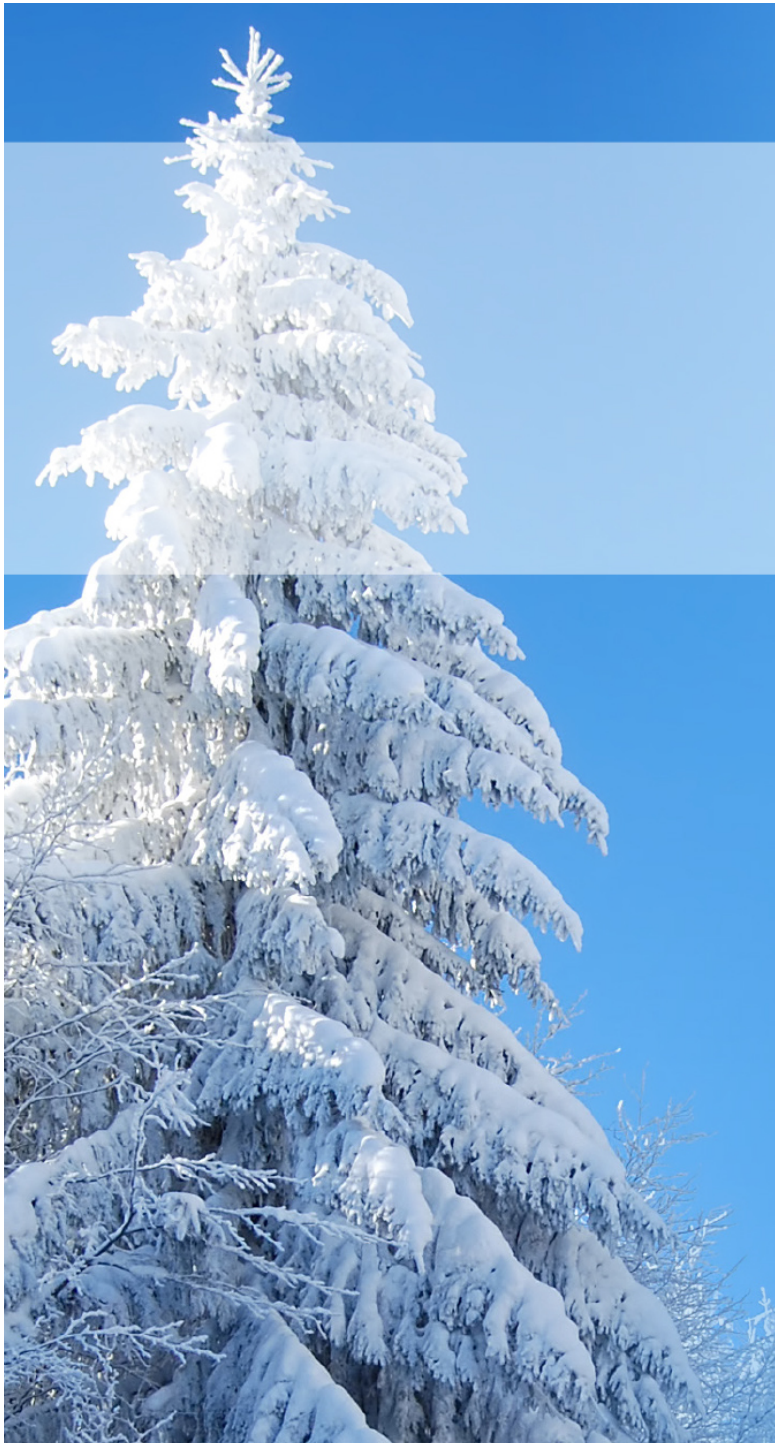


COMPREHENSIVE EDITION

PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 14

Dynamic Data and Linked Lists

Chapter 14 Topics (part one)

- Pointers
- Dynamic memory allocation
- Meaning of an Inaccessible Object
- Meaning of a Dangling Pointer
- Use of a Class Destructor
- Shallow Copy vs. Deep Copy of Class Objects
- Use of a Copy Constructor

Chapter 14 Topics (part two)

- Meaning of a Linked List
- Meaning of a Dynamic Linked List
- Traversal, Insertion and Deletion of Elements in a Dynamic Linked List
- Specification of a Dynamic Linked Sorted List
- Insertion and Deletion of Elements in a Dynamic Linked Sorted List

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`.

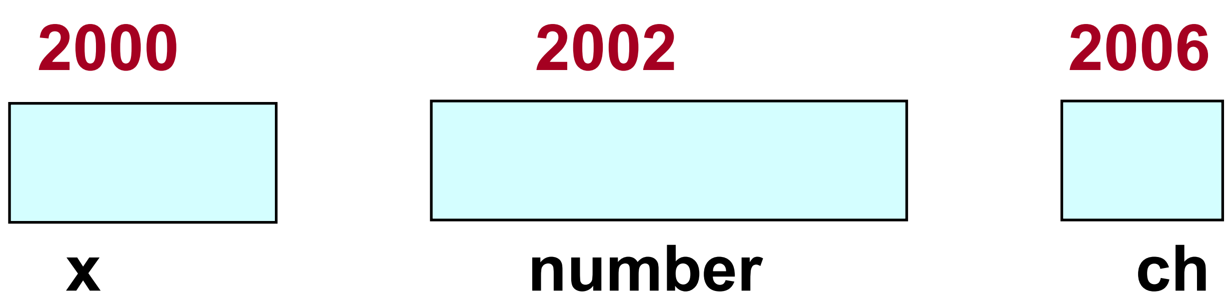
6000

	'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;
```



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;
```

Operator new Syntax

```
new   DataType
```

```
new   DataType  [IntExpression]
```

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

Operator new Syntax, cont...

```
new   DataType
```

```
new   DataType  [IntExpression]
```

Otherwise, program terminates with error message

The dynamically allocated object exists until the delete operator destroys it

The `NULL` Pointer

NULL is a pointer constant 0, defined in header file cstddef, that means that the pointer points to nothing

The NULL Pointer

- It is an error to dereference a pointer whose value is NULL
- Such an error may cause your program to crash, or behave erratically

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

3 Kinds of 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**

3 Kinds of Program Data

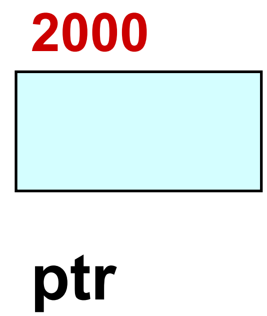
- **Dynamic data:** explicitly allocated and deallocated during program execution by C++ instructions written by programmer using operators `new` and `delete`

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**

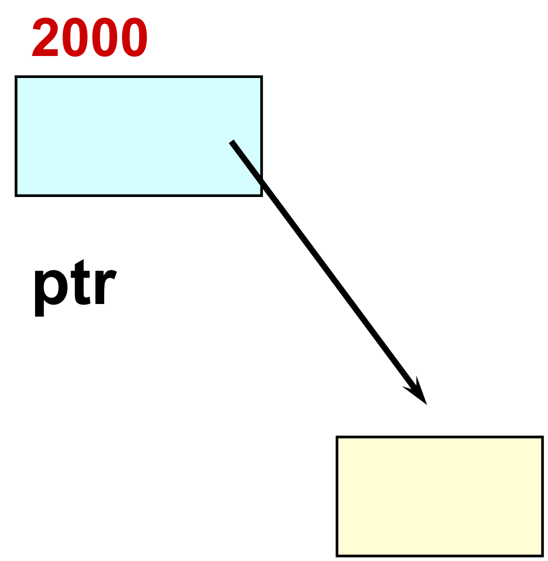
Dynamically Allocated Data

```
char*  ptr;

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



Dynamically Allocated Data

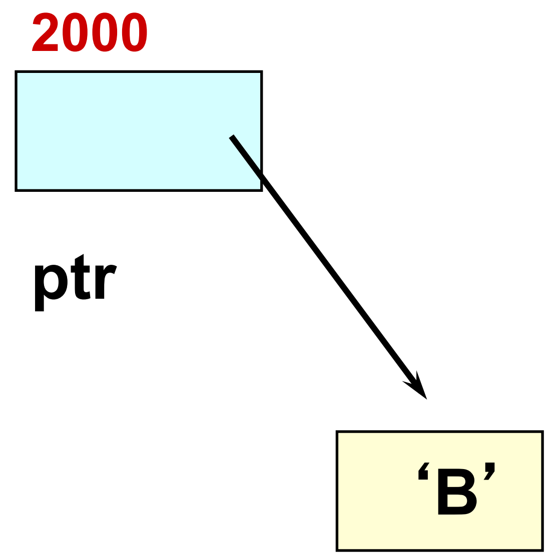
```
char*  ptr;

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;
```

2000

?

ptr

NOTE: delete deallocates the memory pointed to by ptr

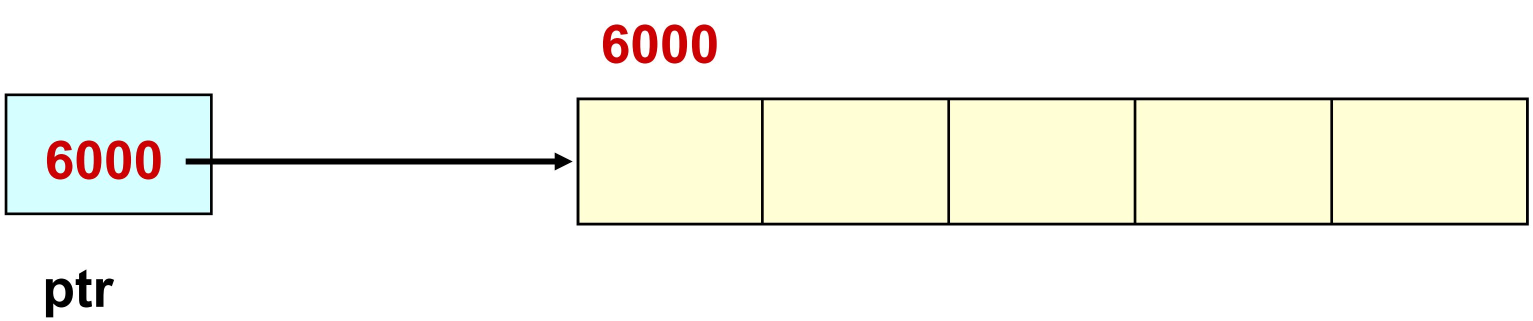
Using Operator `delete`

- Operator delete returns memory to the free store, 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

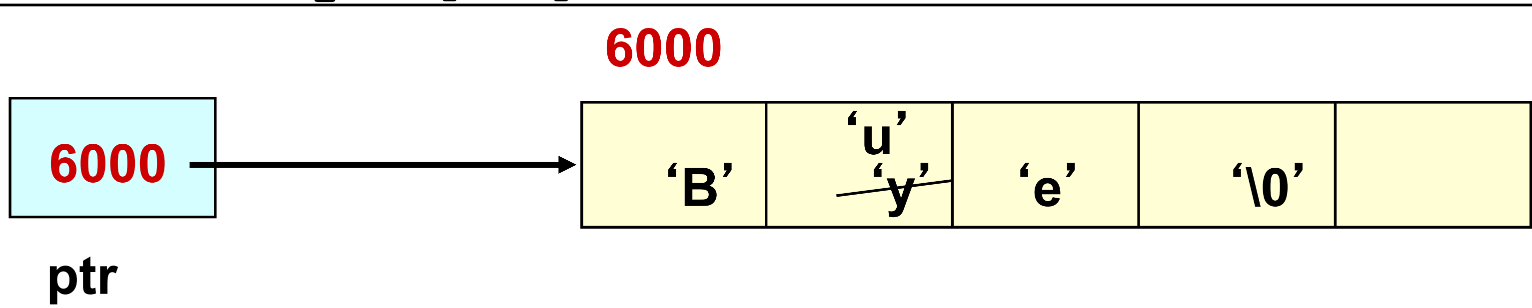
Dynamic Array Allocation

```
char *ptr;// ptr is a pointer variable that
          //  can hold the address of a char
ptr  =  new  char[ 5 ];
// Allocates memory for a 5-character
array //  dynamically at run time 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 NULL there is no effect

Otherwise, the object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined

Operator `delete` Syntax, cont...

delete Pointer

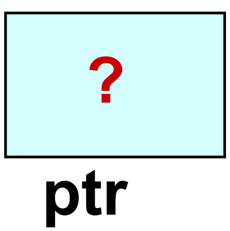
delete [ ] Pointer

The memory is returned to the free store

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

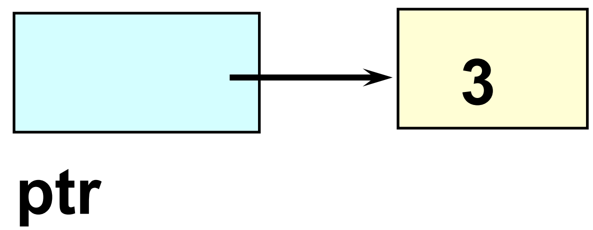
Dynamic Array Deallocation

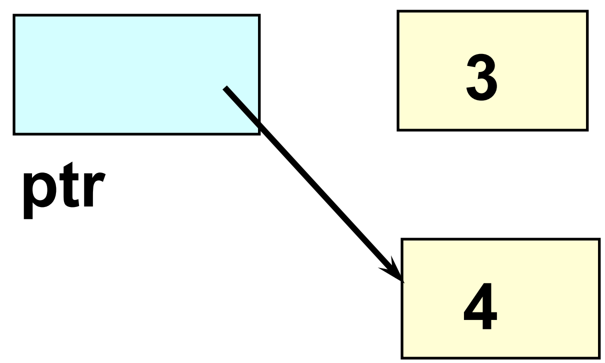
```
char *ptr;

 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
```



What happens here?

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

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

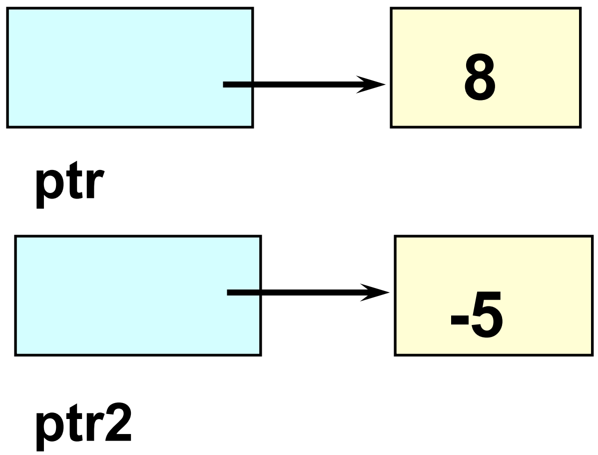




Inaccessible Object

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

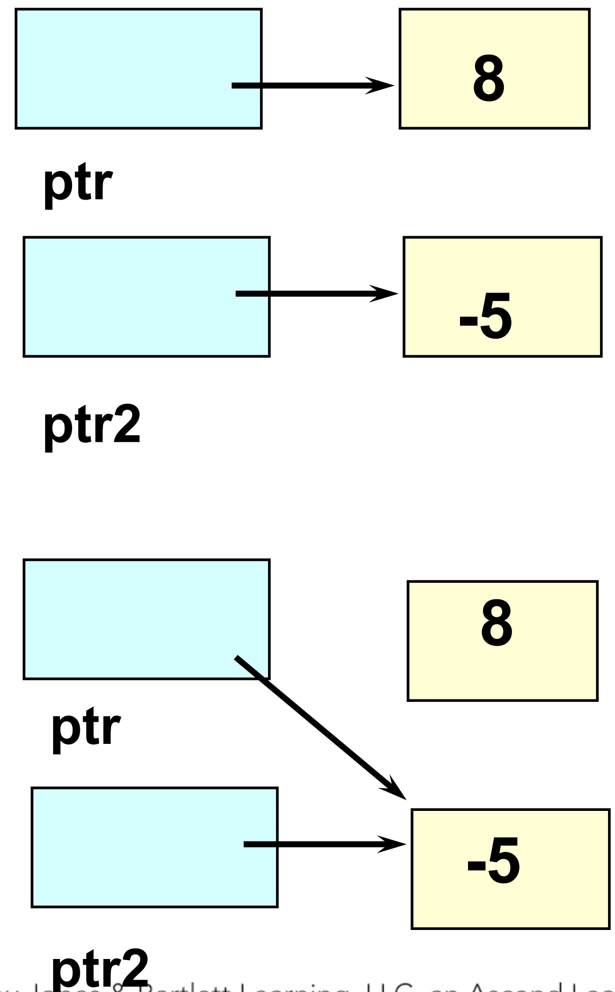
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



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
```



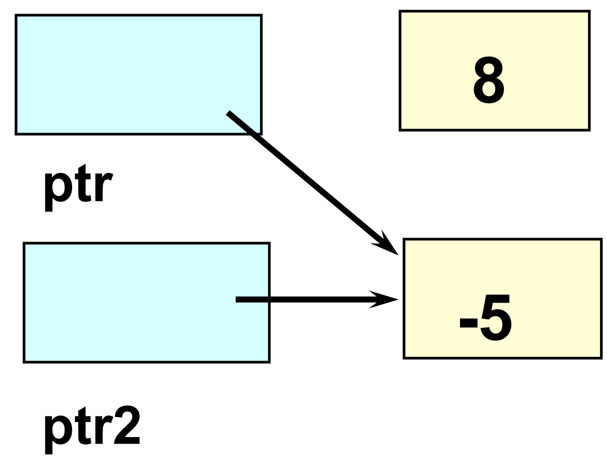
Memory Leak

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

A Dangling Pointer

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

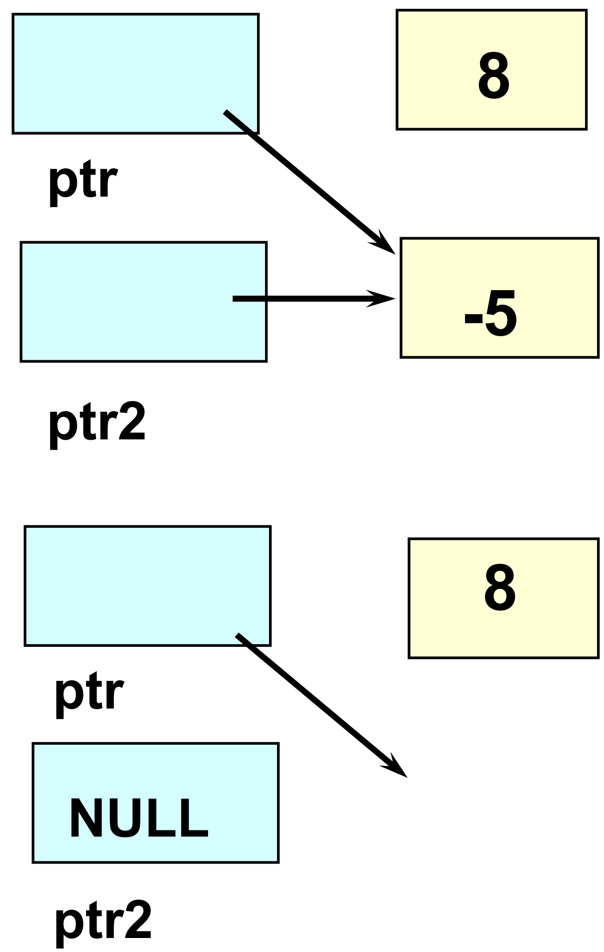
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```



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;
```



```
// Specification file ("dynarray.h")
// Safe integer array class allows run-time
// specification of size, prevents indexes
// from going out of bounds,
// allows aggregate array copying and
// initialization
```

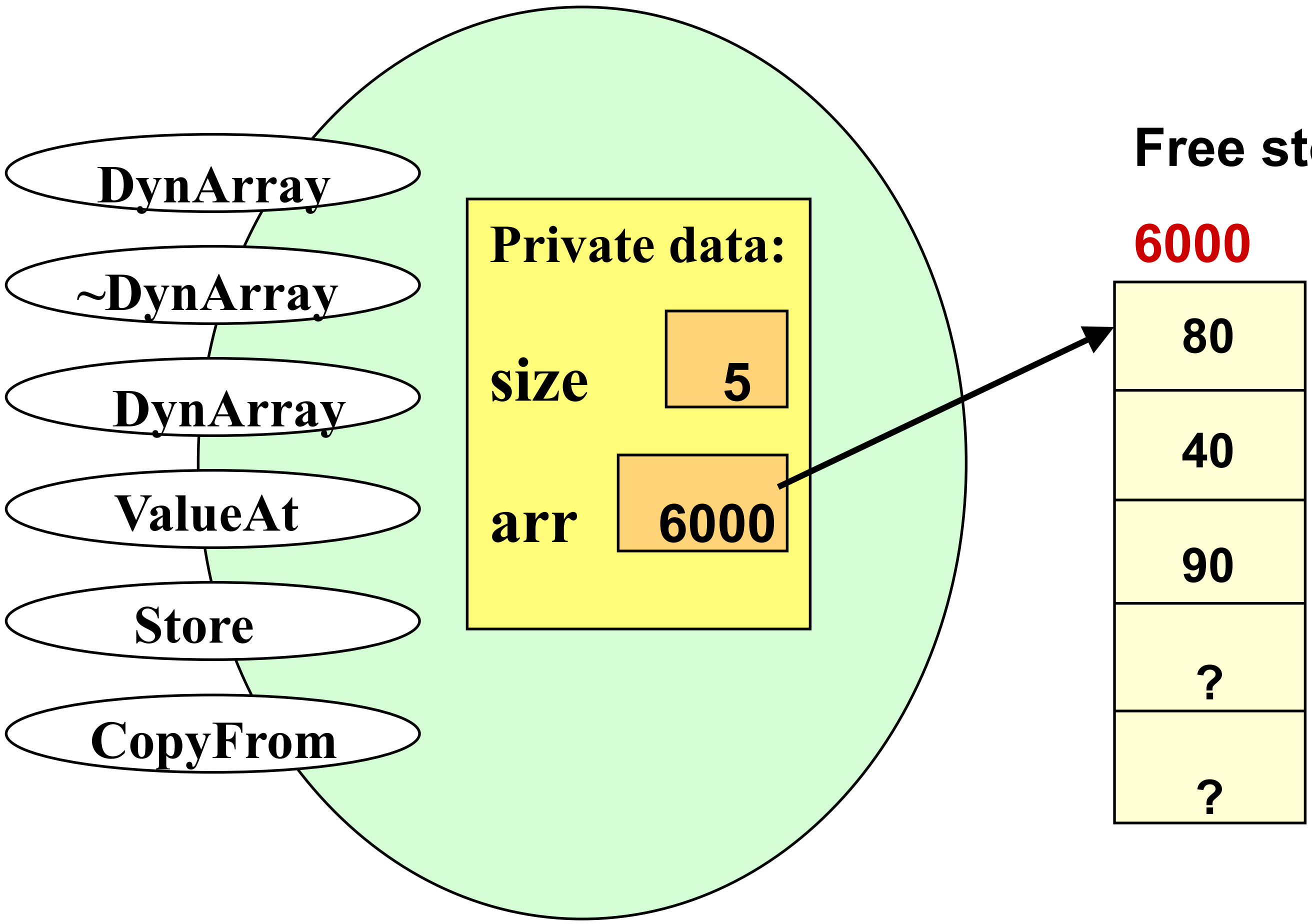
```
// Specification file continued
 class DynArray
{
public:
    DynArray(/* in */  int arrSize);
       // Constructor
      // PRE:  arrSize is assigned
      // POST:  IF arrSize >= 1 && enough memory THEN
      //       Array of size arrSize is created with
      //        all elements == 0  ELSE error message

    DynArray(const DynArray& otherArr);
      // Copy constructor
      // POST: this DynArray is a deep copy of otherArr
      // Is implicitly called for initialization
```

```
// Specification file  continued

    ~DynArray();
      // Destructor
      // POST: Memory for dynamic array deallocated

    int  ValueAt (/* in */ int i)  const;
      // PRE:  i is assigned
      // POST: IF 0 <= i < size of this array THEN
      //       FCTVAL == value of array element at
      //       index i
      //       ELSE error message
```

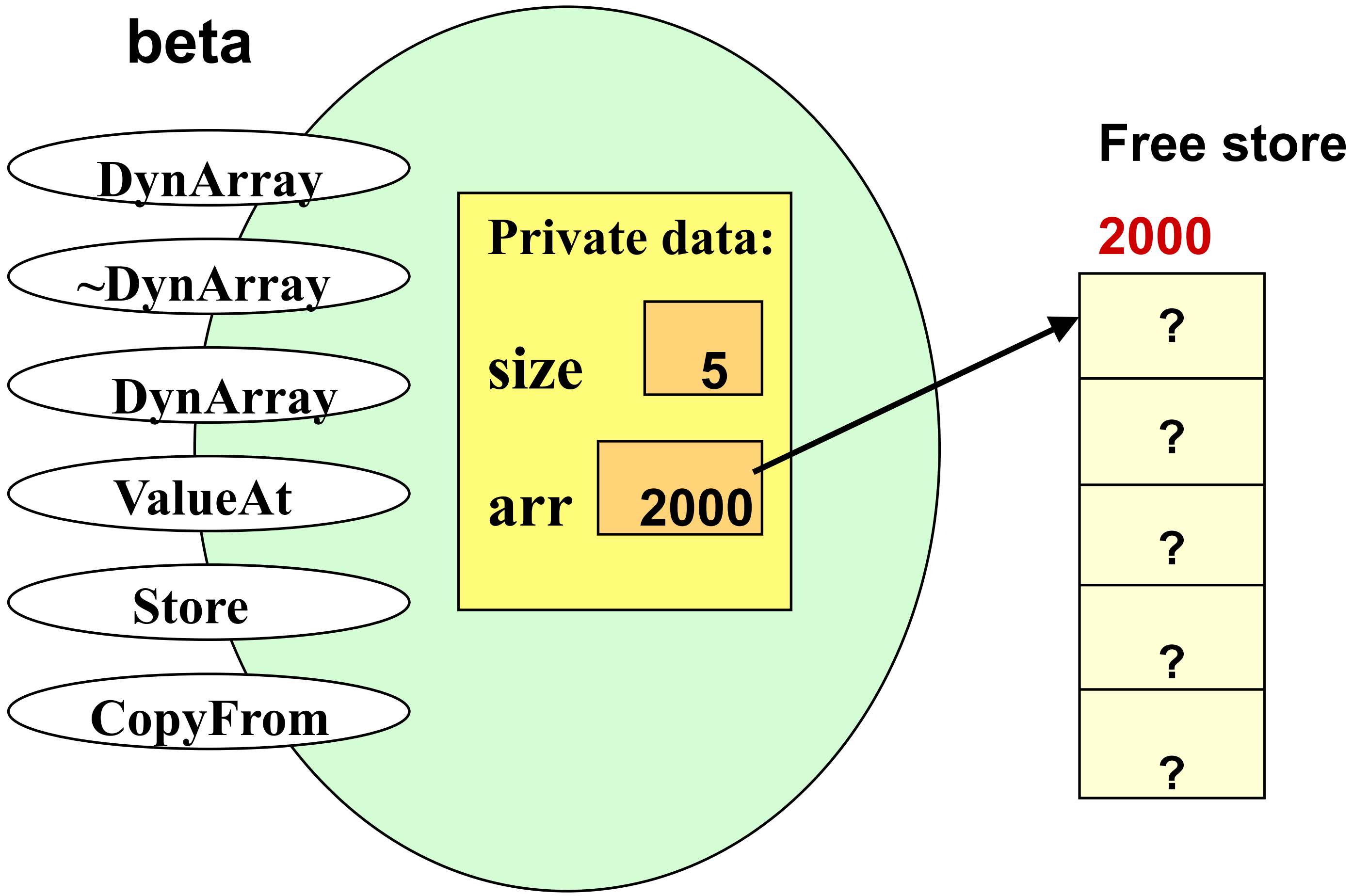
```
 // Specification file  continued
void  Store (/* in */ int val,  /* in */ int i)
    // PRE:  val and i are assigned
    // POST: IF 0 <= i < size of this array THEN
    //       val is stored in array element i
    //       ELSE error message
```

```
// Specification file  continued

    void  CopyFrom (/* in */ DynArray otherArr);
      // POST:  IF enough memory THEN
       //           new array created (as deep copy)
       //           with size and contents
       //           same as otherArr
      //        ELSE error message.

private:
    int*  arr;
    int   size;
};
```

class DynArray



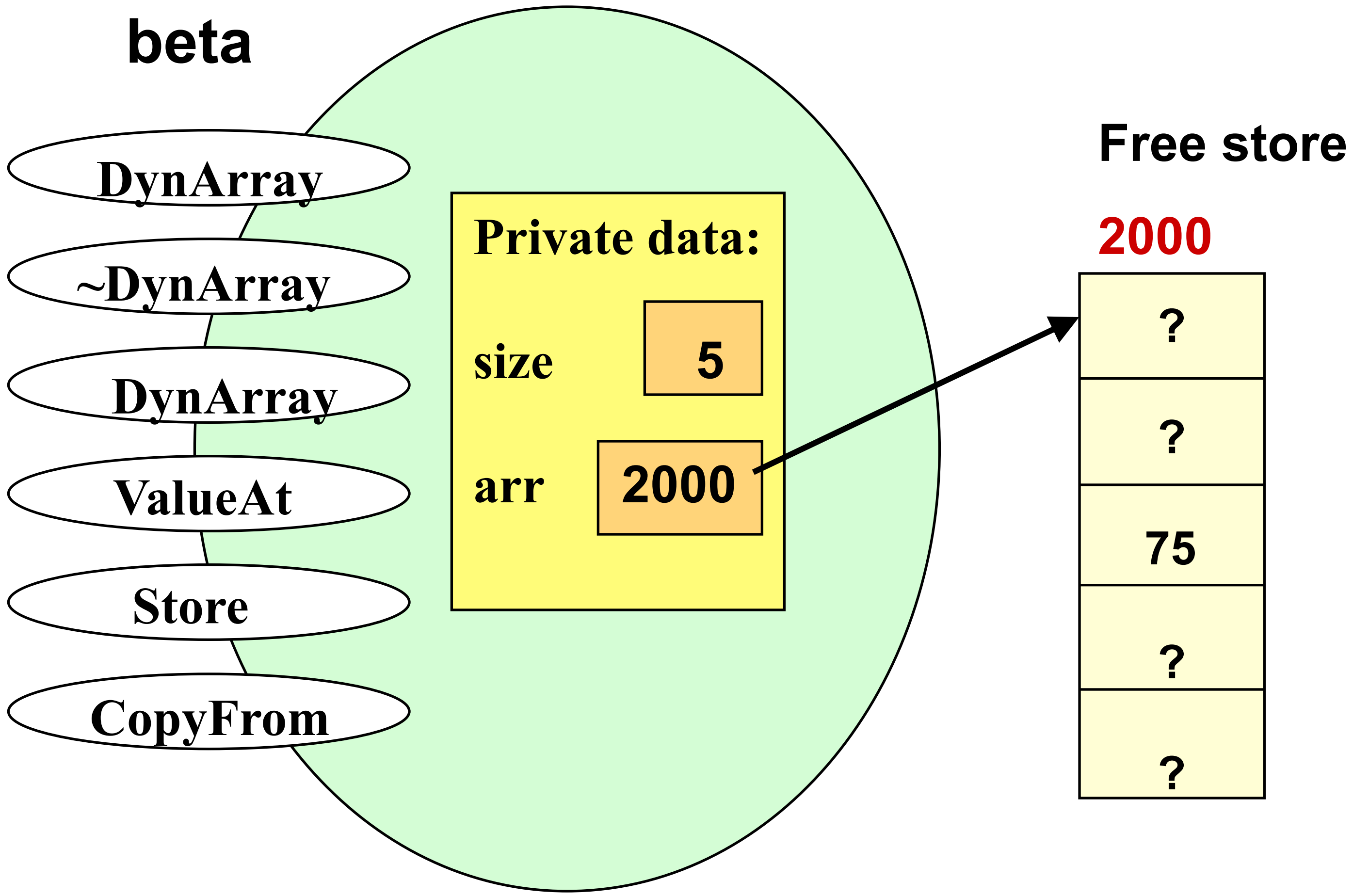
`DynArray beta(5); //constructor`



```
DynArray::DynArray(/* in */  int arrSize)

        // Constructor
       // PRE:  arrSize is assigned
       // POST:  IF arrSize >= 1 && enough memory THEN
       // Array of size arrSize is created with
       // all elements == 0  ELSE error message
```

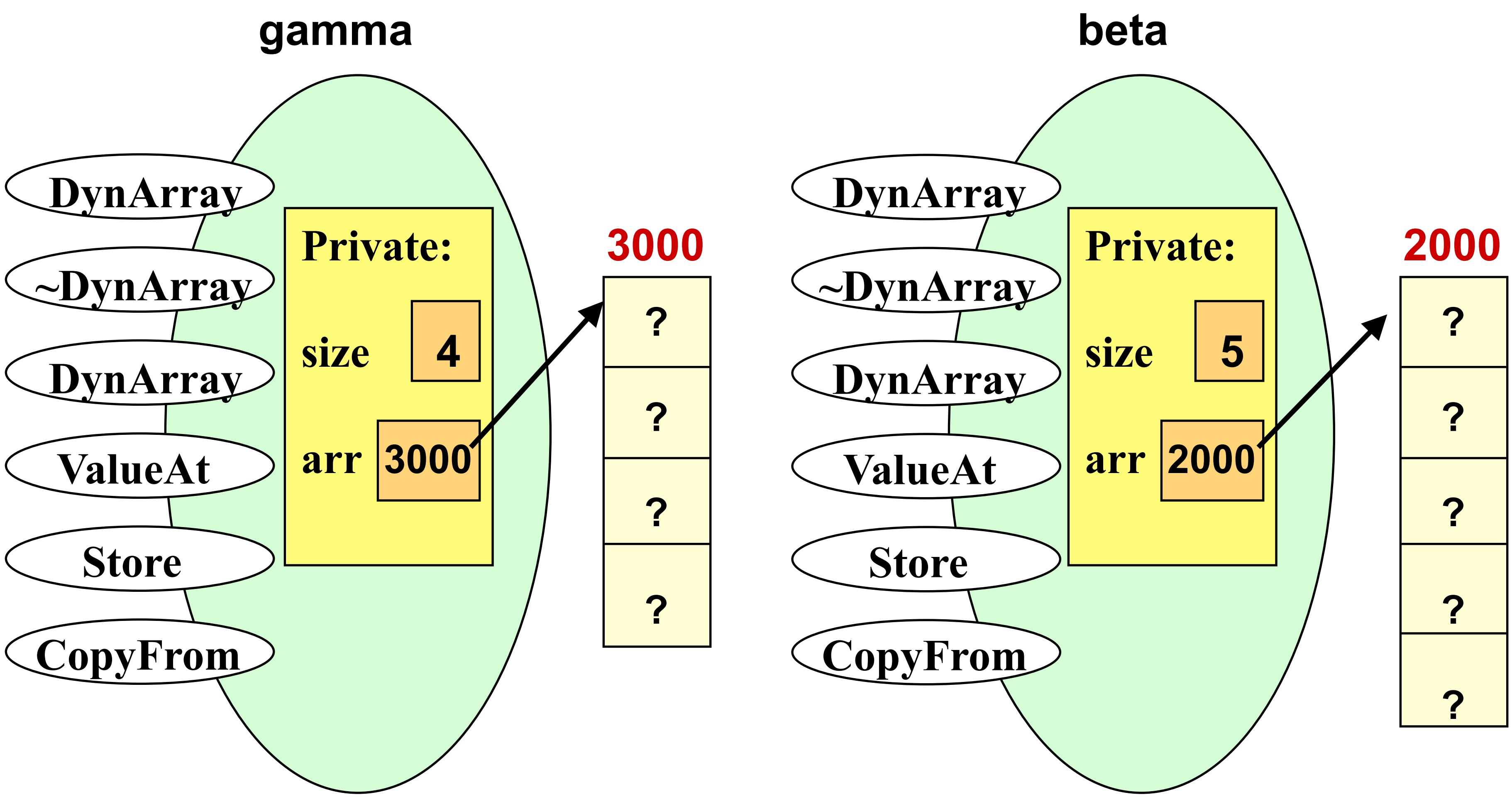
```
{
    int i;
    if (arrSize < 1)
    {
        cerr << "DynArray constructor - invalid size:
  "
             << arrSize << endl;
        exit(1);
    }

    arr = new  int[arrSize];     // Allocate memory

    size = arrSize;

    for (i = 0; i < size;  i++)
        arr[i] = 0;
}
```

`beta.Store(75, 2);`

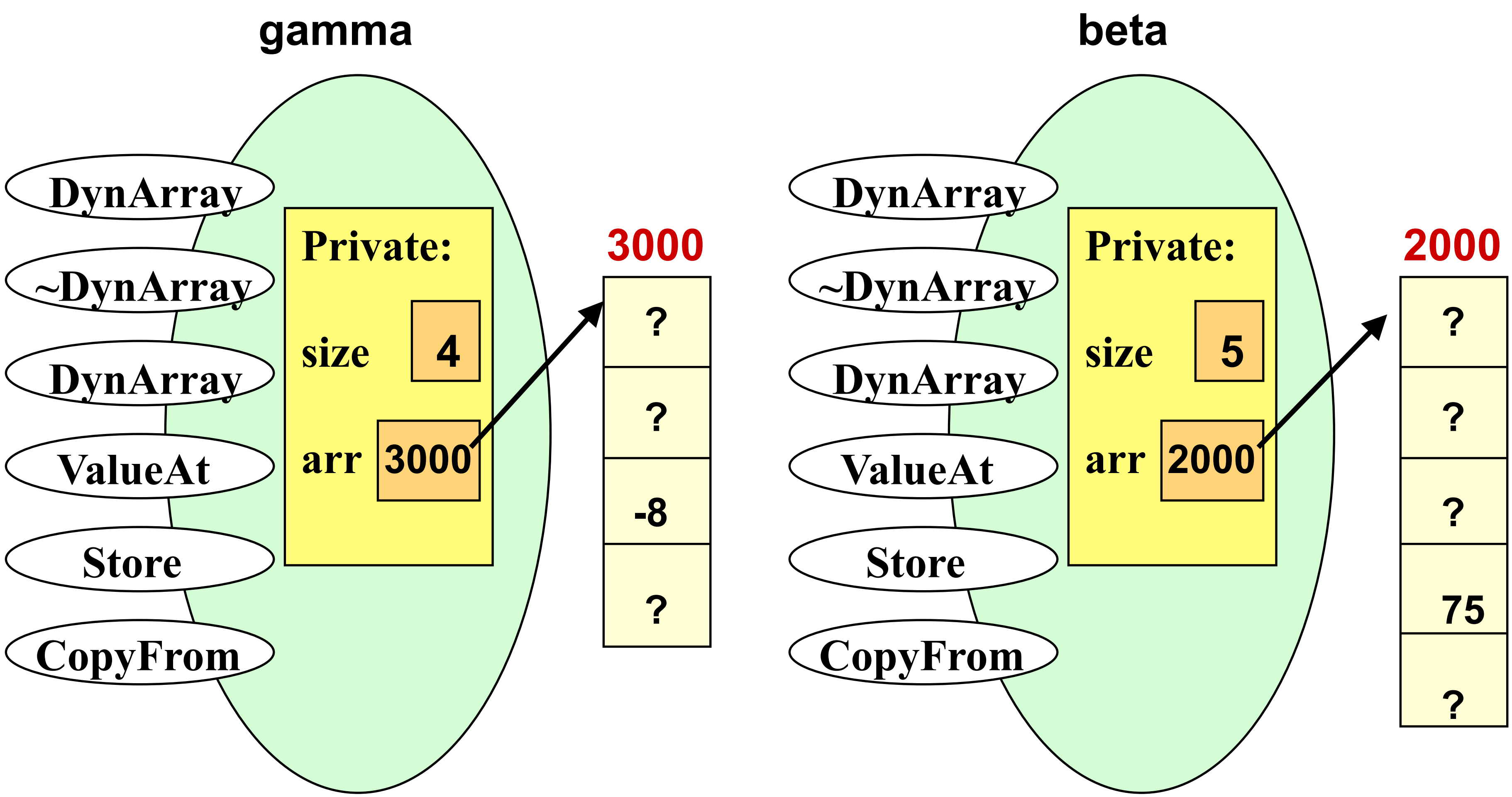


```
void DynArray::Store (/* in */ int val,/* in */ int i)

    // PRE:   val and i are assigned
    // POST: IF 0 <= i < size of this array THEN
    //        arr[i] == val
    //        ELSE error message

{
    if (i < 0 || i >= size)
    {
        cerr << "Store - invalid index: " << i << endl;
        exit(1);
    }

    arr[i] = val;
}
```

```
DynArray gamma(4);//Constructor
DynArray beta(5);
```



```
gamma.Store(-8,2);
Beta.Store(75, 3);
```



```
int  DynArray::ValueAt (/* in */ int i)  const
      // PRE:  i is assigned
      // POST: IF 0 <= i < size THEN
      //       Return value == arr[i]
      //       ELSE halt with error message
{
    if (i < 0 || i >= size)
    {
        cerr << "ValueAt - invalid index : " << i
           << endl;
        exit(1);
    }
    return arr[i];
}
```

Why is a destructor needed?

When a DynArray class variable goes out of scope, the memory space for data members size and pointer `arr` is deallocated

But the dynamic array that `arr` points to is not automatically deallocated

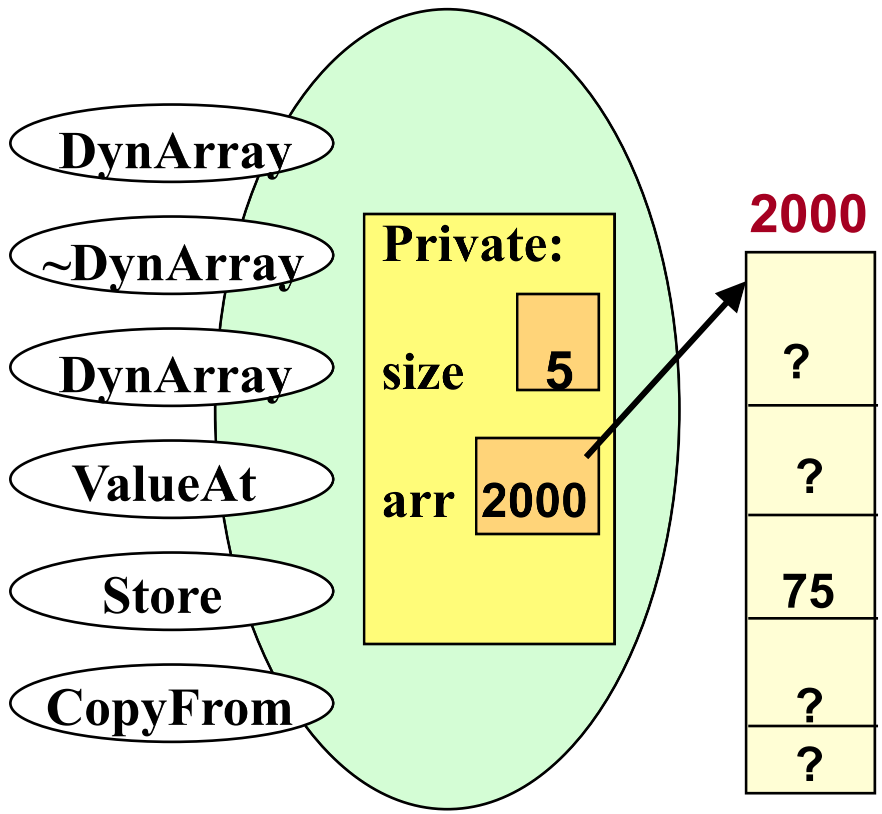
A class destructor is used to deallocate the dynamic memory pointed to by the data member

class DynArray Destructor

```
DynArray::~DynArray();
// Destructor
// POST: Memory for dynamic array deallocated
{
   delete [ ] arr;
}
```

What happens . . .

- ***When a function is called that passes a DynArray object by value, what happens?***

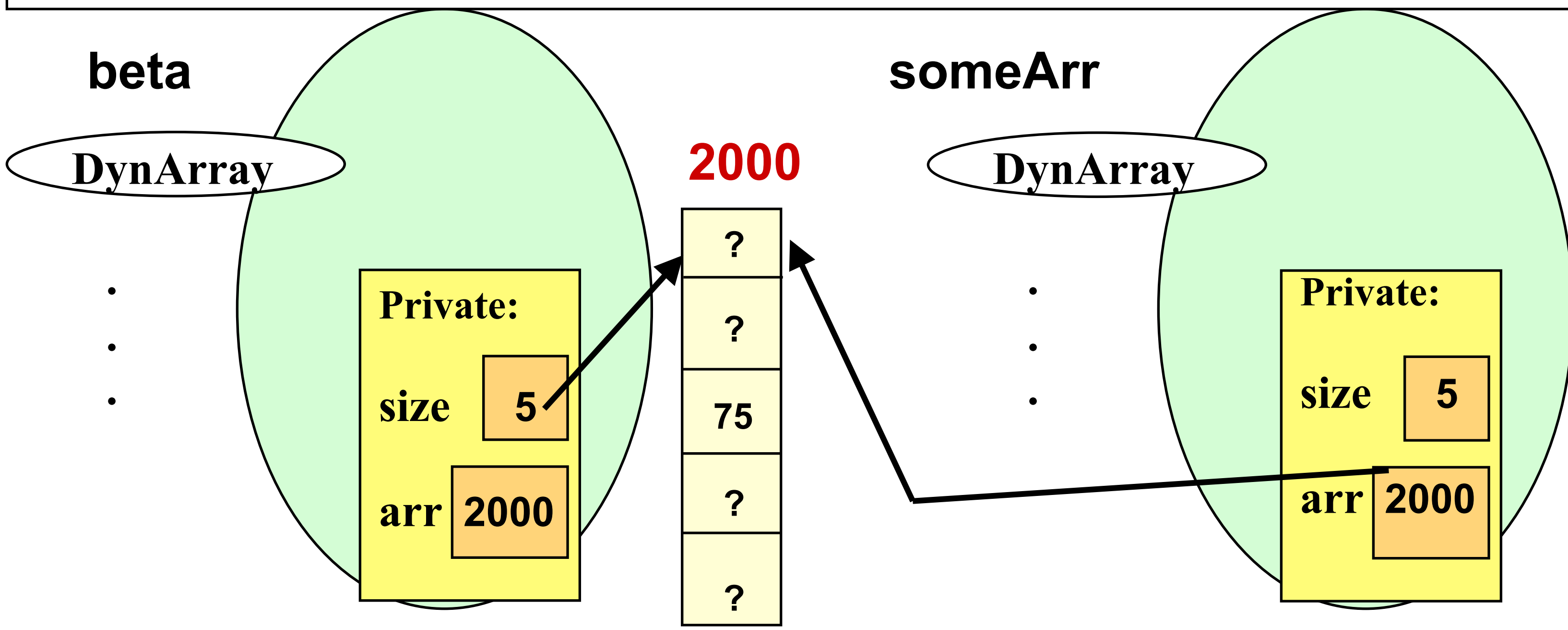


Passing a Class Object by Value

```
 // Function code

void  SomeFunc(DynArray  someArr)
 // Uses pass by value
{
     .
     .
     .
     .
 }
```

By default, Pass-by-value makes a shallow copy

```
DynArray  beta(5);          // Client code
    .
    .
    .
SomeFunc(beta);           //  Function call
```



shallow copy

Shallow Copy vs. Deep Copy

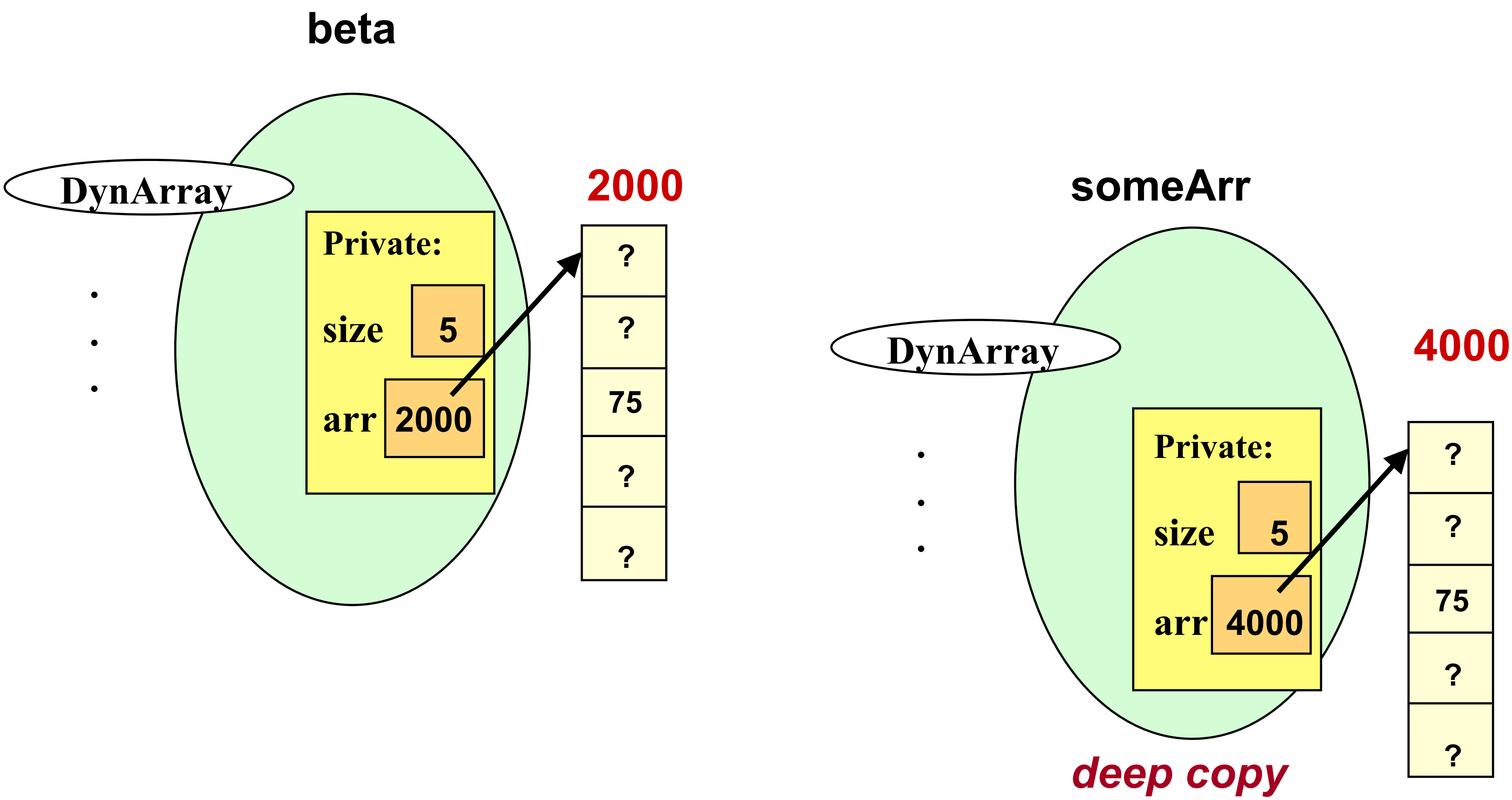
- *A* ***shallow copy*** **copies only the class data members, and does not make a copy of any pointed-to data**
- *A* ***deep copy*** **copies not only the class data members, but also makes a separate stored copy of any pointed-to data**

What's the difference?

- *A shallow copy* shares the pointed to dynamic data with the original class object

- *A deep copy* makes its own copy of the pointed to dynamic data at different locations than the original class object

Making a (Separate) Deep Copy



Initialization of Class Objects

- C++ defines initialization to mean
  - initialization in a variable declaration
  - passing an object argument by value
  - returning an object as the return value of a function
- By default, C++ uses shallow copies for these initializations

As a result . . .

- When a class has a data member that points to dynamically allocated data, you must write what is called a **copy constructor**

- The copy constructor **is implicitly called in initialization situations** and makes a deep copy of the dynamic data in a different memory location

Copy Constructor

Most difficult algorithm so far:

- **If the original is empty, the copy is empty**
- **Otherwise, make a copy of the head with pointer to it**
- **Loop through original, copying each node and adding it to the copy until you reach the end**

See Chapter 18 for an easy, elegant solution

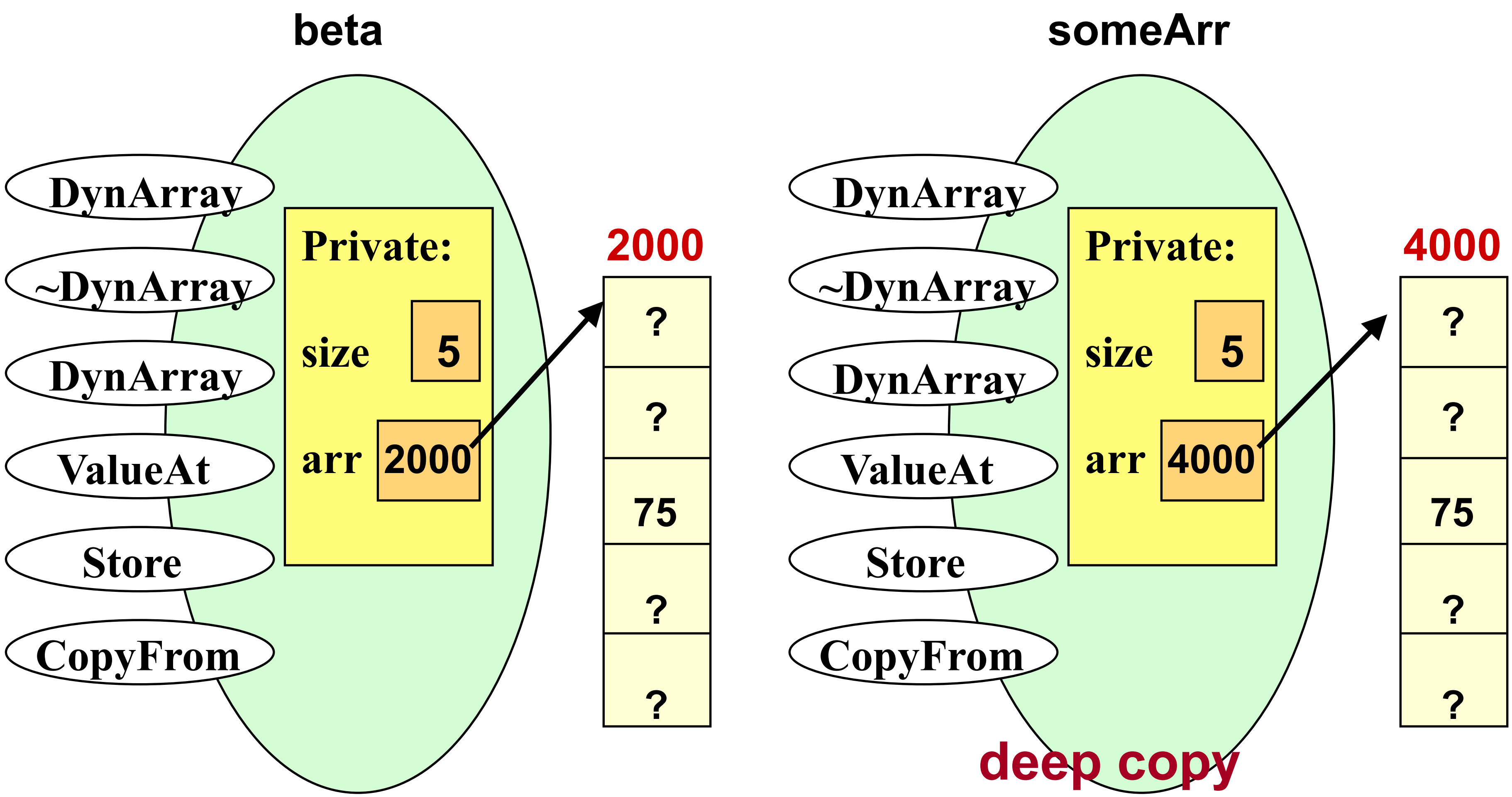
Copy Constructor

- Copy constructor is a special member function of a class that is implicitly called in these three situations:
  - Passing object parameters by value
  - Initializing an object variable in its declaration
  - Returning an object as the return value of a function

More about Copy Constructors

- When you provide (write) a copy constructor for a class, the copy constructor is used to make copies for pass by value
- You do not explicitly call the copy constructor
- Like other constructors, it has no return type
- Because the copy constructor properly defines pass by value for your class, it must use pass by reference in its definition

```
SomeFunc(beta);  // copy-constructor
                 // beta passed by value
```



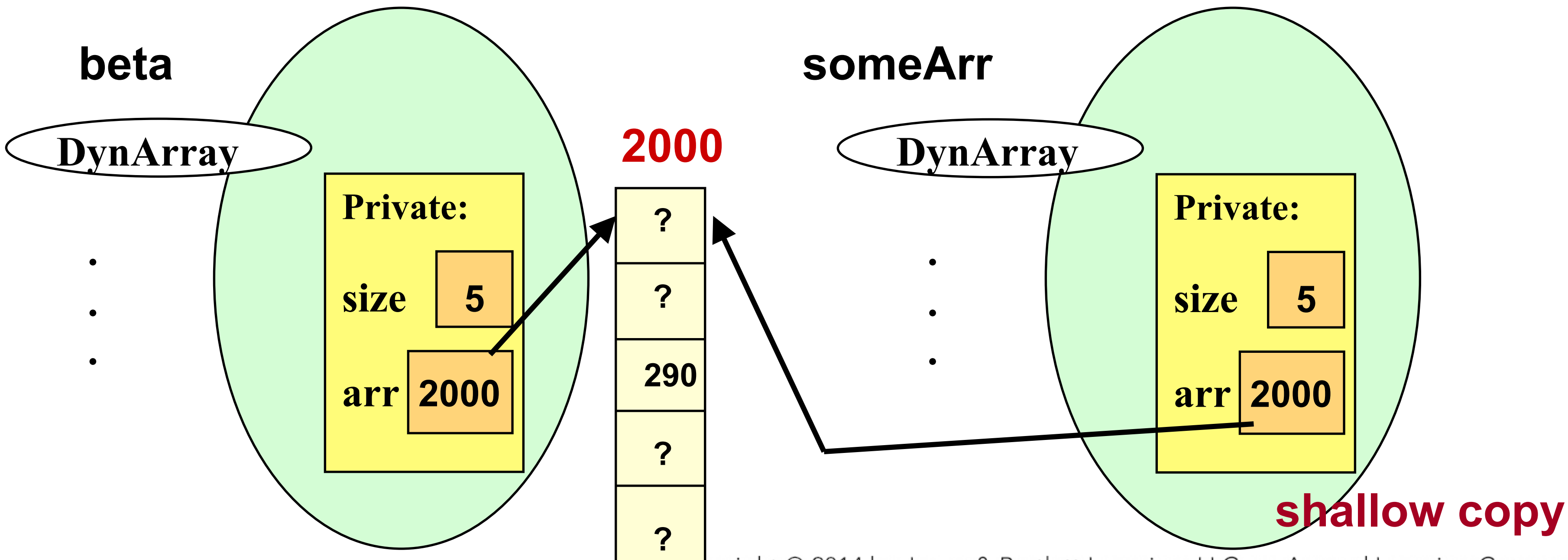
Suppose `SomeFunc` calls Store

```
void  SomeFunc(DynArray  someArr)
// Uses pass by value
{
    someArr.Store(290, 2);
    .
    .
    .

}
```

What happens in the shallow copy scenario?

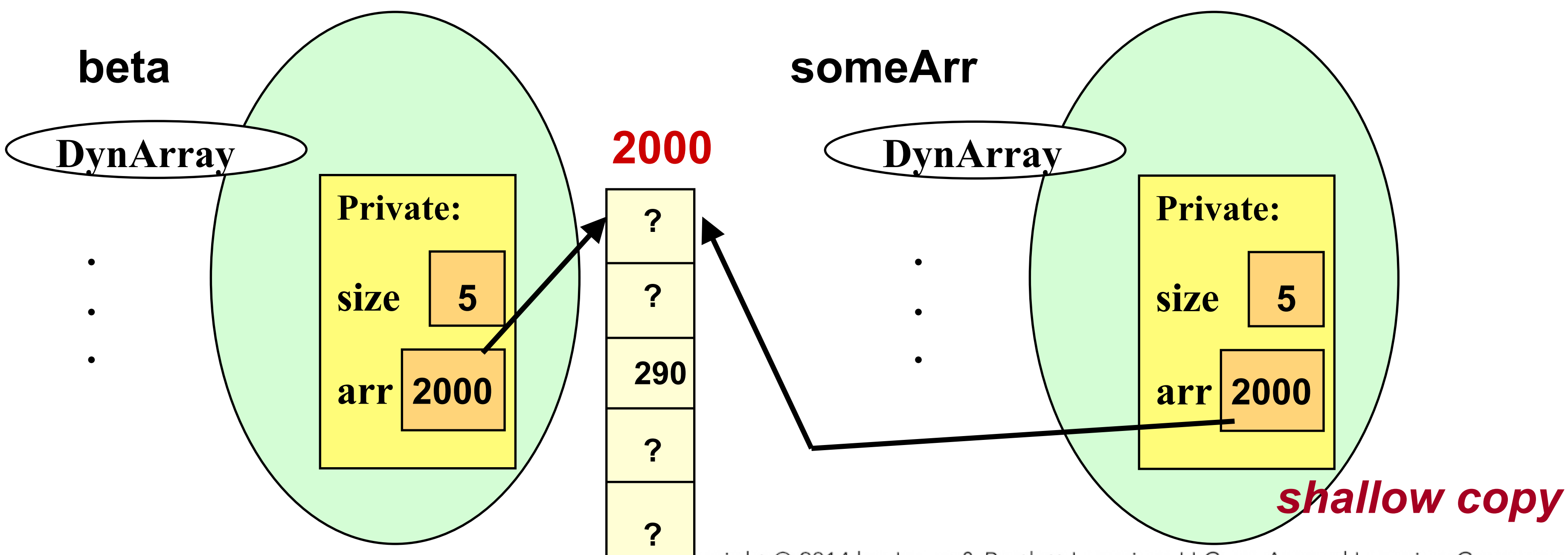
`beta.arr[2]` has changed

```
DynArray  beta(5);          // Client code
      .
      .
      .
SomeFunc(beta);
```



`beta.arr[2]` has changed

Although beta is passed by value, its dynamic data has changed!



Classes with Data Member Pointers Need

CONSTRUCTOR

COPY CONSTRUCTOR

DESTRUCTOR

```
DynArray::DynArray(const DynArray& otherArr)
      // Copy constructor
      // Implicitly called for deep copy in
      // initializations
      // POST:  If room on free store THEN
      //    new array of size otherArr.size is
      //    created
      //    on free store && arr == its base address
      //     && size == otherArr.size
      //    && arr[0..size-1] ==
      //       otherArr.arr[0..size-1]
      //    ELSE error occurs
```

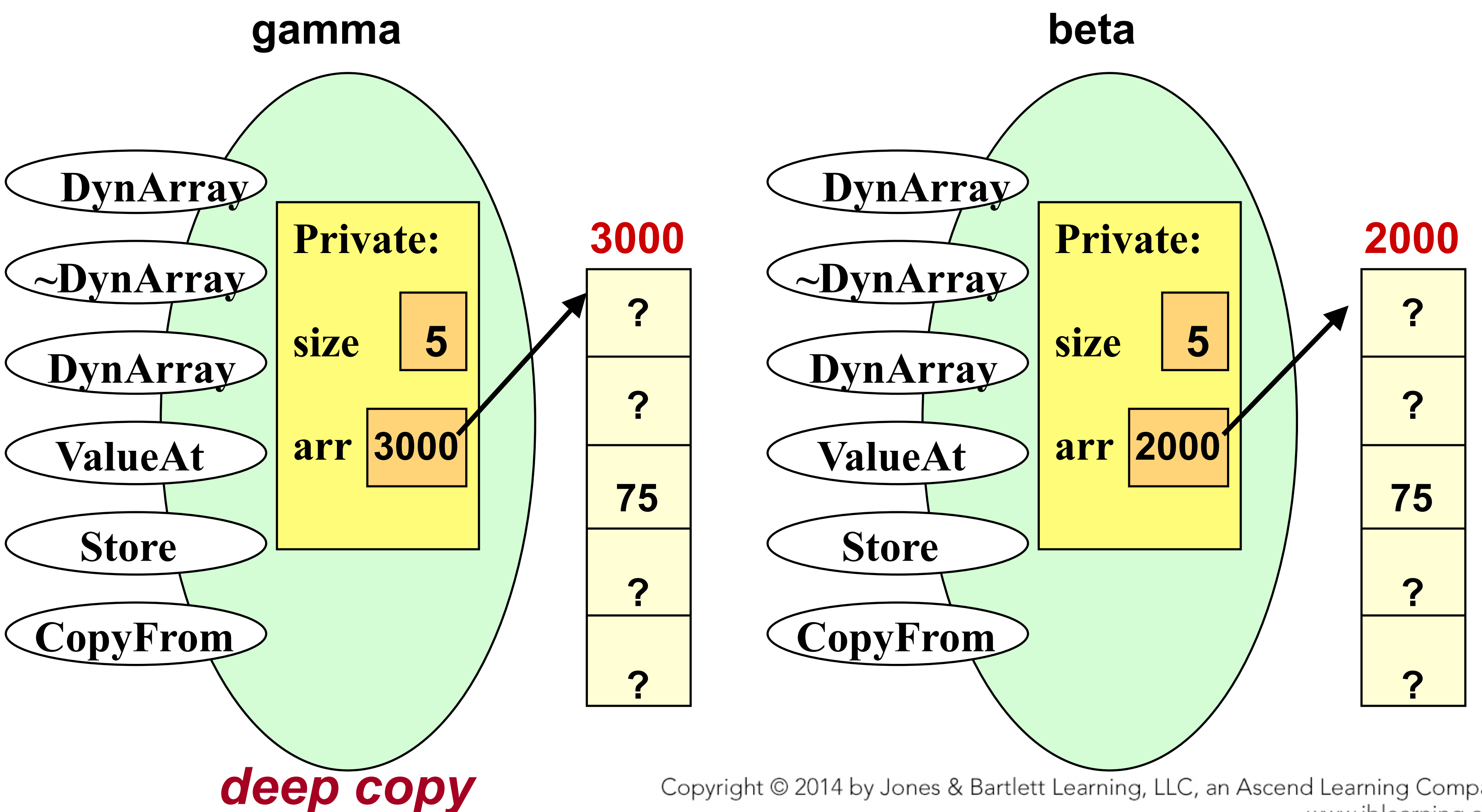
```
{
    int i;
    size = otherArr.size;
    // Allocate memory for copy
    arr = new  int[size];

    // Copies array
    for (i = 0; i< size; i++)
        arr[i] = otherArr.arr[i];

}
```

What about the assignment operator?

- The **default method** used for assignment of class objects makes a **shallow copy**

- If your class has a data member that points to dynamic data, you should write a member function **to create a deep copy** of the dynamic data

gamma.CopyFrom(beta);



```
void  DynArray::CopyFrom (/* in */ DynArray  otherArr)
      // Creates a deep copy of otherArr
      // POST:  Array pointed to by arr@entry
      //        deallocated
      //   &&  IF room on free store
      //       THEN new array is created on free store
      //          && arr == its base address
      //          && size == otherArr.size
      //          && arr[0..size-1] == otherArr[0..size-]
      //        ELSE halts with error message
```

```
{
    int i;

    delete[ ]  arr; // Delete current array
    size = otherArr.size;
    arr = new int [size]; // Allocate new array

    for (i = 0; i< size; i++)  // Deep copy array
        arr[i] = otherArr.arr[i];
}
```

LINKED LIST

What is a List?

- A **list** is a varying-length, linear collection of homogeneous elements
- **Linear** means:
  - Each list element (except the first) has a unique predecessor, and
  - Each element (except the last) has a unique successor

To implement the List ADT

The programmer must:

1) **choose a concrete data representation for the list, and**

2) **implement the list operations**

Recall:
4 Basic Kinds of ADT Operations

- **Constructors** -- create a new instance (object) of an ADT
- **Transformers** -- change the state of one or more of the data values of an instance

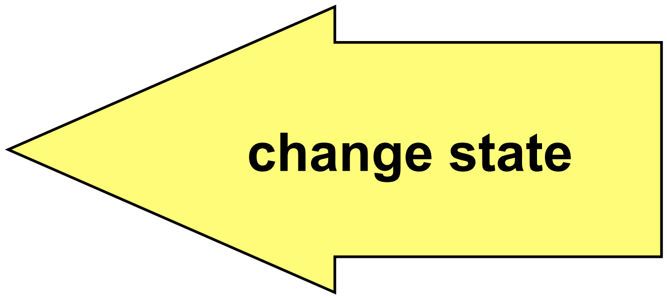
Recall: 4 Basic Kinds of ADT Operations

- **Observers** -- allow client to observe the state of one or more of the data values of an instance without changing them
- **Iterators** -- allow client to access the data values in sequence

List Operations

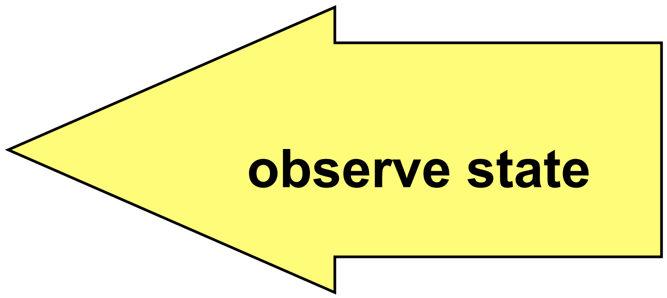
Transformers

- Insert
- Delete
- Sort



Observers

- IsEmpty
- IsFull
- Length
- IsPresent



ADT List Operations

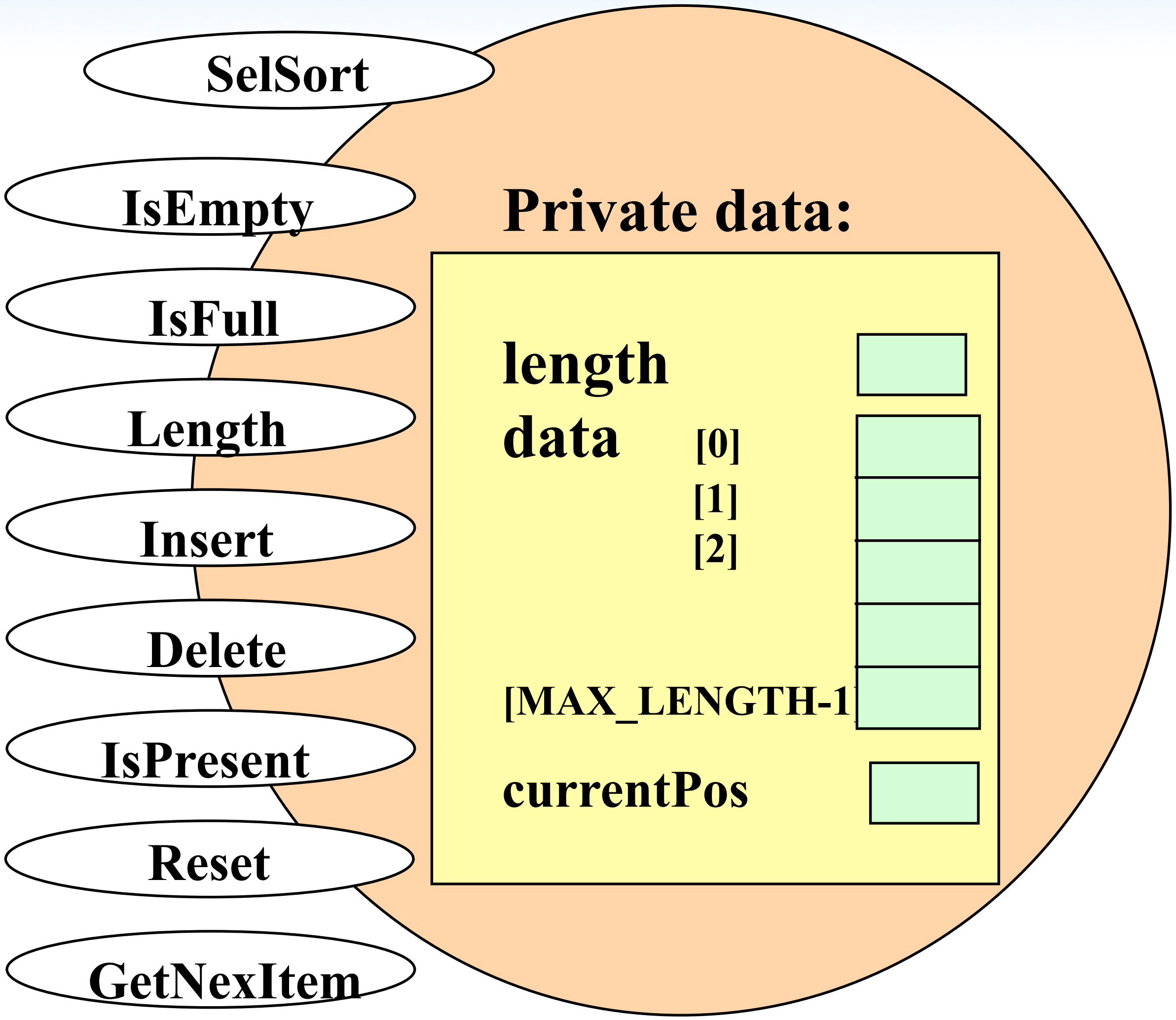
Iterator

- Reset
- GetNextItem

Iteration Pair

- Reset prepares for the iteration
- GetNextItem returns the next item in sequence
- No transformer can be called between calls to GetNextItem *(Why?)*

Array-based `class List`



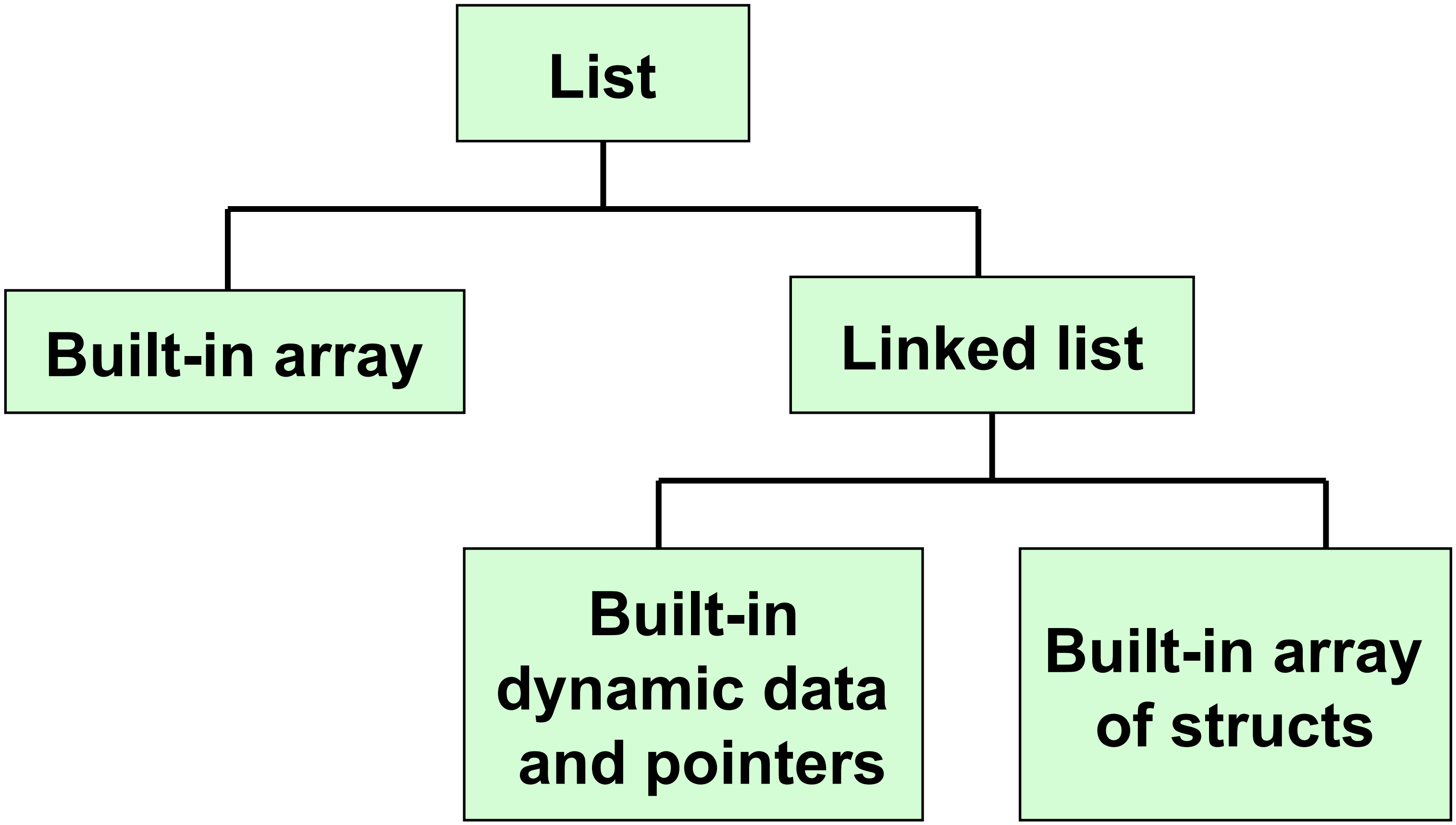
```
// Specification file array-based list ("list.h")
const  int  MAX_LENGTH  =  50;
typedef int   ItemType;

class  List  // Declares a class data type
```

```
{
public:                        // Public member functions

    List();                  // constructor
    bool IsEmpty () const;
    bool IsFull ()  const;
    int  Length ()  const; // Returns length of list
    void Insert (ItemType  item);
    void Delete (ItemType  item);
    bool IsPresent(ItemType  item)  const;
    void SelSort ();
    void Reset ();
    ItemType GetNextItem ();

private:                    // Private data members
     int length;    // Number of values currently stored
     ItemType data[MAX_LENGTH];
    int  CurrentPos;   // Used in iteration
};
```

Implementation Structures

- Use a built-in array stored in contiguous memory locations, implementing operations Insert and Delete by moving list items around in the array, as needed
- Use a linked list in which items are not necessarily stored in contiguous memory locations
- A linked list avoids excessive data movement from insertions and deletions

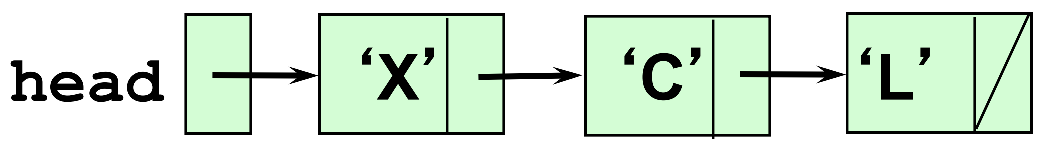
Implementation Possibilities for a List ADT



A Linked List

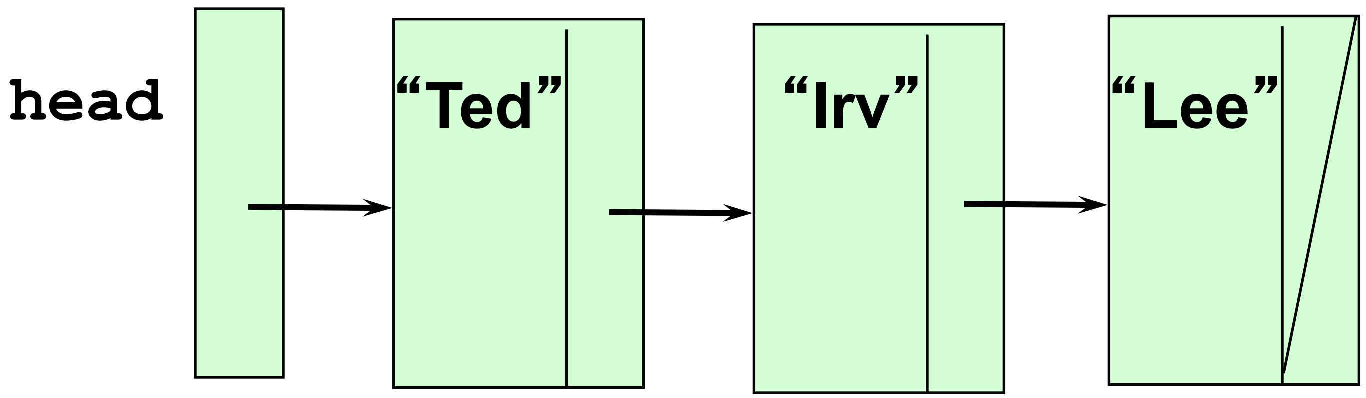
- A **linked list** is a list in which the order of the components is determined by an explicit link member in each node

- Each node is a `struct` containing a data member and a link member that gives the location of the next node in the list



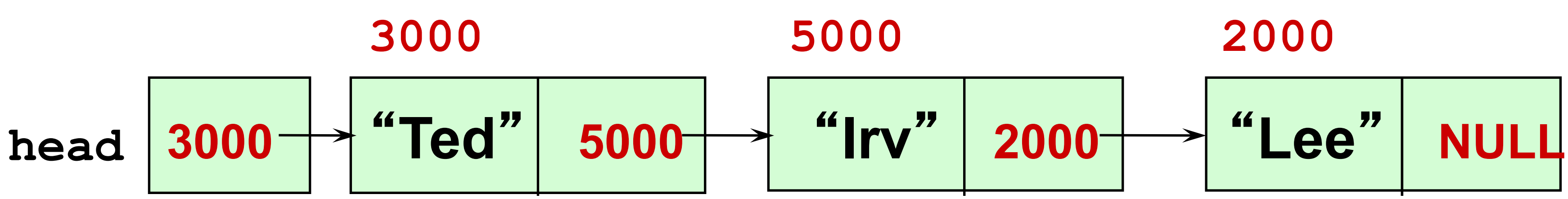
Dynamic Linked List

- A **dynamic linked list** is one in which the nodes are linked together by pointers and an external pointer (or head pointer) points to the first node in the list



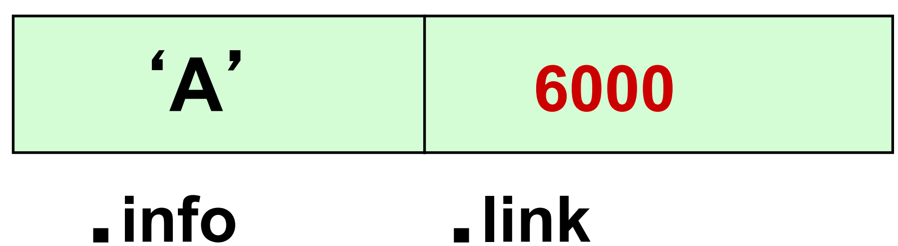
Nodes can be located anywhere in memory

- **The link member holds the memory address of the next node in the list**

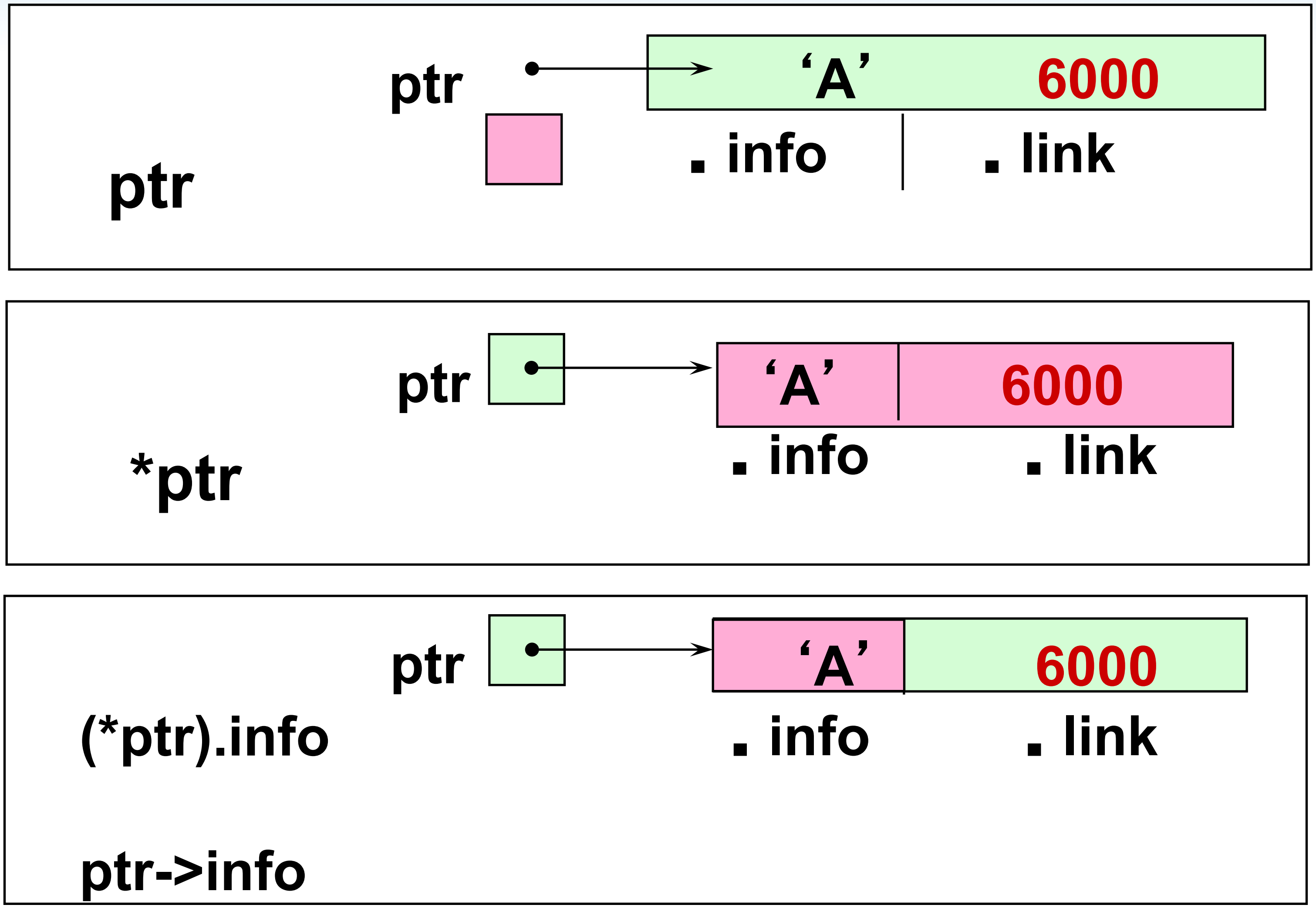


Declarations for a Dynamic Linked List

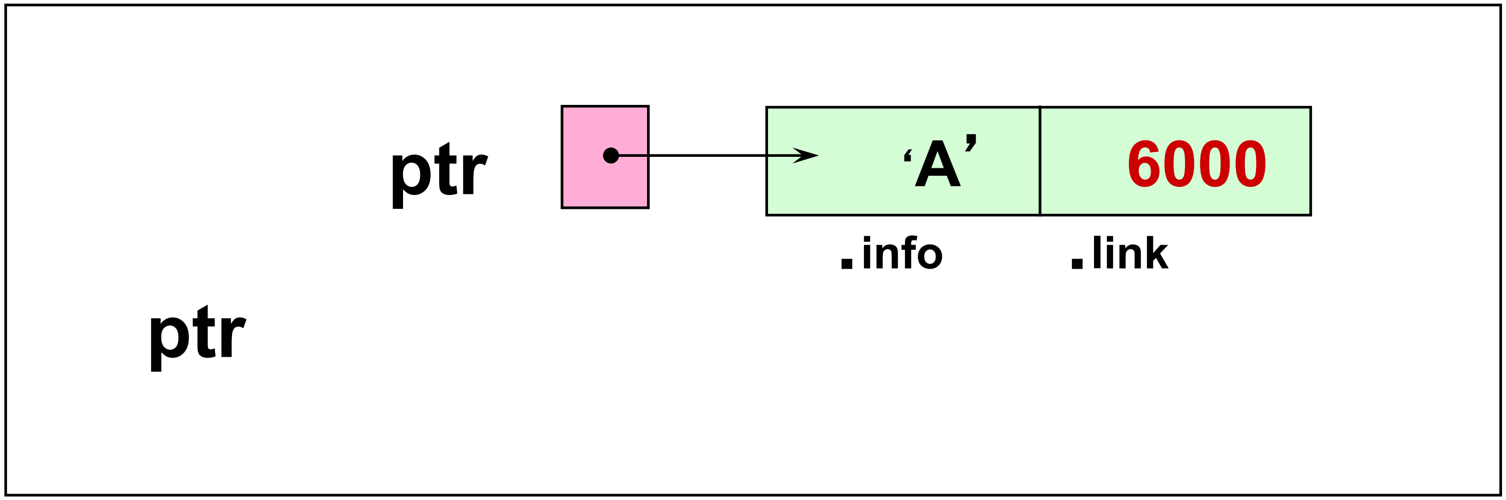
```
// Type declarations

struct NodeType
{
    char info;
    NodeType* link;
}

typedef  NodeType*  NodePtr;

// Variable DECLARATIONS
NodePtr  head;
NodePtr  ptr;
```



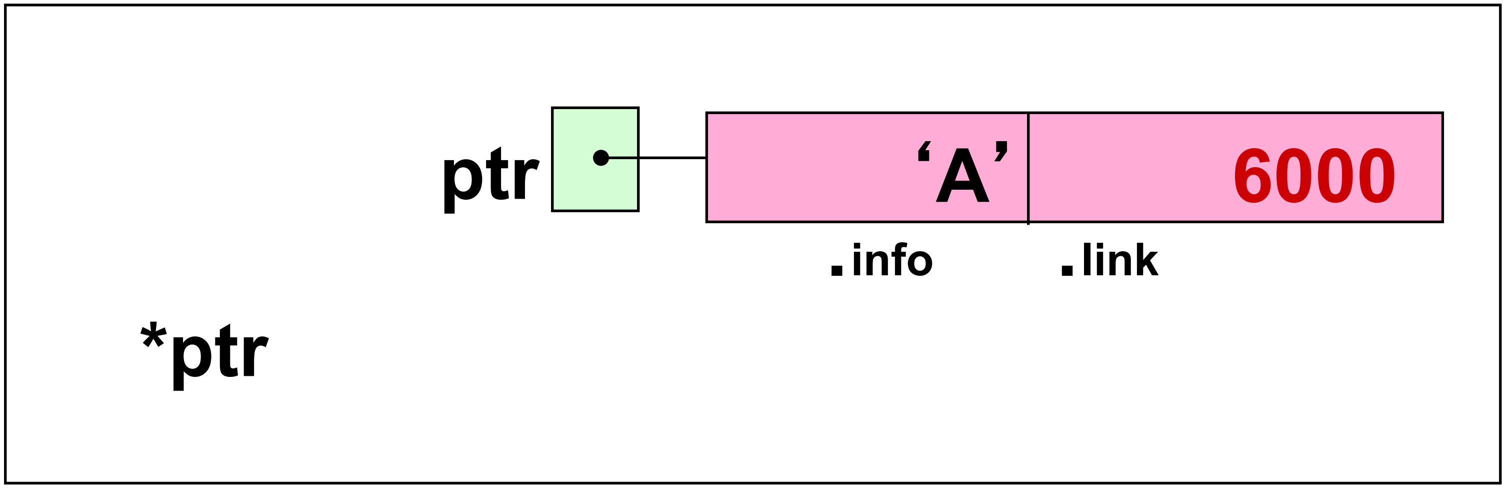
Pointer Dereferencing and Member Selection



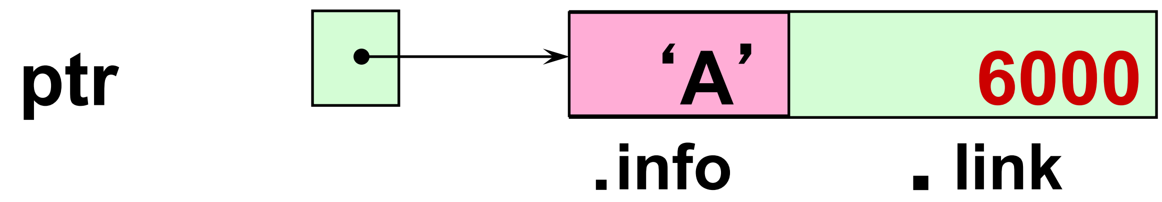
`ptr` is a pointer to a node



*ptr is the entire node pointed to by ptr



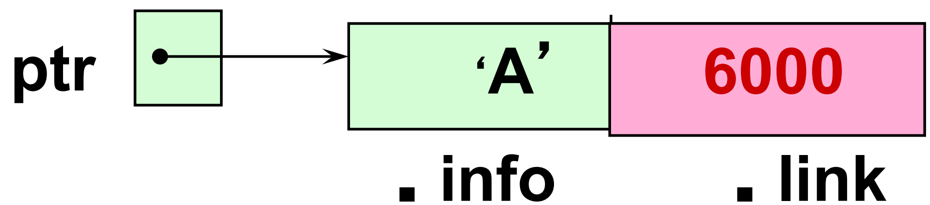
`ptr->info` is a node member



ptr->info

(*ptr).info // Equivalent

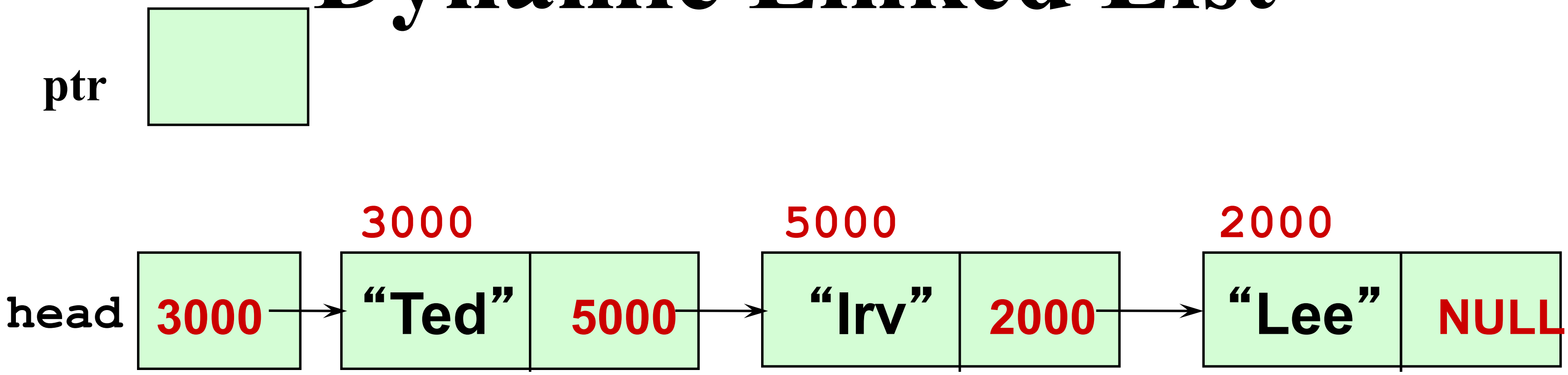
`ptr->link`
is a node member



ptr->link

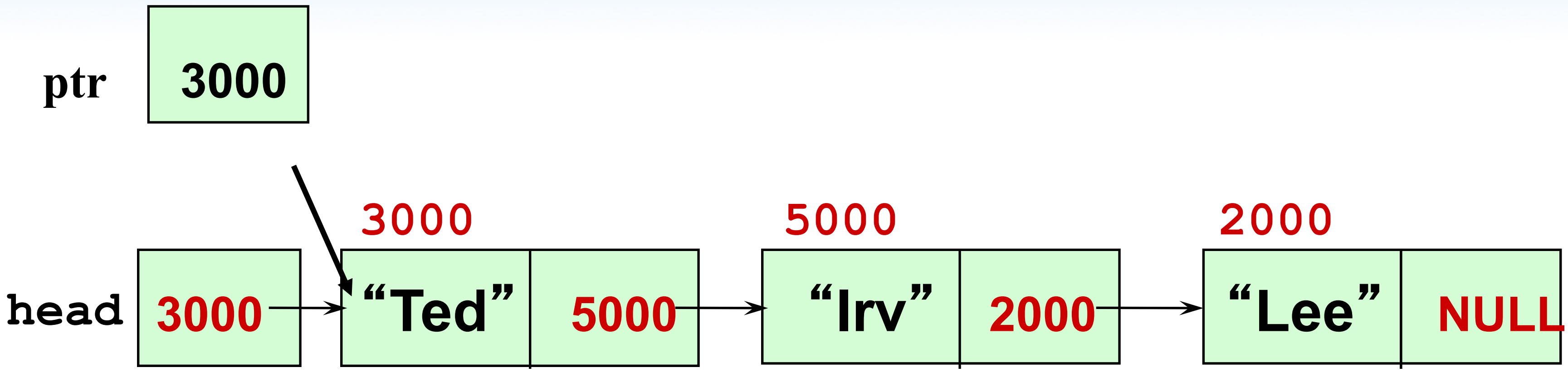
(*ptr).link **// Equivalent**

Traversing a Dynamic Linked List



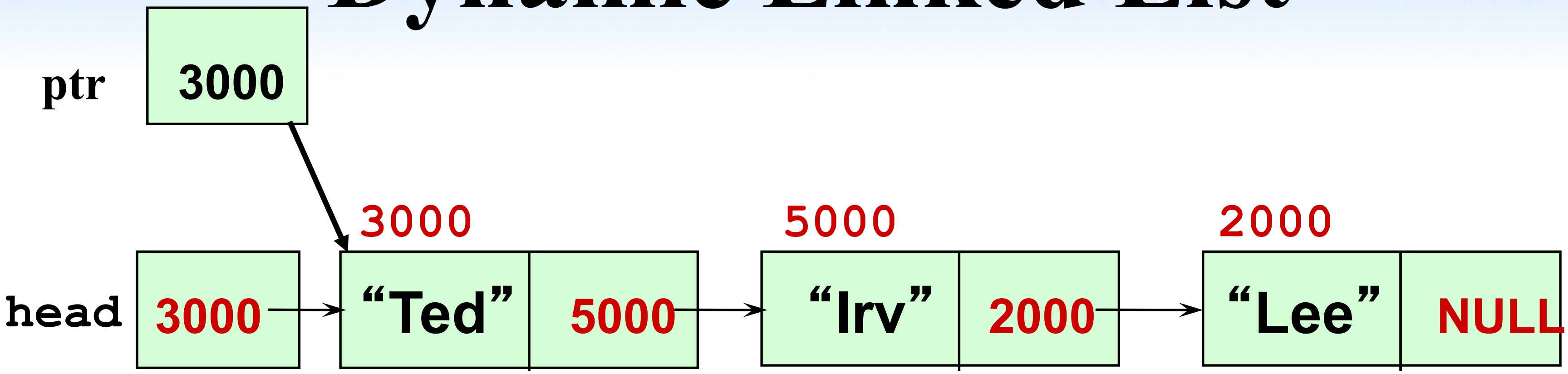

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
    cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



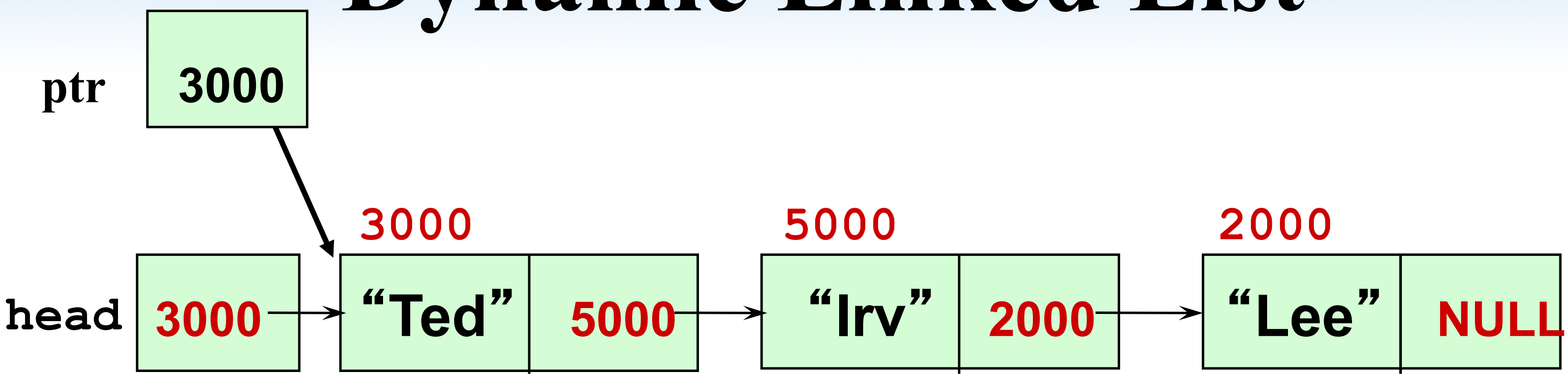

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
    cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



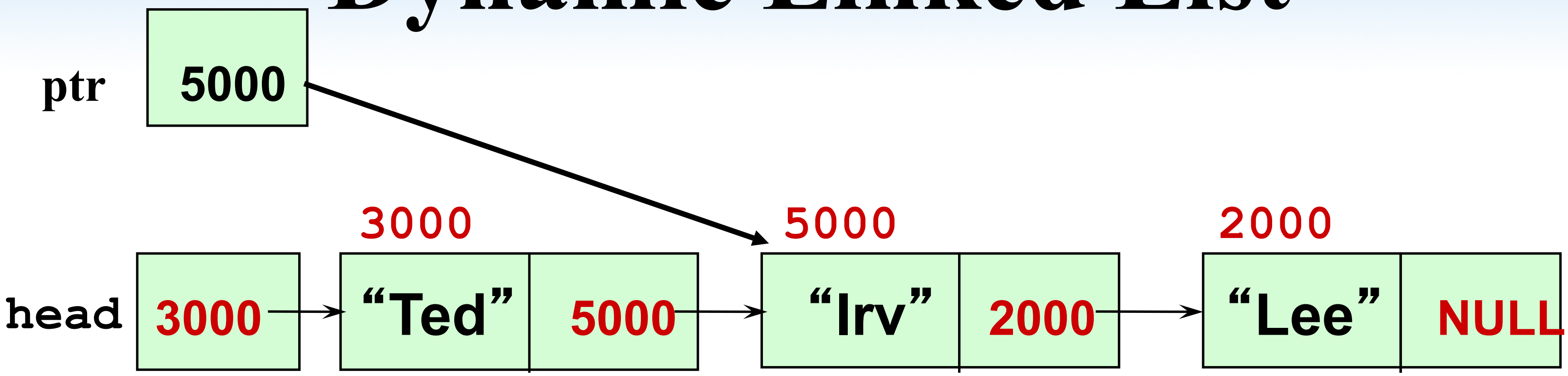

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
  cout  <<  ptr->info;
  // Or, do something else with node *ptr
  ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



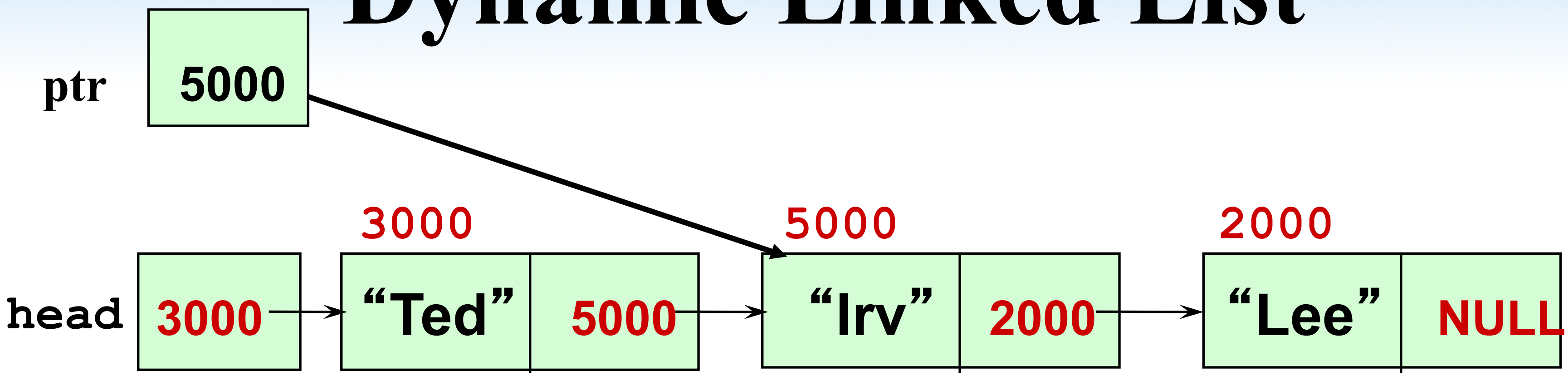

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
     // Or, do something else with node *ptr
     ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



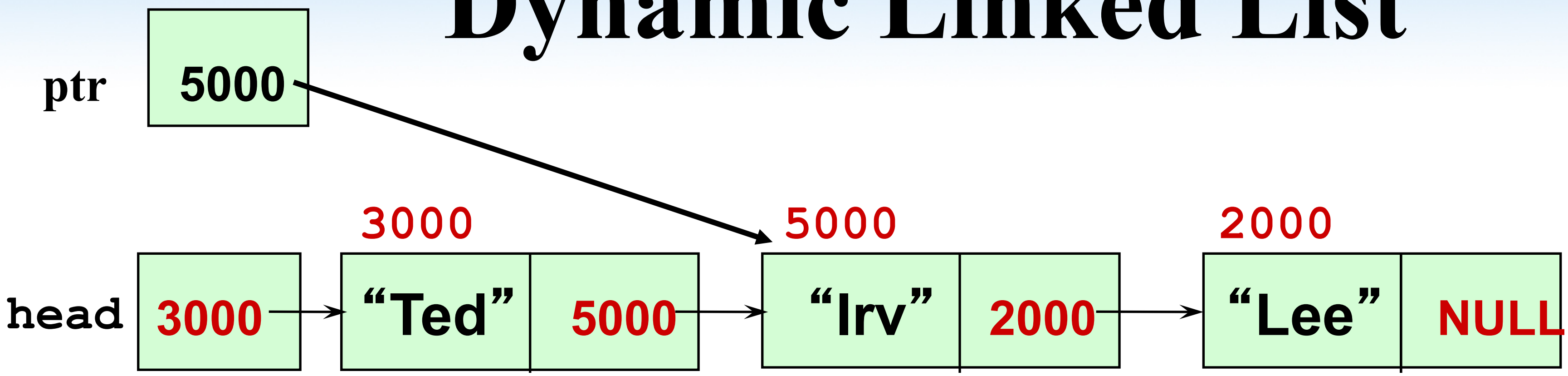

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
    cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



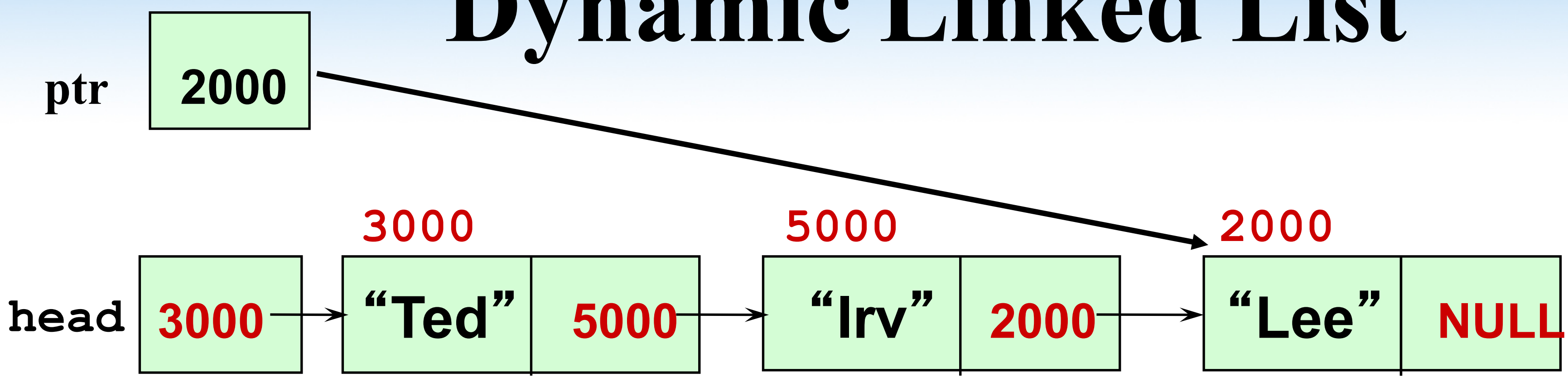

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



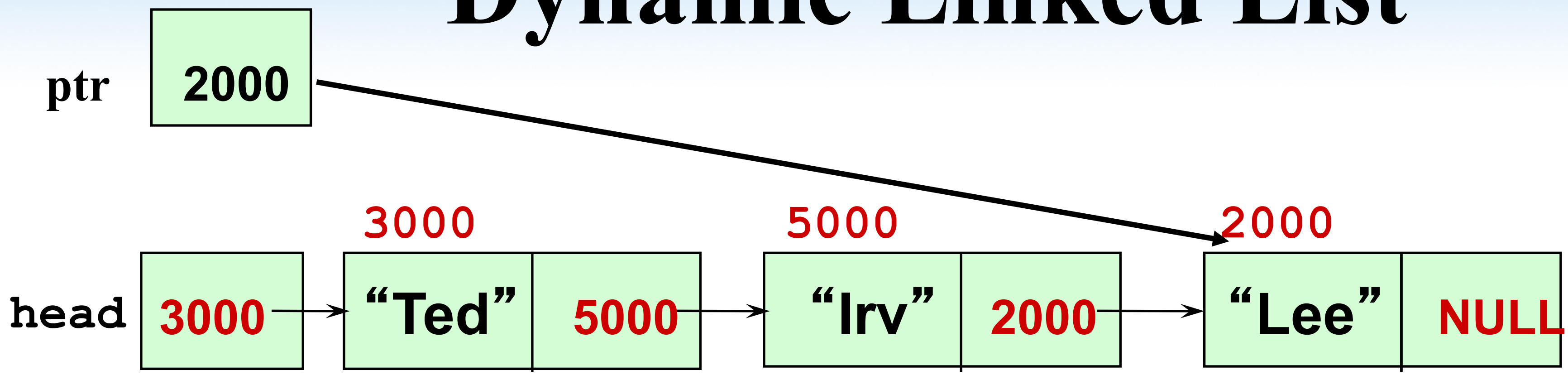

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
     // Or, do something else with node *ptr
     ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



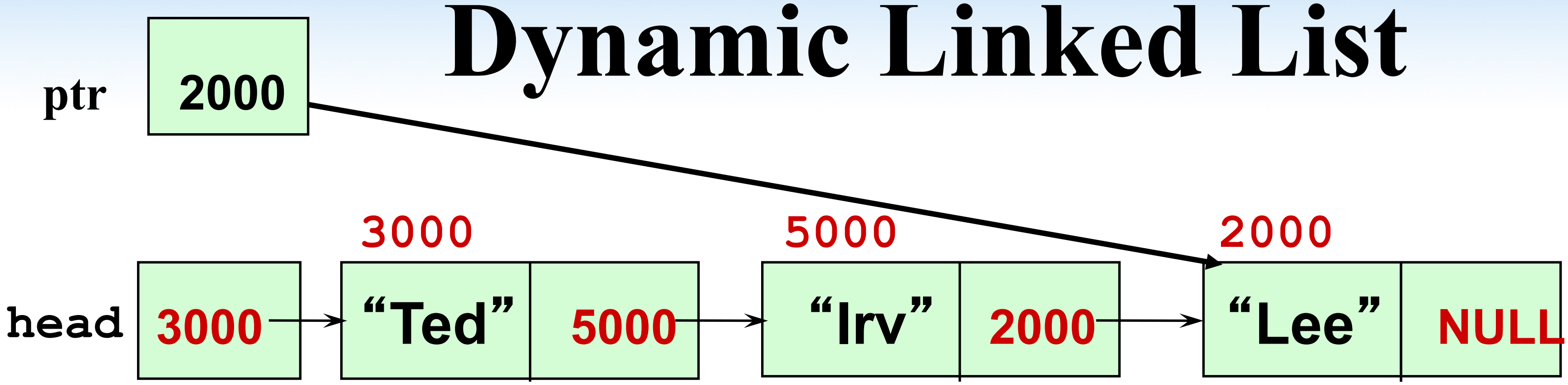

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List



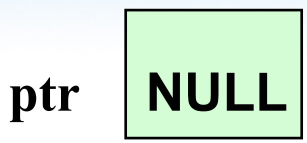

```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
    cout  <<  ptr->info;
   // Or, do something else with node *ptr
   ptr  =  ptr->link;
}
```

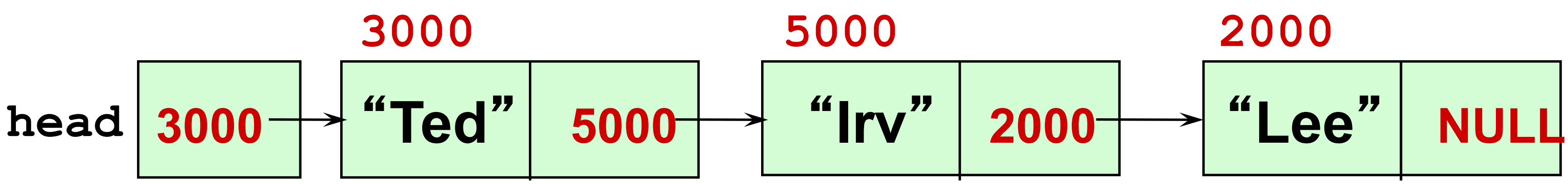
Traversing a Dynamic Linked List




```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
     // Or, do something else with node *ptr
     ptr  =  ptr->link;
}
```

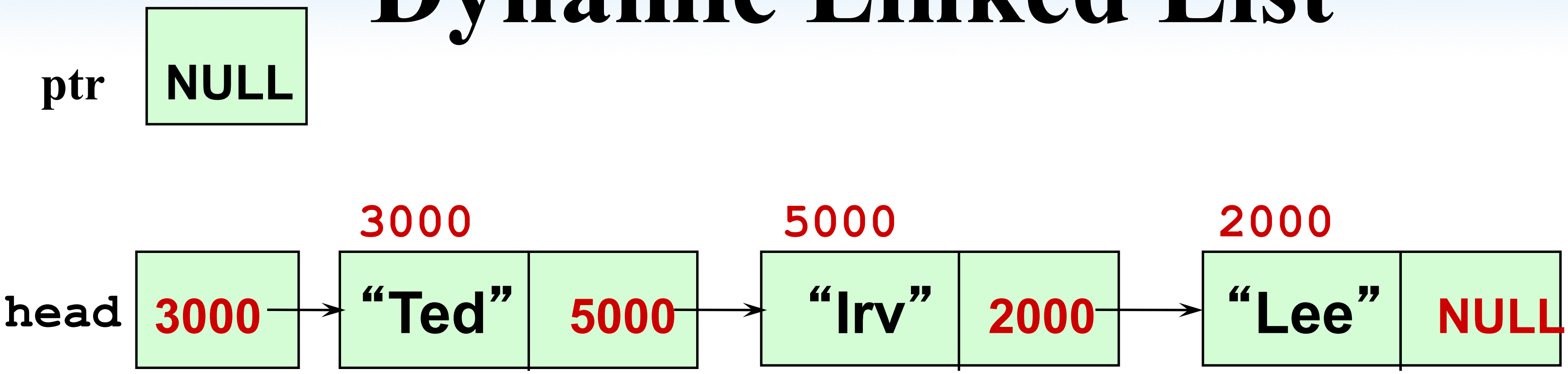
Traversing a Dynamic Linked List






```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
     // Or, do something else with node *ptr
     ptr  =  ptr->link;
}
```

Traversing a Dynamic Linked List




```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
    cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->link;
}
```

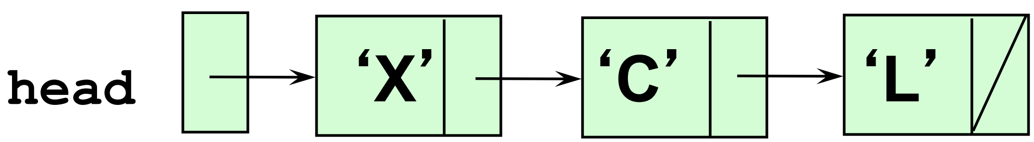
Using Operator `new`

Recall

- If memory is available in the free store (or heap), operator new allocates the requested object, and
- it returns a pointer to the memory allocated
- The dynamically allocated object exists until the delete operator destroys it

Inserting a Node at the Front of a List

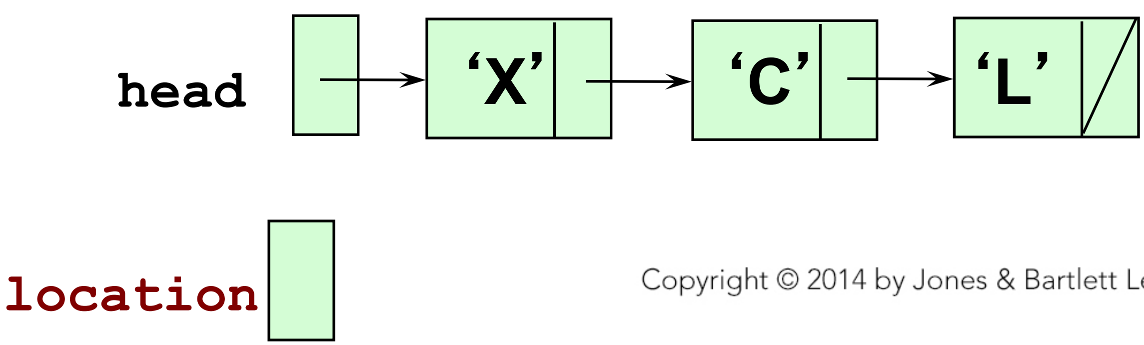
```
char       item = 'B';
NodePtr    location;
location = new  NodeType;
location->info = item;
location->link = head;
head = location;
```



Inserting a Node at the Front of a List

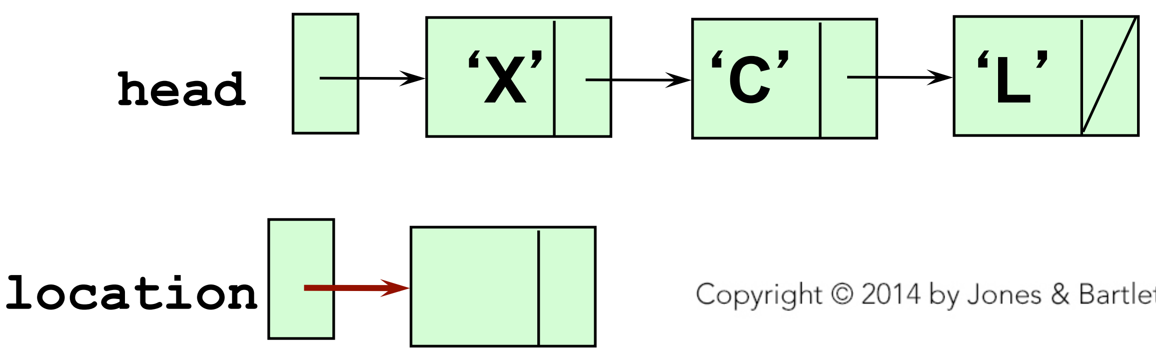
item 


```
char      item = 'B';
NodePtr   location;
location = new  NodeType;
location->info = item;
location->link = head;
head = location;
```



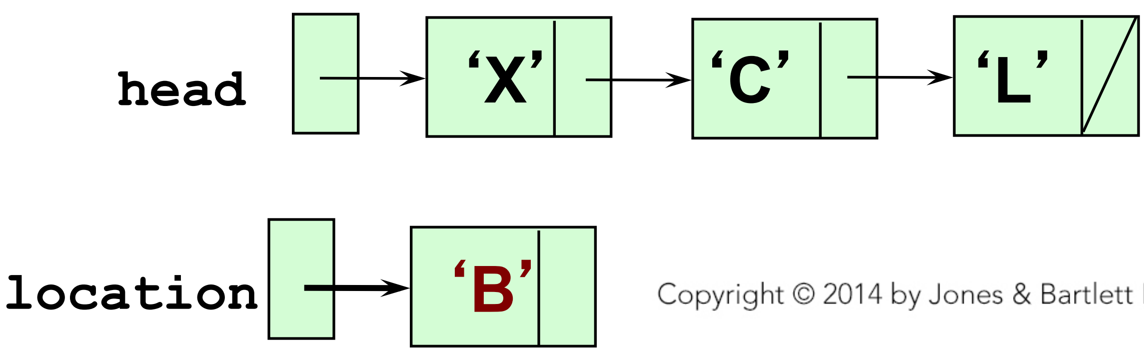
Inserting a Node at the Front of a List

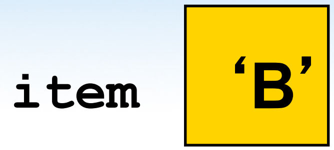

```
char       item = 'B';
NodePtr    location;
location = new  NodeType;
location->info = item;
location->link = head;
head = location;
```



Inserting a Node at the Front of a List

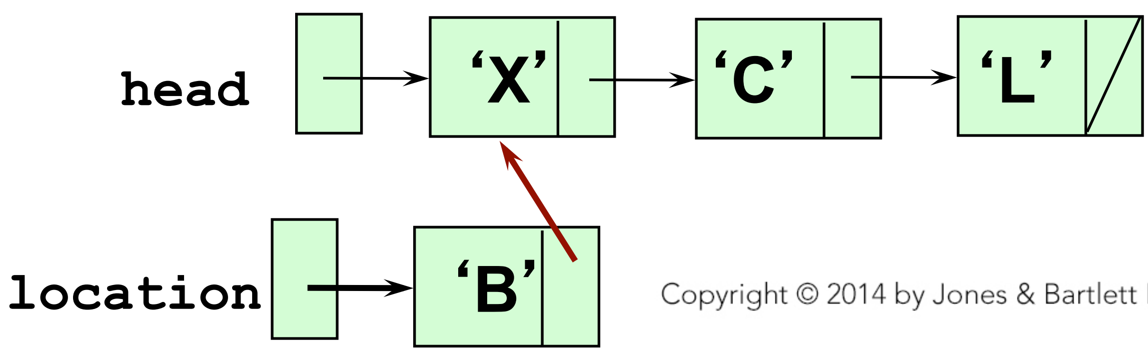

```
char        item = 'B';
NodePtr     location;
location = new  NodeType;
location->info = item;
location->link = head;
head = location;
```





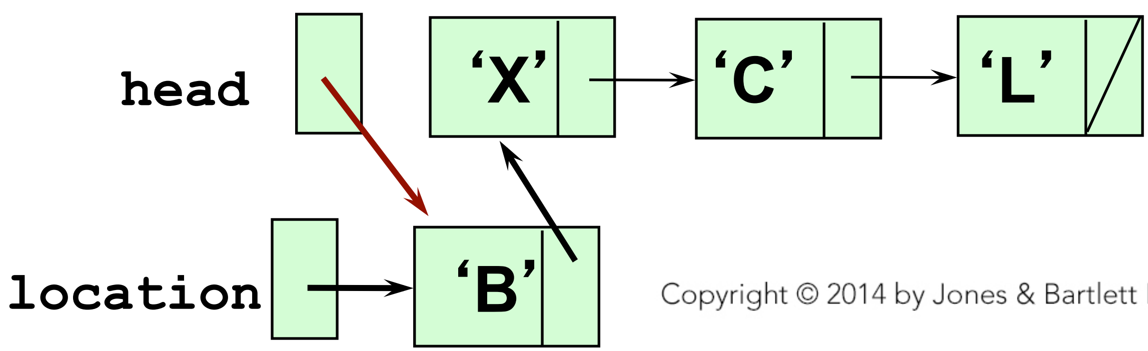
Inserting a Node at the Front of a List

```
char       item = 'B';
NodePtr    location;
location = new  NodeType;
location->info = item;
location->link = head;
head = location;
```



Inserting a Node at the Front of a List


```
char      item = 'B';
NodePtr   location;
location = new  NodeType;
location->info = item;
location->link = head;
head = location;
```



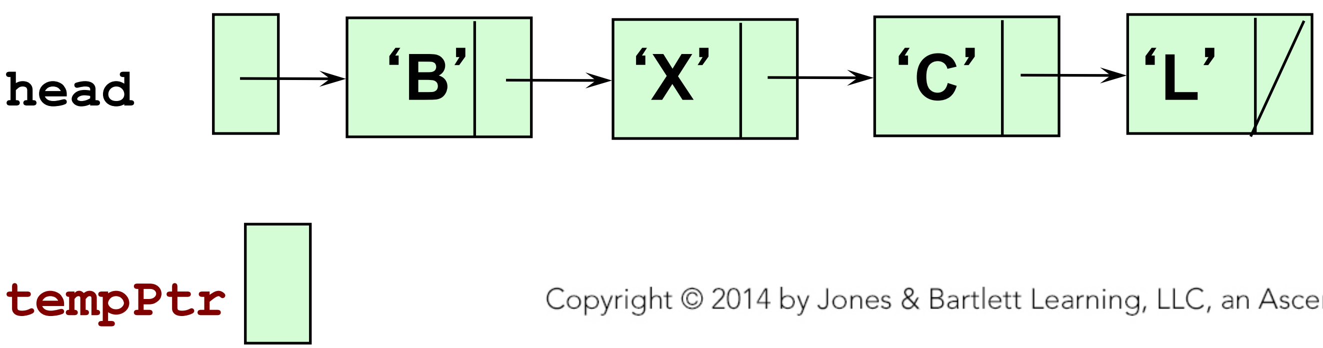
Using Operator `delete`

When you use the operator `delete`:

- **The object currently pointed to by the pointer is deallocated and the pointer is considered undefined**
- **The object's memory is returned to the free store**

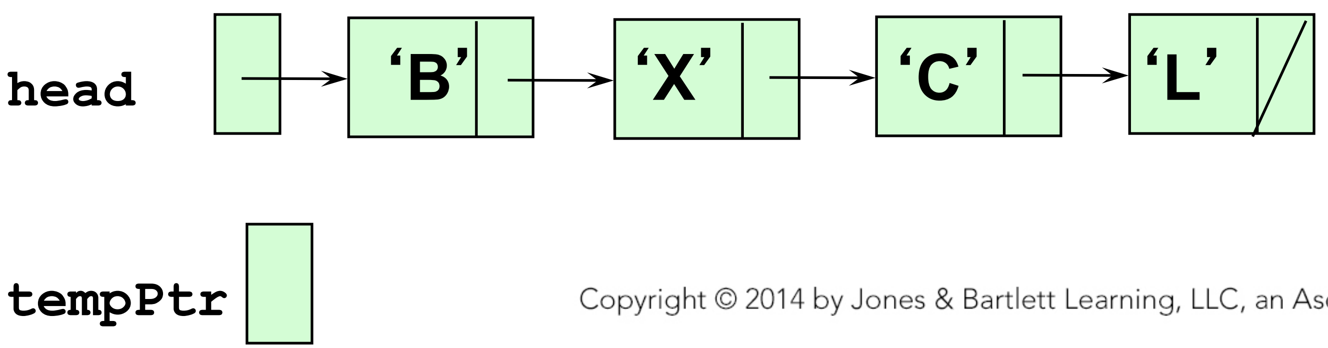
Deleting the First Node from the List


```
NodePtr  tempPtr;

item = head->info;
tempPtr = head;
head = head->link;
delete  tempPtr;
```



Deleting the First Node from the List

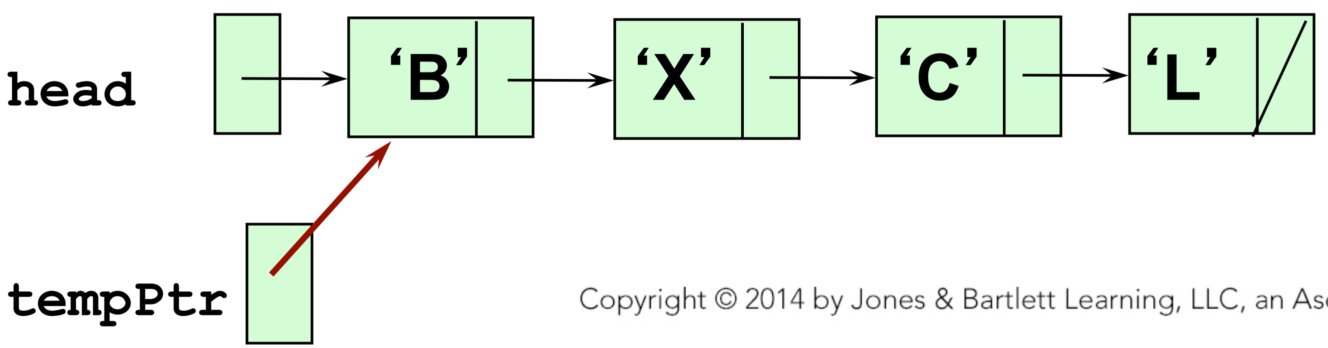
```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->link;
delete  tempPtr;
```



Deleting the First Node from the List

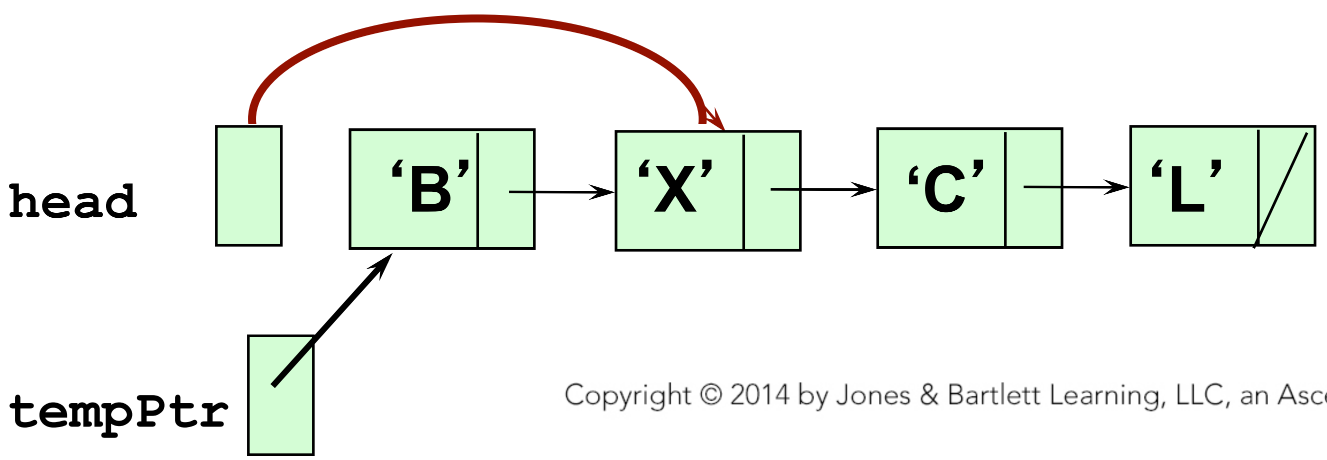
item 'B'

```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->link;
delete  tempPtr;
```



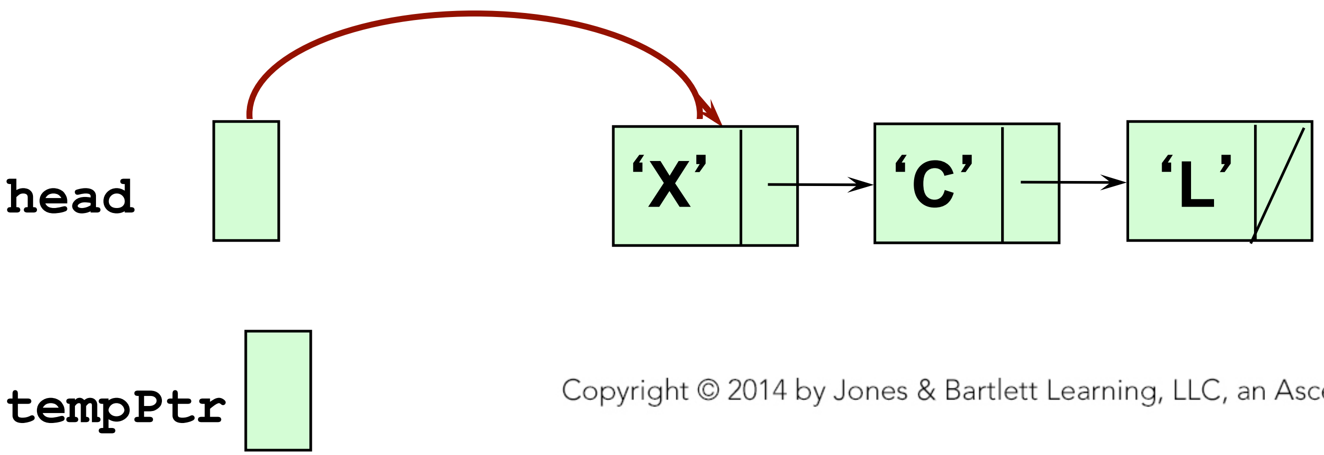
Deleting the First Node from the List


```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->link;
delete  tempPtr;
```



Deleting the First Node from the List


```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->link;
delete  tempPtr;
```



What is a Sorted List?

A sorted list is:

- a variable-length, linear collection of homogeneous elements,
- ordered according to the value of one or more data members
- The transformer operations must maintain the ordering

What is a Sorted List?

In addition to Insert and Delete, let's add two new operations to our list:

InsertAsFirst* and *RemoveFirst

ADT HybridList Operations

Transformers

- InsertAsFirst
- Insert
- RemoveFirst
- Delete

change state

Same observers and iterators as ADT List

Since we have two insertion and two deletion operations, let's call this a Hybrid List

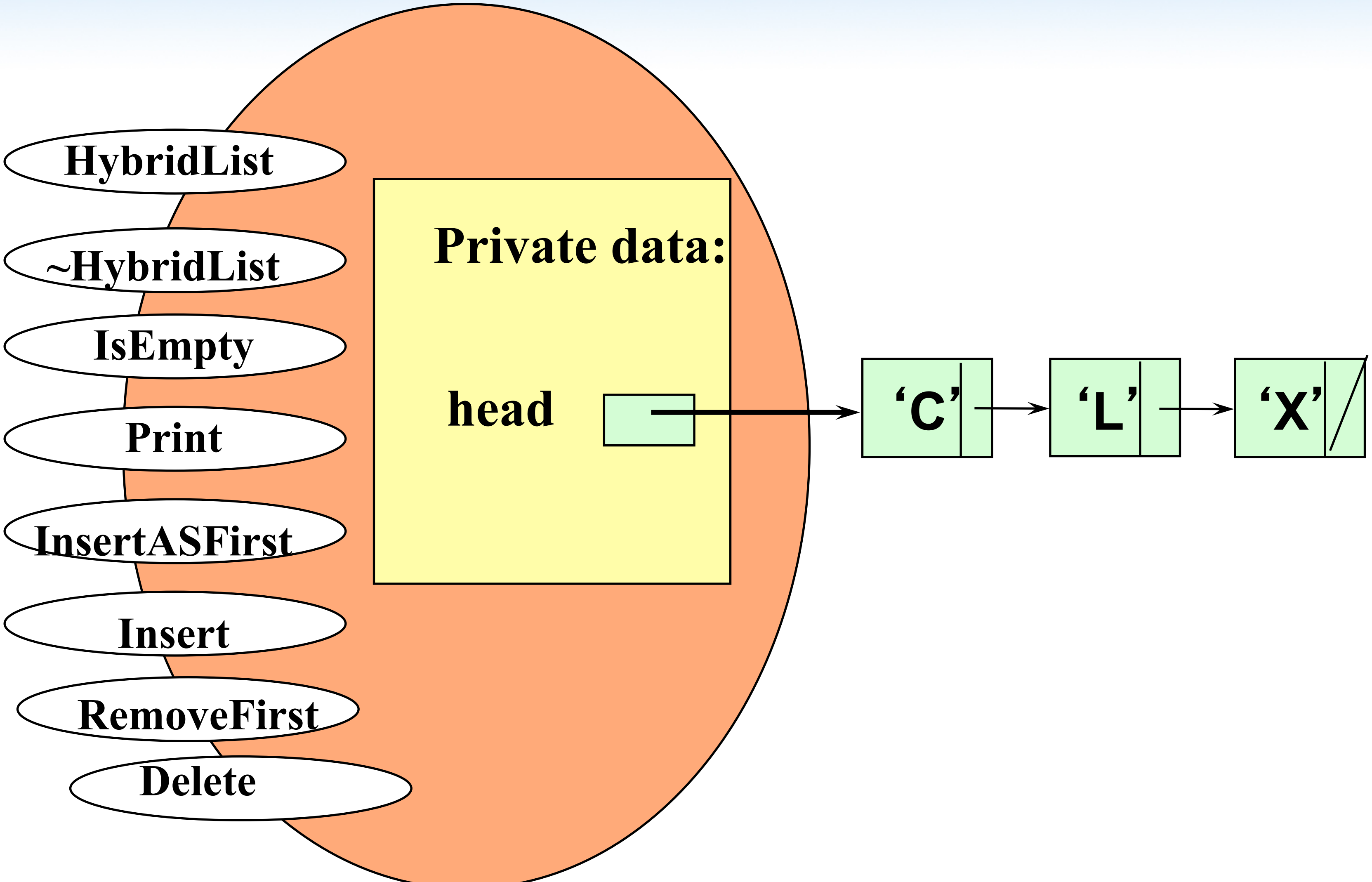
struct NodeType

```
// Specification file sorted list (“slist2.h”)

typedef int ItemType;   // Type of each component is
                        // a simple type or a string
struct  NodeType
{
    ItemType item;      // Pointer to person’s name
    NodeType* link;     // Link to next node in list
};


typedef  NodeType*  NodePtr;
```

```
// Specification file  hybrid sorted list("slist2.h")

class  HybridList
{
public:

     bool IsEmpty () const;

     void InsertAsFirst (/* in */  ItemType  item);

     void Insert (/* in */  ItemType  item);

     void RemoveFirst(/* out */  ItemType&  item);

     void Delete (/* in */  ItemType  item);
     void Print () const;
```

```
// Constructor
HybridList ();
// Destructor
~HybridList ();
// Copy-constructor
HybridList (const HybridList&  otherList);

Private:
    NodeType*  head;
};
```

class HybridList



Insert Algorithm

- ***What will be the algorithm to Insert an item into its proper place in a sorted linked list?***

- ***That is, for a linked list whose elements are maintained in ascending order?***

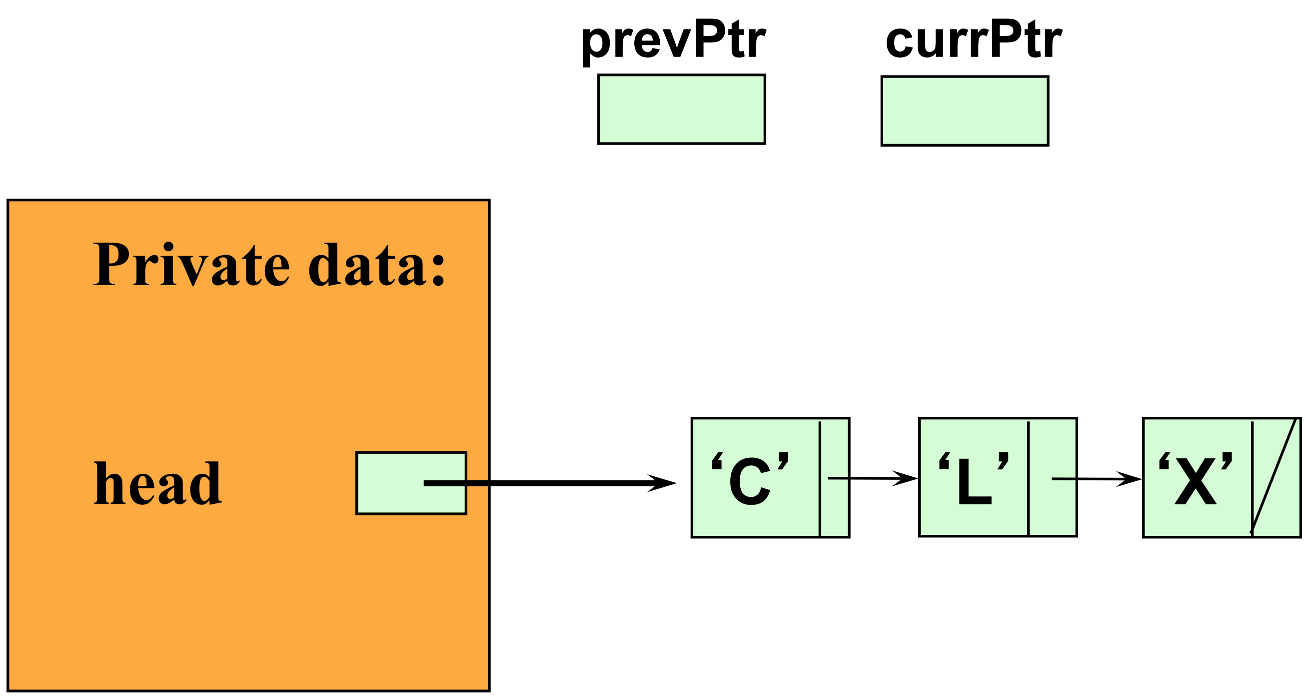
Insert algorithm for `HybridList`

- Find proper position for the new element in the sorted list using two pointers prevPtr and currPtr, where prevPtr trails behind currPtr
- Obtain a new node and place item in it
- Insert the new node by adjusting pointers

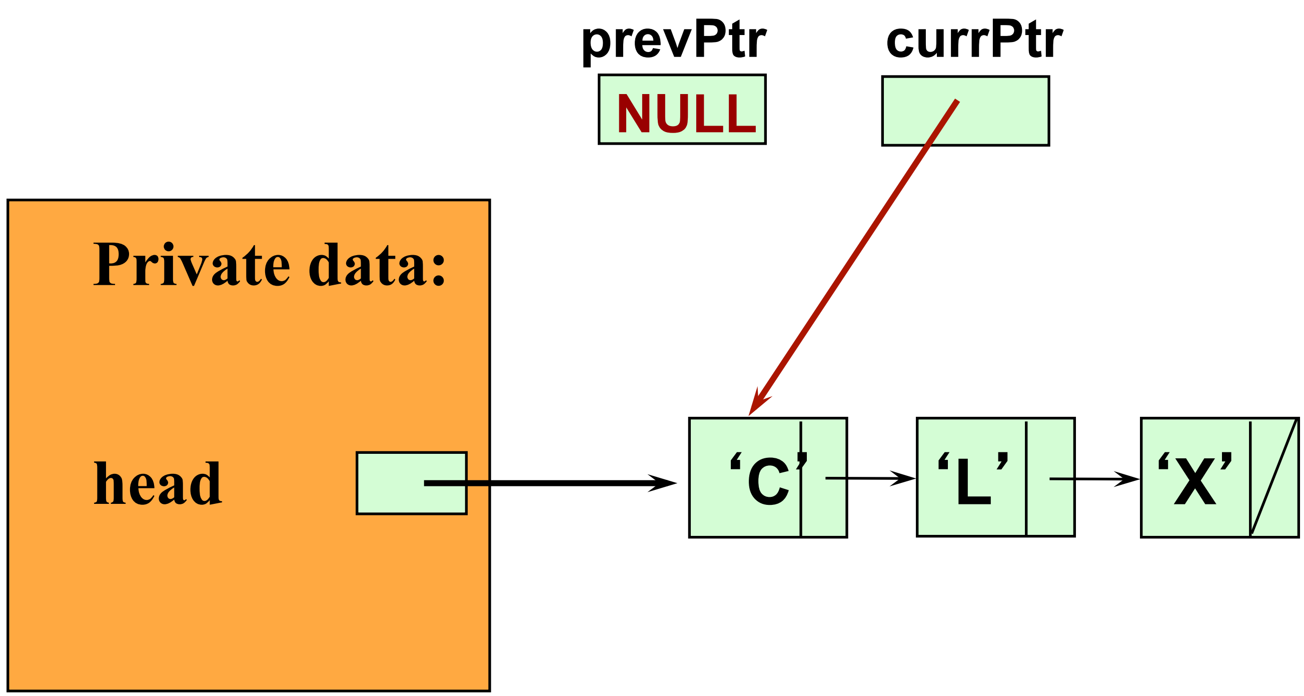
Implementing HybridList Member Function Insert

```
// Dynamic linked list implementation ("slist2.cpp")

void  HybridList::Insert (/* in */  ItemType  item)
// PRE:
//     item is assigned && components in ascending order
// POST:
//     item is in List && components in ascending order
{
             .
             .
             .

}
```

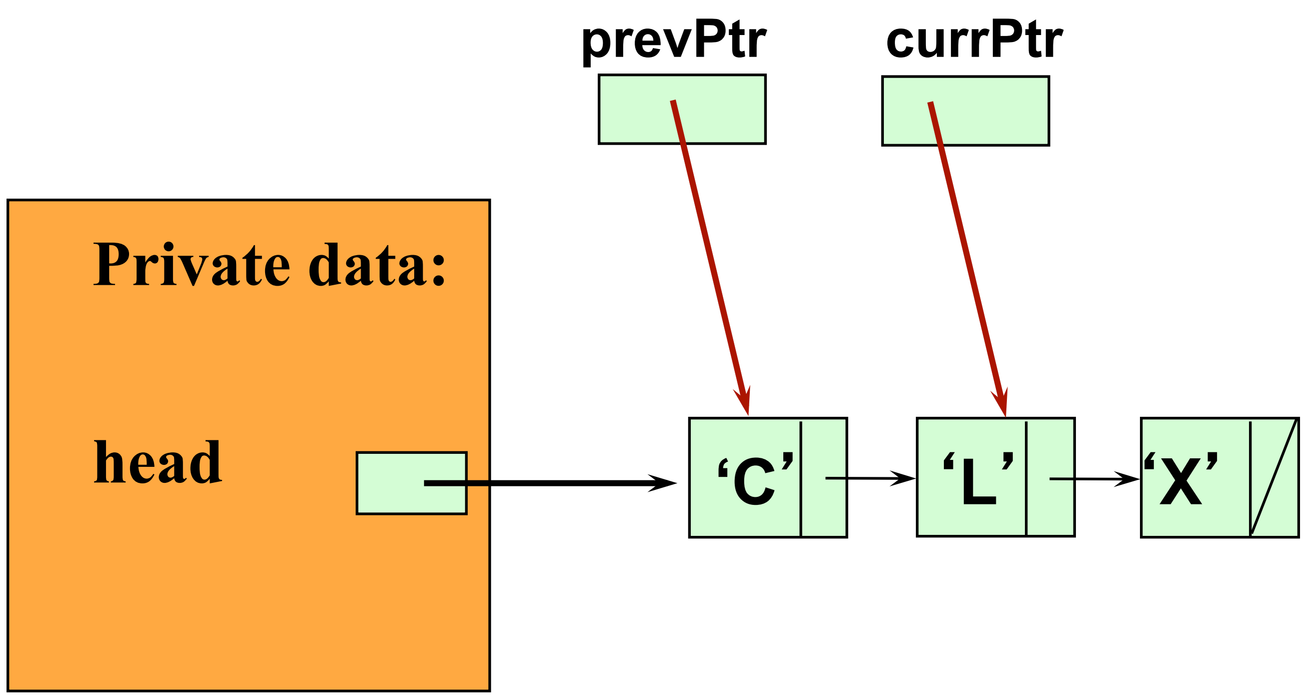
Inserting ‘S’ into a List



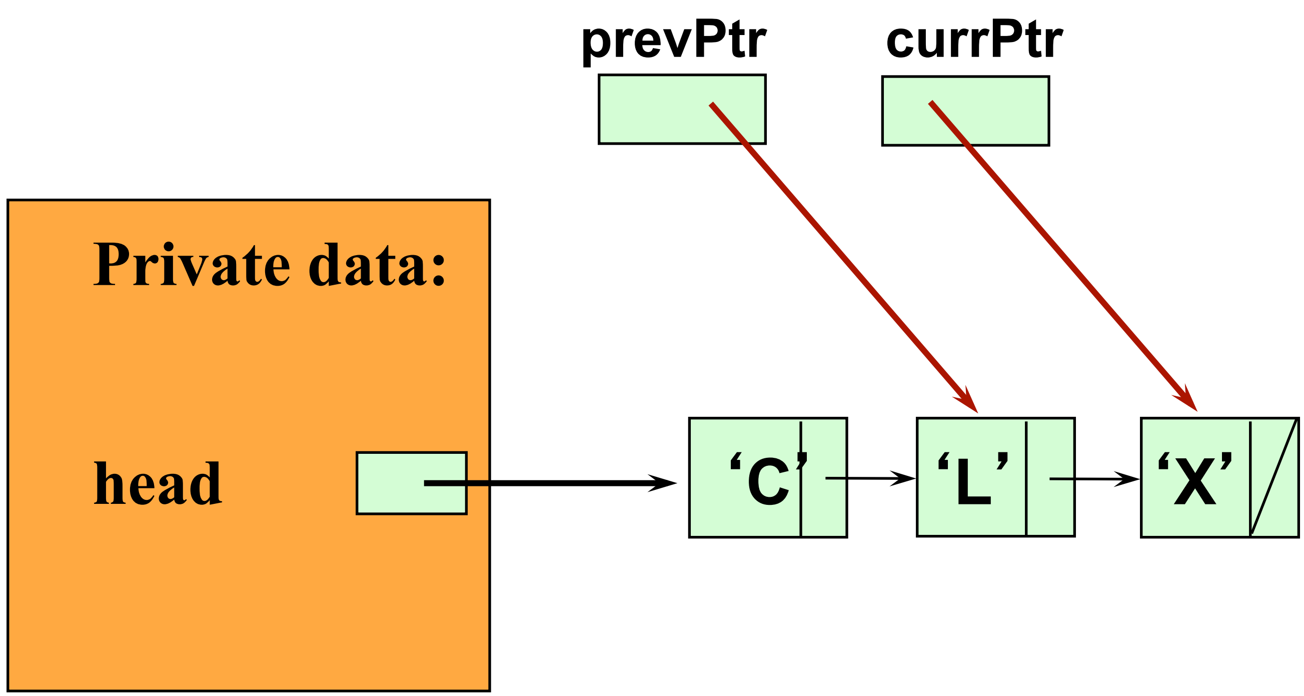
Finding Proper Position for 'S'



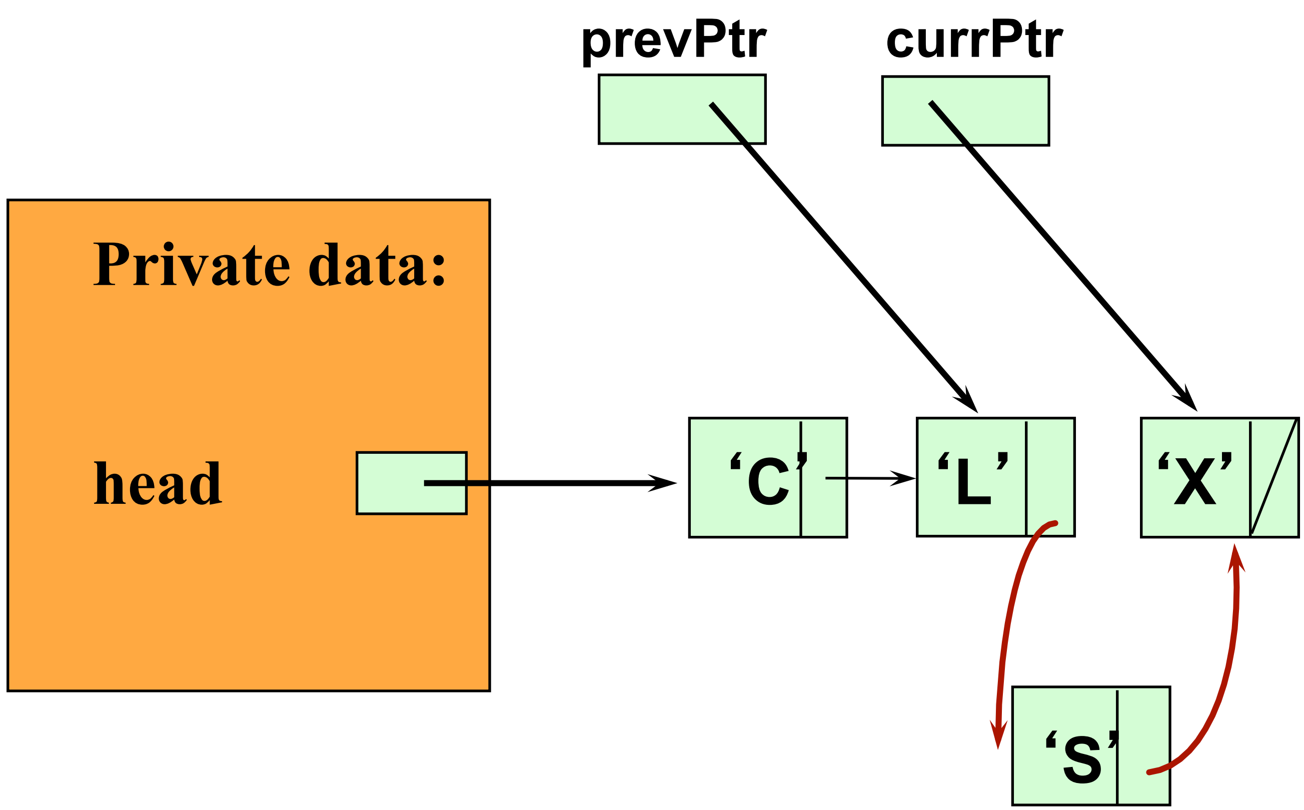
Finding Proper Position for 'S'



Finding Proper Position for 'S'



Inserting ‘S’ into Proper Position



```
// Implementation file for HybridList
   ("slist.cpp")
HybridList::HybridList ()  // Constructor
// Post: head == NULL
{
    head = NULL;
}
```

```
HybridList::~HybridList () // Destructor
// Post: All linked nodes deallocated
{
    ItemType  temp;
    // Keep deleting top node
    while (!IsEmpty)
        RemoveFirst (temp);
}
```

```
void  HybridList::Insert(/* in */  ItemType  item)
// Pre: item is assigned &&  components in
//      ascending order
// Post: new node containing item is in its
//       proper place
//       && components in ascending order
{
    NodePtr currPtr;
    NodePtr prevPtr;
    NodePtr location;
    location = new  NodeType;
   newNodePtr->link = item;
```

```
    prevPtr = NULL;
    currPtr = head;
    while (currPtr != NULL  &&  item > currPtr->info )
    {
         // Advance both pointers
         prevPtr = currPtr;
         currPtr = currPtr->link;
    }

    // Insert new node here
    location->link = currPtr;

    if  (prevPtr == NULL)
         head = location;
    else
         prevPtr->link = location;
}
```

```
void HybridList::InsertAsFirst(/* in */  ItemType  item)
// Pre: item is assigned &&  components in ascending order
// Post: New node containing item is the first item
//       in the list
//       && components in ascending order
{
    NodePtr newNodePtr = new NodeType;

    newNodePtr -> component = item;
    newNodePtr -> link = head;
    head = newNodePtr;
}
```

```
void HybridList::Print() const
// Post: All values within nodes have been printed
{
    // Loop control pointer
    NodePtr currPtr = head;
    while (currPtr != NULL)
    {
        cout << currPtr->component << endl;
        currPtr = currPtr->link;
    }
}
```

```
void HybridList::RemoveFirst (ItemType&  item)
                                  /* out */
// Pre: list is not empty && components in
//      ascending order
// Post: item == element of first list node @ entry
//    &&  node containing item is no longer in list
//    &&  list components in ascending order
{
    NodePtr  tempPtr = head;

    // Obtain item and advance head
    item = head->info ;
    head = head->link;
    delete  tempPtr;
}
```

```
void  HybridList::Delete (/* in */ ItemType  item)

//  Pre: list is not empty && components in
//       ascending order
//        &&  item == component member of some
//       list node
// Post: item == element of first list node @
//       entry
//       &&  node containing first occurrence of
//       item no longer
//       in list   &&  components in ascending
//       order
{
     NodePtr   delPtr;
     NodePtr   currPtr;
     NodePtr   prevPtr;
```

```
    if (item == head->info)
    { // If so, delete first node
        delPtr = head;
        head = head->link;
    }
    else {// Search for item in rest of list
    {
        prevPtr = NULL;
        currPtr = head;
        while(currPtr != NULL && item != currPtr->info )
       {     // Advance both pointers
             prevPtr = currPtr;
             currPtr = currPtr->link;
       }
       if (currPtr != NULL)  {
             delPtr = currPtr;
             prevPtr = currPtr->link;
       }
       else    return;
    }
    delete  delPtr;
}
```