Copying Objects

Slides

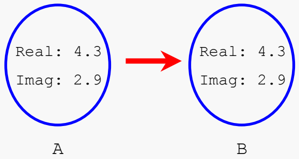
Assignment of Objects

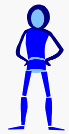
A default assignment operation is provided for objects (just as for struct variables):

```
class Complex {
private:
   double Real, Imag;
public:
   Complex( );
   Complex(double RealPart, double ImagPart);
   // .   .   .
   double Modulus( );
};

Complex A(4.3, 2.9);
Complex B;

B = A;   // copies the data members of A into B
```

The default assignment operation simply copies values of the data members from the "source" object into the corresponding data members of the "target" object.

This is satisfactory in many cases:



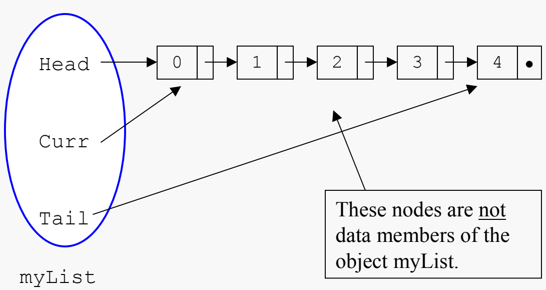


However, if an object contains a pointer to dynamically allocated memory, the result of the default assignment operation is usually not desirable…

Dynamic Content

Consider the `LinkList` class discussed earlier:

```
#include "LinkList.h"

class Integer {
private:
   int Data;
public:
   Integer(int newData=0);
   int  getInt();
   void setInt( int i);
   bool operator==(const Integer& anItem) const;
   bool operator<(const Integer& anItem)  const;
};
typedef Integer Item;

LinkList myList;

for (int Idx = 4; Idx < -1; Idx--) {
   Integer newInteger(Idx);
   myList.PrefixNode(newInteger);
}
```

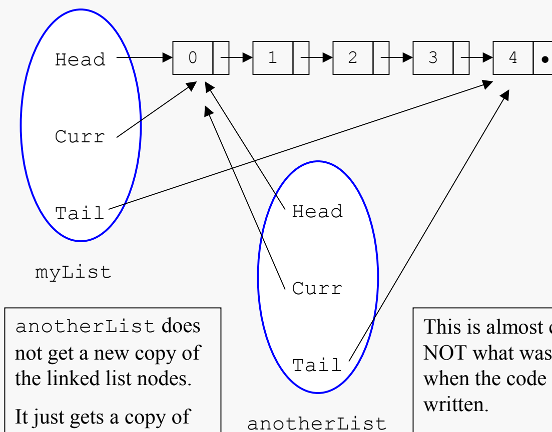


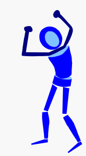
Shallow Copying

Now, suppose we declare another LinkList object and assign the original one to it:

```
LinkList anotherList;

anotherList = myList;
```

Here's what we get:





`anotherList` does not get a new copy of the linked list nodes.

It just gets a copy of the pointers from `myList`.

So both `LinkList` objects share the same dynamic data.

This is almost certainly NOT what was desired when the code above was written.

This is known as making a "**shallow copy**" of the source object.

Deep Copying

When an object contains a pointer to dynamically allocated data, we generally will want the assignment operation to create a complete duplicate of the "source" object. This is known as making a "**deep copy**".

In order to do this, you must provide your own implementation of the assignment operator for the class in question, and take care of the "deep" copy logic yourself. Here's a first attempt:

```
LinkList& LinkList::operator=(const LinkList& otherList) {

   Head = NULL;            // don't copy pointers
   Tail = NULL;
   Curr = NULL;

   LinkNode* myCurr = otherList.Head; //copy head ptr
   while (myCurr != NULL) {
      Item xferData = myCurr->getData();
      if (Head == NULL) //add first node
         PrefixNode(xferData);
      else { //Append to end of list
         Insert(xferData);
         Advance();
      }//else
      myCurr = myCurr->getNext();
   } //while
   return (*this);
}
```

This contains some flaws:

- the "target" object may already be initialized and this doesn't attempt to delete its list, so memory will be "orphaned"
- fixing that will potentially cause a problem if an object is assigned to itself.

…Improved Deep Copy

Here's a somewhat improved version:

```
LinkList& LinkList::operator=(const LinkList& otherList) {

 if (this != &otherList) {      // self-assignment??

   this->~LinkList();           // delete target's list

   Head = Tail = Curr = NULL;   // don't copy pointers

   LinkNode* myCurr = otherList.Head; //copy head ptr
   while (myCurr != NULL) {
      Item xferData = myCurr->getData();
      if (Head == NULL) //add first node
         PrefixNode(xferData);
      else { //Append to end of list
         Insert(xferData);
         Advance();
      }//else
      myCurr = myCurr->getNext();
   }//while
 }//if
 return (*this);
}
```

A more precise copy would involve positioning Curr analogously.

By returning a reference to an object, a member function allows chaining of the the operations. E.g.,

```
LinkList anotherList, anotherCopy;
anotherCopy = anotherList = myList;
```

Add = overload function to the LinkList class.

Passing an Object

When an object is used as an actual parameter in a function call, the distinction between shallow and deep copying can cause seemingly mysterious problems.

```
void PrintList(LinkList& myList, ostream& Out) {
   Item nextValue;
   int Count = 0;

   Out << "Printing list contents: " << endl;
   myList.gotoHead();
   if (myList.isEmpty()) {
      Out << "List is empty" << endl;
      return;
   }

   while (myList.inList()) {
      nextValue = myList.getCurrentData();
      Out << setw(3) << ++Count << ": "
          << nextValue.getSKU() << endl;
      myList.Advance();
   }
   Out << endl;
}
```

This function will print the `Name` fields of a list of objects, (assuming the `InvItem` implementation or something similar for ItemType).

Note that the `LinkList` parameter `myList` is **not** passed by constant reference. That would eliminate risking any modification of the object by the called function. Why is constant reference not used here?

Passing Objects

In the previous example, the object parameter cannot be passed by constant reference because the called function does change the object (although not the content of the list itself).

The object `myList` is passed by reference, which would allow the called function to modify the actual `LinkList` object used in the call.

The advantage of passing by reference is that it eliminates the time and space required to make a copy of the object (if the object were passed by value).

However, since constant reference is not an option here, it would be preferable to eliminate the chance of an unintended modification of the list and pass the `LinkList` parameter by value.

However, that will cause a new problem.

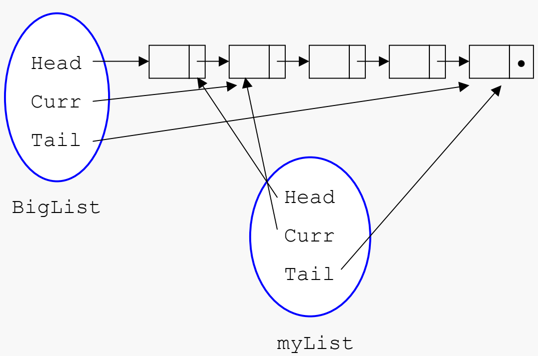
When an object is passed by value, the actual parameter must be copied to the formal parameter (which is a local variable in the called function).

This copying is managed by using a special constructor, called a ***copy constructor***. By default this involves a shallow copy. That means that if the actual parameter involves dynamically allocated data, then the formal parameter will share that data rather than have its own copy of it.

Passing Objects by Value

In this case:

```
// use pass by value:
void PrintList(LinkList myList, ostream& Out) {
  // same implementation
}

void main() {
   LinkList BigList;
   // initialize BigList with some data nodes

   PrintList(BigList, cout);       // print BigList
}
```



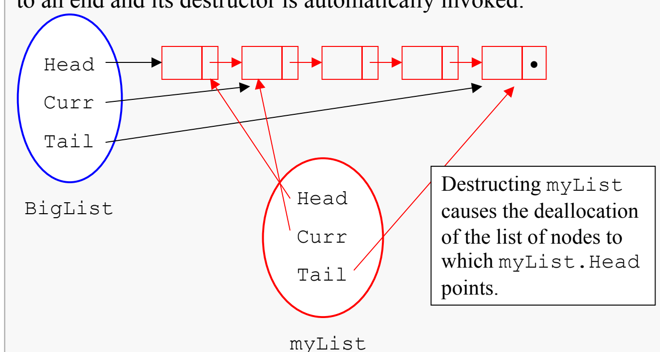
What happens when `PrintList()` is called?
First, a local variable `myList` is created and the data members of `BigList` are copied into `myList`, resulting in the situation shown above.

Passing Objects by Value

As `PrintList()` executes, the `Curr` pointer in `myList` is modified and nodes are printed:

```
void PrintList(LinkList myList, ostream& Out) {

  // operations on myList, which is local
}
```

When `PrintList()` terminates, the lifetime of `myList` comes to an end and its destructor is automatically invoked:



But of course, that's the same list that `BigList` has created. So, when execution returns to `main()`, `BigList` will have lost its list, but `BigList.Head` will still point to that deallocated memory.

Havoc will ensue.

Copy Constructors

There are solutions to this problem:

- always pass objects by reference
- force a deep copy to be made when pass by value is used

The first option is undesirable since it raises the risk of undesired modification of the actual parameter. The second option can be achieved by providing a user-defined copy constructor for the class, and implementing a deep copy. When a user-defined copy constructor is available, it is used when an actual parameter is copied to a formal parameter.

```
LinkList::LinkList(const LinkList& Source) {

   Head = Tail = Curr = NULL;

   LinkNode* myCurr = Source.Head; // copy list
   while (myCurr != NULL) {
      Item xferData = myCurr->getData();
      if (Head == NULL) //add first node
         PrefixNode(xferData);
      else { //Append to end of list
         Insert(xferData);
         Advance();
      }//else
      myCurr = myCurr->getNext();
   }
// add code to logically equate the
// curr pointers for an exact copy
}
```

The copy constructor takes an object of the relevant type as a parameter (constant reference must be used). Implement a deep copy in the body of the copy constructor and the problem described on the previous slides is solved.

Initialization

When an object is declared, it may be initialized with the value of an existing object (of the same type):

```
void main() {
   LinkList aList;  // default construction
   // Now throw some nodes into aList
   // . . .

   LinkList anotherList = aList;  // initialization
}
```

Technically initialization is different from assignment since here we know that the "target" object does not yet store any defined values.

Although it looks like an assignment, the initialization shown here is accomplished by the copy constructor.

If there is no user-defined copy constructor, the default (shallow) copy constructor manages the initialization.

If there is a user-defined copy constructor, it will manage the copying as the user wishes.

Copy constructors also execute when an object is returned by a function as the function return value:

```
object x = getObject(list);
```

Moral

When implementing a class that involves dynamic allocation, if there is any chance that:

- objects of that type will be passed as parameters, or
- objects of that type will be used in initializations

then your implementation should include a copy constructor that provides a proper deep copy.

If there is any chance that:

- objects of that type will be used in assignments

then your implementation should include an overloaded assignment operator that provides a proper deep copy.

This provides relatively cheap insurance against some very nasty behavior.

Class vs. Proc. ADT

Implementation Comparison

- The List Class ADT achieves complete encapsulation/information hiding for the type.
  - Function operation interfaces are simplified as a result of the reduced parameter lists, (i.e. the object is passed implicitly).
- A List Procedural ADT only achieves a certain level of encapsulation/information hiding .
- Due to more localized code, the List Class implements a more reusable ADT.
  - Automatic initialization by constructors eliminates error code checking in the Class ADT.
- Modifications and extensions to the List Class ADT are easier to make.
- *Higher-level* types based on the List Class ADT can be built more readily.