CSE 143

Classes with Member Functions

[pp. 125-140]

Type = Data + Operations

- Data Types have two components
  - Set of possible values
  - Operations that can be applied to values
- Examples
  - Integers: arithmetic operations, printing, etc.
  - Boolean: `AND`, `OR`, `NOT`, test if `true`, etc.
  - Grade Transcript: Add, remove classes and grades, change grades, etc.

Type = Data + Operations

- More Examples:
  - Automatic Teller Machine

```
Data: cash available, machine status
Operations: get account information, dispense
cash, confiscate card, ...
```

  - Telephone network switch

```
Data: line status, call information
Operations: set up and break down calls, send
billing information, test circuits,...
```


Terminology

- Type
  - A category
  - A model for values that have the type
  - Some supplied as part of the programming language, others user-defined
  - Usually few in number

Terminology (cont)

- Instance
  - A specific instantiation of some model (type)
  - Often called a variable or object (these terms have other implications, but when speaking of instances/objects/variables vs types, they mean roughly the same thing)
  - Usually many instances of any particular type

Types in C++

- Primitives: int, double, char, …
- New types created with Class definitions (and, historically, typedef)
- Class members may be
  - Data: underlying representation of values
  - Functions: operations on objects of the class type
- Class instances behave like ordinary variables (if the class is defined properly)

Example: Simple Cell Class

- A Cell contains 4 members, which are defined in class Cell:
  - An integer data value (private)
  - A constructor that initializes newly created Cells
  - Two functions get and set to retrieve and alter the value of a cell
- Source code on the course web site

Using Member Functions

- Use of Cells (in **`main`**, for example)

```
Cell c;         // construct new cell
c.set(-5);      // set cell value to -5
cout << c.get();    // write cell value
```

- A member function is selected with '.' notation, just like data members (*instance.member*)
- When member functions (set, get) are called, the function is automatically linked to the particular cell c used to call it.

Cell class declaration

- In cell.h

```
class Cell {
public:
   // constructor
   cell( );
   // access function
   int get( );
   // modifier function
   void set(int n);
private:
   int val;  // current cell value
};
```

- get and set are member functions, not friends

Cell class implementation

- In cell.cpp:

```
#include "cell.h"
// constructor: initialize this Cell
Cell::Cell( ) {val = 17;}

// yield current value of this Cell
int Cell::get( ) {return val;}

// set value of this Cell to n
void Cell::set(int n) {val = n;}
```

Class name

Member Function name

Scope resolution operator

Scope Resolution Operator

- The same identifier can be used as a member name in different classes.
- The *scope-resolution* operator (::) is needed in the implementation file to indicate which class the member function belongs to

  class_name :: *member_name*
- The same rule applies to constructors. For constructors, *class_name* and *member_name* are the same.

Member Function Notes

- Inside member function implementations:
  - Can refer directly to other data and function members of the current object (the one used to call the function)
  - OK to access any public or private class members directly - the member names refer to those in the current object
  - If one member function calls another, member names in the new function refer to the same object
  - When a member function is called, the object itself is implicitly the "first" parameter to the function. This invisible parameter is how member names can be used to directly access members of the current object.

New Example: Set of Ints

- New class IntSet
- Instances of IntSet hold sets of integers
- Operations:
  - Constructor: Create empty set
  - Others: add a number to a set, delete a number from a set, report size of set, report whether some number is in a set, merge all elements of a set into another set, stream output (<<)

Using IntSets

- An IntSet works like this (in main, for example)

```
bool p;
IntSet a;       // a = { }
IntSet b;       // b = { }
a.add(17);      // a ={17}
a.add(42);      // a ={42,17} or {17,42}
b.add(143);     // b = {143}
cout << a.size();   // output: 2
p = a.isMember(50); // p = false
b.addAllFrom(a);    // b = {143,42,17}
b.remove(143);      // b = {42,17}
```


Public Interface

- In file intset.h

```
class IntSet {
public:
  // constructor: Initialize empty new set
  IntSet( );

  // = number of elements in this set
  int size( ) const;

  // Add n to this set if it is not already
  // present and if there's room.  Do
  // nothing otherwise.
  void add(int n);
```


Public Interface (cont)

```
// Remove n from this set if it is
// present, otherwise do nothing.
void remove(int n);

// = "n is a member of this set"
bool isMember(int n) const;

// Add all elements of set t to this set
void addAllFrom(const IntSet &t);

// Stream output for Intsets.  Print
// contents of set t on stream s
// as {n1,n2,n3,…,nn}
friend ostream& operator<<(ostream &s,
                           const IntSet &t);
```


Interlude: const

- Const is used to declare constant values

```
const int maxSetSize = 100;
```

- It is also used to declare that a member function doesn't change the object when it's called

```
int size() const;   // s.size() won't
                    // change s
```

- It is also used to declare that a function does not alter the value of a reference parameter

```
void addAllFrom(const IntSet &t);
          // s.addAllFrom(t) won't
          // change t
```

- "const correctness" is a good thing - do it

Representation of IntSets

- In file intset.h

```
#include <iostream.h>
const int maxSetSize = 100; // max # elts
class IntSet {
public:
  ...
private:
  int nelts;  // # of elements in this
              // set,0<=nelts<=maxSetSize
  int ints[maxSetSize];
              // set elements are stored
              // in ints[0..nelts-1]
...
};
```

Constructor Implementation

- In file intset.cpp

```
// constructor: Initialize new set to empty
IntSet::IntSet() {
    nelts = 0;
}
```

- This guarantees that all new sets have a sensible initial value (in this case empty). We do not have to rely on the client calling an "initialize" function.
- IntSet representation is private. Clients cannot corrupt an IntSet by directly modifying its fields.

Add, remove, isMember

- All of these operations require examining the set to see if some number is present
- Introduce a (private) function to do the search
- In intset.h

```
private:
  // If n is in this set, return k such
  // that ints[k]=n, else return nelts
  int indexOf(int n) const;
```

- Not part of the interface. Cannot be called by client code; can be called by any member or friend function of IntSet.

indexOf implementation

- In file intset.cpp

```
// If n is in this set, return k such
// that ints[k]=n, else return nelts
int IntSet::indexOf(int n) const {
  int k = 0;
  while (k < nelts && ints[k] != n)
    k++;
  return k;
}
```

- As a member function, indexOf can access members of the current IntSet object directly

Implementation of add

- In file intset.cpp

```
// Add n to this set if it is not already
// present and if there's room.
Void IntSet::add(int n) {
  int loc = indexOf(n);
  if (loc==nelts && nelts < maxSetSize){
    ints[loc] = n;
    nelts++;
  }
}
```

- Implementation of remove is similar (see sample code on the web)

Query functions

- In file intset.cpp

```
// = number of elements in this set
void IntSet::size( ) const {
  return nelts;
}

// = "n is a member of this set"
bool IntSet::isMember(int n) const {
  return indexOf(n) < nelts;
}
```


addAllFrom

- This function adds to the current set everything that appears in another set. It has access to the private members of the other set because it is a member function of class IntSet.

```
// add all elements of set t to this set
Void IntSet::addAllFrom(const IntSet &t){
  for (int k=0; k < t.nelts; k++)
    add(t.ints[k])
}
```

- When `S.addAllFrom(t)` is executing, the function call `add(t.ints[k])` receives `S`, the current object, as its implicit first parameter.

Stream Output for IntSets

- operator<< must be a friend of the class. It cannot be a member, because << cannot have an IntSet as its first parameter (implicit or otherwise)

```
// Write set t on stream s as {x,x,x,x}
ostream& operator<< (ostream &s,
                     const Intset &t){
  s << '{'
  if (t.nelts > 0) s << t.ints[0]
  for (int k=1; k < t.nelts; k++)
    s << ',' << t.ints[k];
  s << '}';
  return s;
}
```

- << can access members of t because it is a friend of class IntSet

Friend vs Member Function

- Should a function be a friend or member of a class? How to choose?
- Friend: Ordinary function with access to private data.
- Declaration:

```
class X {
  ...
  friend X op(X a, X b);
  ...
};
```

- Use:

```
x3 = op(x1,x2);
```


Friend vs Member (cont)

- Member: Function associated with a particular object when it is called.
- Declaration:

```
class X {
  ...
  X opx(X a);
  void opv(X a)
  ...
};
```

- Use:

```
x3 = x1.opx(x2);
x4.opv(x5)
```


Friend vs Member (cont)

- How to choose?
- Member
  - Operation naturally "belongs to" some data object
  - Operation modifies or accesses associated object when executed
- Friend
  - Operation needs access to private implementation details of a class, but…
  - No particularly strong association with one parameter, or
  - Required parameter list won't allow function to have an additional (hidden) object parameter

Example: Class Complex

- Complex class had only friend functions plus constructors so we could postpone dealing with members functions.
- If we did it over, friends or members?
  - Constructors: Complex(a,b), Complex(a), Complex( )
  - Operations: +, -, - (unary), *, /
  - Operations: ==, !=, <, <=, >=, >
  - Access: real( ), imag( )
  - Stream output: <<

Example: Class IntSet

- Suppose we extended class IntSet to provide these additional operations. Friends or members?
  - Iteration operators (pair):

```
reset (start iteration at "beginning" of set)
retrieve next element of iteration
```

  - Set union: **s+t** yields a new set containing all elements found in either **s** or **t**.