CSE 143

Classes with Dynamically Allocated Data

Concepts (this lecture)

- Constructors and destructors for dynamic data
- The **`this`** pointer
- Deep vs shallow copy
- Defining assignment (**`operator=`**)
- Copy constructor
- Standard components of a C++ class (how to create a proper user-defined type)

Example: List of ints w/dups

- New class IntList
- Instances of IntList hold collections of integers
  - An int may appear more than once in an IntList
- Operations
  - Constructors: Create empty IntList; create IntList with room for n elements initially
  - Queries: determine size of an IntList; find out if a given number appears in an IntList

List of ints w/dups (cont.)

- Operations (cont)
  - Modify: add a number to an IntList , expanding the list to make room if needed
  - Addition: create a new list that contains all of the elements from two other lists
  - Stream output: <<

Using IntLists

- An IntList works like this

```
IntList L, M, N;
L.add(17); L.add(42);
if (L.isMember(17))
   cout << “works” << endl;
else
   cout << “is broken” << endl;
M.add(143);
N = M + L;
```


IntList Representation

- In file intlist.h

```
class IntList {
 public:
   ...
 private:
   int arraysz; // current size of array L
   int sz;      // current # items in list
   int *L;      // array containing list
                // elements in L[0..sz-1].
                // L[sz..arraysz-1] is
                // free space.
 ...
 };
```

IntList Interface

- In file intlist.h

```
class IntList {
public:

   // constructors
   IntList( );     // create empty IntList
   IntList(int n); // create empty IntList
                   //   with free space
                   //   for n ints

   // destructor
   ~IntList( );
```


IntList Interface (cont)

```
   // = # elements in this IntList
   int size( ) const;
   // = "n is in this IntList"
   bool isMember(int n) const;
   // add n to this IntList
   void add(int n);
   // = new IntList with all elements from
   //   s followed by all elements from t
   friend IntList operator+
     (const IntList &s, const IntList &t);
   // stream output
   friend ostream &operator<<(ostream &s,
                const IntList &t);
   ...
};
```


Constructors

- Need to initialize private data, including allocating an array to hold the list data. Initial size of array is some small number if the client doesn't request anything specific.
- In file intlist.cpp

```
static const int initialsz = 4;
   // initial size of array if none given
   // (unrealistically small to make it
   // easier to test/demonstrate operation
   // of IntList::add)
```


Constructors (cont)

- In file intlist.cpp (cont)

```
// create empty list
IntList::IntList( ) {
    arraysz = initialsz;
    L = new int[arraysz];
    sz = 0;
}

// create empty IntList w/space for n ints
IntList::IntList(int n) {
    arraysz = n;
    L = new int[arraysz];
    sz = 0;
}
```


Dynamic Data Management

- An IntList refers to dynamically allocated data. The array needs to be deleted when the IntList no longer exists.

```
Void f( ) {
    IntList L1;
    ...
}
```

Array allocated here by constructor

Automatic storage used by L1 reclaimed here

- When function **f** terminates, the space holding ints and pointers of **L1** is recycled. If nothing further is done, the array is still allocated, but not accessible. (Memory leak)

Destructors

- A class may have *exactly one* destructor.
- The destructor is automatically executed for each object when that object is about to be destroyed
- The destructor should do any cleanup needed before the object vanishes, including releasing dynamically allocated storage used by the object
- Destructor name is ***~ClassName***
- Destructor for IntLists (in intlist.cpp):

```
   // destructor
   IntList::~IntList( ) {
      delete [ ] L;
   }
```

List Operations

- **size** and **isMember** present no surprises

```
// = # items in this list
int IntList::size( ) const {
   return sz;
}

// = "n is in this list"
bool IntList::isMember(int n) const {
   int k = 0;
   while (k<sz && n!=L[k])
      k++;
   return k<sz;
}
```

- Stream output (<<) is similar to previous versions.

this Pointer

- In a member function, references to member data are done via a pointer variable **this**, which is an implicit parameter to every member function.
- **this** is automatically initialized when a member function is called, and can be used explicitly if desired.

```
// = "n is in this list"
bool IntList::isMember(int n) const {
   int k = 0;
   while (k < this->sz
            && n != this->L[k])
      k++;
   return k < this->sz;
}
```


Function add

- **add** is easy if there is room in the array for the new int value.

```
// add n to this IntList
int IntList::add( ){

   if (sz == arraysz) {
      // array is full, make it bigger
      ...
   }

   // add n at end of array
   L[sz] = n;
   sz++;
}
```


Function add (cont)

- If the array is full, we allocate a new one twice as large and copy the existing values to it.

```
if (sz == arraysz) {
   // array is full; double its size
   arraysz = 2 * arraysz;
   int * newL = new int[arraysz];
   for (int k = 0; k < sz; k++)
      newL[k] = L[k];
   delete [ ] L;
   L = newL;
}
// add n at end of array
L[sz] = n;
sz++;
```

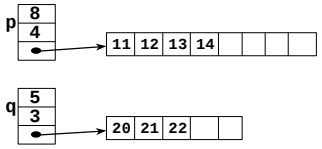

operator+

- To compute the "sum" of two lists, construct a new list large enough to hold both IntLists.

```
// = new list with contents of s and t
IntList operator+(const IntList &s,
                  const IntList &t) {
  IntList ans(s.arraysz + t.arraysz);
  int j, k;
  for (k=0; k<s.sz; k++)      // copy s
    ans.L[k]=s.L[k];
  for (j=0; j<t.sz; j++) {    // copy t
    ans.L[k] = t.L[j];
    k++;
  }
  ans.sz = s.sz + t.sz;
  return ans;
}
```


Are we done?

- Almost.
- But suppose we have the following two IntLists.



- What happens if we execute assignment **p=q;**?

Problems after **p=q;**

- Aliasing: **p.L** and **q.L** refer to the same array
  - Changes to one will affect the other
  - Information about array sizes and number of IntList elements in **p** and **q** can become inconsistent.
- Memory leak: What happened to the array with IntList **p**'s old data?
- Multiple destructor problem. When **p** and **q** are eventually deleted, **IntList::~IntList** will be called twice, which will attempt to delete the (one) array twice.

Shallow vs Deep Copy

- The problem is that the default assignment operator only copies the top-level data in the object, and does not clone any data pointed to from there. This is known as a *shallow copy*.
- What we want is a *deep copy*, which copies the entire object, including any dynamic data it references.
- Solution: Redefine assignment for IntLists.

Assignment: **operator=**

- First implementation

```
// assign contents of rhs to this IntList
int IntList::operator=
                    (const IntList &rhs){
   // if old array is not large enough,
   // replace it with one the size of rhs.L
   if (arraysz < rhs.sz){
      delete [ ] L;
      arraysz = rhs.arraysz;
      L = new int[arraysz];
   }
   // copy values to this array;
   sz = rhs.sz;
   for (int k = 0; k < sz; k++)
      L[k] = rhs.L[k];
}
```


Are we done? (2)

- Close.
- Problem 1: What happens if we execute **p=p;** ?
- Problem 2: C/C++ allows multiple assignments. The meaning of

```
w = x = y = z;
```

is supposed to be

```
w = (x = (y = z));
```

So **operator=** needs to return a reference to the value assigned so it can be used as the source for further assignments.

Fixing **operator=**

- Solution 1: Check to see if this IntSet is the same as the one on the right side of the assignment.

```
IntList::operator=(const IntList &rhs){
   ...
   if (this == &rhs)
      return without doing anything
   ...
```

- Solution 2: **operator=** should return a reference to the object it has just updated.

```
   ...
   // update this object
   ...
   // return reference to new value
   return *this;
```


Revised **operator=**

```
// assign contents of rhs to this IntList
IntList & IntList::operator=
                    (const IntList &rhs){

   // exit if attempting to assign to self
   if (this == &rhs)
      return *this;

   // if old array is not large enough,
   // replace it with one the size of rhs.L
   if (arraysz < rhs.sz){
      delete [ ] L;
      arraysz = rhs.arraysz;
      L = new int[arraysz];
   }
```

Revised **operator=** (cont)

```
   // copy values to this array;
   sz = rhs.sz;
   for (int k = 0; k < sz; k++)
      L[k] = rhs.L[k];

   // return reference to result
   return *this;
}
```


Are we done? (3)

- Much closer
- Last wrinkle. What if we declare and initialize an IntList like **L2** below?

```
IntList L1;
L1.add(10);
IntList L2 = L1;
```

- Or suppose we have an IntList value parameter

```
void f(IntList L3) { ... }
```


Are we done? (3) (cont)

- Or suppose a function returns an (anonymous) IntList created by copying a local variable.

```
IntList g() {
   IntList result;
   // calculate answer and store in result
   ...
   return result;
}
```

- In these situations, we're creating a new, never-initialized IntList, not updating one that already has a value. So **operator=** isn't quite right.

Copy Constructor

- A constructor with a **const &** parameter of the same class.
- Executed whenever a new, never initialized object is created with a specified initial value.

```
// copy constructor
IntList::IntList(const IntList &val){
   // allocate array same size as initial
   // value, then copy data
   arraysz = val.arraysz;
   L = new int[arraysz];
   sz = val.sz;
   for (int k = 0; k < sz; k++)
      L[k] = rhs.L[k];
}
```


Defining a New C++ Type

- A complete C++ class should contain the following members
  - Constructor(s) - including one with no parameters (null constructor)
  - Copy constructor
  - Destructor
  - Assignment (**operator=**)
- Instances of such classes will work like variables of any built-in type (including declaration, assignment, and use as function parameters)
- Additional operations and redefinitions of others (**operator==**, for example) will also usually be needed/desired, depending on the application