Pointers and Dynamic Objects

Mechanisms for developing
flexible list representations

Usefulness

- Mechanism in C++ to pass command-line parameters to a program
  - This feature is less important now with the use of graphical interfaces
- Necessary for dynamic objects
  - Objects whose memory is acquired during program execution as the result of a specific program request
    - Dynamic objects can survive the execution of the function in which they are acquired
  - Dynamic objects enable variable-sized lists

Categorizing Expressions

- Lvalue expressions
  - Represent objects that can be evaluated and modified
- Rvalue expressions
  - Represent objects that can only be evaluated
- Consider

```
int a;
vector<int> b(3);
int c[3];
a = 1;              // a: lvalue
c[0] = 2*a + b[0];  // c[0], a, b[0]: lvalues
```

- Observation
  - Not all lvalues are the names of objects

Basics

- Pointer
  - Object whose value represents the location of another object
  - In C++ there are pointer types for each type of object
    - Pointers to int objects
    - Pointers to char objects
    - Pointers to RectangleShape objects
  - Even pointers to pointers
    - Pointers to pointers to int objects

Syntax

- Examples of uninitialized pointers

Indicates pointer object

```
int *iPtr;        // iPtr is a pointer to an int
char *s;          // s is a pointer to a char
Rational *rPtr;   // rPtr is a pointer to a
                  // Rational
```

- Examples of initialized pointers

```
int i = 1;
char c = 'y';
int *ptr = &i;    // ptr is a pointer to int i
char *t = &c;     // t is a pointer to a char c
```

Indicates to take the address of the object

Memory Depiction

```
int i = 1;
char c = 'y';
int *ptr = &i;
char *t = &c
```

Indirection Operator

- An asterisk has two uses with regard to pointers
  - In a definition, it indicates that the object is a pointer

```
char *s; // s is of type pointer to char
```

  - In expressions, when applied to a pointer it evaluates to the object to which the pointer points

```
int i = 1;
int *ptr = &i;         // ptr points to i
*ptr = 2;
cout << i << endl;     // display a 2
```

* indicates indirection or dereferencing

*ptr is an lvalue

Address Operator

- & use is not limited to definition initialization

```
int i = 1;
int j = 2;
int *ptr;
ptr = &i;      // ptr points to location of i
*ptr = 3;      // contents of i are updated
ptr = &j;      // ptr points to location of j
*ptr = 4;      // contents of j are updated
cout << i << " " << j << endl;
```

Null Address

- 0 is a pointer constant that represents the empty or null address
  - Its value indicates that pointer is not pointing to a valid object
  - Cannot dereference a pointer whose value is null

```
int *ptr = 0;
cout << *ptr << endl; // invalid, ptr
                      // does not point to
                      // a valid int
```

Member Indirection

- Consider

```
Rational r(4,3);
Rational rPtr = &r;
```

- To select a member of r using rPtr and member selection, operator precedence requires

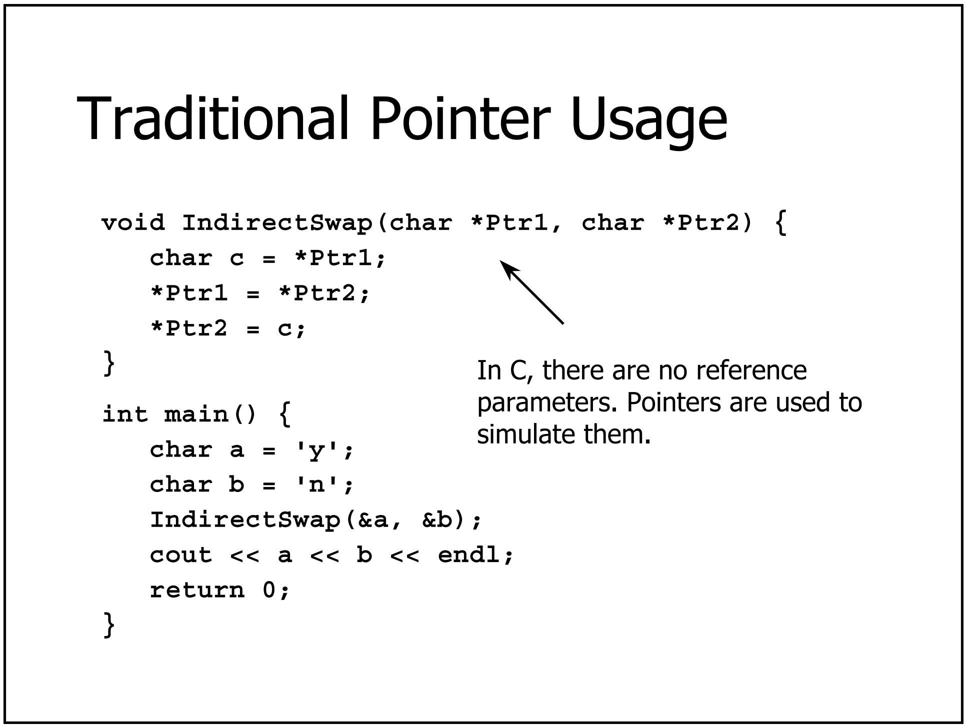
```
(*rPtr).Insert(cout);
```

Invokes member Insert() of the object to which rPtr points (r)

- This syntax is clumsy, so C++ provides the indirect member selector operator ->

```
rPtr->Insert(cout);
```

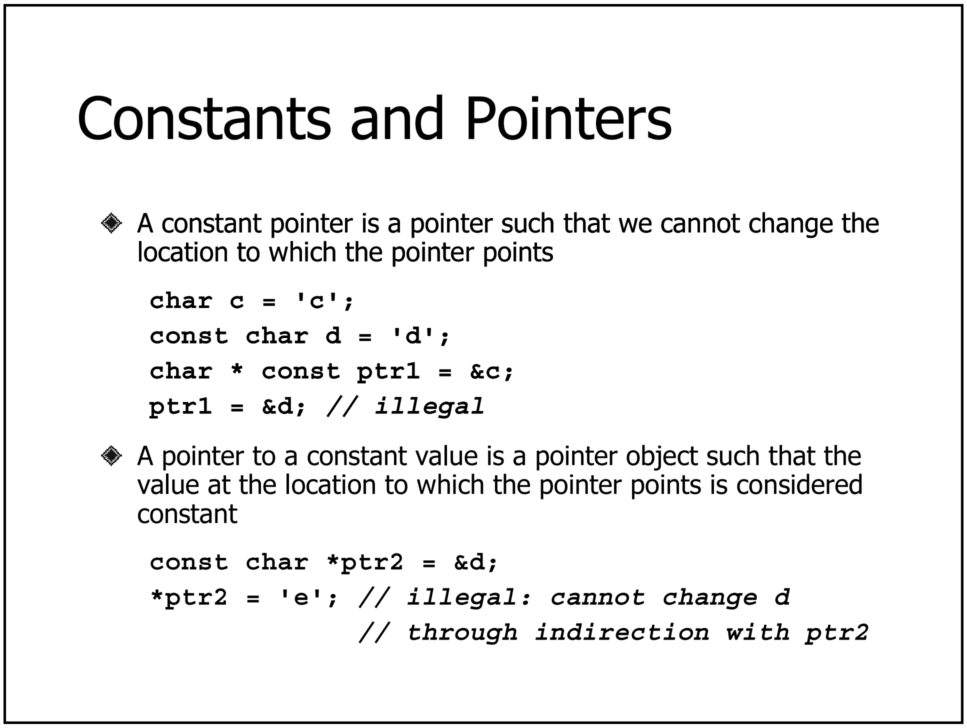
Invokes member Insert() of the object to which rPtr points (r)



Traditional Pointer Usage

```
void IndirectSwap(char *Ptr1, char *Ptr2) {
   char c = *Ptr1;
   *Ptr1 = *Ptr2;
   *Ptr2 = c;
}
int main() {
   char a = 'y';
   char b = 'n';
   IndirectSwap(&a, &b);
   cout << a << b << endl;
   return 0;
}
```

In C, there are no reference parameters. Pointers are used to simulate them.



Constants and Pointers

- A constant pointer is a pointer such that we cannot change the location to which the pointer points

```
char c = 'c';
const char d = 'd';
char * const ptr1 = &c;
ptr1 = &d; // illegal
```

- A pointer to a constant value is a pointer object such that the value at the location to which the pointer points is considered constant

```
const char *ptr2 = &d;
*ptr2 = 'e'; // illegal: cannot change d
             // through indirection with ptr2
```

Differences

- Local objects and parameters
  - Object memory is acquired automatically
  - Object memory is returned automatically when object goes out of scope

- Dynamic objects
  - Object memory is acquired by program with an allocation request
    - new operation
  - Dynamic objects can exist beyond the function in which they were allocated
  - Object memory is returned by a deallocation request
    - delete operation

General New Operation Behavior

- Memory for dynamic objects
  - Requested from the free store
    - Free store is memory controlled by operating system
- Operation specifies
  - The type and number of objects
- If there is sufficient memory to satisfy the request
  - A pointer to sufficient memory is returned by the operation
- If there is insufficient memory to satisfy the request
  - An exception is generated
    - An *exception* is an error state/condition which if not handled (corrected) causes the program to terminate

The Basic New Form

- Syntax

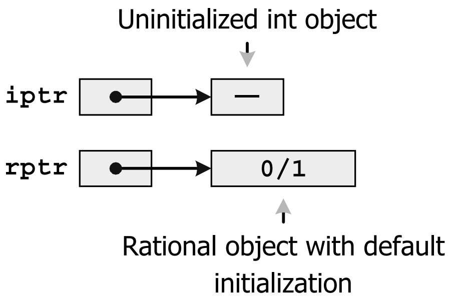
  ```
  Ptr = new SomeType ;
  ```

  - Where
    - Ptr is a pointer of type SomeType

- Beware
  - The newly acquired memory is uninitialized unless there is a default SomeType constructor

Examples

```
int *iptr = new int;
Rational *rptr = new Rational;
```



Uninitialized int object

iptr

—

rptr

0/1

Rational object with default initialization

Another Basic New Form

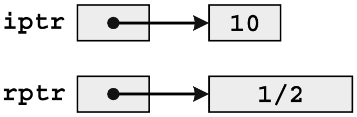
- Syntax

```
SomeType *Ptr = new SomeType(ParameterList);
```

  - Where
    - Ptr is a pointer of type SomeType
- Initialization
  - The newly acquired memory is initialized using a SomeType constructor
  - ParameterList provides the parameters to the constructor

Examples

```
int *iptr = new int(10);
Rational *rptr = new Rational(1,2);
```



The Primary New Form

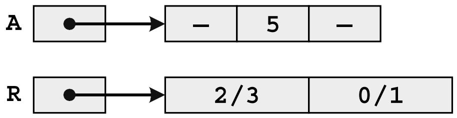
- Syntax

```
P = new SomeType [Expression] ;
```

  - Where
    - P is a pointer of type SomeType
    - Expression is the number of contiguous objects of type SomeType to be constructed -- we are making a list
  - Note
    - The newly acquired list is initialized if there is a default SomeType constructor
- Because of flexible pointer syntax
  - P can be considered to be an array

Examples

```
int *A = new int [3];
Rational *R = new Rational[2];
A[1] = 5;
Rational r(2/3);
R[0] = r;
```



Right Array For The Job

```
cout << "Enter list size: ";
int n;
cin >> n;
int *A = new int[n];
GetList(A, n);
SelectionSort(A, n);
DisplayList(A, n);
```

- Note
  - Use of the container classes of the STL is preferred from a software engineering viewpoint
    - Example vector class

Delete Operators

- Forms of request

```
delete P;     // used if storage came from new
delete [] P;  // used if storage came from new[]
```

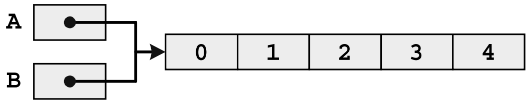
  - Storage pointed to by P is returned to free store
    - P is now undefined

Cleaning Up

```
int n;
cout << "Enter list size: ";
cin >> n;
int *A = new int[n];
GetList(A, n);
SelectionSort(A, n);
DisplayList(A, n);
delete [] A;
```

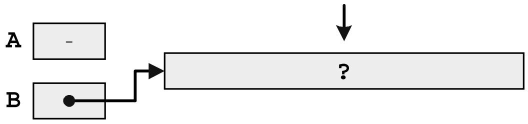
Dangling Pointer Pitfall

```
int *A = new int[5];
for (int i = 0; i < 5; ++i) A[i] = i;
int *B = A;
```



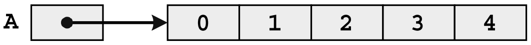
```
delete [] A;
```

Locations do not belong to program



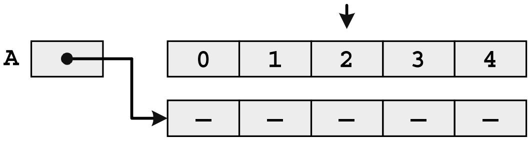
Memory Leak Pitfall

```
int *A = new int [5];
for (int i = 0; i < 5; ++i) A[i] = i;
```



```
A = new int [5];
```

These locations cannot be accessed by program



A Simple Dynamic List Type

- What we want
  - An integer list data type IntList with the basic features of the vector data type from the Standard Template Library
- Features and abilities
  - True object
    - Can be passed by value and reference
    - Can be assigned and copied
  - Inspect and mutate individual elements
  - Inspect list size
  - Resize list
  - Insert and extract a list

Sample IntList Usage

```
IntList A(5, 1);
IntList B(10, 2);
IntList C(5, 4);
for (int i = 0, i < A.size(); ++i) {
  A[i] = C[i];
}
cout << A << endl; // [ 4 4 4 4 4 ]
A = B;
A[1] = 5;
cout << A << endl; // [ 5 2 2 2 2 2 2 2 2 2 ]
```

IntList Definition

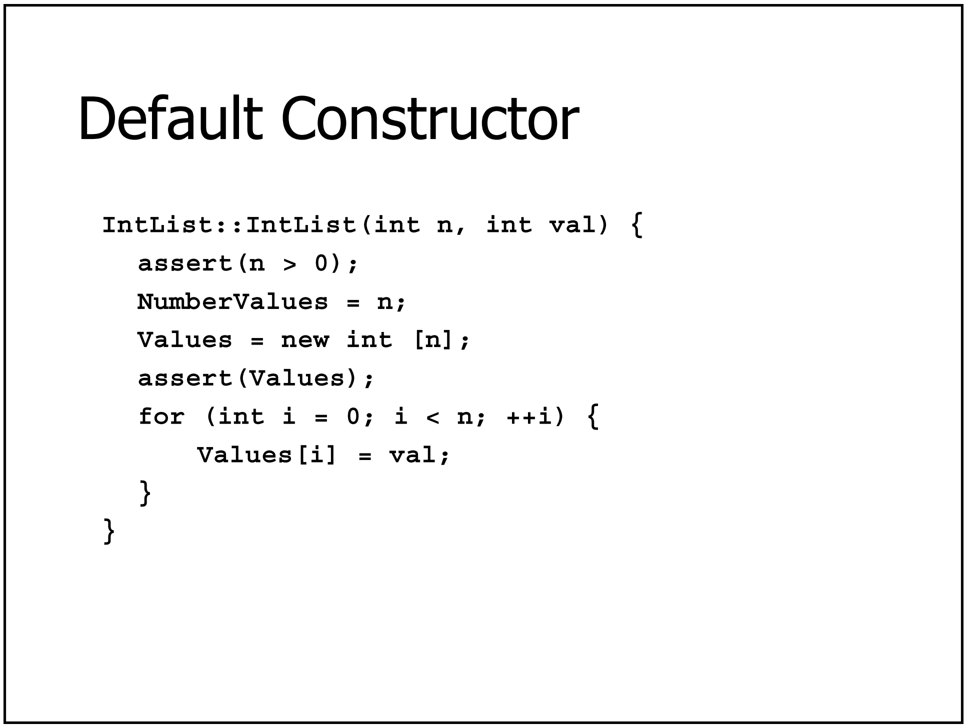
```
class IntList {
  public:
      // constructors
      IntList(int n = 10, int val = 0);
      IntList(const int A[], int n);
      IntList(const IntList &A);
      // destructor
      ~IntList();
      // inspector for size of the list
      int size() const;
      // assignment operator
      IntList & operator=(const IntList &A);
```

IntList Definition (continued)

```
  public:
      // inspector for element of constant list
      const int& operator[](int i) const;
      // inspector/mutator for element of
      // nonconstant list
      int& operator[](int i);
      // resize list
      void resize(int n = 0, int val = 0);
      // convenience for adding new last element
      void push_back(int val);
```

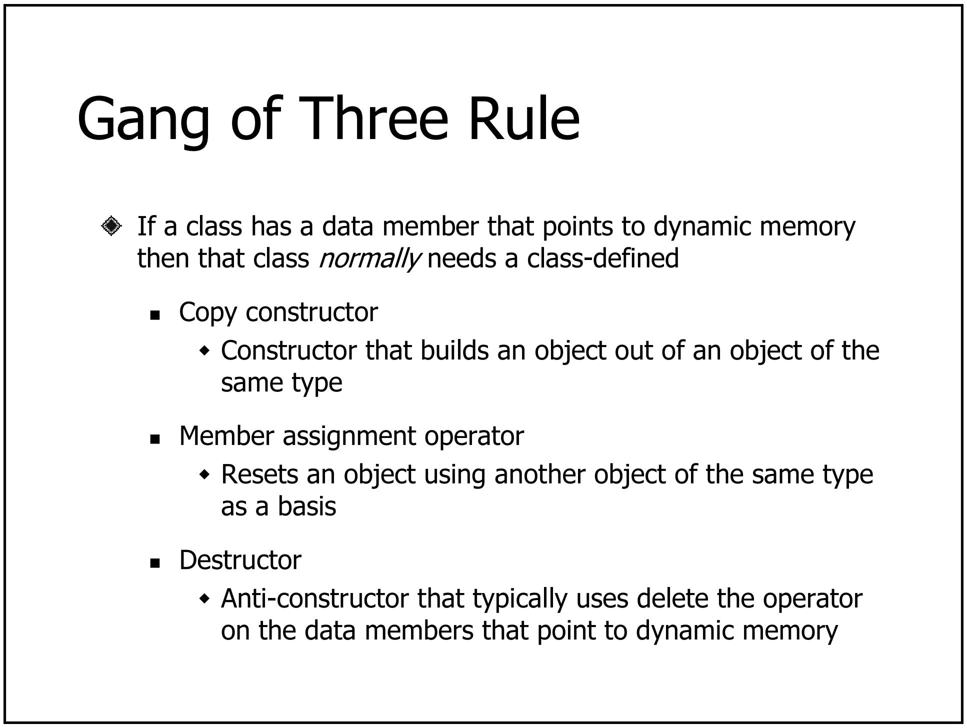
IntList Definition (continued)

```
  private:
      // data members
      int *Values;      // pointer to elements
      int NumberValues; // size of list
};

// IntList auxiliary operators -- nonmembers

ostream& operator<<(ostream &sout, const IntList &A);

istream& operator>>(istream &sin, IntList &A);
```



Default Constructor

```
IntList::IntList(int n, int val) {
   assert(n > 0);
   NumberValues = n;
   Values = new int [n];
   assert(Values);
   for (int i = 0; i < n; ++i) {
        Values[i] = val;
   }
}
```



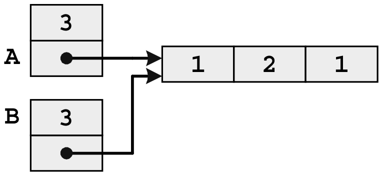
Gang of Three Rule

- If a class has a data member that points to dynamic memory then that class *normally* needs a class-defined
  - Copy constructor
    - Constructor that builds an object out of an object of the same type
  - Member assignment operator
    - Resets an object using another object of the same type as a basis
  - Destructor
    - Anti-constructor that typically uses delete the operator on the data members that point to dynamic memory

Why A Tailored Copy Constructor

- Suppose we use the default copy constructor

```
IntList A(3, 1);
IntList B(A);
```



- And then

```
A[2] = 2;
```

- Then
  - `B[2]` is changed!
  - Not what a client would expect
- Implication
  - Must use tailored copy constructor

Tailored Copy Constructor

```
IntList::IntList(const IntList &A) {
   NumberValues = A.size();
   Values = new int [size()];
   assert(Values);
   for (int i = 0; i < size(); ++i)
        Values[i] = A[i];
}
```

What kind of subscripting is being performed?

Gang Of Three

- What happens when an IntList goes out of scope?
  - If there is nothing planned, then we would have a memory leak
- Need to have the dynamic memory automatically deleted
  - Define a destructor
    - A class object going out of scope automatically has its destructor invoked

Notice the tilde

```
IntList::~IntList() {
   delete [] Values;
}
```

First Assignment Attempt

- Algorithm
  - Return existing dynamic memory
  - Acquire sufficient new dynamic memory
  - Copy the size and the elements of the source object to the target element

Initial Implementation (Wrong)

```
IntList& operator=(const IntList &A) {
  NumberValues = A.size();
  delete [] Values;
  Values = new int [NumberValues ];
  assert(Values);
  for (int i = 0; i < A.size(); ++i)
     Values[i] = A[i];
  return A;
}
```

- Consider what happens with the code segment

```
IntList C(5,1);
C = C;
```

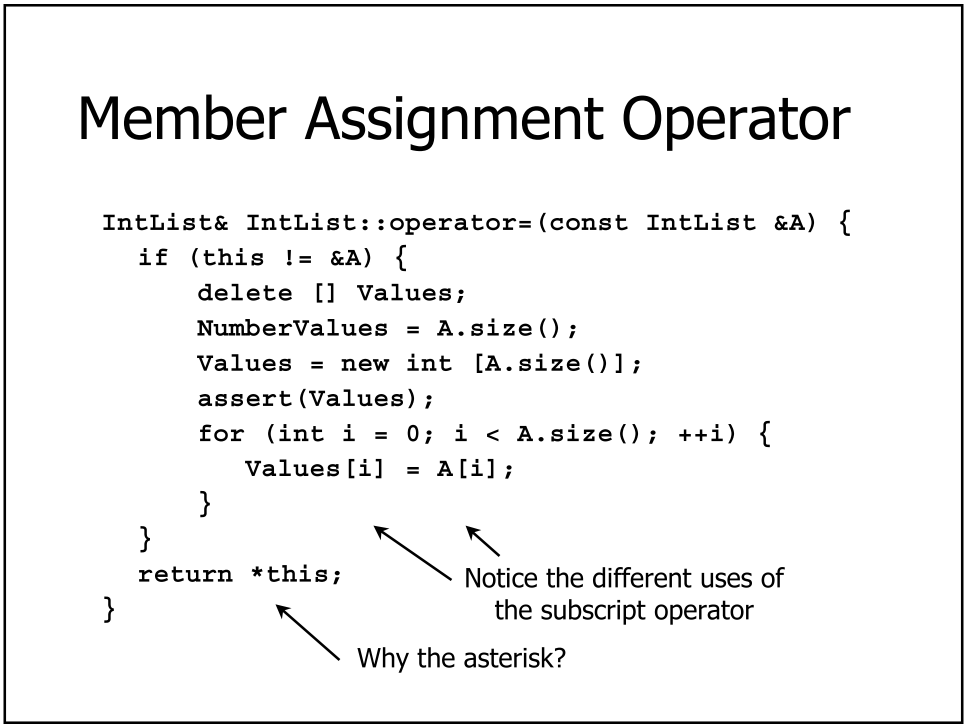
This Pointer

- Consider
  - `this`
- Inside a member function or member operator this is a pointer to the invoking object

```
IntList::size() {
   return NumberValues;
}
```

or equivalently

```
IntList::size() {
   return this->NumberValues;
}
```

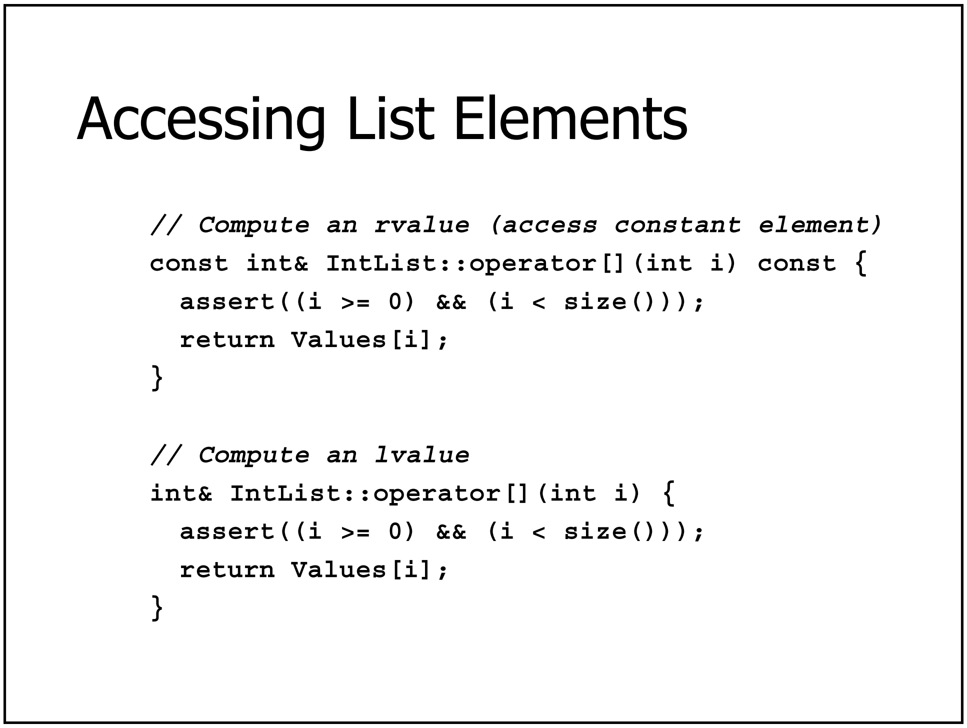


Member Assignment Operator

```
IntList& IntList::operator=(const IntList &A) {
   if (this != &A) {
       delete [] Values;
       NumberValues = A.size();
       Values = new int [A.size()];
       assert(Values);
       for (int i = 0; i < A.size(); ++i) {
          Values[i] = A[i];
       }
   }
   return *this;
}
```

Notice the different uses of the subscript operator

Why the asterisk?



Accessing List Elements

```
// Compute an rvalue (access constant element)
const int& IntList::operator[](int i) const {
  assert((i >= 0) && (i < size()));
  return Values[i];
}

// Compute an lvalue
int& IntList::operator[](int i) {
  assert((i >= 0) && (i < size()));
  return Values[i];
}
```

Stream Operators

- Should they be members?

```
class IntList {
  // ...
  ostream& operator<<(ostream &sout);
  // ...
};
```

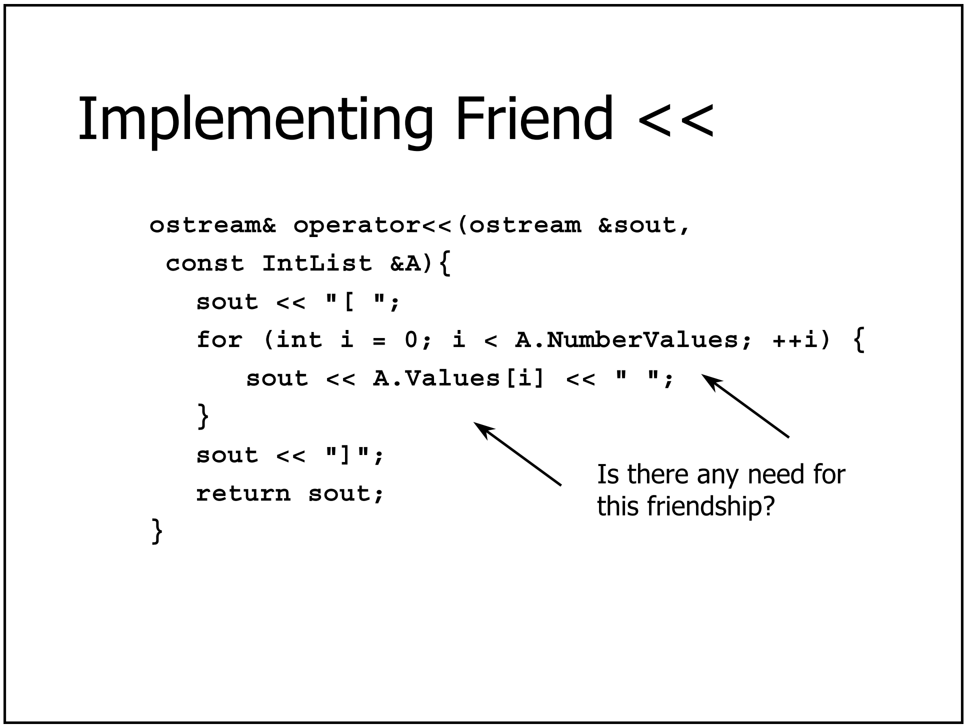
- Answer is based on the form we want the operation to take

```
IntList A(5,1);
A << cout;  // member form (unnatural)
cout << A;  // nonmember form (natural)
```

Beware of Friends

- If a class needs to
  - Can provide complete access rights to a nonmember function, operator, or even another class
    - Called a friend
- Declaration example

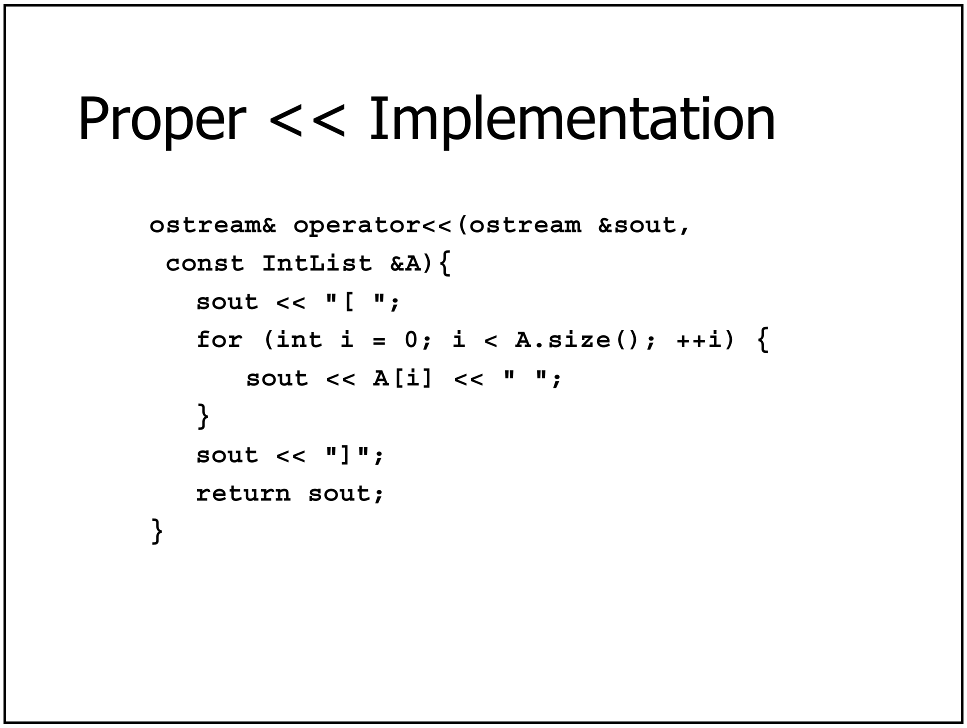
```
class IntList {
  // ...
  friend ostream& operator<< (
    ostream &sout, const IntList &A);
  // ...
};
```



Implementing Friend <<

```
ostream& operator<<(ostream &sout,
 const IntList &A){
   sout << "[ ";
   for (int i = 0; i < A.NumberValues; ++i) {
      sout << A.Values[i] << " ";
   }
   sout << "]";
   return sout;
}
```

Is there any need for this friendship?



Proper << Implementation

```
ostream& operator<<(ostream &sout,
 const IntList &A){
   sout << "[ ";
   for (int i = 0; i < A.size(); ++i) {
      sout << A[i] << " ";
   }
   sout << "]";
   return sout;
}
```