C++ Plus Data Structures

Nell Dale

David Teague

Chapter 6

Lists Plus

ADT Sorted List Operations

Transformers

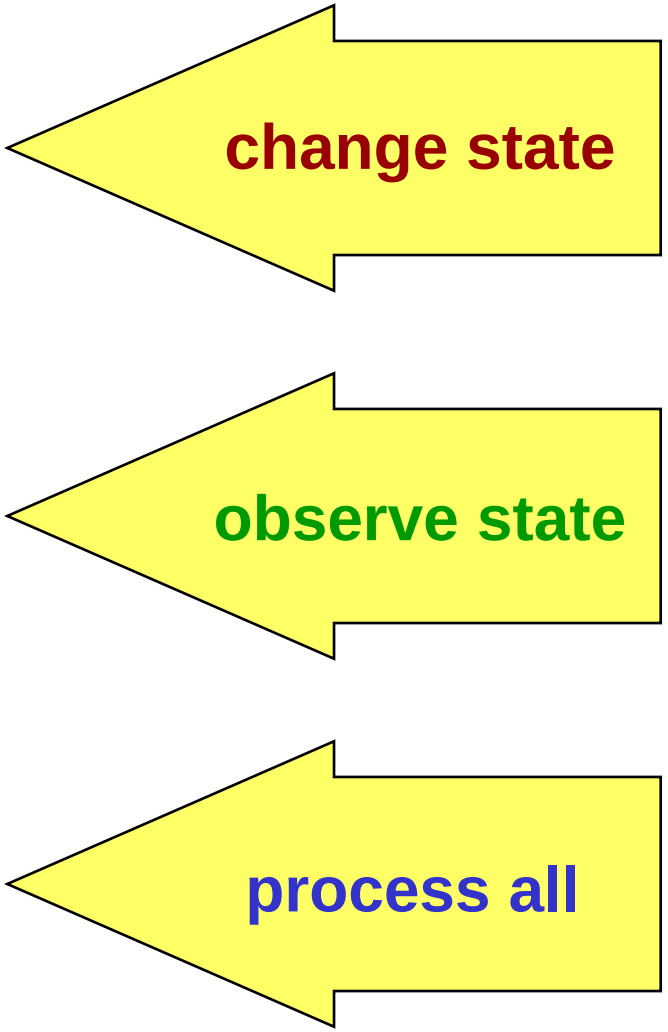
- MakeEmpty
- InsertItem
- DeleteItem

Observers

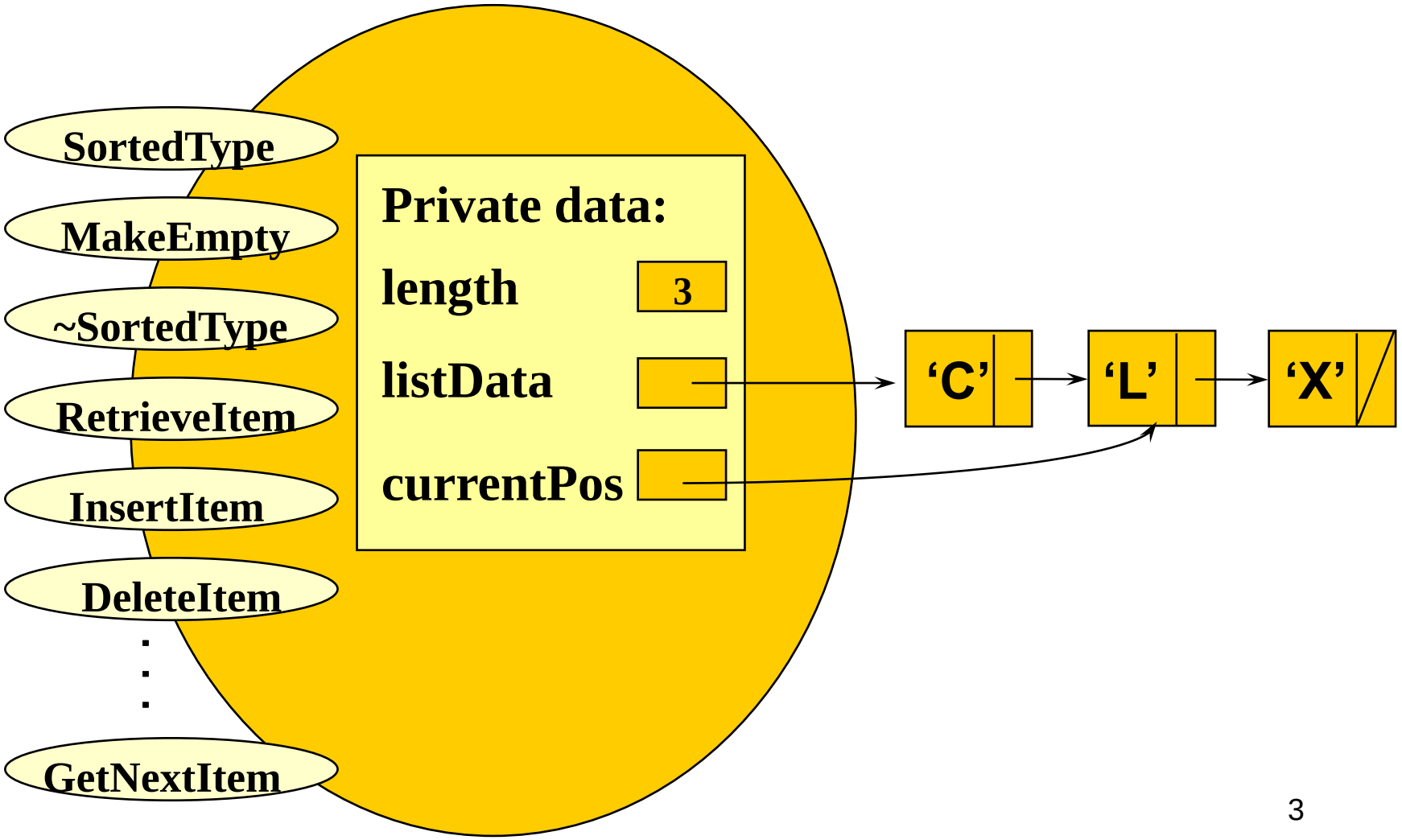
- IsFull
- LengthIs
- RetrieveItem

Iterators

- ResetList
- GetNextItem

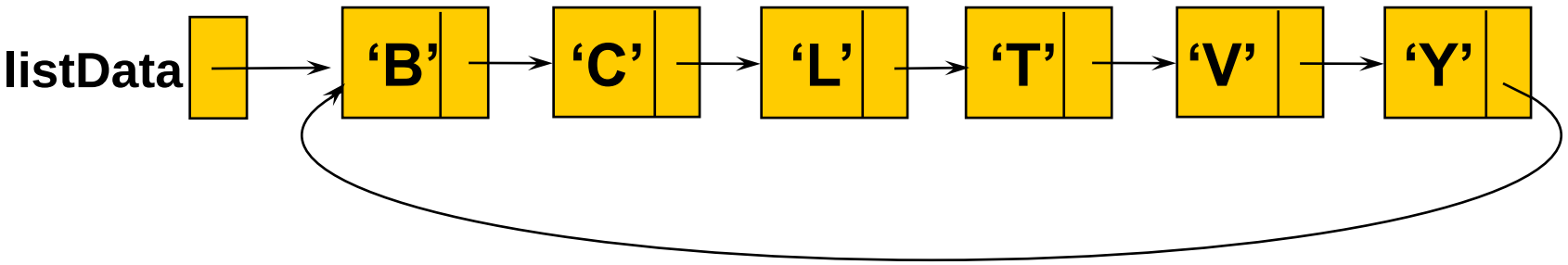


class SortedType<char>



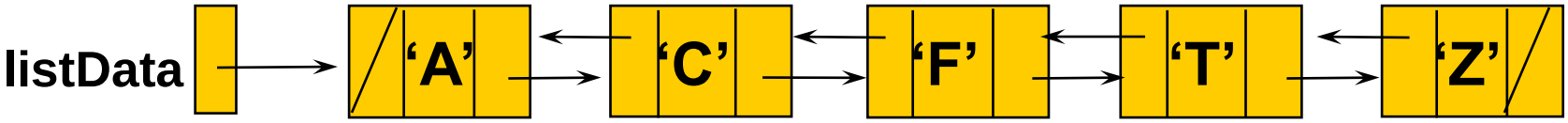
What is a Circular Linked List?

A circular linked list is a list in which every node has a successor; the "last" element is succeeded by the "first" element.



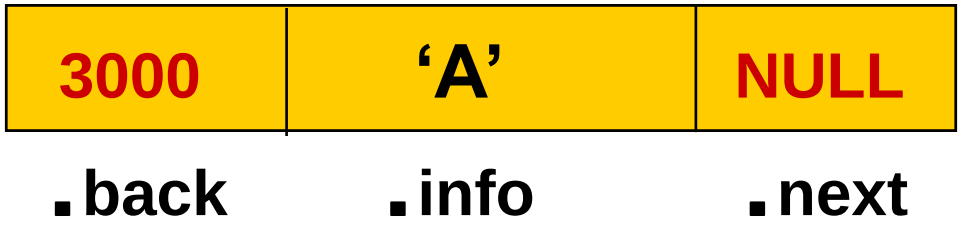
What is a Doubly Linked List?

A doubly linked list is a list in which each node is linked to both its successor and its predecessor.



Each node contains two pointers

```
template< class  ItemType >
struct   NodeType  {

   ItemType     info;                    // Data member
   NodeType<ItemType>*  back;   // Pointer to predecessor
   NodeType<ItemType>*  next;   // Pointer to successor

};
```

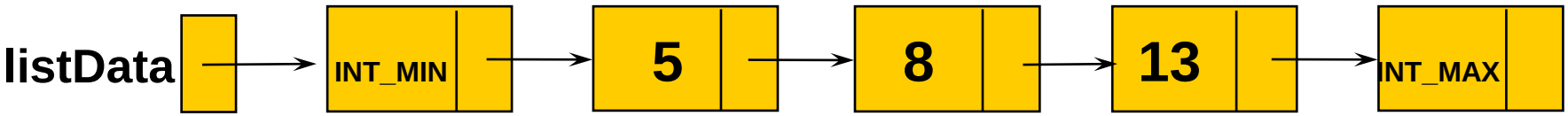


What are Header and Trailer Nodes?

A Header Node is a node at the beginning of a list that contains a key value smaller than any possible key.

A Trailer Node is a node at the end of a list that contains a key larger than any possible key.

Both header and trailer are placeholding nodes used to simplify list processing.



Recall Definition of Stack

Logical (or ADT) level: A stack is an ordered group of homogeneous items (elements), in which the removal and addition of stack items can take place only at the top of the stack.

A stack is a LIFO "last in, first out" structure.

Stack ADT Operations

MakeEmpty -- Sets stack to an empty state.

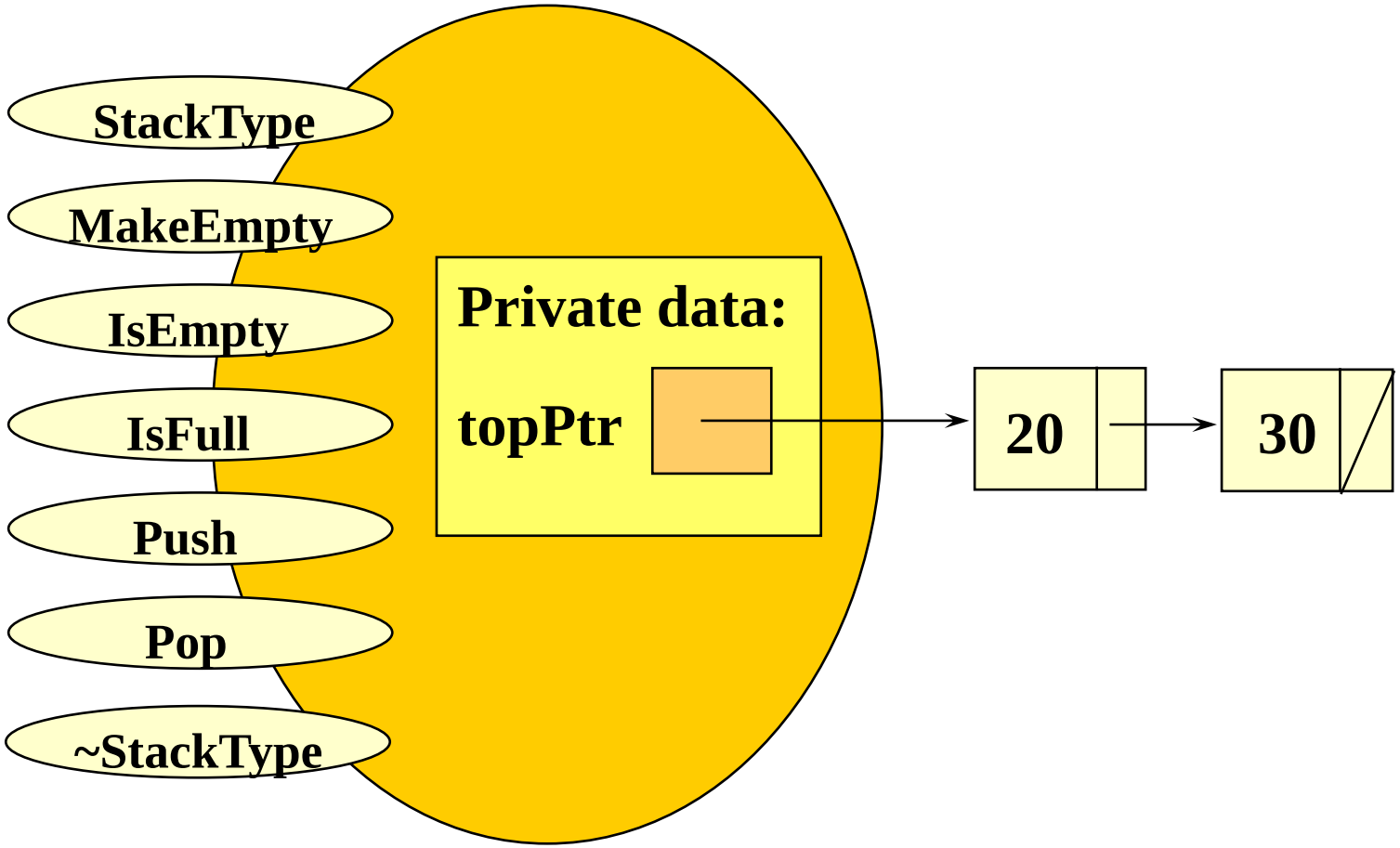
IsEmpty -- Determines whether the stack is currently empty.

IsFull -- Determines whether the stack is currently full.

Push (ItemType newItem) -- Adds newItem to the top of the stack.

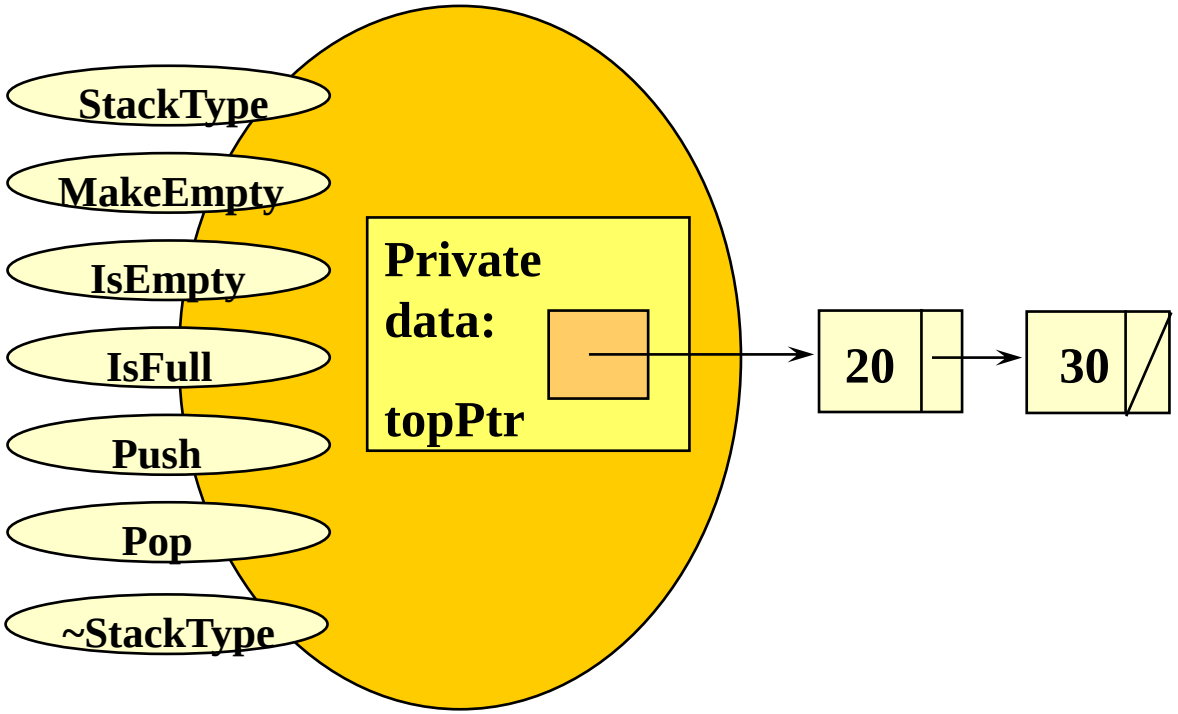
Pop (ItemType& item) -- Removes the item at the top of the stack and returns it in item.

class StackType<int>



What happens . . .

When a function is called that uses pass by value for a class object like our dynamically linked stack?

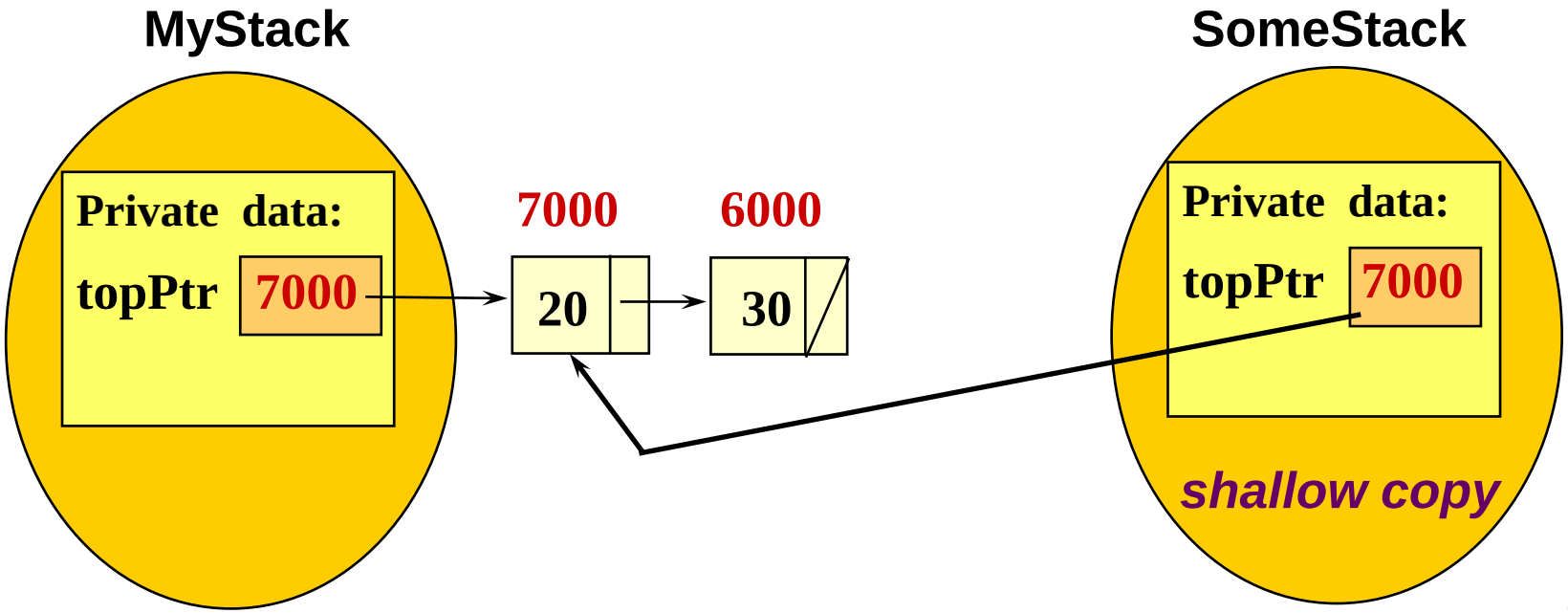


Passing a class object by value

```
                                        // FUNCTION CODE
template<class ItemType>
void  MyFunction( StackType<ItemType> SomeStack )
   // Uses pass by value
{
        .
        .
        .
        .
}
```

Pass by value makes a shallow copy

```
StackType<int>  MyStack;          // CLIENT CODE
      .
      .
      .
MyFunction( MyStack );            // function call
```



Shallow Copy vs. Deep Copy

A shallow copy copies only the class data members, and does not copy any pointed-to data.

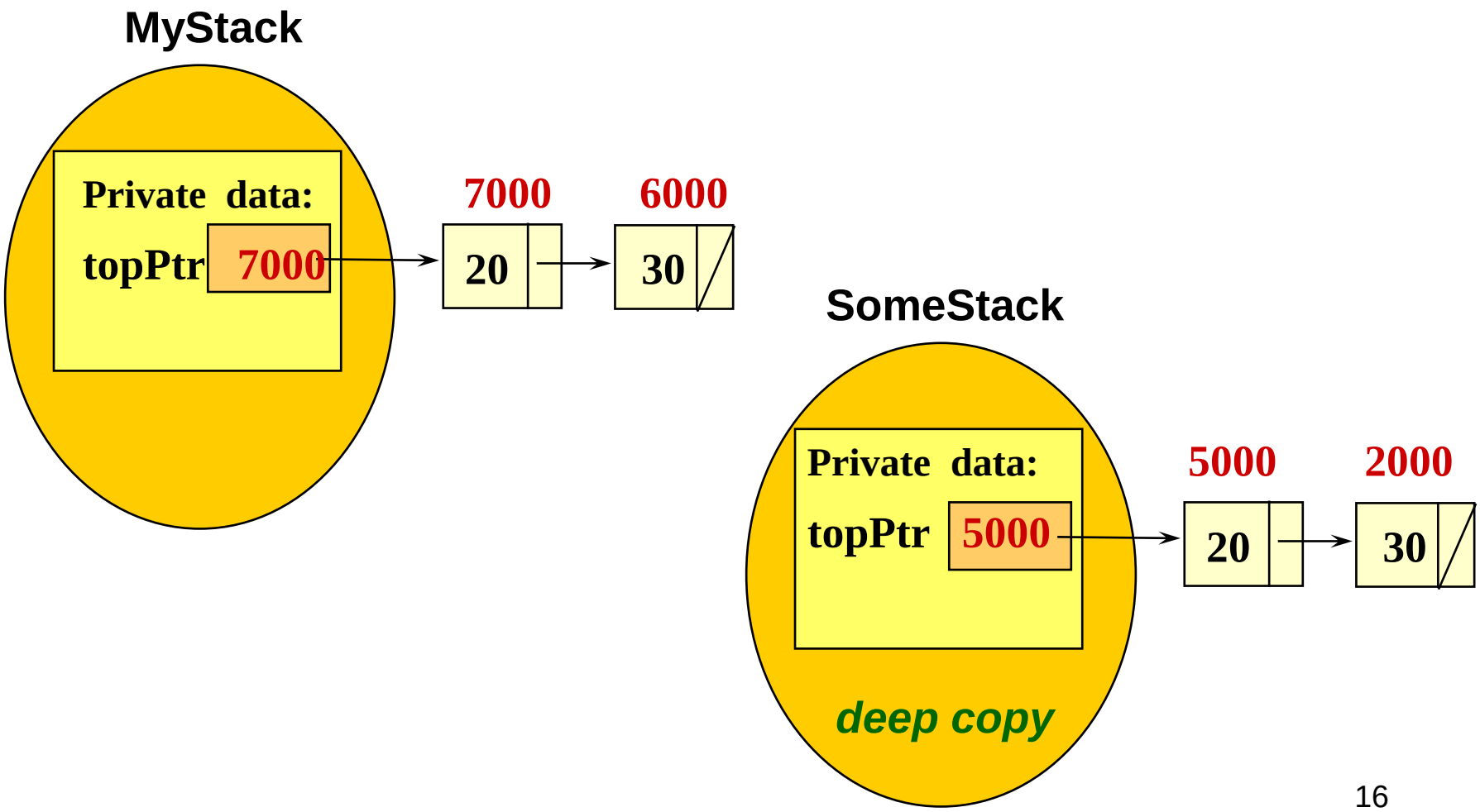
A deep copy copies not only the class data members, but also makes separately stored copies of any pointed-to data.

What's the difference?

A shallow copy shares the pointed to data with the original class object.

A deep copy stores its own copy of the pointed to data at different locations than the data in the original class object.

Making a deep copy



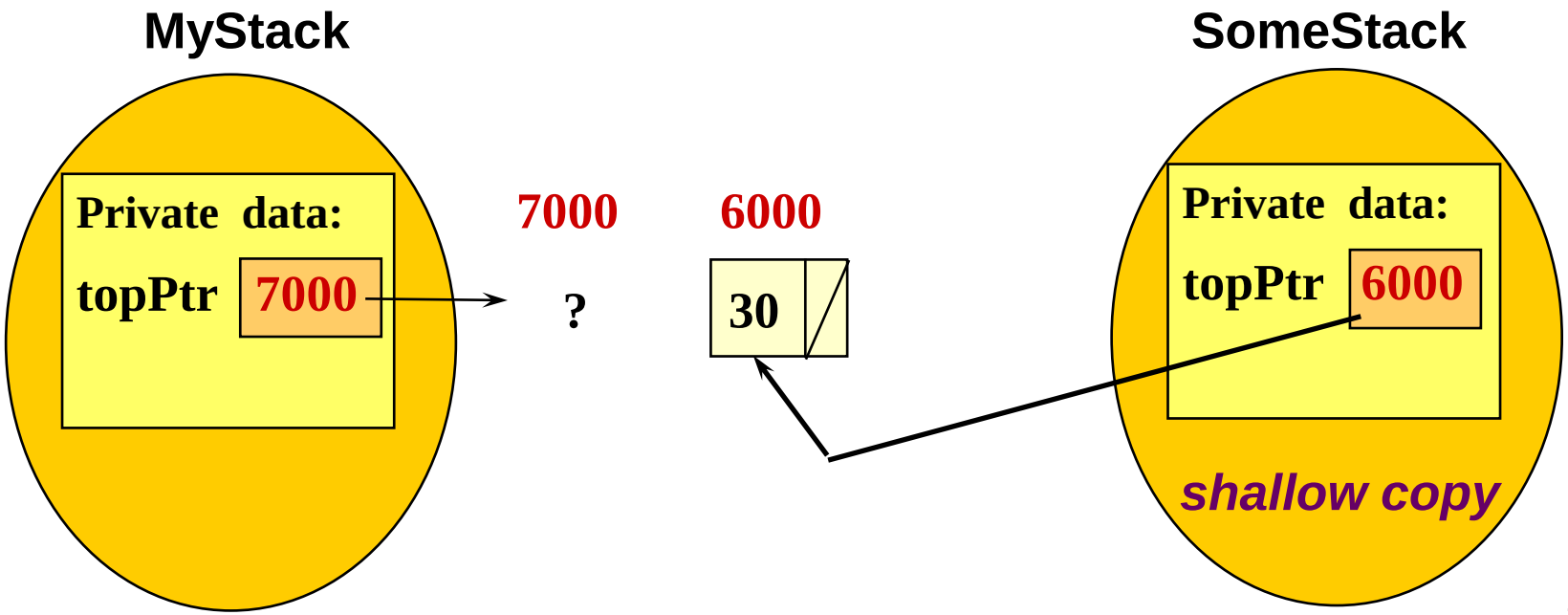
Suppose `MyFunction` Uses Pop

```
                                        // FUNCTION CODE
template<class ItemType>
void  MyFunction( StackType<ItemType> SomeStack )
   // Uses pass by value
{
      ItemType  item;
      SomeStack.Pop(item);
      .
      .
      .
}
```

WHAT HAPPENS IN THE SHALLOW COPY SCENARIO?

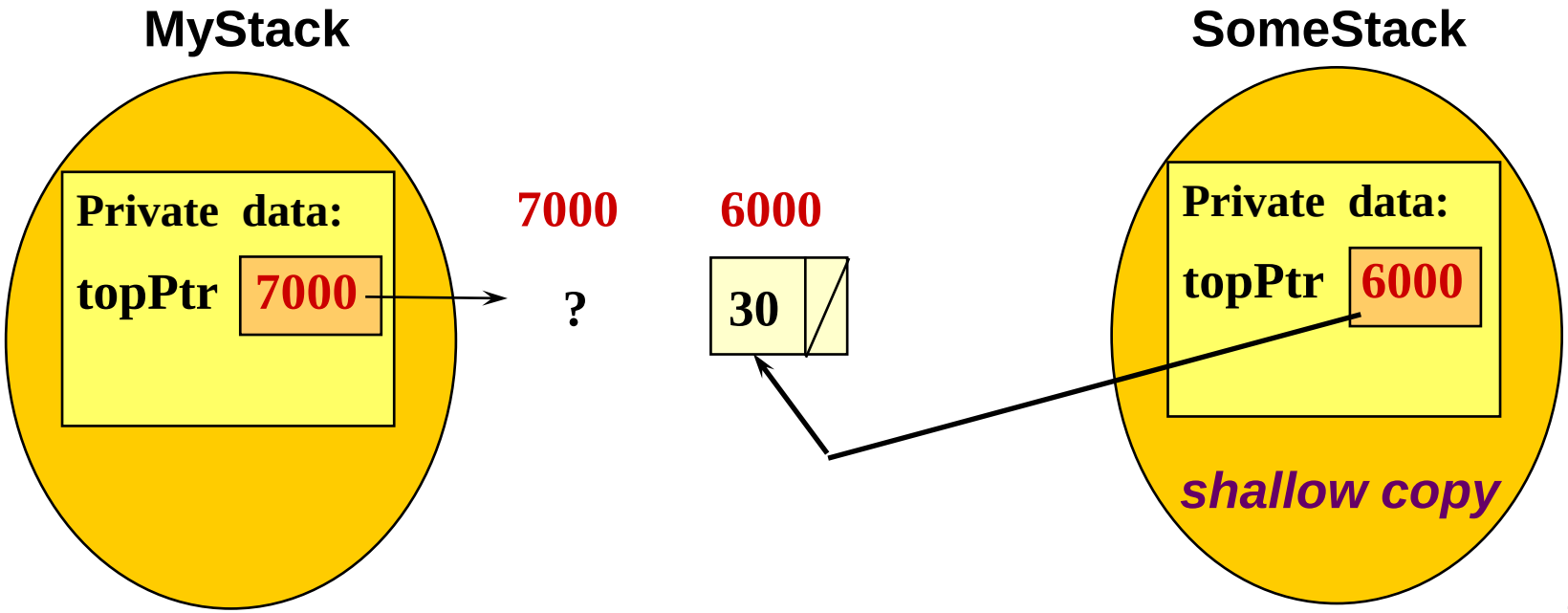
`MyStack.topPtr` is left dangling

```
StackType<int>  MyStack;      // CLIENT CODE
      .
      .
      .
MyFunction( MyStack );
```



`MyStack.topPtr` is left dangling

NOTICE THAT NOT JUST FOR THE SHALLOW COPY, BUT ALSO FOR ACTUAL PARAMETER MyStack, THE DYNAMIC DATA HAS CHANGED!



As a result . . .

This default method used for pass by value is not the best way when a data member pointer points to dynamic data.

Instead, you should write what is called a copy constructor, which makes a deep copy of the dynamic data in a different memory location.

More about copy constructors

When there is a copy constructor provided for a class, the copy constructor is used to make copies for pass by value.

You do not 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.

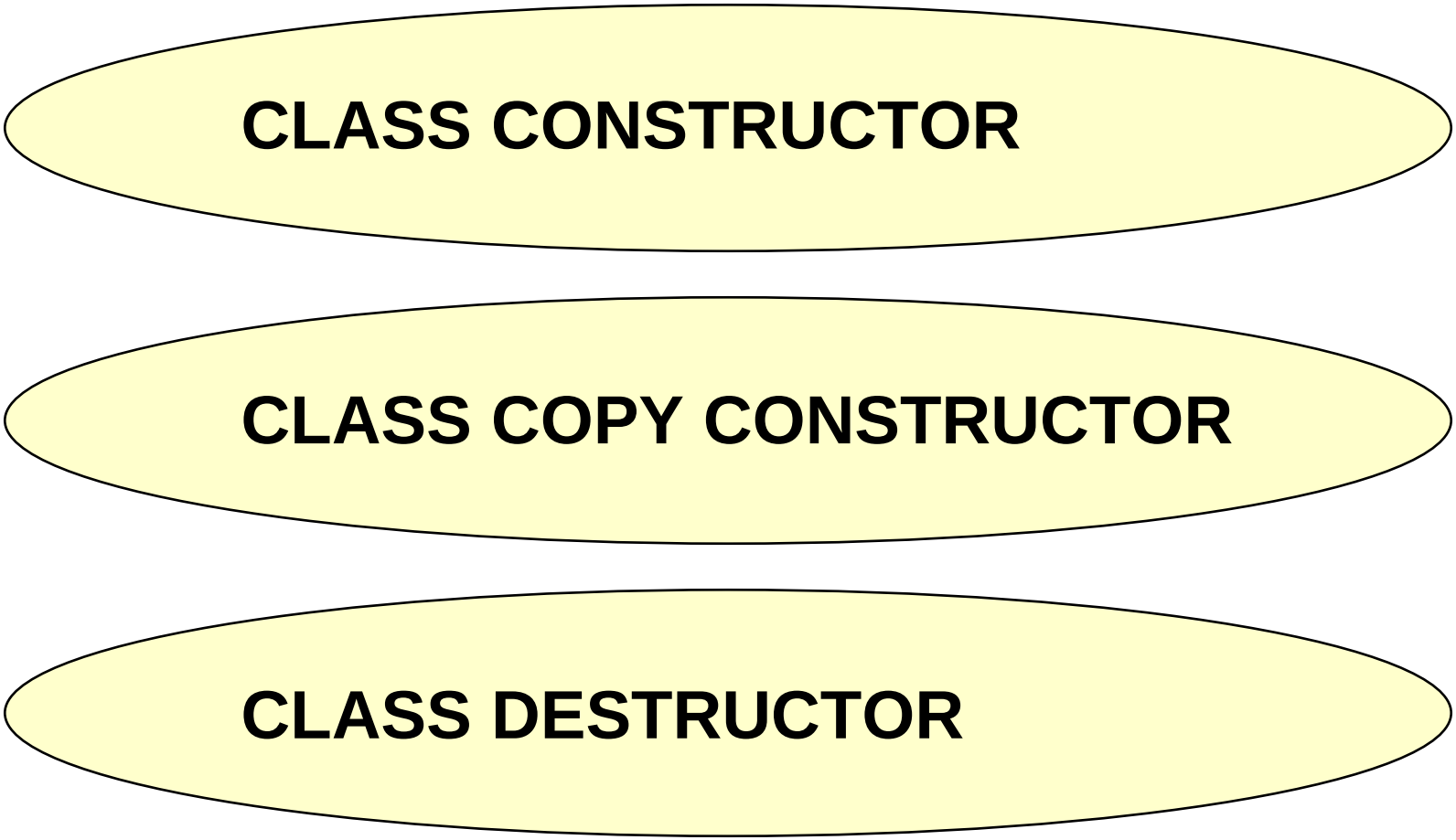
Copy Constructor

Copy constructor is a special member function of a class that is implicitly called in these three situations:

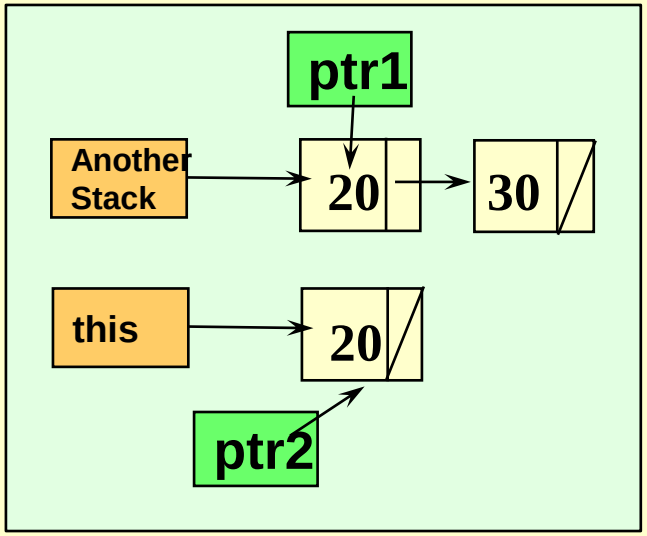
- **passing object parameters by value,**
- **initializing an object variable in a declaration, (ex: int a=b;)**
- **returning an object as the return value of a function.**

```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK

template<class ItemType>
class StackType  {
public:
   StackType( );
        // Default constructor.
        // POST:  Stack is created and empty.
   StackType( const StackType<ItemType>& anotherStack );
        // Copy constructor.
        // Implicitly called for pass by value.

   .
   .
   .
   ~StackType( );
        // Destructor.
        // POST: Memory for nodes has been deallocated.
private:
   NodeType<ItemType>*  topPtr ;
};
```

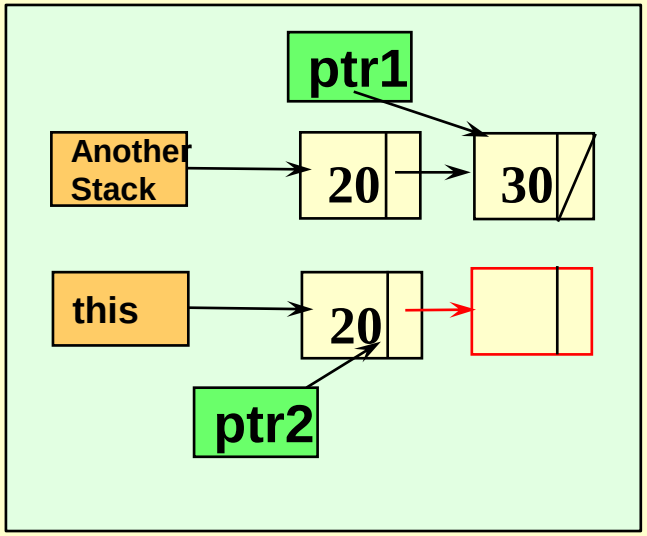
Classes with Data Member Pointers Need



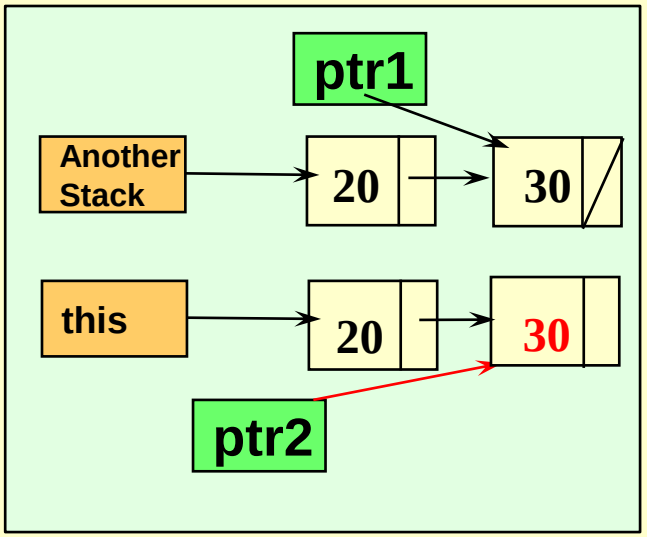
```
template<class ItemType>          // COPY CONSTRUCTOR
StackType<ItemType>::
StackType( const StackType<ItemType>& anotherStack )
{  NodeType<ItemType>* ptr1 ;
   NodeType<ItemType>* ptr2 ;
   if ( anotherStack.topPtr == NULL )
        topPtr = NULL ;
   else
   {    ptr2 = new NodeType<ItemType> ;
        topPtr = ptr2;
        ptr1 = anotherStack.topPtr;
        ptr2->info = ptr1->info ;
        ptr1 = ptr1->next;
        while ( ptr1 != NULL )      // deep copy other nodes
        {       ptr2->next = new NodeType<ItemType> ;
                ptr2->info = ptr1->info ;
                ptr2 = ptr2->next ;
                ptr1 = ptr1->next ;
        }
        ptr2->next = NULL ;
   }}
```



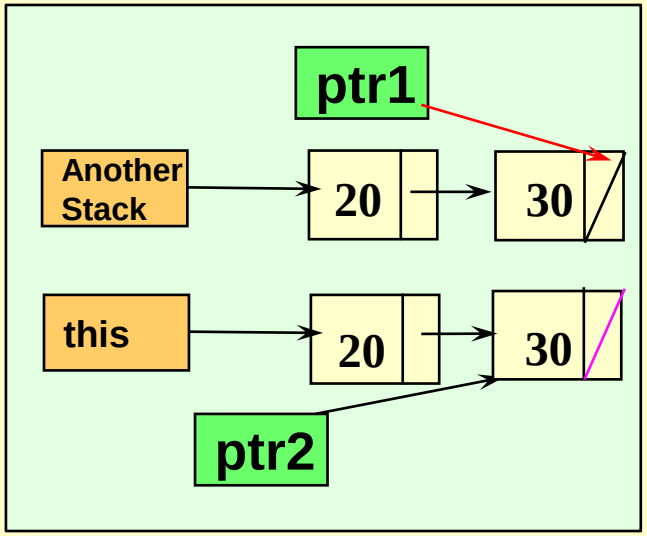
```
template<class ItemType>          // COPY CONSTRUCTOR
StackType<ItemType>::
StackType( const StackType<ItemType>& anotherStack )
{  NodeType<ItemType>* ptr1 ;
   NodeType<ItemType>* ptr2 ;
   if ( anotherStack.topPtr == NULL )
        topPtr = NULL ;
   else
   {    ptr2 = new NodeType<ItemType> ;
        topPtr = ptr2;
        ptr1 = anotherStack.topPtr;
        ptr2->info = ptr1->info ;
        ptr1 = ptr1->next;
        while ( ptr1 != NULL )      // deep copy other nodes
        {       ptr2->next = new NodeType<ItemType> ;
                ptr2->info = ptr1->info ;
                ptr2 = ptr2->next ;
                ptr1 = ptr1->next ;
        }
        ptr2->next = NULL ;
   }}
```



```
template<class ItemType>          // COPY CONSTRUCTOR
StackType<ItemType>::
StackType( const StackType<ItemType>& anotherStack )
{  NodeType<ItemType>* ptr1 ;
   NodeType<ItemType>* ptr2 ;
   if ( anotherStack.topPtr == NULL )
        topPtr = NULL ;
   else
   {    ptr2 = new NodeType<ItemType> ;
        topPtr = ptr2;
        ptr1 = anotherStack.topPtr;
        ptr2->info = ptr1->info ;
        ptr1 = ptr1->next;
        while ( ptr1 != NULL )      // deep copy other nodes
        {       ptr2->next = new NodeType<ItemType> ;
                ptr2->info = ptr1->info ;
                ptr2 = ptr2->next ;
                ptr1 = ptr1->next ;
        }
        ptr2->next = NULL ;
   }}
```



```
template<class ItemType>          // COPY CONSTRUCTOR
StackType<ItemType>::
StackType( const StackType<ItemType>& anotherStack )
{  NodeType<ItemType>* ptr1 ;
   NodeType<ItemType>* ptr2 ;
   if ( anotherStack.topPtr == NULL )
        topPtr = NULL ;
   else
   {    ptr2 = new NodeType<ItemType> ;
        topPtr = ptr2;
        ptr1 = anotherStack.topPtr;
        ptr2->info = ptr1->info ;
        ptr1 = ptr1->next;
        while ( ptr1 != NULL )      // deep copy other nodes
        {       ptr2->next = new NodeType<ItemType> ;
                ptr2->info = ptr1->info ;
                ptr2 = ptr2->next ;
                ptr1 = ptr1->next ;
        }
        ptr2->next = NULL ;
   }}
```



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 pointer to dynamic data, you should write a member function to overload the assignment operator to make a deep copy of the dynamic data.

```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK

template<class ItemType>
class StackType  {
public:
   StackType( );
        // Default constructor.
   StackType( const StackType<ItemType>& anotherStack );
        // Copy constructor.
   void  operator= ( StackType<ItemType> );
        // Overloads assignment operator.

   .
   .
   .

   ~StackType( );
        // Destructor.
private:
   NodeType<ItemType>*  topPtr ;
};
```

C++ Operator Overloading Guides

1. **All operators except these `::` `.` `sizeof` `?:` may be overloaded.**
2. **At least one operand must be a class instance.**
3. **You cannot change precedence, operator symbols, or number of operands.**
4. **Overloading ++ and -- requires prefix form use by default, unless special mechanism is used.**
5. **To overload these operators `=` `()` `[]` member functions (not friend functions) must be used.**
6. **An operator can be given multiple meanings if the data types of operands differ.**

Using Overloaded Binary operator+

When a Member Function was defined

myStack + yourStack

myStack.operator+(yourStack)

When a Friend Function was defined

myStack + yourStack

operator+(myStack, yourStack)

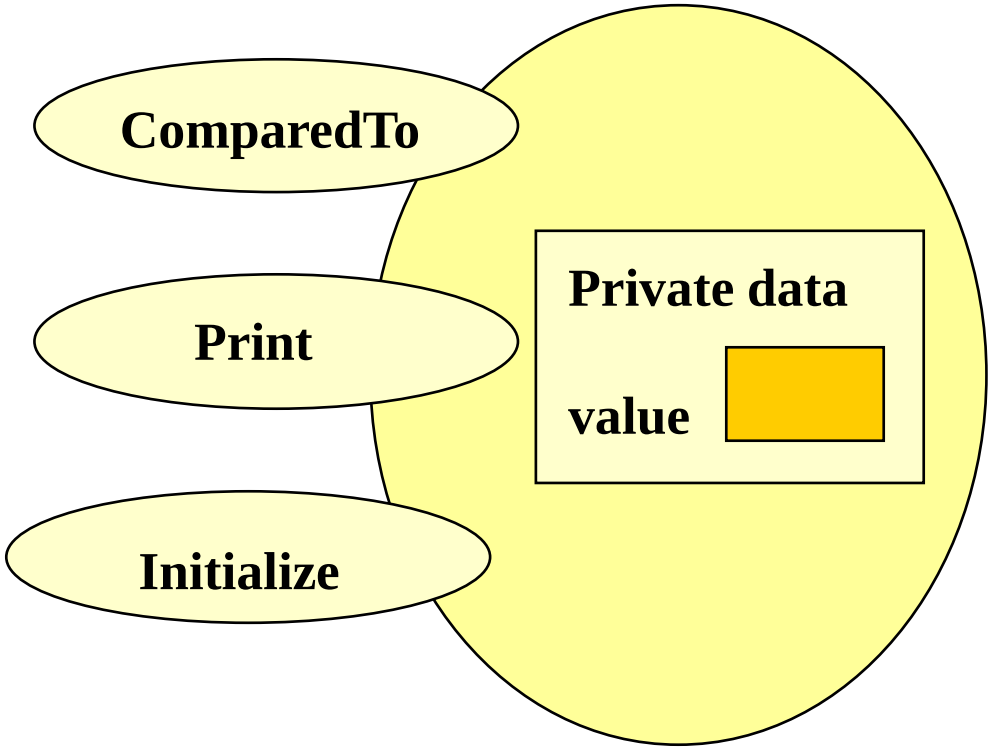
Composition (containment)

Composition (or containment) means that an internal data member of one class is defined to be an object of another class type.

A FAMILIAR EXAMPLE . . .

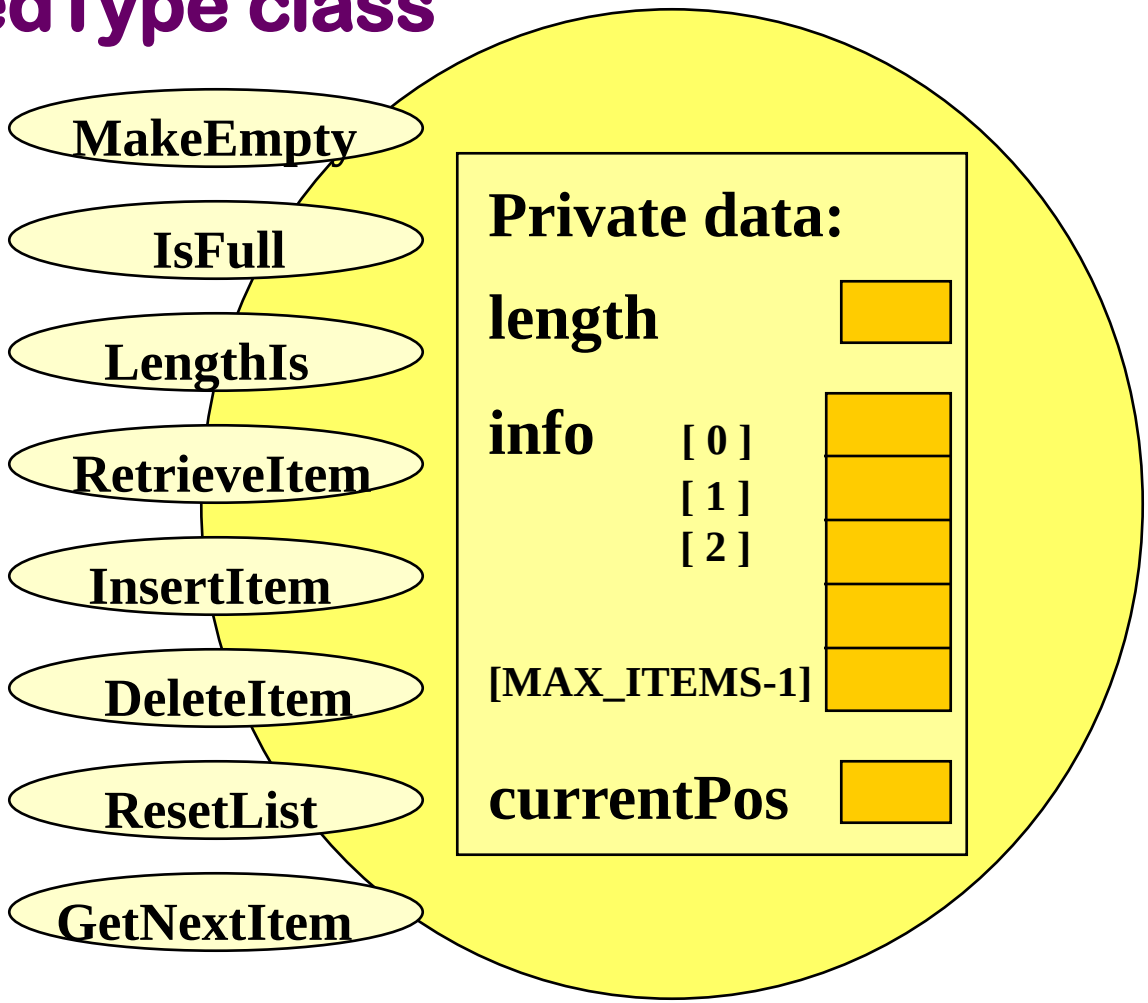
ItemType
Class Interface Diagram

class ItemType



Sorted list contains an array of ItemType

SortedType class



Inheritance

Inheritance is a means by which one class acquires the properties--both data and operations--of another class.

When this occurs, the class being inherited from is called the Base Class.

The class that inherits is called the Derived Class.

AN EXAMPLE . . .

Recall Definition of Queue

Logical (or ADT) level: A queue is an ordered group of homogeneous items (elements), in which new elements are added at one end (the rear), and elements are removed from the other end (the front).

A queue is a FIFO "first in, first out" structure.

Queue ADT Operations

MakeEmpty -- Sets queue to an empty state.

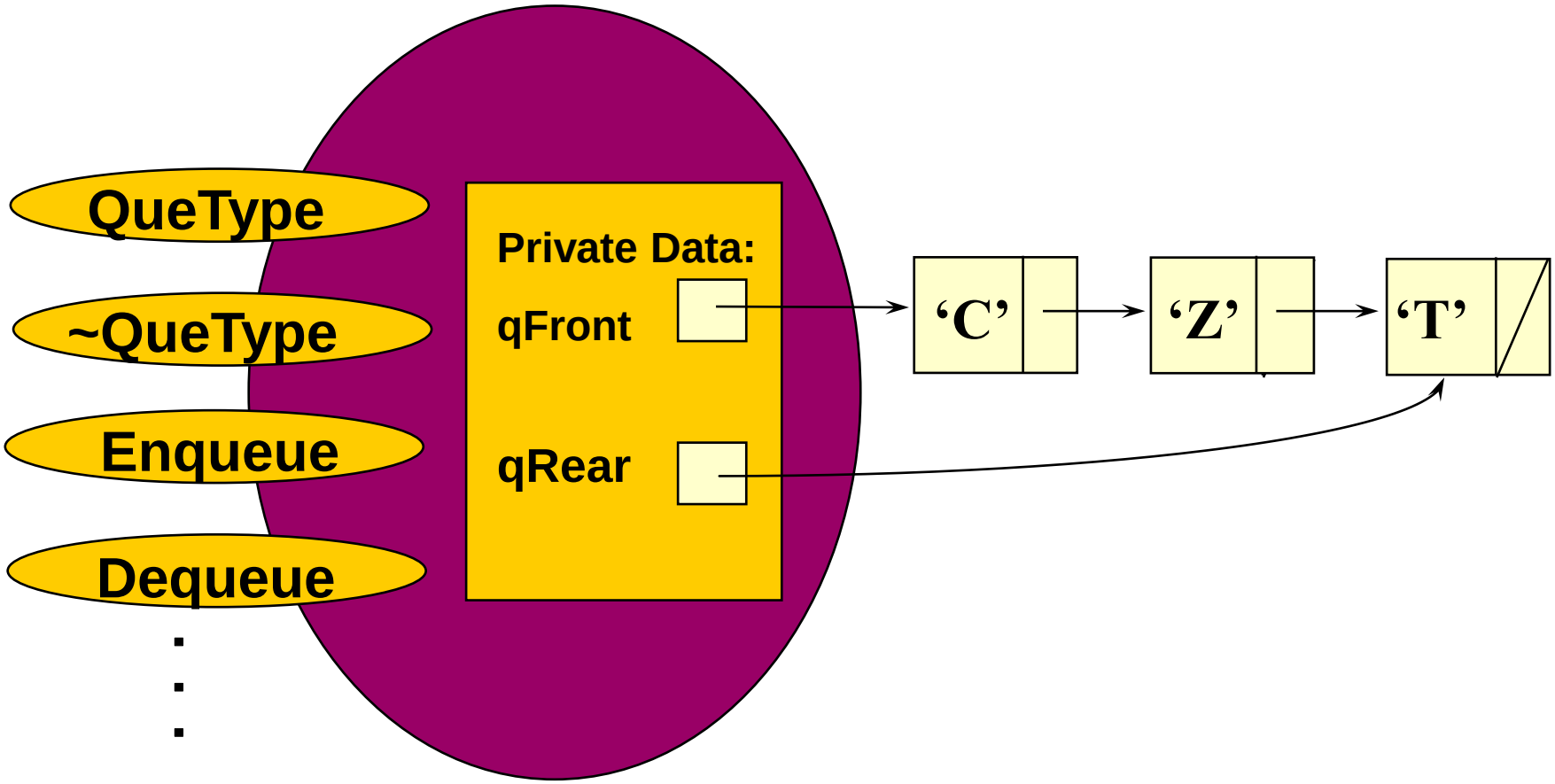
IsEmpty -- Determines whether the queue is currently empty.

IsFull -- Determines whether the queue is currently full.

Enqueue (ItemType newItem) -- Adds newItem to the rear of the queue.

Dequeue (ItemType& item) -- Removes the item at the front of the queue and returns it in item.

class QueType<char>

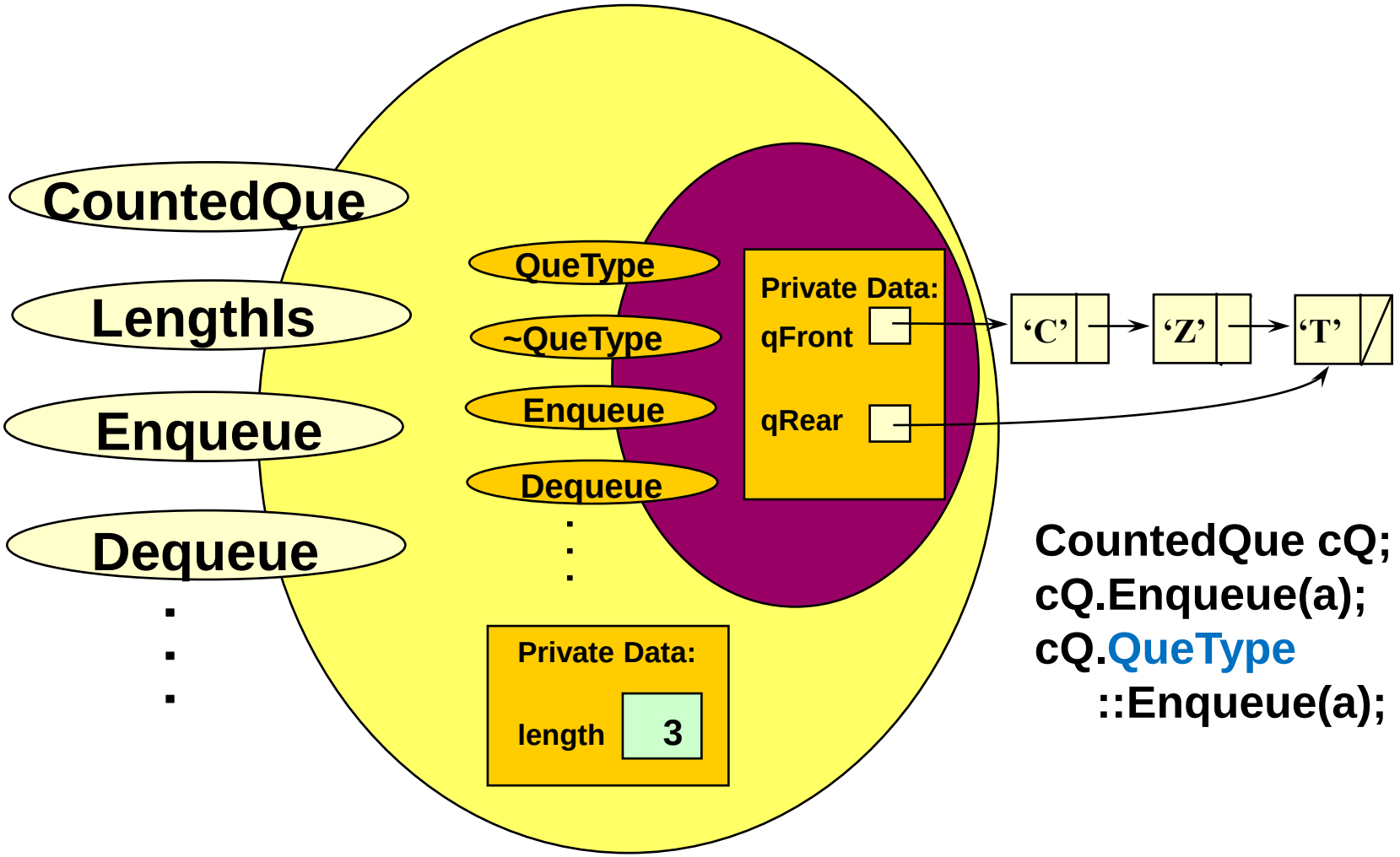


```
// DYNAMICALLY LINKED IMPLEMENTATION OF QUEUE

#include "ItemType.h"       // for ItemType

template<class ItemType>
class QueType  {
public:
   QueType( );                  // CONSTRUCTOR
   ~QueType( ) ;                // DESTRUCTOR
   bool IsEmpty( ) const;
   bool IsFull( ) const;
   void Enqueue( ItemType item );
   void Dequeue( ItemType&  item );
   void MakeEmpty( );
private:
   NodeType<ItemType>*  qFront;
   NodeType<ItemType>*  qRear;
};
```

SAYS ALL PUBLIC MEMBERS OF QueType CAN BE INVOKED FOR OBJECTS OF TYPE CountedQue

```
// DERIVED CLASS CountedQue FROM BASE CLASS QueType

template<class ItemType>
class CountedQue : public QueType<ItemType>
{
public:
   CountedQue( );
   void Enqueue( ItemType newItem );
   void Dequeue( ItemType&  item );
   int LengthIs( ) const;
   // Returns number of items on the counted queue.

private:
   int length;
};
```

class CountedQue<char>




```
CountedQue cQ;
cQ.Enqueue(a);
cQ.QueType
    ::Enqueue(a);
```

```
// Member function definitions for class CountedQue

template<class ItemType>
CountedQue<ItemType>::CountedQue( ) : QueType<ItemType>( )
{
   length = 0 ;
}


template<class ItemType>
int CountedQue<ItemType>::LengthIs( ) const
{
   return  length ;
}
```

```
template<class ItemType>
void CountedQue<ItemType>::Enqueue( ItemType newItem )
       // Adds newItem to the rear of the queue.
       // Increments length.
{
   length++;

   QueType<ItemType>::Enqueue( newItem );
}

template<class ItemType>
void CountedQue<ItemType>::Dequeue(ItemType& item )
       // Removes item from the rear of the queue.
       // Decrements length.
{
   length--;

   QueType<ItemType>::Dequeue( item );
}
```

Iterator Class

It is designed to scan the list sequentially.

Up to now, we used the member functions (ResetList(), GetNextItem()) and the member variable(currentPos) of List class.

- Since there is only one pointer to the node, we can not access several nodes simultaneously.

Iterator Class separates the iteration from the List Class.

- Can access more than one node simultaneously.

Iteration Class Design

⑴ Declare IteratorType<Type> as a friend class of ListType<Type> and NodeType<Type> to enable IteratorType<Type> access the members of these two classes.

⑵ IteratorType<Type> includes a reference variable, listData, pointing to the list to be scanned. Variable listData is initialized when the iterator class object is created.

⑶IteratorType<Type> includes a private data member currentPos pointing to a node in the list.

⑷ IteratorType<Type> has public member functions including NotNull(), NextNotNull(), First(), and Next(), to access members of the list.

Reference Variable

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    int aint = 10;
    string astring = " Old String";
    int &bint = aint;
    string &bstring = astring;

    cout << "aint: " << aint << "\t\t bint: " << bint << endl;
    cout << "astring: " << astring << "\t bstring: " << bstring << endl;
    cout << "\n reference 변수를 이용하여 변경\n";
    bint = 15;
    bstring = " New String";
    cout << "aint: " << aint << "\t\t bint: " << bint << endl;
    cout << "astring: " << astring << "\t bstring: " << bstring << endl;
}
```

```
aint: 10                bint: 10
astring:  Old String    bstring:  Old String

 reference 변수를 이용하여 변경
aint: 15                bint: 15
astring:  New String    bstring:  New String
```

Iterator Class ADT

```
template <class Type> class IteratorType {
public:
    IteratorType(const ListType<Type> &iList): itrlist(iList),
     current(iList.listData) {};
    Boolean NotNull(); // Check that the current pointer is not Null
    Boolean NextNotNull(); // Check that next pointer is not null
    void ResetList(); // set the current pointer to the first node
    // Get the current node and update the current pointer
    int GetCurrentItem(Type item);
private:
    const ListType<Type> &itrList; //connect itrList to the list
    NodeType<Type> *current; // pointer to current node
};
```

List Class와 Interator class

```
template <class Type> class NodeType {
friend class IteratorType<Type>;
private:
      Type data;
      NodeType *link;
};
template <class Type> class ListType {
friend class IteratorType<Type>;
public:
      List() {listData = 0;};
     ... 기타 연산자들
private:
      NodeType<Type> *listData;
                                          };
```

Definition of Member functions

```
// Check that current pointer is not null
template <class Type>
Boolean IteratorType<Type>::NotNull() {
      if(current) return TRUE;
      else return FALSE;
}
 // check that the next node is not null.
template <class Type>
Boolean IteratorType<Type>::NextNotNull() {
      if(current && current->next) return TRUE;
      else return FALSE;
}
```

Defintion of Member Functions

```
// let current point to the first node of the list
template <class Type>
void IteratorType<Type>::ResetList() {
    current = itrList.listData;
}
// return the record pointed by current pointer. If the current
    nod is not null, return 1. Otherwise, return 0
template <class Type>
int IteratorType<Type>::GetCurrentItem(Type& item) {
     if(NotNull()) {
          item = current->data;
          current = current->next;
          return 1;
     }
     else return 0;
}
```

Iterator Class Example

```
// sum up all the values in the list
int sum(const List<int>& list)
{
    Iterator iter(list);
    int sum=0;

    iter.ResetList();
    while (iter.GetCurrentItem(item))
        sum + = item;
}
```