More class design with C++

Starting Savitch Chap. 11

Member or non-member function?

- Class operations are typically implemented as member functions
  - Declared inside class definition
  - Can directly access private members
  - Usually the task involves only one object (this)
- But some operations are more appropriate as ordinary (nonmember) functions
  - Declared outside any class definition
  - Usually the task involves more than one object
  - Cannot access private members of a class though
    - Unless they are `friend`s of the class

Implementing an ordinary function

- Consider an equality function for DayOfYear
  - Comparing two objects, so a non-member function

```
bool equal(DayOfYear date1, DayOfYear date2) {
   return date1.get_month() == date2.get_month()
          && date1.get_day() == date2.get_day();
}
```

- Why is function equal not very efficient?
  - Each call to a public accessor function requires "overhead" costs – to manage new stack frames
  - Accessing `date1.month` is simpler, more efficient
    - But it is also illegal! Unless …

friends

- Can be a function or (rarely) a whole other class
- Not class members, but can access private members of a class that has declared it as a friend
- Declared inside class by keyword `friend`

  ```
  class DayOfYear {
  public:
      friend bool equal(DayOfYear date1,
                        DayOfYear date2);
  ```

- Implement without `DayOfYear::`
  - Okay to use private members of DayOfYear though

A Money class with a friend

```
class Money {
public:
    friend Money add (Money, Money);
    ...
private:
    long cents;
};
Money add (Money amt1, Money amt2) {
    Money temp;
    temp.cents = amt1.cents + amt2.cents;
    return temp;
}
```

- Why is this still inefficient? How to improve it?

Parameter passing efficiency

- The add function uses “call-by-value” parameters
  - *Copies* of objects are created and then later destroyed
- Using “call-by-reference” parameters is more efficient – no copies (at that stage anyway):

```
friend Money add (Money &, Money &);
...
Money add (Money &amt1, Money &amt2) {...}
```

- But a new problem now: can’t pass it constant objects – even though it doesn’t change them

const

- Part of an object’s type in C++

```
const int x = 12;
          // must initialize on creation; can never change afterwards
someFunction(x);
          // error if parameter is int& without const
```

- Good classes support constant objects: “SCO”

```
friend Money add (const Money &, const Money &);
...
Money add(const Money &amt1, const Money &amt2){…}
```

- But what about `amt1.getCents()` inside add?
  - Answer: won’t compile! Unless getCents() is const too:

```
long getCents() const;
...
long Money::getCents const { return cents; }
```

Operator function overloading

- Example: `ADT operator+(const ADT &, const ADT &);`
  - Overloads `+` to return an `ADT` object (hopefully the sum of the two `ADT` arguments – best to not change operator's meaning)
- Can overload almost any C++ operator
  - At least one argument must be a user-defined type
  - Precedence, "narity", and associativity rules apply as usual
    - e.g., `+` has usual precedence, is binary or unary, l-r
    - e.g., `=` has lower precedence, is binary only, r-l
  - See other rules on page 629 of the Savitch text
- But "just because you can does not mean you should"
  - e.g., a bad idea to overload `,` or `&&` or `||` even if legal
  - And should always maintain the expected operator behavior

Operator functions for Money

- Replace add function with operator +

```
friend Money operator+
      (const Money &, const Money &);
...
Money operator+(const Money &amt1, const
   Money &amt2){ /* same implementation as add */ }
```

- Replace equal function with operator ==

```
friend bool operator== (const Money &,
   const Money &);
...
bool operator== (const Money &amt1,
   const Money &amt2) {
    return amt1.cents == amt2.cents;
}
```

2 ways to use operator functions

```
Money a(100), b(50); // two Money objects
```

- Can add/compare by functional notation:

```
Money sum1 = operator+(a, b);
if ( operator==(a, b) ) … // false in this case
```

- But now can use infix notation too:

```
Money sum2 = a + b;
if ( sum1 == sum2 ) … // true in this case
```

- By the way: C++ will try to convert any function argument to match the parameter type

```
if ( sum1 == 150 ) … // still true! See next slide.
```

Implicit type conversion in C++

- Converting ctors – e.g., `Money(long dollars);`
  - Any ctor that takes exactly one argument
  - Invoked whenever an argument of that type is passed to a function that expects an object
    - In the case on previous slide – 150 converted to Money(150)
- Operator conversion functions – inverse idea
  - Specify types to which an object may be converted
  - Say class Money has `operator double() const;`
    - Means a Money object can be implicitly converted to `double` in certain circumstances, like `cout << sum1;`
  - Better to overload `<<` instead for this purpose though

Member vs. non-member ops

- Recall that some functions are more naturally defined as class members
  - Specifically, any function that needs a `this` pointer:
    - e.g., ++, +=, … all need to change the object
  - And there are four operators that can only be overloaded as class members: `=`, `( )`, `[ ]`, and `->`
- Sometimes non-member functions better though
  - e.g., binary functions, where the order of the arguments doesn't matter:
    - e.g., `==`, `<`, …, and binary forms of `+`, `-`, `*`, `/`, `%`
  - Also when must access other types – like << and >> that require access to ostream and istream (`cout`, `cin`)

Overloading << and >>

- Want to do: `cout << cost << endl;`
  - Need: `friend ostream& operator<< (ostream& outs, const Money& amount);`

```
...
ostream& operator<<( ostream& outs, const
  Money& amount) {
    // print to outs (e.g., outs << amount.cents;)
    return outs; // must return the ostream reference
}
```

- Want to do: `cin >> price >> tax;`
  - Need: `friend istream& operator>> (istream& ins, Money& amount);`

About member operator functions

- First argument is `this` – but it’s hidden
  - Always the left argument of binary operations
  - So there can be no implicit conversion of left argument – must be object of the correct type
  - Is the only argument of unary operations
- Often return `*this` to allow operation chaining
  - e.g., imagine a Money += (compound assignment op)

```
Money& operator+= (const Money &right);
...
Money& Money::operator+= (Money const &right) {
    return *this = *this + right;
}  // assuming operator= and operator+ are both already defined
```

- Note: two versions of `operator++` and `operator--`
- And usually want two versions of `operator[]`

Three free member operators

- By default, for any class C (even `class C {};`), the compiler supplies three member operators
- An assignment operator

```
C& operator=(const C &);
```

  - Like a free copy ctor … makes a shallow copy
  - So often necessary to redefine it to make a deep copy
- And two different address-of operators
  - One for mutable objects:

```
C* operator&();
```

  - And one for constant objects:

```
const C* operator&() const;
```

  - No good reason to redefine either of these functions!

Classes with dynamic memory

- Must properly manage – to avoid memory leaks
  - C++ does not have an automatic garbage collector – so C++ programmers are responsible for returning memory to the free store
- Example class from text (Display 11.11): `StringVar`

```
 ...
private:
  char *value; // pointer to dynamic array of characters
  int max_length; //declared max length of array
```

  - Point is to hold/manage a C-string of any length

Second Exam
Friday, May 3