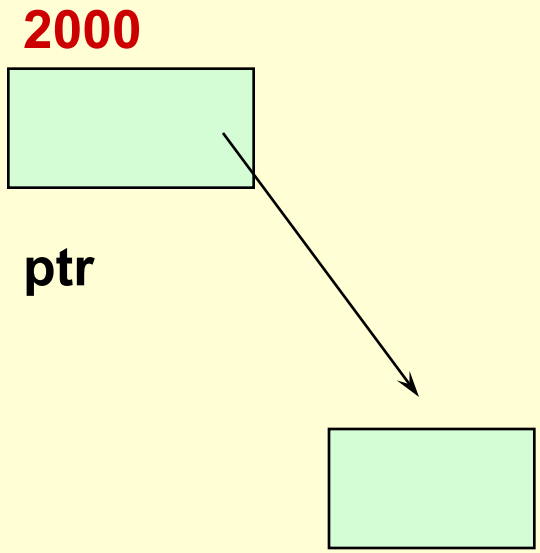
Dynamically Allocated Data

```
char*  ptr;


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



Dynamically Allocated Data

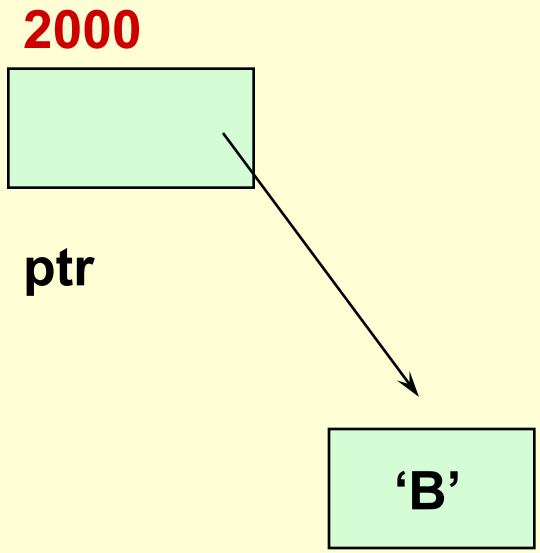
```
char*  ptr;

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



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

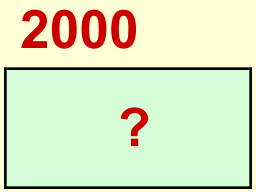
```
char*  ptr;


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`

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.

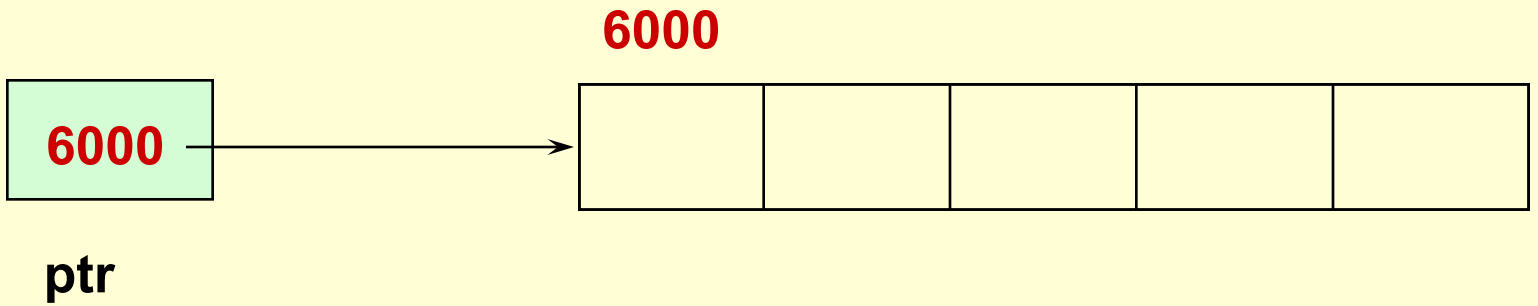
Some C++ pointer operations

Precedence

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

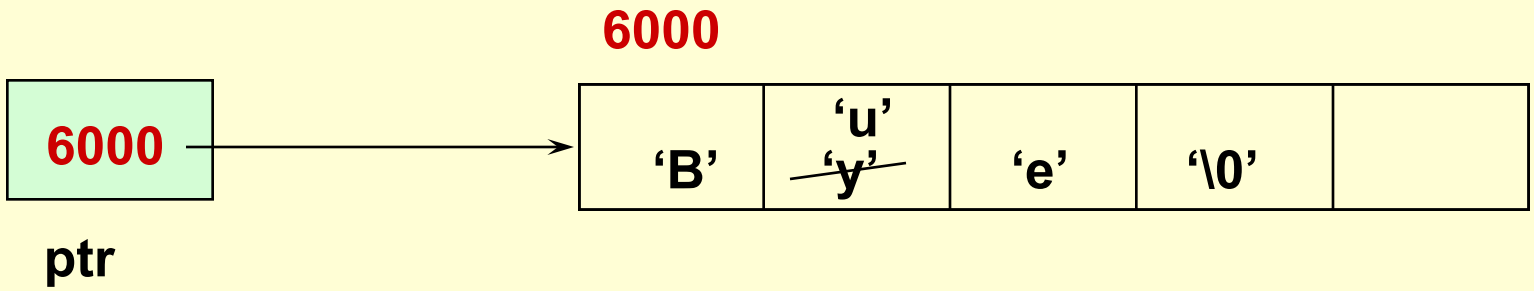
Dynamic Array Allocation

```
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 5 characters and places into
            // the contents of ptr their beginning address
```



Dynamic Array Allocation

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, "Bye" );

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

cout  << ptr[ 2] ;
```



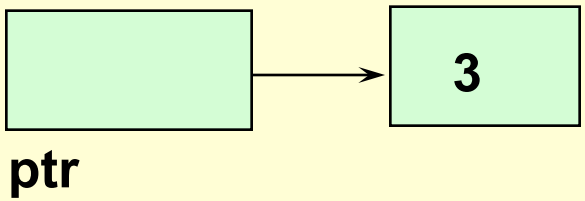
Dynamic Array Deallocation

```
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, but
                    // the value of ptr is considered unassigned
```

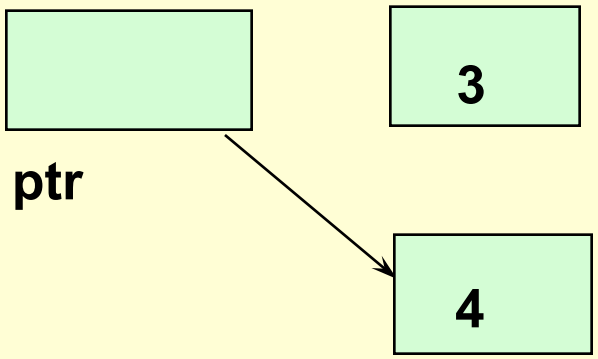
ptr

What happens here?

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



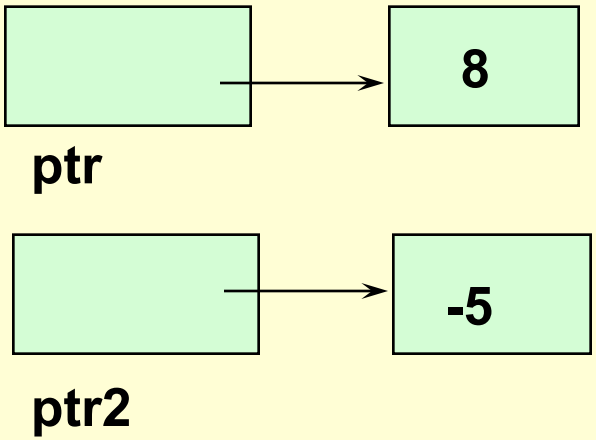

```
ptr = new int;     // changes value of ptr
*ptr = 4;
```



Memory Leak

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

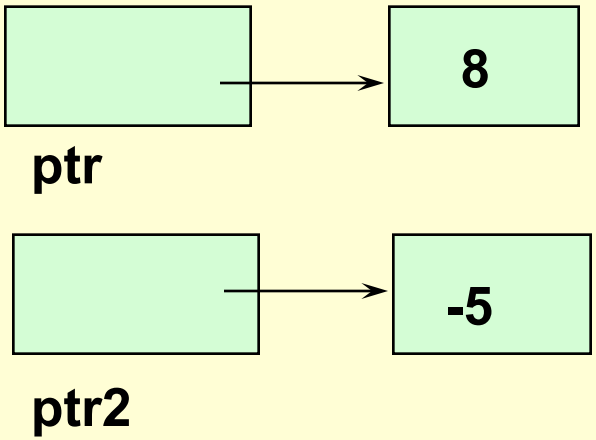
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



How else can an object become inaccessible?

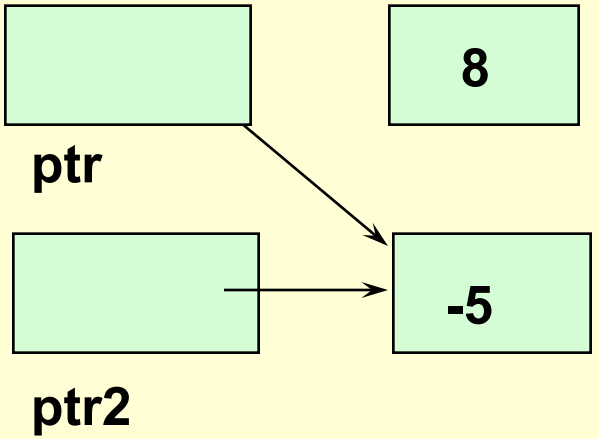
Causing a Memory Leak

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




```
ptr = ptr2;
```

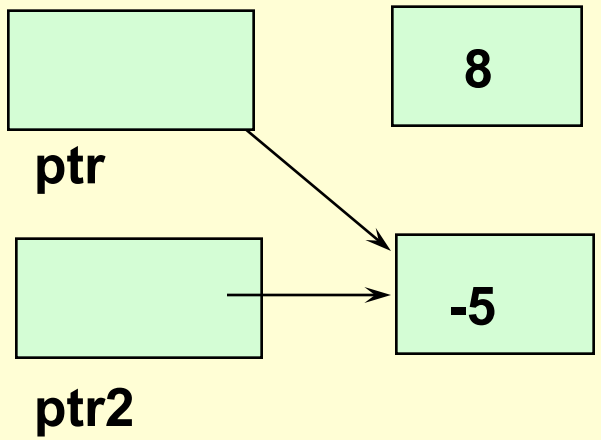
// here the 8 becomes inaccessible



A Dangling Pointer

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

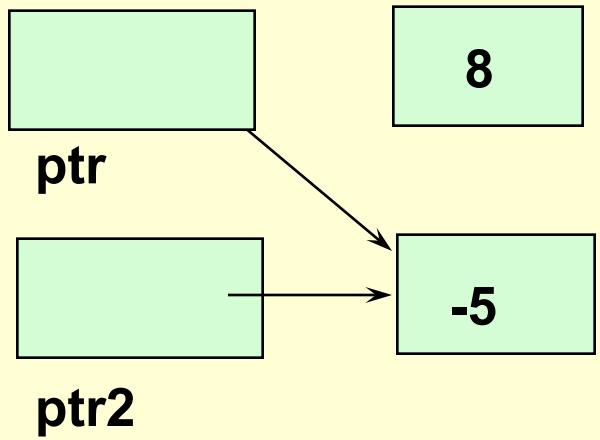
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```

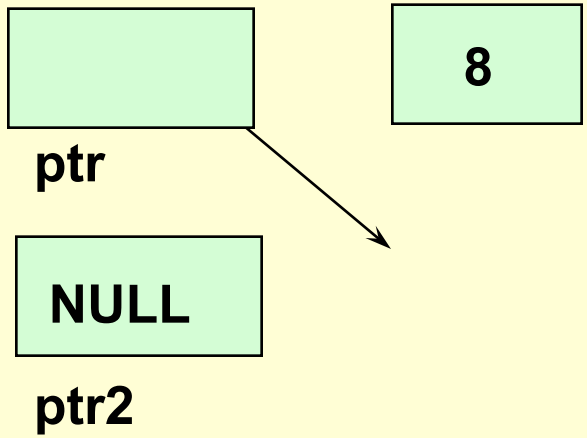


FOR EXAMPLE,

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;
```





Valid struct operations

- Operations valid on an entire struct type variable:

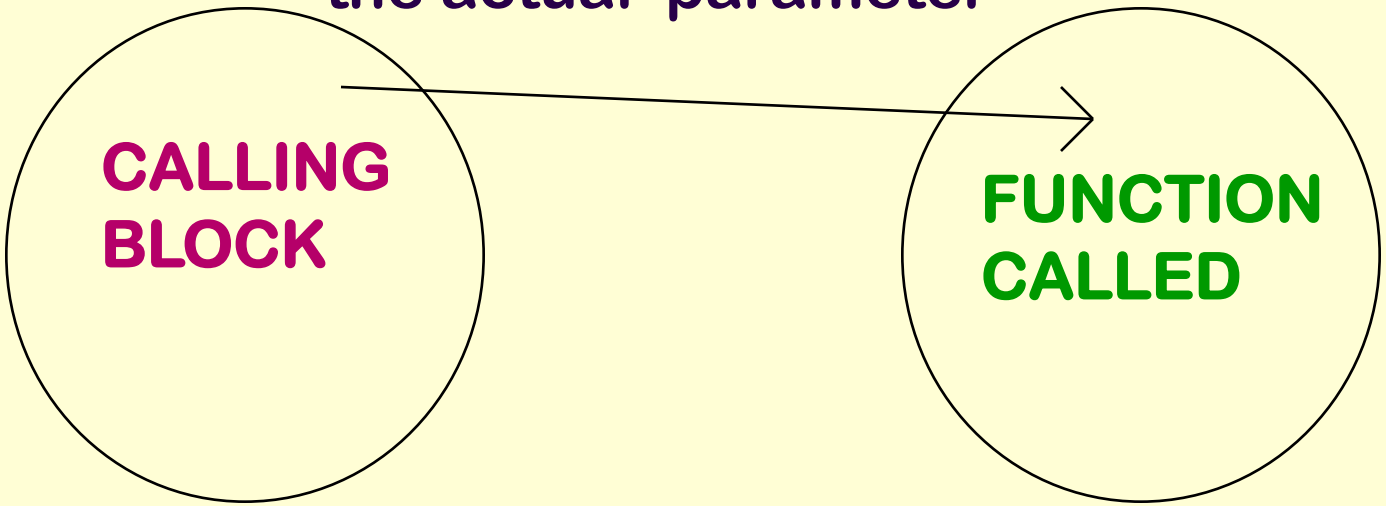
  assignment to another struct variable of same type,

  pass as a parameter to a function
  (either by value or by reference),

  return as the value of a function.

Pass-by-value

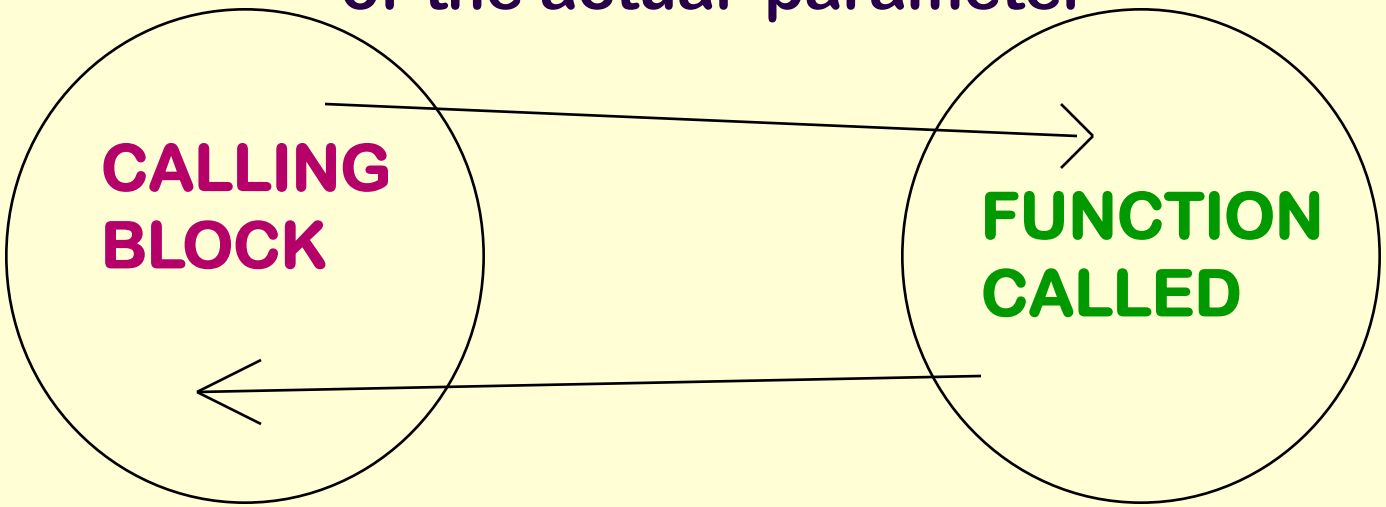
sends a *copy*
of the contents of
the actual parameter



So,
the actual parameter cannot be changed by the function.

Pass-by-reference

sends the *location*
(memory address)
of the actual parameter



can change value of
actual parameter

Using struct type
Reference Parameter to change a member

```
void  AdjustForInflation(CarType&  car,  float  perCent)
// Increases price by the amount specified in perCent
{
        car.price = car.price * perCent + car.price;
} ;
```

SAMPLE CALL

```
AdjustForInflation(myCar, 0.03);
```

Using struct type
Value Parameter to examine a member

```
bool  LateModel(CarType  car,  int date)

//  Returns true if the car's model year is later than or
//  equal to date;  returns false otherwise.
{
        return  ( car.year  >=  date ) ;
} ;
```

SAMPLE CALL

```
if ( LateModel(myCar, 1995) )
   cout << myCar.price << endl ;
```

One-Dimensional Array at the Logical Level

A one-dimensional array is a structured composite data type made up of a finite, fixed size (*known at compile time*) collection of homogeneous (*all of the same data type*) elements having relative positions and to which there is direct access (*any element can be accessed immediately*).

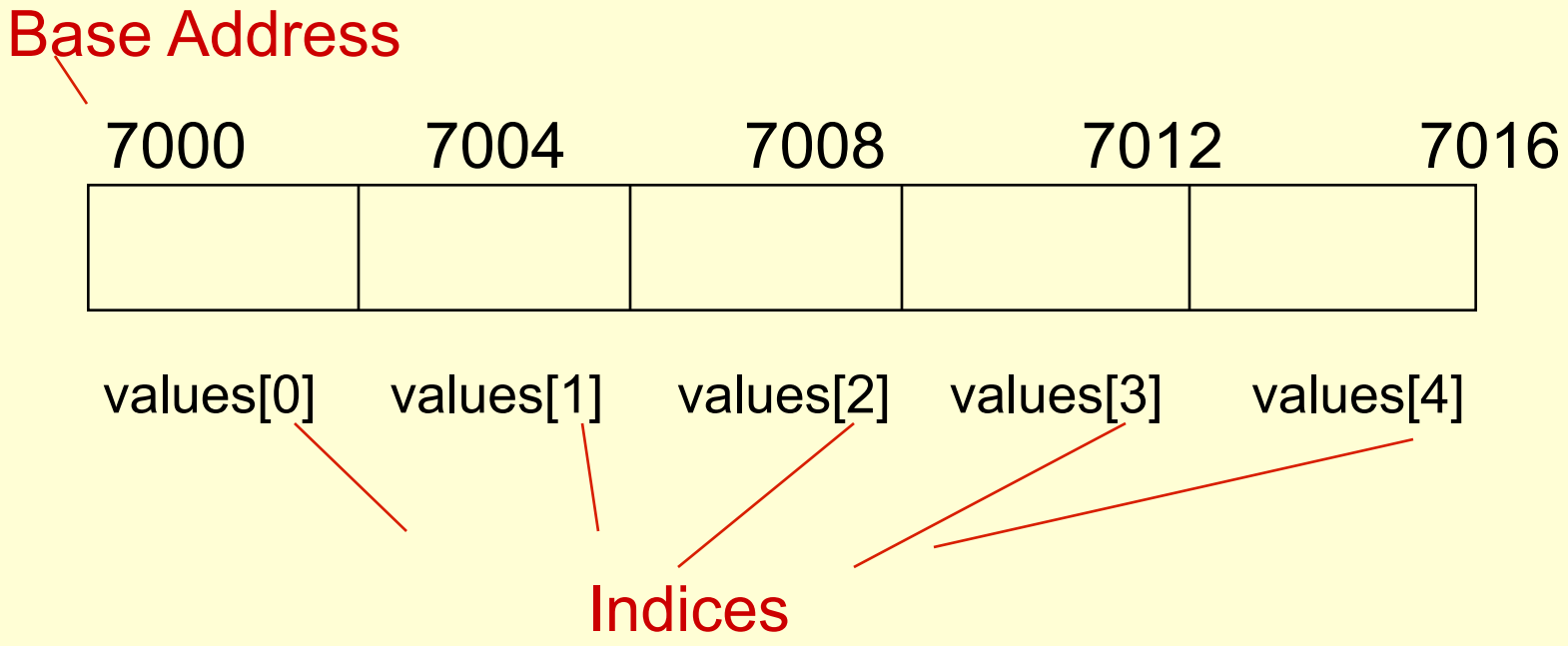
Array operations (*creation, storing a value, retrieving a value*) are performed using a declaration and indexes.

Implementation Example

This ACCESSING FUNCTION gives position of values[Index]

Address(Index) = BaseAddress + Index * SizeOfElement

float values[5]; ***// assume element size is 4 bytes***



One-Dimensional Arrays in C++

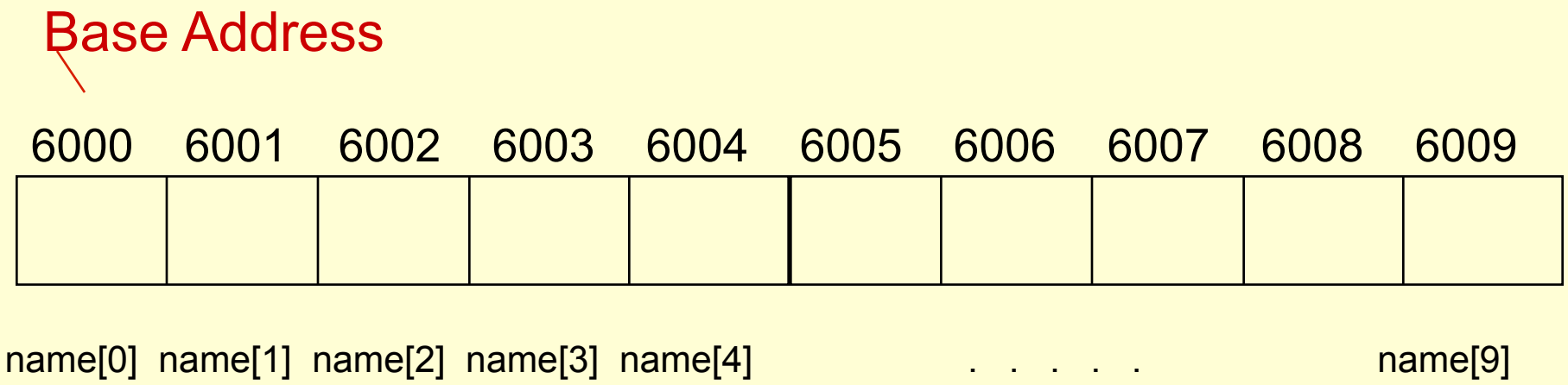
- The index must be of an integral type (char, short, int, long, or enum).
- The index range is always 0 through the array size minus 1.
- Arrays cannot be the return type of a function.

Another Example

This ACCESSING FUNCTION gives position of name[Index]

Address(Index) = BaseAddress + Index * SizeOfElement

char name[10]; ***// assume element size is 1 byte***



Passing Arrays as Parameters

- In C++, arrays are *always* passed by reference, and & is not used with the formal parameter type.

- Whenever an array is passed as a parameter, its base address is sent to the called function.

const array parameter

Because arrays are always passed as reference parameters, you can protect the actual parameter from unintentional changes by using **const** in formal parameter list and function prototype.

FOR EXAMPLE . . .

```
                                        // prototype

float  SumValues(const  float  values[ ],
                        int  numOfValues );
```

```
float  SumValues (const  float  values[ ],
                                 int   numOfValues )
// Pre:   values[ 0] through values[numOfValues-1]
//          have been assigned
// Returns the sum of values[0] through
// values[numOfValues-1]
{
   float  sum  = 0;

    for ( int  index = 0;  index < numOfValues; index++ )
   {
         sum += values [ index ] ;
   }
    return  sum;
}
```

Copy Structure

(see Sec 6.4, Text)

- **Shallow copy:** an operation that copies one class object to another without copying any pointed-to data

- **Deep copy**: an operation that not only copies one class object to another but also makes copies of any pointed-to data