Lecture Material

- Pointers
- Linked list class
- Parameter passing
- Shallow and deep copying
- Copy constructor
- Assignment operator
- Operator overloading

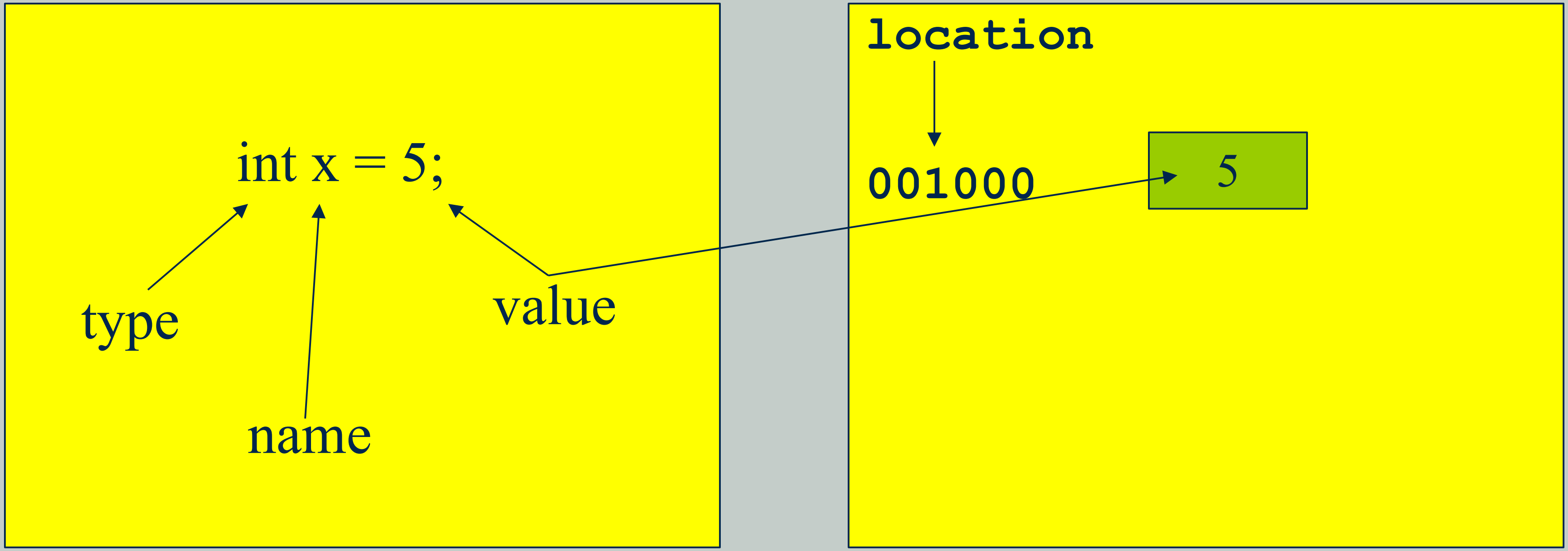
Pointers

- Four attributes of a variable
  - name
  - type
  - value
  - location (address)
- Pointer is a type of value
  - stored in a variable
  - is just a number!
- Operator * means:
  - take value stored in variable, and use it as address of another variable
- Operator & means:
  - take address of variable (NOT the value of it)

```
int x = 5;
```

Pointers

- Variable
  - name, type, value, location (address)

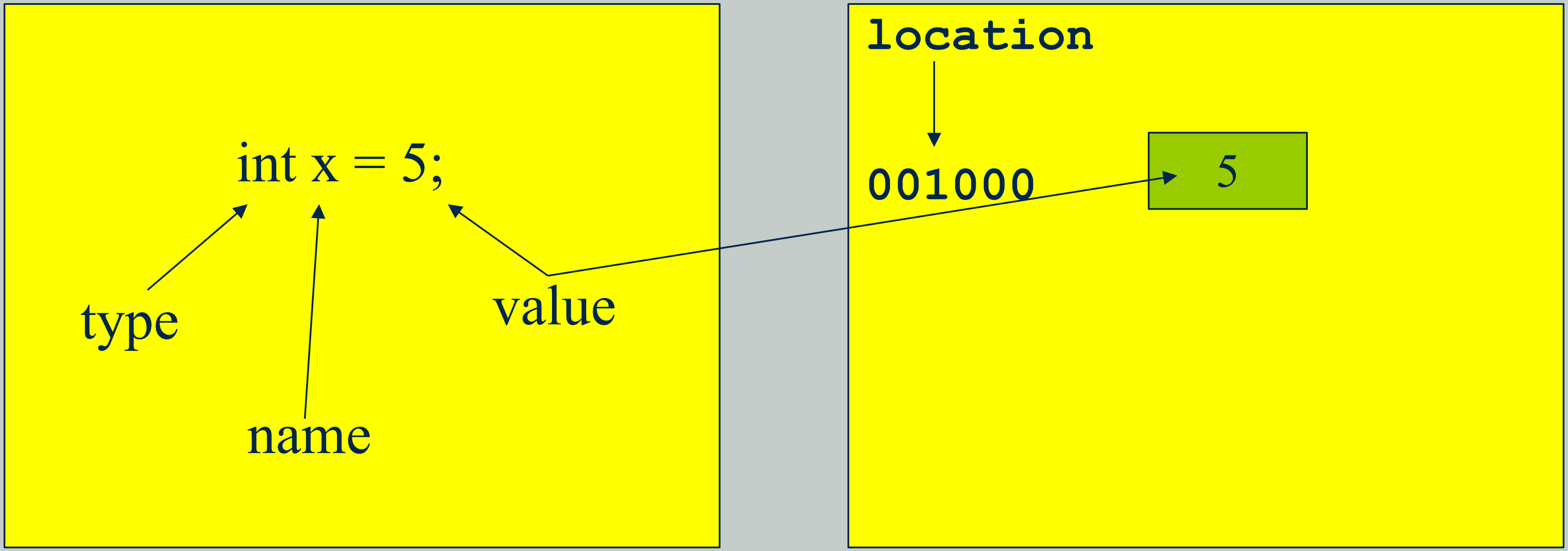


In program   In memory, at runtime

- Which variable at what address? How much memory? Who decides?

Pointers

- What is the value of the following expressions? Are they all legal?
  - x
  - &x
  - *x



In program

In memory, at runtime

Pointers

- What is the value of the following expressions? Are they all legal?
  - x, &x, *x
  - p, &p, *p
  - q, &q, *q
  - ip, &ip, *ip

```
int x=5;
char *p="hello";
char *q;
int *ip;
ip=&x;
```

In program

location	value	name
001000	5	int x
001004	3000	char*p
001008	?	char*q
001012	?	int* ip
...	...	
003000	hello\0	

In memory, at runtime

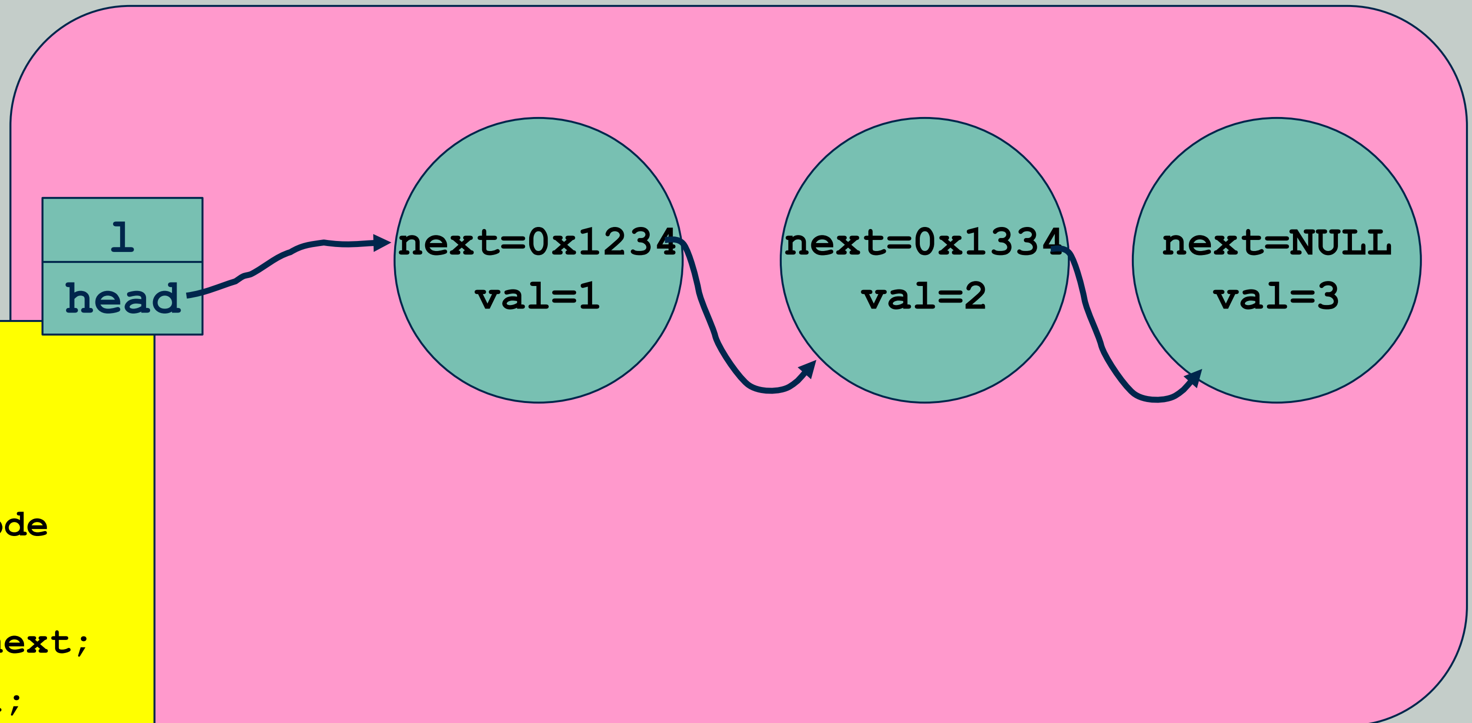
Function Pointers

```
int* f1(int*, const int*);
int* (*fp1)(int*, const int*);
int* (*f2(int))(int*, const int*);
int* (*(*fp2)(int))(int*, const int*);

fp1=f1;
fp1=&f1;
fp1=&fp1; /* wrong */
fp2=f2;
fp2=&f2;
```

```
int a,b,*c;

c=f1(&a,&b);
c=fp1(&a,&b);
c=(*fp1)(&a,&b);
c=*fp1(&a,&b); /* wrong */
c=(f2(3))(&a,&b);
c=(*f2(3))(&a,&b);
c=(fp2(3))(&a,&b);
c=(*fp2(3))(&a,&b);
c=(*(*fp2)(3))(&a,&b);
```

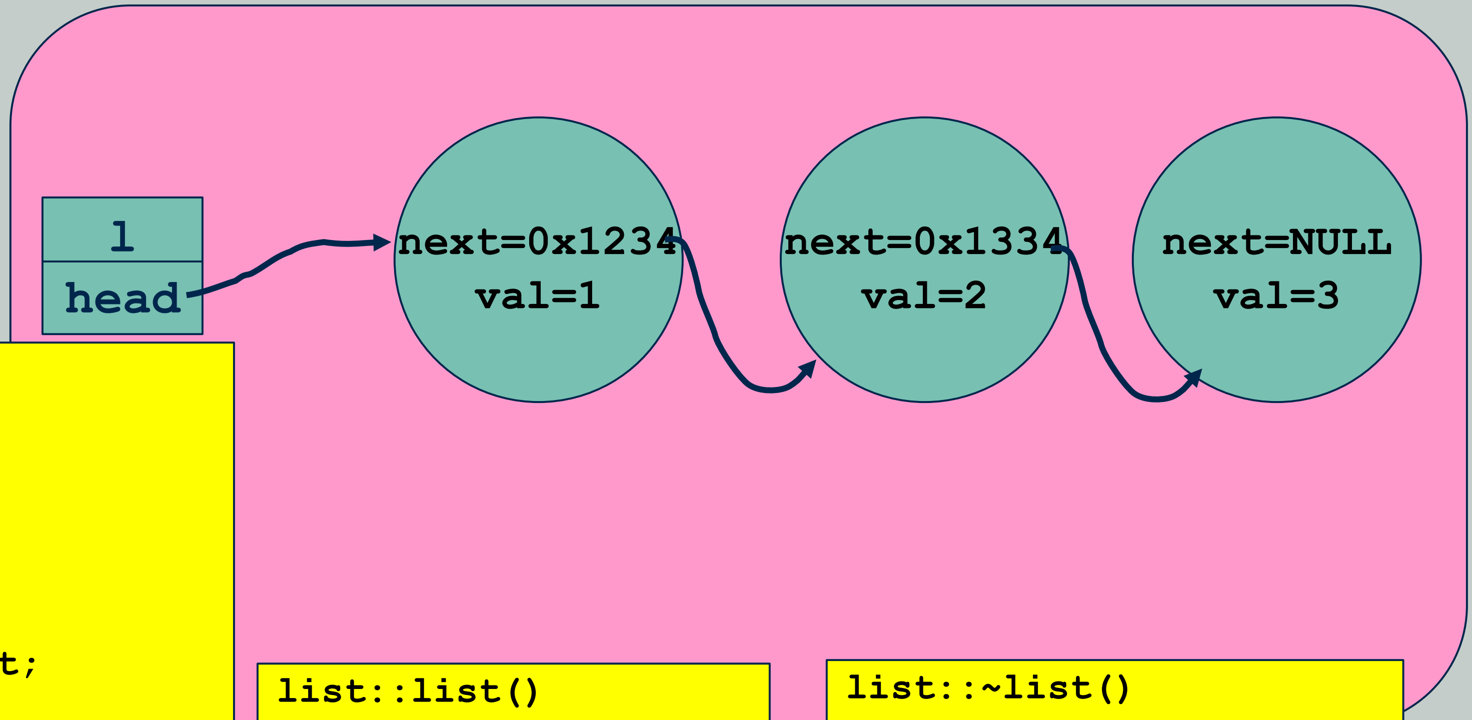
Class *list*

Unidirectional linked list




```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
};
```

Class *list* - Constructor and Destructor

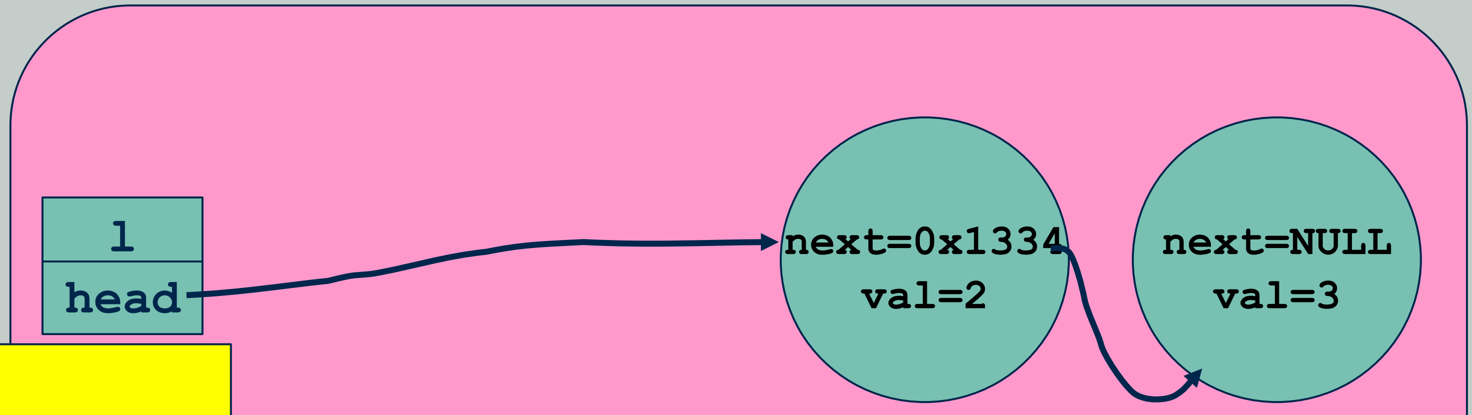



```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
public:
  list ();
  ~list ();
  void insert (int a);
}
```

```
list::list()
{
  head = NULL;
}
```

```
list::~list()
{
  while(head)
  {
    node* t=head->next;
    delete head;
    head=t;
  };
}
```

Class *list* - Destructor

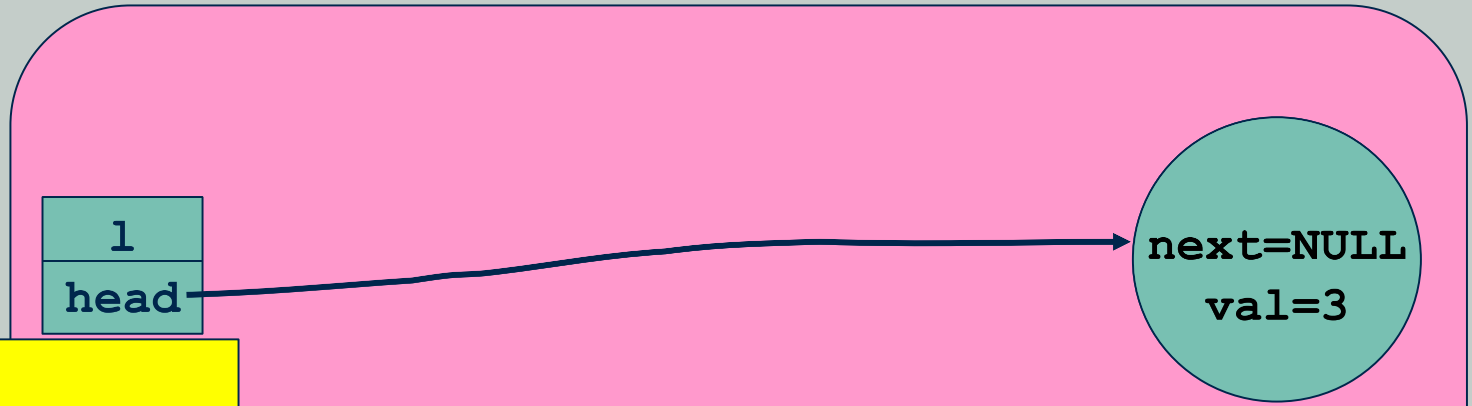



```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
public:
  list ();
  ~list ();
  void insert (int a);
}
```

```
list::list()
{
  head = NULL;
}
```

```
list::~list()
{
  while(head)
  {
    node* t=head->next;
    delete head;
    head=t;
  };
}
```

Class *list* - Destructor




```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
public:
  list ();
  ~list ();
  void insert (int a);
}
```

```
list::list()
{
  head = NULL;
}
```

```
list::~list()
{
  while(head)
  {
    node* t=head->next;
    delete head;
    head=t;
  };
}
```

Class *list* - Destructor

l

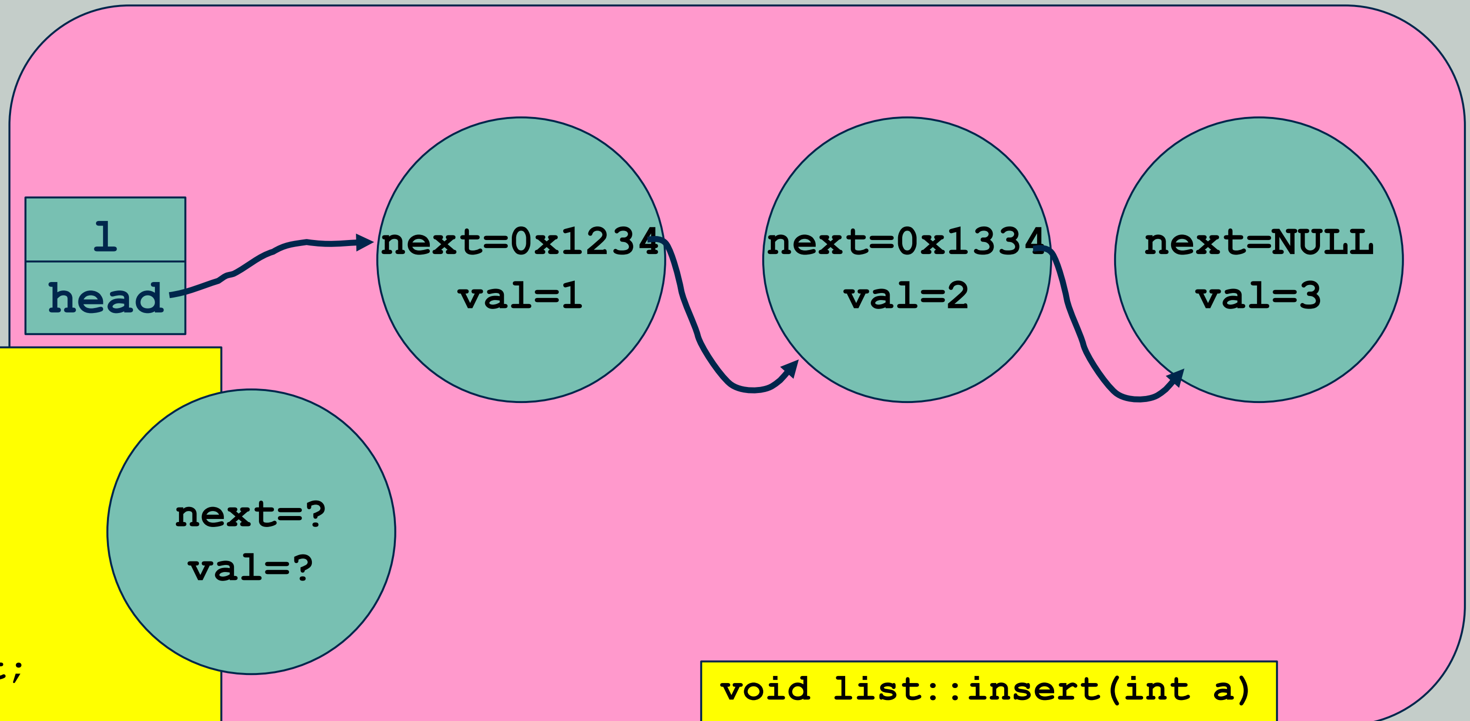
head=NULL

```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
public:
  list ();
  ~list ();
  void insert (int a);
}
```

```
list::list()
{
  head = NULL;
}
```

```
list::~list()
{
  while(head)
  {
    node* t=head->next;
    delete head;
    head=t;
  };
}
```

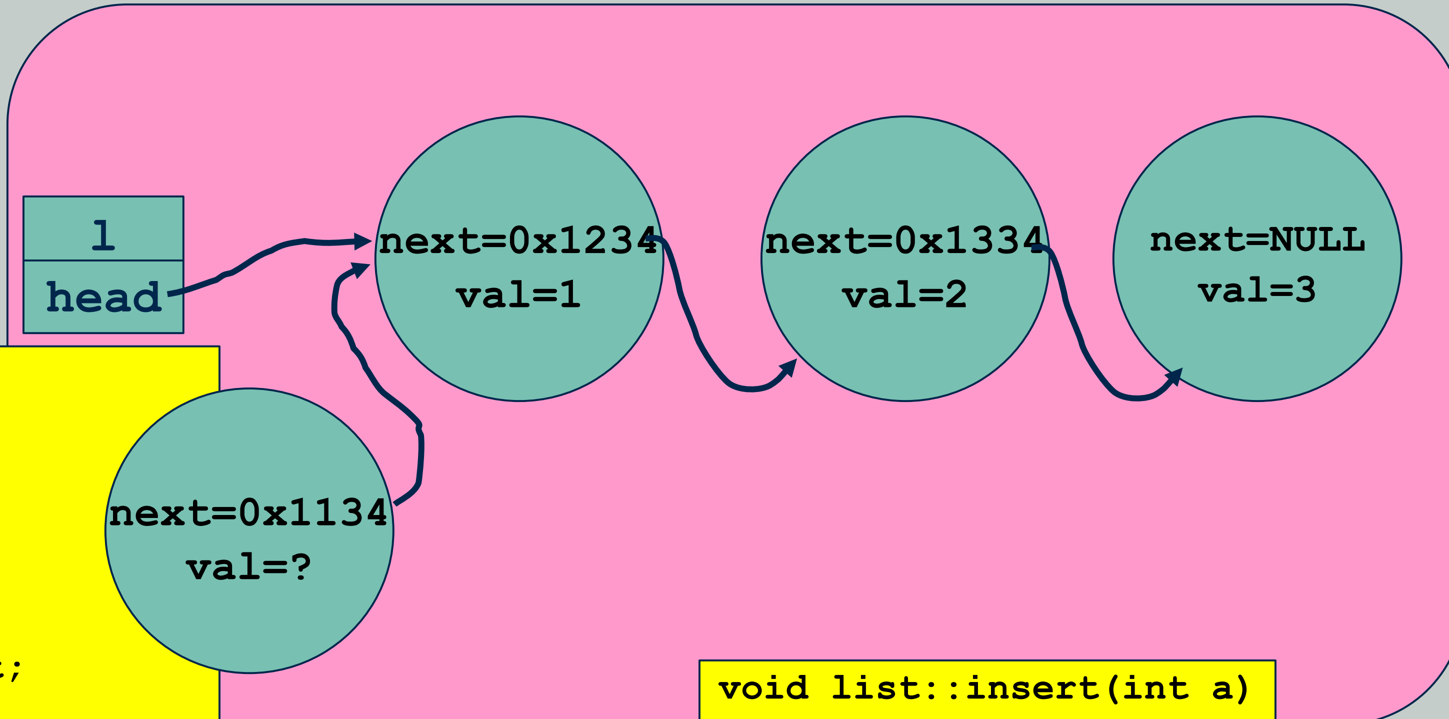
Class *list* - Insert




```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
public:
  list ();
  ~list ();
  void insert (int a);
}
```

```
void list::insert(int a)
{
  node* t=new node;
  t->next=head;
  head = t;
  head->val = a;
}
```

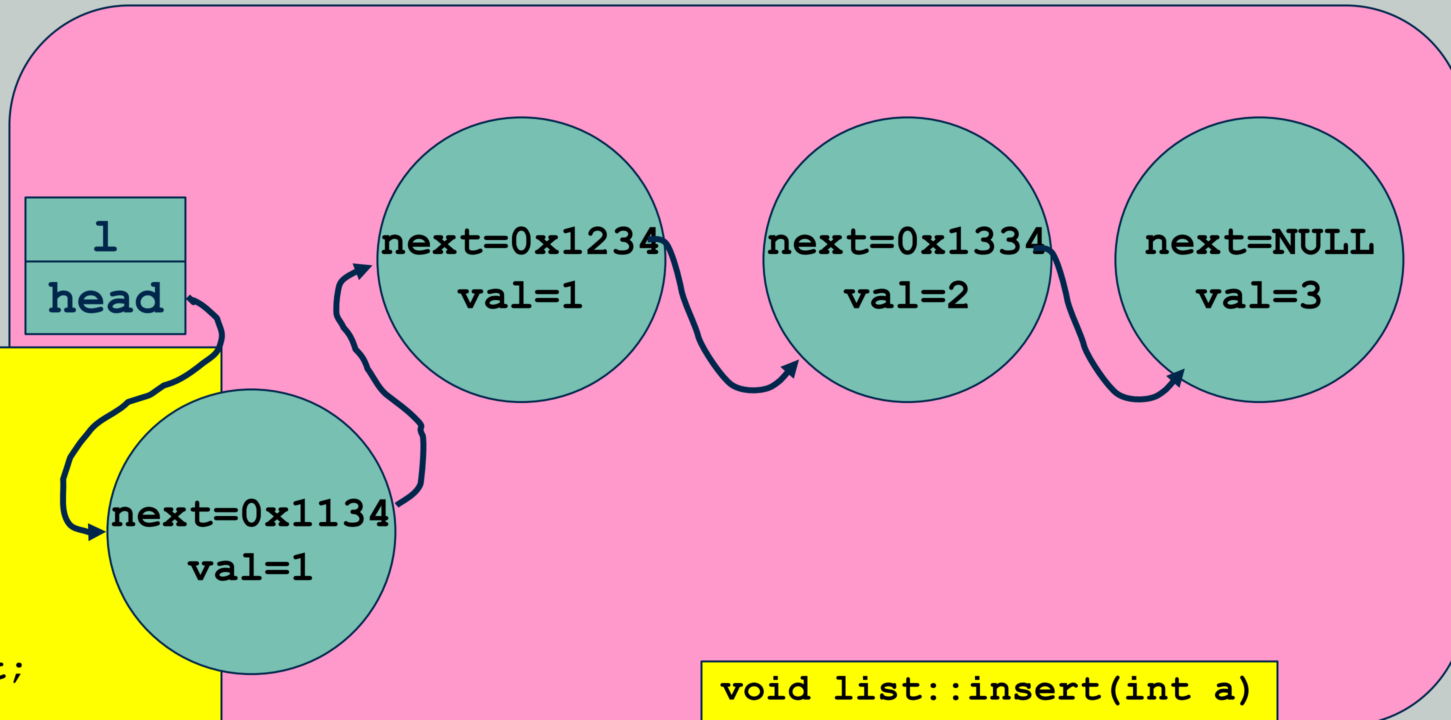
Class *list* - Insert




```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
public:
  list ();
  ~list ();
  void insert (int a);
}
```

```
void list::insert(int a)
{
  node* t=new node;
  t->next=head;
  head = t;
  head->val = a;
}
```

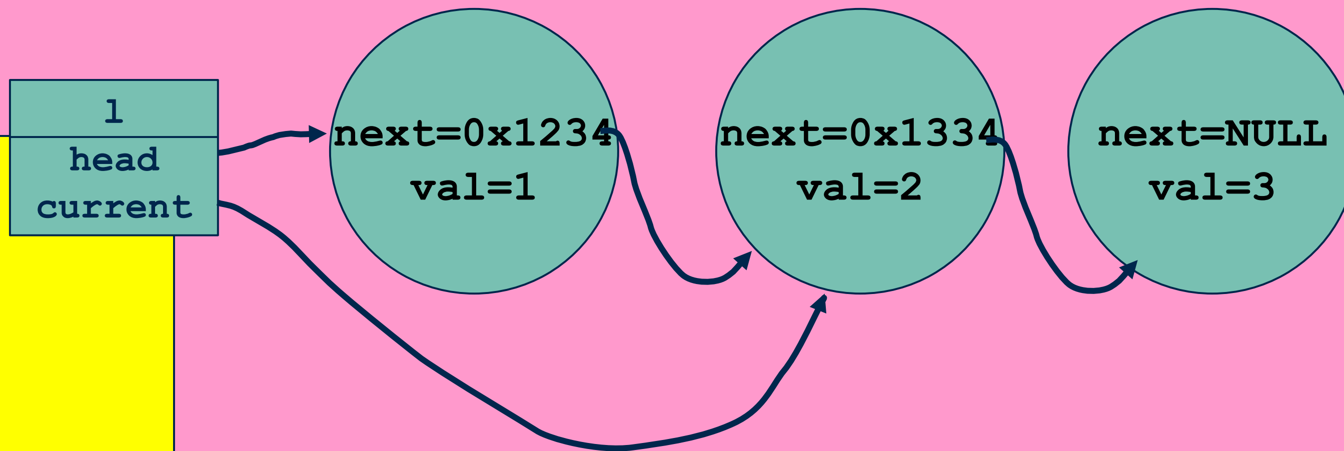
Class *list* - Insert




```
class list
{
private:
  struct node
  {
    node *next;
    int val;
  };
  node * head;
public:
  list ();
  ~list ();
  void insert (int a);
}
```

```
void list::insert(int a)
{
  node* t=new node;
  t->next=head;
  head = t;
  head->val = a;
}
```

Class *list* - Iterator




```
class list
{
private:
...
  node * head;
  node * current;
public:
...
  void goToHead ();
  int getCurrentData();
  void advance ();
  bool moreData ();
};
```

```
#include <iostream>
using namespace std;
#include "list.h"
int main()
{
 list l;
 l.insert(3);
 l.insert(2);
 l.insert(1);
 l.goToHead();
```

```
while(l.moreData())
 {
   int val;
   val=l.getCurrentData();
   cout << val << " ";
   l.advance();
 }
 cout << endl;
};
```

Passing of Function Parameters

- Passing by value
  - formal parameters are the copies of actual parameters
- Passing by reference
  - formal parameters are the references to the actual parameters, i.e. all operations on formal parameters refer to actual parameters
- C and C++ by default pass all arguments by value

```
void d1(int x)
{ x = 10; }
void d2(int *p)
{ (*p) = 10;}
void d3(int *p)
{ p = new int(4);}
```

```
int main() {
  int y = 2;
  d1(y); cout << y;
  d2(&y); cout << y;
  d3(&y); cout << y;
}
```

Passing of Function Parameters

- By value
  - value of parameter is passed to function
- By reference
  - reference of parameter is passed to function, thus value can be modified
- By constant reference
  - reference of parameter is passed to function for efficiency reasons, but value cannot be modified (verified by compiler)

```
void f1(int x) { x = x + 1; }
void f2(int& x) { x = x + 1; }
void f3(const int& x) { x = x + 1; }
void f4(int *x) { *x = *x + 1; }
int main() {
  int y = 5;
  f1(y);
  f2(y);
  f3(y);
  f4(&y);
}
```

- Which is which in this example?
- What is the value of y after each call?
- On the last one (f4), what is being passed as an argument? Is it passed by value or reference?
- Can you pass a pointer by reference?

Passing Parameters to Functions

- Objects are no different than anything else passed to a function
  - Classes provide support to modify the behavior
- Three ways of doing it: by value, by reference, by constant reference
- By Value
  - Copy constructor will be used on the argument
- By Reference
  - A reference to the object will be passed
- By Constant Reference
  - A constant reference will be passed. Only const methods in the class can be called on this argument.

Passing an Object as a Parameter

- 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 (list & toPrint, ostream & Out)
{
  int nextValue;
  Out << "Printing list contents: " << endl;
  toPrint.goToHead ();
  if (!toPrint.moreData ())
    {
      Out << "List is empty" << endl;
      return;
    }
  while (toPrint.moreData ())
    {
      nextValue = toPrint.getCurrentData ();
      Out << nextValue << " ";
      toPrint.advance ();
    }
  Out << endl;
}
```

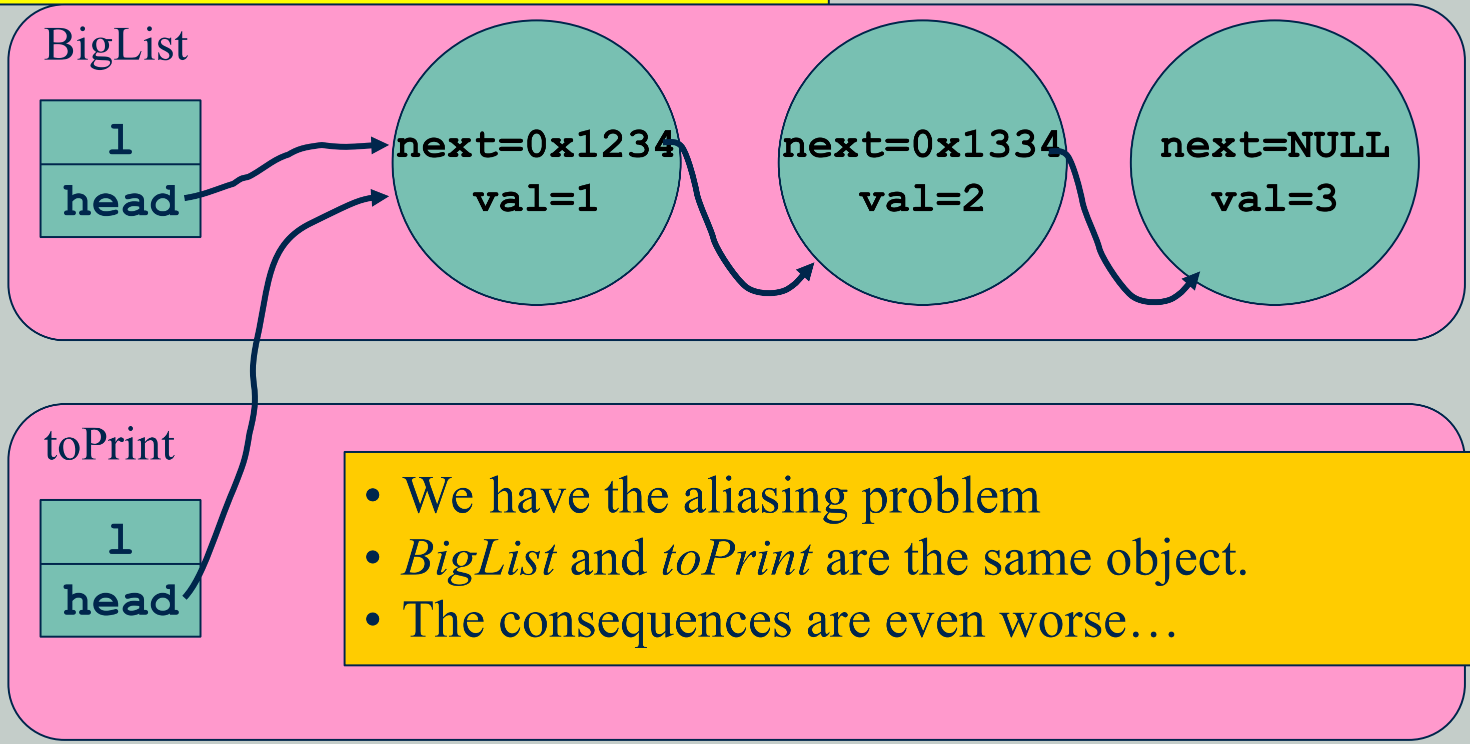
- The list object is passed by reference because it may be large, and making a copy would be inefficient.
- What if we used pass by constant reference?
- What if we used pass by value?

Passing Objects

- In the previous example, the object parameter cannot be passed by constant reference because the called function does change the object (the current position pointer)
- However, since constant reference is not an option here, it may be preferable to eliminate the chance of an unintended modification of the list and pass the list parameter by value
  - This solution will be inefficient (Why?)
  - Might cause problem if you don’t have a copy constructor (Why?)

Passing Objects by Value

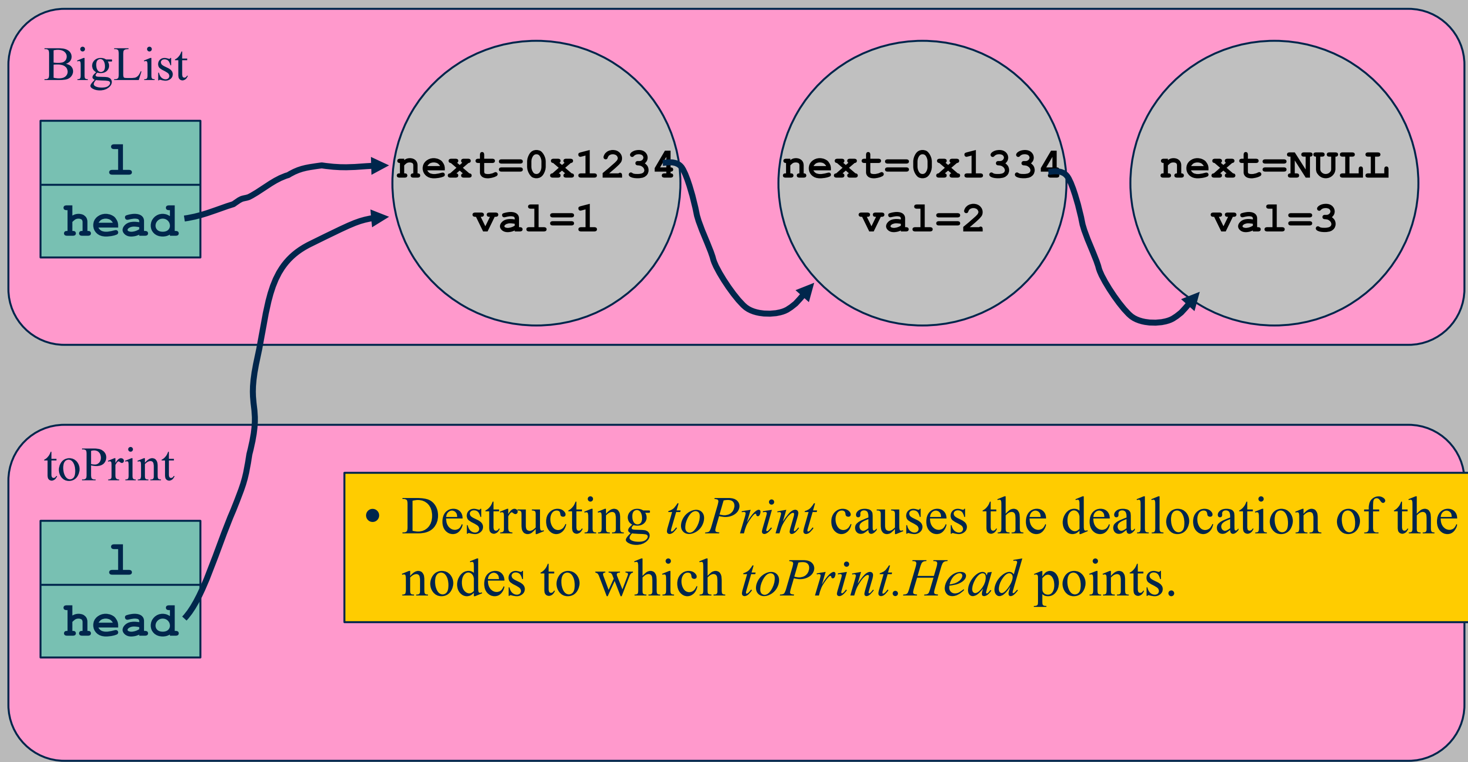
```
void
PrintList (list toPrint, ostream & Out)
{
   // same implementation
}
void main()
{
   list BigList;
   // initialize BigList with some data nodes
   PrintList(BigList, cout);
}
```



- We have the aliasing problem
- *BigList* and *toPrint* are the same object.
- The consequences are even worse…

Passing Objects by Value

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



- Destructing *toPrint* causes the deallocation of the list of nodes to which *toPrint.Head* points.

- But of course, that's the same list that *BigList* has created. So, when execution returns to *main()*, *BigList* will have been destroyed, but *BigList.head* will still point to that deallocated memory

Assignment of Objects

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

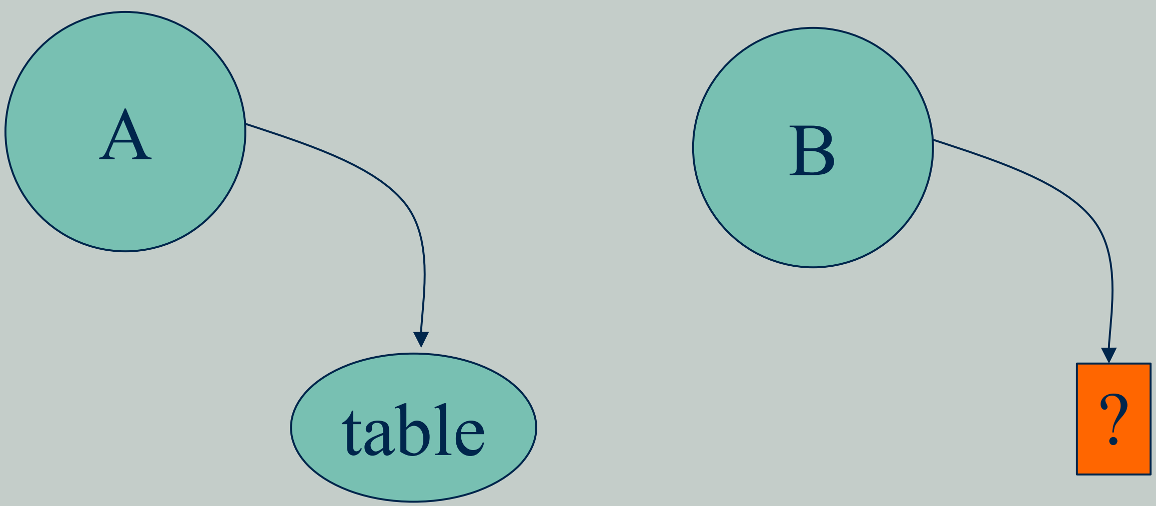
```
class DateType {
  public:
   // constructor
   DateType();
   DateType(int newMonth, int newDay, int newYear);
   ...
};
. . .
DateType A(1, 22, 2002);
DateType B;
B = A; // copies the data members of A into B
```

- 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…

Problems with Assignment of Pointers

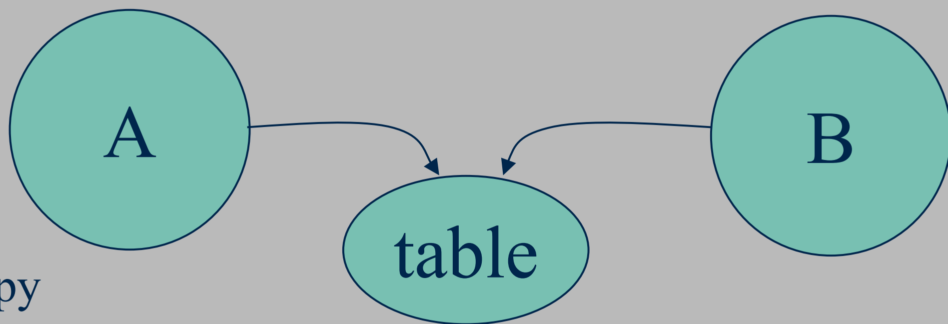
```
class Wrong {
private:
   int *table; // some data here
public:
   // constructor
   Wrong() {table = new int[1000]; }
   ~Wrong() { delete [] table; }
};
. . .
Wrong A;
Wrong B;
B = A; // copies the data members of A into B
```

- What type of data does *Wrong* store?
- Is *int *table* the same as *int table[ ]*?
- What happens when it is copied?
- What problems do we encounter?
- How can it be solved?

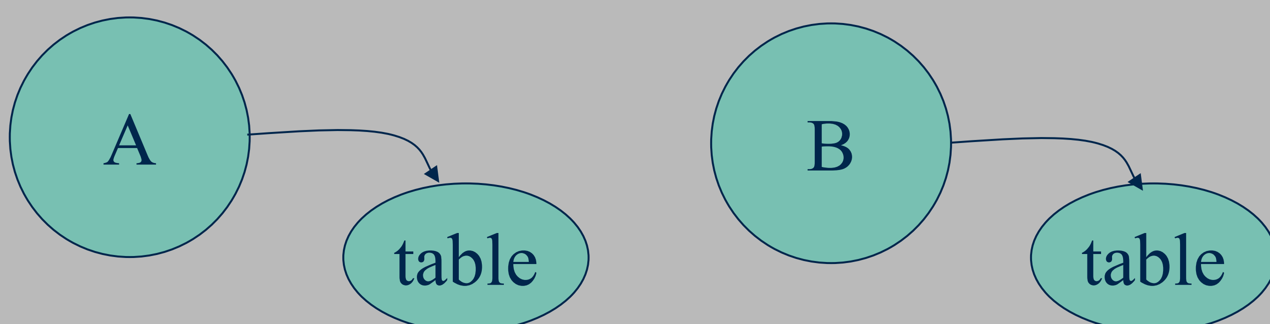


Assignment: Types of Copying

- Two types of copying objects with pointer members and its contents when they are being assigned
- Shallow Copy
  - Copy all member variables (including the pointers)
  - This results in copying of the pointers but not what the pointers point to

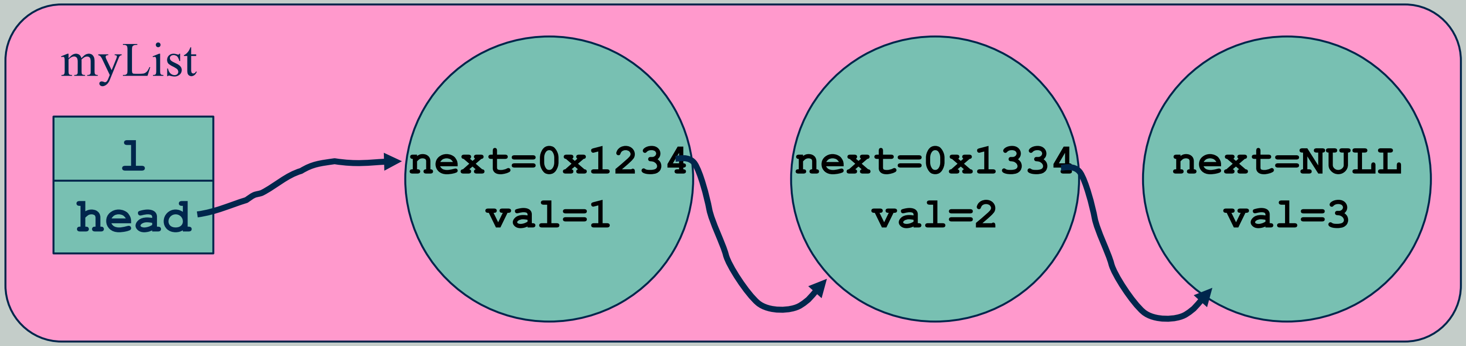


- Deep Copy
  - New memory allocation for all pointers
  - Copy contents pointed by pointers to new locations
  - Copy remaining member variables (non pointers)



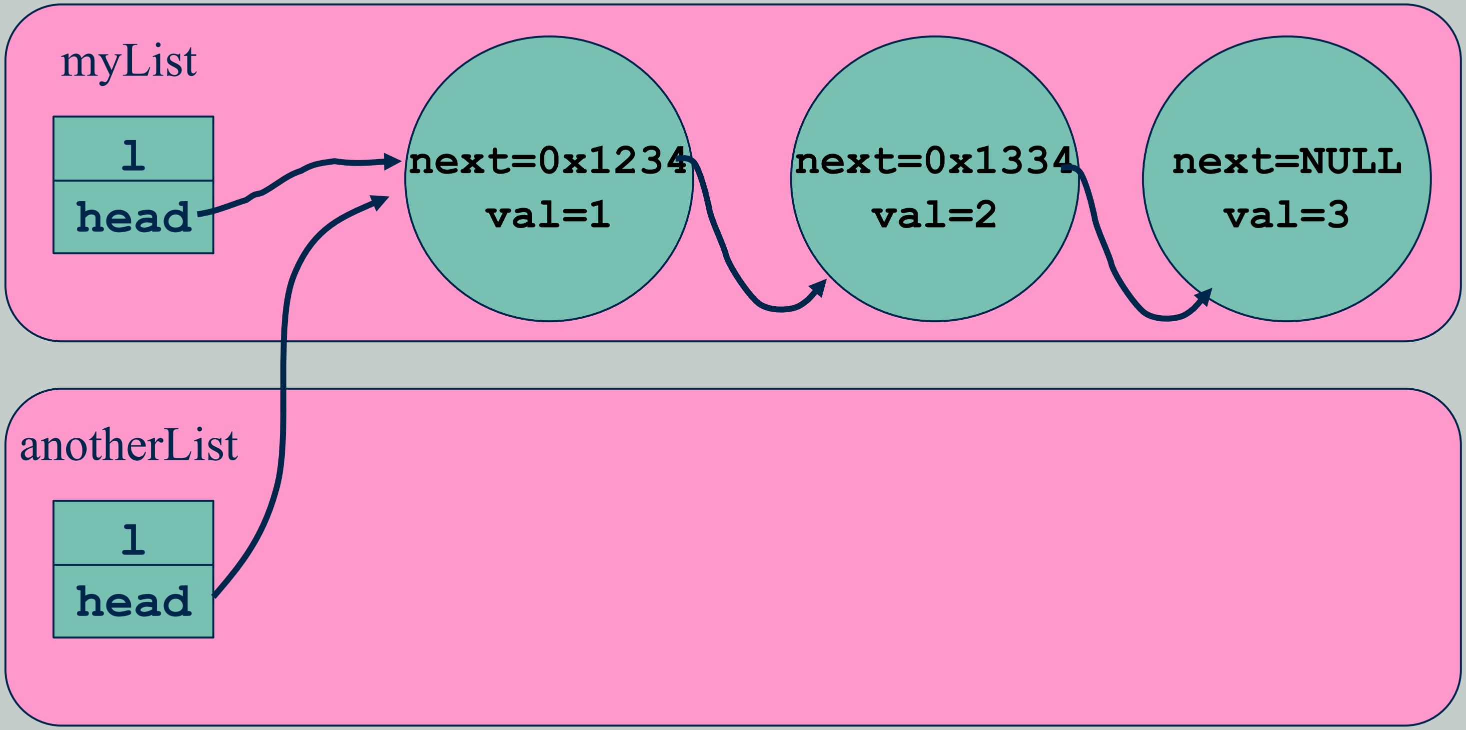
Problems with Shallow Copying

```
list myList;
myList.insert (3);
myList.insert (2);
myList.insert (1);
```



Problems with Shallow Copying

```
list myList;
myList.insert (3);
myList.insert (2);
myList.insert (1);

list anotherList;
anotherList=myList;
```



Deep Copying Essentials

- When an object contains a pointer to dynamically allocated data, define the assignment operation to make a deep copy
  - Define assignment operator for the class in question

    AType& AType::operator=(const AType& otherObj)
  - In the assignment operator take care of the following special situations
    - Are you assigning something to itself? For example *A=A*:

      if (this == &otherObj) // if true, do nothing
    - Call the "delete" operation on the receiving object.

      delete this->...
    - Allocate new memory for values being copied
    - Copy the assigned values
    - Return **this*

Copy Constructor vs. Assignment Operator

- Copy constructor is used to create a new object from scratch
- It has the following signature:
  - *AType::AType(const AType& otherObj)*
- Is simpler than the assignment operator - does not have to check the assignment to itself neither free the previous contents.
- Is used to copy actual parameter to formal parameter when passing by value
- When creating a new object, it can be initialized with the existing object of the same type. Copy constructor is invoked then.

```
int main() {
  list a;
  //...
  list b(a); //copy constructor called
  list c=a; //copy constructor called
};
```

Anonymous Objects

- An anonymous object is a nameless (i.e. unnamed) object
  - Object is created but there is no named variable holding it
- Useful:
  - for temporary use (parameter in a method call, return, expression term)
  - as default value for an object parameter
- Anonymous objects are created by a direct invocation of a class constructor
- Consider a method receiving an *Address* object

```
void Person::setAddress(Address addr);
```

- Argument could be passed as follows...

```
Person joe;
joe.setAddress(Address("Disk Drive"...));
```

- Instead of ...

```
Person joe;
Address joeAddress("Disk Drive"...);
joe.setAddress(joeAddress);
```

Example: Anonymous Objects as Parameters

- Without anonymous objects, we have a mild mess:

```
Name JBHName("Joe", "Bob", "Hokie");
Address JBHAddr("Oak Bridge Apts", "#13",
"Blacksburg","Virginia", "24060");
Person JBH(JBHName, JBHAddr, MALE);
. . .
```

- With anonymous objects we reduce pollution of the local namespace:

```
Person JBH(Name("Joe", "Bob", "Hokie"),
           Address("Oak Bridge Apts", "#13",
           "Blacksburg","Virginia", "24060"),
           MALE);
. . .
```

Example: Anonymous Objects as Defaults

- Used as default parameter values, anonymous objects provide a relatively simple way to control initialization and reduce class interface clutter:

```
Person::Person(Name N = Name("I", "M", "Nobody"),
Address A = Address("No Street", "No Number",
"No City", "No State", "00000"), Gender G =
GENDERUNKNOWN) {
  Nom = N;
  Addr = A;
  Spouse = NULL;
  Gen = G;
}
```

Different Ways to Create Objects

- Automatic variables

```
Atype a; // default constructor
```

- Automatic variables with arguments

```
Atype a(3); // constructor with (int) signature
```

- Passing arguments to functions by value

```
void f(Atype b) {...}
Atype a; // default constructor
...
f(a); // copy constructor
```

- Assigning values to variables

```
Atype a,b;
...
a=b; // assignment operator
```

- Initialization of new objects

```
Atype b; // default constructor
...
Atype a=b; // copy constructor (NOT assignment operator)
```

- Returning values from functions

```
Atype f() {
  Atype a; // default constructor
  ...
  return a; // copy constructor
}
```

Features of a Solid C++ Class

- Explicit default constructor
  - Guarantees that every declared instance of the class will be initialized in some controlled manner

    ```
    ClassName::ClassName() { ... }
    ```
- If objects of the class contain pointers to dynamically-allocated storage:
  - Define an explicit destructor
    - Prevents memory waste. Release resources when object is destroyed.

      ```
      ClassName::~ClassName() { ... }
      ```
  - Define an assignment operator
    - Implicitly used when an object is assigned to another. Prevents destructor aliasing problem.

      ```
      ClassName & ClassName::operator=(const ClassName& obj) { ... }
      ```
  - Define a copy constructor
    - Implicitly used when copying an object during parameter passing or initialization. Prevents destructor aliasing problem.

      ```
      ClassName::ClassName(const ClassName& obj) { ... }
      ```

Overloading

- Overloading - having multiple "definitions" for the same name
  - Multiple functions under just one name
- In C++, overloaded names are differentiated by number of arguments and type of arguments
  - (and inheritance)
- This is called the signature of a function
  - return types are not considered, so this would be illegal:

```
double fromInt(int x)
float fromInt(int x)
```

- Most common use of overloading is for operators

Overloading & Polymorphism

- Overloading is considered “ad-hoc” polymorphism.
- Can define new meanings (functions) of operators for specific types.
- Compiler recognizes which implementation to use by signature (the types of operands used in the expression).
- Overloading is already supported for many built-in types and operators:

```
17 * 42
4.3 * 2.9
cout << 79 << 'a' << "overloading is profitable" << endl;
```

- The implementation used depends upon the types of operands.

Reasons for Overloading

- Support natural, suggestive usage:

```
Complex A(4.3, -2.7), B(1.0, 5.8);
Complex C;
C = A + B; // '+' means addition for this
  type as well as int, etc.
```

- Semantic integrity (assignment for objects with dynamic content must ensure a proper deep copy is made).
- Able to use objects in situations expecting primitive values

Operators That Can Be Overloaded

- Only the following operator symbols can be overloaded:

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td></tr>
<tr><td>|</td><td>~</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td></tr>
<tr><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td></tr>
<tr><td><<</td><td>>></td><td>>>=</td><td><<=</td><td>==</td><td>!=</td><td><=</td></tr>
<tr><td>>=</td><td>&&</td><td>||</td><td>++</td><td>--</td><td>->*</td><td>,</td></tr>
<tr><td>-></td><td>[]</td><td>()</td><td>new</td><td>new[]</td><td>delete</td><td>delete []</td></tr>
</table>

- Operators =, ->, [], () must be non-static members

Operator Overloading Guidelines

- Avoid violating expectations about the operator:

```
Complex Complex::operator~() const {
   return ( Complex(Imag, Real) );
}
```

- Provide a complete set of properly related operators: a = a + b and a+= b have the same effect and it makes sense to support both if either is supplied.
- Define the operator overload as a class member unless it's necessary to do otherwise.
- If the operator overload cannot be a class member, then make it a friend rather than add otherwise unnecessary member accessors to the class.

Syntax for Overloading Operators

- Declared and defined like other methods or functions, except that the keyword *operator* is used.
- As method of the Name class:

```
bool Name::operator== (const Name& RHS) {
  return ((First == RHS.First) &&
  (Middle == RHS.Middle) &&
  (Last == RHS.Last) );
}
```

- As nonmember function:

```
bool operator==(const Name& LHS, const Name& RHS) {
  return ((LHS.First == RHS.First) &&
  (LHS.Middle == RHS.Middle) &&
  (LHS.Last == RHS.Last) );
}
```

- It is probably most natural here to use the member operator approach.

Using Overloaded Operators

- If *Name::operator==* defined as member function, then

```
nme1 == nme2
```

  is the same as

```
nme1.operator==(nme2)
```

- If *operator==* defined as nonmember function, then

```
nme1 == nme2
```

  is the same as

```
operator==(nme1, nme2)
```

Binary Operator as a Member

- A class member subtract operator for *Complex* objects:

```
Complex Complex::operator-(const Complex& RHS) const {
  return ( Complex(Real - RHS.Real, Imag - RHS.Imag) );
}
```

- To be a class member, the left operand of an operator must be an object of the class type:

```
Complex X(4.1, 2.3), Y(-1.2, 5.0);
int Z;
```

OK: `X + Y;`

Not OK: `Z + X;`

- It is typical to pass by constant reference to avoid the overhead of copying the object.

Binary Non-Member Operators

- A non-member subtract operator for *Complex* objects:

```
Complex operator-(const Complex& LHS, const Complex& RHS) {
   return ( Complex(LHS.getReal() - RHS.getReal(),
         LHS.getImag() - RHS.getImag()) );
}
```

- As a non-member, this subtract operator must use the public interface to access the private data members of its parameters…
  - … unless the class *Complex* declares it to be a friend.
- If an operator or function is declared to be a friend of a class then it can access private members as if it were a member function.

```
class Complex
{
   ...
    friend Complex operator+ (const Complex&, const Complex&);
   ...
};
```

Unary Operators

- A negation operator for the *Complex* class:

```
Complex Complex::operator-() const {
  return ( Complex(-Real, -Imag) );
}
```

```
Complex A(4.1, 3.2); // A = 4.1 + 3.2i
Complex B = -A; // B = -4.1 - 3.2i
```

- Note that a unary member operator takes NO parameters

Pre- and Postincrementation

```
class Value {
  private:
      int x;
  public:
      Value(int i = 0) : x(i) {}
  int get() const { return x; }
  void set(int x) ( this->x = x; }
  Value& operator++();
  Value operator++(int Dummy);
}
```

- Preincrementation operator

```
Value& Value::operator++() {
  x = x + 1;
  return *this;
}
```

- Postincrementation operator

```
Value Value::operator++(int Dummy) {
  x = x + 1;
  return Value(x-1); // return previous value
}
```

Multiple Overloading

- We can have two addition operators in a class:

```
Complex Complex::operator+(double RHS) const {
   return (Complex(Real + RHS, Imag));
}
Complex Complex::operator+(Complex RHS) const {
   return (Complex(Real + RHS.Real, Imag + RHS.Imag));
}
```

- This lets us write mixed expressions, like:

```
Complex X(4.1, 2.3);
double R = 1.9;
Complex Y = X + R; // Y.Real is 6.0
```

- Signature of function used to resolve which is used:

```
Complex Z = Y + R; // complex plus double
Complex W = Y + X; // complex plus complex
```

Multiple Overloading

- Constructor can be used as a conversion operator

```
Complex Complex::operator+(Complex RHS) const {
  return (Complex(Real + RHS.Real, Imag + RHS.Imag));
}
Complex:: Complex (double co)
{
  Real = co;
  Imag = 0;
};
```

```
Complex X(4.1, 2.3);
double R = 1.9;
Complex Y = X + R; // Y = X.operator+(Complex(R));
```

- Will not work, if left operand is *double*

```
Complex X(4.1, 2.3);
double R = 1.9;
Complex Y = R + X; // syntax error
```

- Better to implement binary operator as nonmember

Multiple Overloading

- Nonmember will work also when *double* is at the left

```
friend Complex operator+(Complex LHS, Complex RHS) {
  return (Complex(LHS.Real + RHS.Real, LHS.Imag + RHS.Imag));
}
```

```
Complex X(4.1, 2.3);
double R = 1.9;
Complex Y = X + R; // Y = operator+(X,Complex(R));
Complex Z = R + X; // Y = operator+(Complex(R),X);
```

- When to implement operators as nonmembers
  - When working with basic data types,

    e.g. ***`Complex operator+(int LHS, const Complex& RHS);`***
  - When we cannot modify the original class,

    e.g. ***`ostream`***

Provide a Reasonable Set of Operators

- In some cases, whole categories of operators make sense for a type.
- For instance, it makes sense to overload all of the arithmetic operators for the class *Complex*. It also makes sense to overload all six relational operators for the class *Name*.
- Often the implementation of one operator can "piggyback" off of another:

```
Complex operator + (Complex s1, Complex s2)
{
  Complex n (s1);
  return n += s2;
}
```

Stream I/O Operators

- We do not have access to the *istream* or *ostream* class code, so we cannot make overloadings of << or >> members of those classes.
- We also cannot make them members of a data class because the first parameter must then be an object of that type.
- Therefore we must define *operator*<< as non-member function.
- However, it must access private members of the data class, so we will typically make it a friend of that class. The alternative would be to have accessor functions for all the data members that will be written, and that is frequently unacceptable.
- The general signature will be:

```
ostream& operator<<(ostream& Out, const Data& toWrite)
```

operator<< for *Complex* Objects

- This overloaded *operator*<< will write a nicely formatted *Complex* object to any output stream:

```
ostream& operator<<(ostream& Out, const Complex& toWrite) {
   const int Precision = 2;
   const int FieldWidth = 8;
   Out << setprecision(Precision);
   Out << setw(FieldWidth) << toWrite.Real;
   if (toWrite.Imag >= 0)
        Out << " + ";
   else
        Out << " - ";
   Out << setw(FieldWidth) << fabs(toWrite.Imag);
   Out << "i";
   Out << endl;
   return Out;
}
```

operator>> for *Complex* Objects

This overloaded *operator>>* will read a complex number formatted in the manner used by *operator<<*:

```
istream& operator>>(istream& In, Complex& toRead) {
  char signOfImag;
  In >> toRead.Real;
  In >> signOfImag;
  In >> toRead.Imag;
  if (signOfImag == '-')
    toRead.Imag = -toRead.Imag;
  In.ignore(1, 'i');
  return In;
}
```

- Of course, this depends on knowing exactly how the *Complex* objects are formatted in the input stream.
- We could make this a lot more complicated if we had multiple formats to deal with.

Indexing Operator Overloading

```
class vector
{
  int *data;
  unsigned int size;
public:
  vector(int n); //creates n-element vector
  ~vector();
  int& operator[] (unsigned int pos);
  int operator[] (unsigned int pos) const
  //copy constructor, assignment operator, ...
};
int& vector::operator[] (unsigned int pos)
{
  if (pos >= size)
    abort ();
  return data[pos];
}
int vector::operator[] (unsigned int pos) const
{
  if (pos >= size)
    abort ();
  return data[pos];
}
```

Provides expected functionality, allowing us to write:

```
vector a(10);
a[5]=10;
cout << a[4]<<endl;
```

Relational Operators in General

- If objects of a class will routinely be stored in a container, the class should provide overloadings for at least some of the relational operators.
- In order to perform searches and sorts, the container object must be able to compare the stored objects. There are several approaches:
  - use accessor members of the stored objects and compare data members directly
  - use comparison member functions of the stored objects, as opposed to operators, to compare the data members
  - use overloaded relational operators provided by the stored objects
- The first requires the container to know something about the types of the data members being compared.
- The second requires the stored objects to provide member functions with constrained interfaces.
- The third allows natural, independent design on both sides.