6 Designing classes

Class design is a big topic in C++, with many aspects to consider. Since any one class does not illustrate all of the aspects of class design, we use two example classes to illustrate the issues. The first example class is `Rational`, which implements fractions and provides examples of the issues that arise in developing a numerical or algebraic class. The second example class is `Account`, which – with a bit of artificial manipulation – provides examples of memory management and other problems.

161

Rational. Figure 58 on the next page shows the declaration of class `Rational`. Comments that would normally be included in such a declaration have been omitted to save space. The accompanying text provides adequate explanation. Note that this `Rational` class is similar to, but not identical with, the rational example in (Weiss 2004, Section 5.5).

162
163
164

An instance of `Rational` is a rational number, or "fraction", represented by two long integers. The fraction $\frac{n}{d}$ is represented by the pair (n, d). The class provides assignment, copying, arithmetic operations (`+`, `-`, `*`, `/`), and exponentiation (`^`), the associated assignment operators (`+=`, `-=`, `*=`, `/=`), and comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`) for fractions. It also provides stream operators for reading and writing fractions.

Rational numbers are stored in normalized form: the pair (n, d) is in ***normal form*** if $d > 0$ and $\gcd(n, d) = 1$. Thus we can consider $d > 0 \wedge \gcd(n, d) = 1$ to be a ***class invariant***. For example, the construction `Rational(4,-6)` yields the pair $(-2, 3)$. An attempt to create a fraction with zero denominator raises an exception.

Account. The other example class, `Account`, is rather different. Some of the differences are due to the application and some are due to an intentional complication: `Account` has a data member that is a pointer.

An account is associated with a person who has a name. In class `Account`, the name is represented as a `char*` (pointer to array of characters). This introduces various problems, ***all of which could be avoided*** just by using the standard class `string` instead of `char*`. We use `char*`, however, just to show what the problems are when designing classes with pointer members, and how they can be solved.

Other differences between `Rational` and `Account` concern comparison, accessors, and mutators.

165
166

```
#ifndef RATIONAL_H
#define RATIONAL_H

#include <iostream>

class Rational
{
public:
    Rational(long n = 0, long d = 1);
    const Rational & operator+= (const Rational & right);
    const Rational & operator-= (const Rational & right);
    const Rational & operator*= (const Rational & right);
    const Rational & operator/= (const Rational & right);
    const Rational operator-() const;
    const Rational operator^ (int e) const;
    friend const bool operator== (const Rational & left,
                                  const Rational & right);
    friend const bool operator!= (const Rational & left,
                                  const Rational & right);
    friend const bool operator<  (const Rational & left,
                                  const Rational & right);
    friend const bool operator>  (const Rational & left,
                                  const Rational & right);
    friend const bool operator<= (const Rational & left,
                                  const Rational & right);
    friend const bool operator>= (const Rational & left,
                                  const Rational & right);
    friend std::istream & operator>> (std::istream & is,
                                      Rational & r);
    friend std::ostream & operator<< (std::ostream & os,
                                      const Rational & r);
    double toDouble() const;
    enum Exceptions { BAD_INPUT, ZERO_DENOMINATOR };

private:
    void normalize();
    long num;
    long den;
};

const Rational operator+ (const Rational & left, const Rational & right);
const Rational operator- (const Rational & left, const Rational & right);
const Rational operator* (const Rational & left, const Rational & right);
const Rational operator/ (const Rational & left, const Rational & right);

#endif
```

Figure 58: Declaration for class `Rational`

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <string>

class Account
{
public:
    Account();
    Account(char *name, long id, long balance = 0);
    Account(const Account & other);
    ~Account();
    Account & operator=(const Account & other);
    long getBalance() const;
    void deposit(long amount);
    void withdraw(long amount);
    void transfer(Account & other, long amount);
    friend bool operator==(const Account & left,
                           const Account & right);
    friend bool operator!=(const Account & left,
                           const Account & right);
private:
    void storeName(char *s);
    char *name;
    long id;
    long balance;
};

#endif
```

Figure 59: Declaration for class `Account`

Figure 59 shows the declaration of class `Account`. The data members of an account are: the `name` of the owner or client; an `id` number; and the `balance` in the account.

C++ does not have a type (like `Currency`) that is really suitable for financial work.[22] Curiously, the Boost library developers have never accepted a currency class, although examples exist[23] that could be raised to the standard of a Boost class fairly easily.

never use a float variable for financial applications!

Accountants do not like `double` because rounding prevents precise balancing of accounts. Integers used to represent cents are quite good, provided that input and output functions make appropriate conversions between dollars and cents. 32-bit integers permit values between $\pm\$21,474,836.47$, which is not enough for a bank president's annual income. 64-bit integers are necessary for realistic financial calculations, as the following table shows (but remember that the concrete maximum values are platform-dependent).

[22] Surprisingly few languages do have such a type. `Java` provides `BigDecimal`.
[23] See, for example, http://www.colosseumbuilders.com/sourcecode.htm.

Type	Maximum signed value ($)
`int`	327.67
`long`	21,474,836.47
`long long`	92,233,720,368,547,758.07

long long

New `long long` type in C++11

The integer type `long long` has been standard in C since 1999, to support 64-bit integers. It has finally been approved for C++11 (together with `unsigned long long`), but some compilers allowed it for a few years now, including GCC, even though it was not officially part of the language. If you are interested in the technical details of adding a type to a standardized language, see http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2005/n1811.pdf.

We avoid the issue of a currency class in our example by using `long`.

There are other ways, too, in which the account class given here is unrealistic and nothing like a class that would be used in a banking application, for example. Nevertheless, the example provides some useful insights into class design.

6.1 Constructors

All classes have constructors. If you do not provide a constructor, the compiler generates a default constructor that allocates memory for the object but doesn't do anything else. Except in rare cases, for very simple classes, it is best to provide one or more constructors. If you provide any constructor at all, default or otherwise, the compiler does ***not*** generate a default constructor.

A ***default constructor*** is a constructor without any parameters. There are a number of situations in which a default constructor is required; for example, an array declaration is allowed only for types that have a default constructor.

> ***Define a default constructor for every class.***

calling constructors

167

Warning. There is a small but important inconsistency in C++ notation. Suppose that `C` is a class with two constructors, one of which is a default constructor. Now consider these statements:

```
C c1(45);
C c2();
```

It is natural to read the statements like this:

- `c1` is an instance of `C` constructed from an integer;
- `c2` is an instance of `C` constructed using the default constructor.

Unfortunately, this interpretation is wrong! In fact, `c2` has been declared (but not defined) as a function that takes no arguments and returns a `C`. To invoke the default constructor, you must ***leave out the parentheses***:

```
C c1(45);
C c2;
```

Rational. Class `Rational` requires only one constructor. It is a general purpose constructor with two `long` parameters, corresponding to the numerator and denominator of the fraction, stored as `num` and `den`, respectively. Both parameters have default values (`n=0, d=1`), which implies that this constructor is considered to be a default constructor.

The following declarations are equivalent: in each case, a `Rational` object is constructed with `r.num` = 0 and `r.den` = 1: 168

```
Rational r;
Rational r(0);
Rational r(0, 1);
Rational r = Rational(0);
Rational r = Rational(0, 1);
```

The constructor calls `normalize` to put the fraction in normal form. Note that the default parameter values are ***not*** repeated in the function definition. 169

```
Rational::Rational(long n, long d)
    : num(n), den(d)
{
    normalize();
}
```

The data members are set using initializers rather than assignments. This is the best way to initialize data members and you should use it whenever possible.

constructor initializers

Use initializers in constructors.

The private member function `normalize` (see Figure 60 on the next page) is called whenever a new fraction is created. If `den` = 0, it throws the exception `ZERO_DENOMINATOR`, which is one of the values of the enumeration `Exceptions` declared in the class. If `num` = 0, it sets `den` = 1, to ensure that the fraction zero has the unique representation (0, 1). If `den` < 0, it changes the sign of both numerator and denominator. Finally, it divides both by their greatest common divisor to cancel common factors. 170 171 throw

Providing default values for the parameters of the constructor has an important consequence. If N is an expression of type `long`, the `Rational(`N`)` is the rational number $(N, 1)$. The C++ compiler interprets this as permission to silently convert `long` to `Rational` whenever such a

default values in constructors and automatic type conversions

```
long gcd(long i, long j)
{
    assert(i > 0 && j > 0 && "Error in gcd arguments");
    while (true)
    {
        long tmp = j % i;
        if (tmp == 0)
            return i;
        j = i;
        i = tmp;
    }
    assert(false && "Logical error in gcd");
}

void Rational::normalize()
{
    if (den == 0)
        throw Rational::ZERO_DENOMINATOR;
    if (num == 0)
    {
        den = 1;
        return;
    }
    if (den < 0)
    {
        num = -num;
        den = -den;
    }
    assert(num != 0 && den > 0 && "Logical error in normalize");
    long g = gcd(abs(num), den);
    num /= g;
    den /= g;
}
```

Figure 60: Functions to normalize instances of class `Rational`

conversion is required. Since other integral types, such as `char` and `int`, may also be implicitly converted to `long`, these types may be implicitly converted to `Rational`.

172 This is convenient, because it allows us to use "mixed-mode" arithmetic. For example:

```
Rational r(1, 2);
r += 2;
cout << r << endl;       // Writes "5/2"
cout << 1/r << endl;     // Writes "2/5"
```

Automatic conversion also leads to less expected results:

```
Rational r;
r = 'a';
cout << r << endl; // Writes "97"
```

Account. The normal constructor for an `Account` is passed the name, identification code, and opening balance for the new account. If the opening balance is omitted, the balance is set to zero.

The name is passed as a `char*`. Simply copying the pointer would be a serious mistake. As one of the many ways in which things could go wrong, consider Figure 61. Since the bad guys use buffer overflow as the basis of many attacks, it is always a mistake to read characters into a character array. To make it hard for them, we can use a large buffer (which is not a properly secured solution either). The real problem with this function, however, is that the buffer is destroyed when the function returns.

173

```
Account * createAccount()
{
    char buffer[10000];
    cout << "Please enter your name: ";
    cin >> buffer;
    int id;
    cout << "Please enter your account ID: ";
    cin >> id;
    return new Account(buffer, id);
}
```

Figure 61: Pointer problems

It would be unreasonable to require the user of the `Account` class to take care of this problem. We would have to provide an arcane restriction along the lines of "the name of the client passed to the constructors must have a longer lifetime than the `Account` object". Instead, we write a constructor that makes its copy of the name, allocates memory for it, and stores it. It turns out that storing the name is something that we will have to do several times. Consequently, it makes sense to put it into a function, called `storeName` here. Here is the constructor:

174

```
Account::Account(char *name, long id, long balance)
    : id(id), balance(balance)
{
    storeName(name);
}
```

Space for the name is allocated dynamically (i.e., using `new`, with space taken from the heap) and we must remember to allocate one character position for the terminator, `'\0'`. (We are using the old-style functions `strlen` and `strcpy`. As mentioned above, these problems do not arise with the more modern class `string`.)

```
void Account::storeName(char *s)
```

```
{
    name = new char[strlen(s) + 1];
    strcpy(name, s);
}
```

The constructor above is ***not*** a default constructor because the parameters do not have default values. We could provide default values or we could define another constructor for use as the
175 default, like this:

```
Account::Account()
    : id(0), balance(0)
{
    storeName("");
}
```

copy constructor

The ***copy constructor*** is an important component of every C++ class. It is used whenever an object has to be copied. Common uses of the copy constructor include:

- Initialized declarations of the form

  ```
  Account anne = bill;
  Account anne(bill);
  ```

- Passing an object by value.
- Returning an object by value.

If we do not define a copy constructor, the compiler generates one for us. This ***default copy constructor*** simply copies the values of the data members of the object. The default copy constructor for `Rational` does exactly what we want and there was no need to define our own version. Class `Account` is different.

Class `Account` has the data member `name`, which is a pointer. The default copy constructor just copies the pointer, not the object it points to. This would be a disaster for accounts, because we would end up with more than one pointer pointing to the same name. If one account is deleted, the others would be left with ***dangling pointers***. (Technically, the copy constructor performs a ***shallow copy*** but what we need is a ***deep copy***.)

The copy constructor must therefore behave like the constructor, making a new copy of the name. Fortunately, we have a function `storeName` that does exactly the right thing.

The signature (or prototype) of the copy constructor for a general class `T` is `T::T(const T &)`. For our class, `Account`, the implementation looks like this:

```
Account::Account(const Account & other)
      : id(other.id), balance(other.balance)
{
    storeName(other.name);
}
```

> ***Define a copy constructor for any class that has pointer members.***

There is an alternative way of defining a copy constructor that is sometimes useful:

- Declare the copy constructor `T(const T &)` in the `private` part of the class declaration.
- Do ***not*** provide a definition of the copy constructor.

preventing copying an object

The effect is to prevent copying of the object. By declaring the copy constructor, we prevent the compiler from generating it. By making the copy constructor private, we prevent outsiders from calling it. By not defining the copy constructor, we prevent member functions from using it. The result is that any initialized declaration, passing by value, or returning by value, will be flagged as an error by the compiler.

It is quite possible that a real-life banking application might choose to prevent copying of accounts. This is the mechanism that could be used.

Note that preventing copying does not make `Account` a Singleton; we can have as many accounts as we need, but we cannot make copies of them.

6.2 Destructors

A ***destructor*** is a member function that is called when an object is to be deallocated or "destroyed". The compiler provides a default destructor if you don't. The default destructor releases the memory held by the object and does nothing else. There are two circumstances in which you ***must*** define a destructor:

1. The class has members that are pointers.
2. The class will be used as a base class.

We will discuss the second aspect in detail in Lecture 8.

Rational. Class `Rational` does not have any pointer members and is not intended to be used as a base class. Consequently, we do not define a destructor for it.

Account. We must define a destructor for class `Account` because it has a pointer member. Later, we will discuss using `Account` as a base class, which provides another reason for having a destructor.

> ***Define a destructor for any class that has pointer members.***

`delete x` vs. `delete [] x`

The destructor must destroy any data that was created dynamically (using `new`) by the constructor. Destruction is performed by `delete`, which has two forms:

- `delete x` for simple objects
- `delete [] x` for arrays

The distinction between the two kinds of destruction is very important, because the compiler may not detect the error if you are wrong. In `Account`, we must use `delete [] name`; if we wrote `delete name` instead, only the first character of the name would be deallocated.

Use* `delete` *for simple variables and* `delete[]` *for arrays.

176 Here is the destructor for class `Account`:

```
Account::~Account()
{
    delete [] name;
}
```

6.3 Operators

C++ allows most operators to be overloaded. This is an extremely useful feature of the language, but it should not be abused. If operators are defined, they should be defined consistently and they should behave in a reasonable way.

Rational. Class `Rational` is an obvious candidate for overloaded operators, because fractions are numbers and users will expect to use arithmetic operations with fractions. To respect C++ conventions, if you provide `+`, you should also provide `+=`, and similarly for the other operators. It turns out to be easier, and more efficient, to define the assignment operators (`+=`, `-=`, `*=`, `/=`) as member functions and then to use them in the definitions of the simple operators (`+`, `-`, `*`,
177 `/`).

178 Figure 62 on the next page shows the implementation of the assignment operators. Each one normalizes its result and returns a reference to the new fraction. Returning a reference avoids making an unnecessary copy of the object. This is another convention that you should follow; it allows users to write statements such as

```
p += q *= r;
```

assuming that they can figure out what such expressions mean.

But why did we declare the return type `const` for the operators, as in

```
const Rational & Rational::operator+= (const Rational & right)
const Rational operator+ (const Rational & left, const Rational & right);
```

If the return types weren't `const`, a user of our `Rational` class would be allowed to write code like

```
(r + s) = t;
```

which is clearly nonsense. To prevent mistakes as in `if(r+s = t)` (instead of `==`), we define the return types of all our operators as `const`. As Meyers recommends (Meyers 2005, Item 3):

Use* `const` *whenever possible.

The keyword `this` can be used only in a member function (i.e., method of a class). It is a constant pointer to the current object. To obtain the whole object, we dereference `this` with the `*` operator. The functions in Figure 62 use this trick to return the current object as `*this`.

this

```
const Rational & Rational::operator+= (const Rational & right)
{
    num = num * right.den + den * right.num;
    den *= right.den;
    normalize();
    return *this;
}

const Rational & Rational::operator-= (const Rational & right)
{
    num = num * right.den - den * right.num;
    den *= right.den;
    normalize();
    return *this;
}

const Rational & Rational::operator*= (const Rational & right)
{
    num *= right.num;
    den *= right.den;
    normalize();
    return *this;
}

const Rational & Rational::operator/= (const Rational & right)
{
    num *= right.den;
    den *= right.num;
    normalize();
    return *this;
}
```

Figure 62: Arithmetic assignment operators for class `Rational`

Account. Since adding and subtracting accounts does not make much sense, we do not provide operators for class `Account`.

6.3.1 Assignment (operator=)

By default, the compiler will generate a default assignment operator (operator=). The effect of this operator will be to copy all of the fields of the object. If this is what you want, you do not need to define your own version of operator=.

Rational. The default assignment operator is just what we need for Rational objects: the statement r = s will copy the numerator and denominator of s to r. Consequently, we do not define operator=.

Account. Since Account has a pointer member, we ***must*** define an assignment operator for it. Not doing so will lead to the same problems as not having a copy constructor: we will end up with many accounts all pointing to the same name. The assignment operator looks rather like the copy constructor, but there are two important differences. Consider the ***incorrect*** version
179 of the assignment operator shown in Figure 63.

```
Account & Account::operator=(const Account & other)
{
    storeName(other.name);
    id = other.id;
    balance = other.balance;
    return *this;
}
```

Figure 63: An ***incorrect*** implementation of operator=

Note first that the type of the assignment operator is Account& and that the function returns *this. This is conventional for assignment operators and it allows statements such as

```
a1 = a2 = a3 = a4 = a5;
```

for programmers who want to write such statements.

memory leaks

The definition of Account::operator= above has a memory leak. When storeName is called, the old name of the account is lost.

There is another problem. Suppose that the programmer writes

```
a = a;
```

and think about the effect in Account::operator= as defined above. storeName allocates space for the name, and copies the name into. The old name is lost – another memory leak! If we insert the statement delete [] name to avoid the memory leak, things get even worse: storeName would attempt to copy the deleted name! The solution is to check for ***self-assignment***. These considerations lead to the correct assignment operator shown in Figure 64 on the facing page.

self-assignment

180

You might ask: "What programmer could be so stupid as to write a = a?" The answer is that a programmer might not write this assignment as such, but it could easily be generated by template expansion. Also, a programmer might quite reasonably write

```
Account & Account::operator= (const Account & other)
{
    if (this == &other)
        return *this;
    delete [] name;
    storeName(other.name);
    id = other.id;
    balance = other.balance;
    return *this;
}
```

Figure 64: Assignment operator for class `Account`

```
a[i] = a[j];
```

and not feel it necessary to check `i != j`.

Define an assignment operator for any class that has pointer members. The assignment operator must check for self-assignment and return a reference to `*this`.

If you want to prevent assignment of instances of a class, declare `operator=` as a `private` member function and do not provide an implementation for it.

6.3.2 Arithmetic

If it makes sense to perform arithmetic operations on instances of the class, we can provide the appropriate operators. These are usually implemented in conjunction with the corresponding assignment operators, described above.

In general, you should implement the arithmetic functions so that they obey standard laws of algebra. For example, `-` is the inverse of `+`, `/` is the inverse of `*`, and so on.

There are occasional exceptions to this rule. For example, `string` uses `+` for concatenation, even though concatenation is neither commutative nor associative, and has no inverse. But people seem to accept `+` as a concatenation operator, so this is perhaps excusable.

implementing operators: impacts of free vs. member functions

Arithmetic operators, such as `operator+`, can be implemented as member functions with one parameter or as free functions with two parameters. The form of declaration for a member function is:

```
Rational operator+(const Rational & rhs);
```

The left operand is the current object (`*this`) and the right operand is the value passed to the parameter `rhs`.

If `operator+` is implemented as a member function, then 181

```
x + y
```

is effectively translated as

```
x.operator+(y)
```

The compiler will use the static type of `x` to choose the appropriate overload of `operator+` and may convert `y` to match the type of `x`. Suppose that we implemented `+` for class `Rational` in this way and that `i` is an `int` and `r` is a `Rational`. Then `r+i` converts `i` to `Rational` and adds the resulting fractions but `i+r` does not compile because the integer version of `operator+` cannot accept a `Rational` right argument. In other words, the advantage of defining comparison operators as free functions is that the order of operands does not matter.

There is another issue to consider when we choose to implement free functions associated with a class: do we provide access functions for data members of the class, or do we declare the free functions as `friend`s? The choice depends very much on the particular application. For example, if the class already provides accessor functions for some reason, the free functions can make use of them. If security is important, and data members should not be exposed, then `friend` functions may be a better choice.

182

183 **Rational.** Figure 65 on the next page shows the standard arithmetic operators for class `Rational`, implemented as free functions that use the corresponding assignment operators. Even though class `Rational` does not export the numerator and denominator of a fraction, these (free) functions are not declared as `friend`s of the class, because they are implemented using the corresponding assignment member functions (`+=`, `-=`, `*=`, `/=`).

If you give users the arithmetic operators, they will expect unary minus as well. Unary minus is best implemented as a member function with no arguments. It does ***not*** negate the value of the fraction, but instead returns the negated value, while the object remains unchanged.

184 Consequently, we can qualify it with `const`:

```
const Rational Rational::operator- () const
{
    return Rational(-num, den);
}
```

C++ programmers expect to use `pow` for exponentiation. But `pow` takes arguments of many types, even for the exponent. For fractions, we take the slightly daring approach of providing `^` as an exponential operator. The exponent must be an integer, but the integer may be positive or negative: if e is negative, then r^e is evaluated as $\left(\frac{1}{r}\right)^{-e}$.

overloading operator^ and precedence

If $r = 0$ and $e = 0$, then $r^e = 1$. However, if $r = 0$ and $e < 0$, then r^e evaluates $\frac{1}{r}$, which throws an exception.

The function uses the identity $x^{2e} = (x^2)^e$ when e is even to achieve complexity $\mathcal{O}(\log e)$ rather than $\mathcal{O}(e)$. Since exponentiation is not commutative, and the left operand must be a `Rational`,

185 we implement `operator^` as a member function, as shown in Figure 66.

There is a minor problem with using `operator^` as an exponent operator. Although we can overload operators in C++, we cannot change their precedence. As it happens, `operator^`, which is normally used as exclusive-or on bit strings, has a ***lower*** precedence than the other arithmetic operators. Its precedence is even lower than that of `operator<<` and `operator>>`. So we had better warn our users to put exponential expressions in parentheses!

```
const Rational operator+ (const Rational & left, const Rational & right)
{
    Rational result = left;
    result += right;
    return result;
}

const Rational operator- (const Rational & left, const Rational & right)
{
    Rational result = left;
    result -= right;
    return result;
}

const Rational operator* (const Rational & left, const Rational & right)
{
    Rational result = left;
    result *= right;
    return result;
}

const Rational operator/ (const Rational & left, const Rational & right)
{
    Rational result = left;
    result /= right;
    return result;
}
```

Figure 65: Arithmetic operators for class `Rational`, implemented using the arithmetic assignments of Figure 62 on page 109

```
const Rational Rational::operator^ (int e) const
{
    if (e < 0)
        return (1/ *this)^(-e);
    else if (e == 0)
        return 1;
    else if (e % 2 == 0)
        return (*this * *this)^(e/2);
    else return *this * (*this^(e - 1));
}
```

Figure 66: An exponent function for class `Rational`

Account. Class `Account` does not provide any operators.

6.3.3 Comparison

There are two important kinds of comparison: ***equality*** and ***ordering***. Identity comparison corresponds to the operators `==` and `!=` and is useful for many kinds of objects. Ordering corresponds to the operator `<` and its friends and is useful only for objects for which some kind of ordering makes sense.

By convention, the ordering operators return a Boolean value `true` or `false`. They can implement only a ***total ordering*** in which, for any objects x and y, one of the following must be true: $x < y$, $x = y$, or $x > y$. They cannot be used to implement a partial ordering, in which two objects may be unrelated.

Rational. Fractions are totally ordered. We can define the ordering by

$$\frac{n_1}{d_1} > \frac{n_2}{d_2} \iff n_1 \times d_2 > n_2 \times d_1.$$

186 187 Figure 67 on the facing page shows the corresponding functions for class `Rational`. They are implemented as free, `friend` functions.

Account. Considering the comparison operators for class `Account` raises interesting questions about equality.

- Does it make sense to say that two accounts are "equal"?
- If it does make sense, what does it mean to say "account A equals account B"?

Comparing balances probably is not very helpful. Comparing names is unsafe, because two people might have the same name.[24] There are two comparisons that might be reasonable:

extensional vs. intensional equality

- Two accounts are equal if they have the same ID (***extensional equality***)
- Two accounts are equal if they are the same object (***intensional equality*** or ***identity***)

The following comparison functions check for identity (accounts are equal only if they are the same object). Note that this is not really realistic because, in a practical application, account objects would spend most of their lives on disks and would only occasionally be brought into memory. 188

```
bool operator==(const Account & left, const Account & right)
{
    return &left == &right;
}

bool operator!=(const Account & left, const Account & right)
{
    return &left != &right;
}
```

[24] This is, in fact, the cause of many "mistaken identity" problems.

```
const bool operator== (const Rational & left, const Rational & right)
{
    return left.num * right.den == right.num * left.den;
}

const bool operator!= (const Rational & left, const Rational & right)
{
    return !(left == right);
}

const bool operator<  (const Rational & left, const Rational & right)
{
    return left.num * right.den < right.num * left.den;
}

const bool operator>  (const Rational & left, const Rational & right)
{
    return left.num * right.den > right.num * left.den;
}

const bool operator<= (const Rational & left, const Rational & right)
{
    return left < right || left == right;
}

const bool operator>= (const Rational & left, const Rational & right)
{
    return left > right || left == right;
}
```

Figure 67: Comparison operators for class `Rational`

It would be straightforward to compare IDs instead.

Accounts could be ordered by name or by ID; this would allow us to sort a `vector` of accounts, for example. These operators are easy to add if needed.

6.3.4 Input and output

For many classes, it is useful to provide the insertion operator (`<<`), if only for debugging purposes. The extraction operator, `>>`, is less often needed, but can be provided as well. When these operators are provided, they are commonly implemented as `friend`s.

Rational. For fractions, both input and output operators make sense, as we implement them as friends. Both present complications.

For input, the first decision we have to make is: what should the user enter? Let's say that the
189 user must enter two integers separated by a slash:

```
Enter a rational:  2/3
```

The next decision is: how does the program respond if the user does ***not*** enter the data in the form that we expect? There are many possible solutions, ranging from accepting only input that is exactly correct to parsing what the user enters and figuring out what was meant.

In the following function, we follow a middle way: we read an integer, a character, and another integer, and we throw an exception if the character is not `'/'`. Although rationals can be constructed from integers, the user is required to enter a complete fraction, even if it is `3/1`.

```
istream & operator>> (istream & is, Rational & r)
{
    long m, n;
    char c;
    is >> m >> c >> n;
    if (c != '/')
        throw Rational::BAD_INPUT;
    r = Rational(m, n);
    return is;
}
```

Output is usually more straightforward than input, but there is one problem. Suppose that we
190 implement the inserter in the "obvious" way

```
os << num << '/' << den;
```

and the user writes

```
cout << setw(12) << r;
```

in which `r` is a `Rational`. Our function will use 12 columns to write the numerator and then will write the slash and the denominator using the default field width of zero – probably not what the user expected!

Of the various ways to correct this error, the simplest is to format the entire fraction, in the obvious way, into a buffer, and then to insert the buffer into the output stream. This ensures that any width modifiers will be applied to the complete fraction, not just to the numerator.

The remaining issue is how to write whole numbers (fractions with denominator 1). To avoid sillinesses like `3/1`, we will not write the slash or the denominator for whole numbers. This reasoning leads to the following function.

```
ostream & operator<< (ostream & os, const Rational & r)
{
    ostringstream buffer;
    buffer << r.num;
    if (r.den != 1)
        buffer << '/' << r.den;
    return os << buffer.str();
}
```

String streams can be used for either output (`ostringstream`) or input (`istringstream`) and require the directive

string stream

```
#include <sstream>
```

- An output string stream behaves like any other output stream (e.g., `cout`).
- After writing things to it, you can extract the string of formatted data using the function `str`, which returns a `string`.
- After `str` has been invoked on an `ostringstream`, the stream is ***frozen*** and does not permit further write operations.
- When the string stream is deleted (usually at the end of the current scope), data associated with it is deleted.

An input string stream can be used to parse a string. In the following code, the input string stream `isstr` is initialized with `myString`. The `isstr` is used, like any other input stream (e.g., `cin`), with extract operators.

191

```
string myString = "3 4 5 words";
istringstream isstr(myString);
int a, b, c;
string w;
isstr >> a >> b >> c >> w;
```

Hence, string streams in C++ perform a similar function to the `StringTokenizer` class in Java – see (Weiss 2004, Section 9.7) for a comparison.

6.4 Conversions

Conversions are a delicate issue in C++. Programmers don't like writing explicit conversions and want the compiler to do the dirty work for them. Too many conversions, however, can be a bad thing.

Rational. It seems reasonable to convert rationals to floating-point numbers. Sometimes, for example, we might want 0.66667 rather than 2/3. C++ provides a powerful way of implementing such conversions, using operator notation. We can define a conversion from `Rational` to `double` 192 by adding this member function to class `Rational`:

```
operator double () const
{
    return double(num) / double(den);
}
```

implicit type conversions

This function provides an ***implicit conversion*** from `Rational` to `double`: in any context where the compiler expects to find an expression of type `double`, but in fact finds an expression of type `Rational`, it will insert a call to this function to perform the conversion.

Similarly, we could provide implicit conversion from `Rational` to `bool`, so that we could write

```
if (r) ...
```

as an abbreviation for

```
if (r != 0) ...
```

This conversion would be written

```
operator bool () const
{
    return num != 0;
}
```

ambiguity problems with implicit conversions

Unfortunately, both of these functions are a bit ***too*** effective! If we write `1/r`, for example, the compiler complains about ambiguity: Given `1/r`, the compiler can either convert `1` to `Rational` and evaluate the reciprocal of `r` as a `Rational` ***or*** it can convert both `1` and `r` to `double` and evaluate the reciprocal of `r` as a `double`. Since we would like to keep the convenience of being able to write `1/r` for the `Rational` reciprocal of a `Rational`, we choose ***not*** to provide `operator double`. A similar argument applies to `operator bool`.

It is not hard to find a better alternative: we simply provide the same function with a funny name to prevent the compiler from using it implicitly. The following function does what we need 193 and will be invoked only when the user explictly requests it by writing `r.toDouble()`.

```
const double Rational::toDouble() const
{
    return double(num) / double(den);
}
```

6.5 Accessors

An ***accessor*** or ***inspector*** is a member function that returns information about an object without changing the object. It follows immediately from this definition that accessors should always return a non-`void` value and should be declared `const`.

Rational. The only candidates for accessors for class `Rational` are `getNum` and `getDen`, implemented in the obvious way. The decision ***not*** to provide these was that the representation of a rational number is a "secret" and providing these accessors would give away part of the secret.

If the class did provide `getNum` and `getDen`, the `friend` functions would no longer need to be declared as friends, because they could use the accessors instead.

Account. There are several candidates for accessors for class `Account`. We provide just an accessor for the account balance, to demonstrate the general idea. 194

```
long Account::getBalance() const
{
    return balance;
}
```

6.6 Mutators

A ***mutator*** is a member function that changes the state of the object. Usually, the return type of a mutator is `void` (otherwise the mutator would be a function with a side-effect).

Rational. Any function that changes the state of a rational should do so in a way that makes semantic sense. Thus `+=` is acceptable, because it changes the state by adding another rational. Arbitrary mutations of the numerator and denominator do not make semantic sense, and so we do not provide mutators for class `Rational`.

Account. There are several ways in which the state of an account may reasonably be changed. We provide three "obvious" functions: `deposit`, `withdraw`, and `transfer`. Of these, `deposit` is straightforward. 195

```
void Account::deposit(long amount)
{
    balance += amount;
}
```

For withdrawals, we have to decide what to do when the balance would go negative. The solution adopted here is to throw an exception. Since we do not know who will be handling the exception, we must ensure that sufficient information is provided. To achieve this, we declare a special exception class (described below) and store the account ID and a helpful message in the exception object.

```
void Account::withdraw(long amount)
{
    if (amount > balance)
        throw AccountException(id,
              "Withdrawal: amount greater than balance");
    else
        balance -= amount;
}
```

what()

Exceptions. Standard exception classes provide a function `what` that returns a description of the exception. Although we could inherit from one of the standard exception classes, we
196 choose not to do so here, but we provide `what` anyway. Figure 68 shows the declaration and
197 implementation of the class `AccountException`.

The third mutator for class `Account` is `transfer`, which transfers money from one account to another. We allow an account to transfer funds ***to*** another account but not to transfer funds ***from*** another account. This function will throw an exception if the transfer would leave the
198 giving account with a negative balance.

```
void Account::transfer(Account & other, long amount)
{
    withdraw(amount);
    other.deposit(amount);
}
```

```
class AccountException
{
public:
    AccountException(long id, std::string reason);
    std::string what();
private:
    long id;
    std::string reason;
};


AccountException::AccountException(long id, string reason)
    : id(id), reason(reason)
{}

string AccountException::what()
{
    ostringstream ostr;
    ostr << "Account " << id << ".  " << reason << '.';
    return ostr.str();
}
```

Figure 68: Declaration and implementation of class `AccountException`

The function `transfer` is implemented using the previously defined functions `withdraw` and `deposit`. It is always a good idea to use existing code when it applies, rather than introducing more code, with possible errors.

In real-life banking, an operation such as `transfer` must be implemented very carefully: either the transaction must succeed completely, or it must fail completely and have no effect. (Database gurus are familiar with this problem of ***transactional integrity***.) Without pretending that `transfer` is really a secure function, we note that there is only one (obvious) way that it can fail – `withdraw` may throw an exception – and that this failure leaves both accounts unchanged.

6.7 Odds and ends

6.7.1 Indexing (`operator[]`)

C++ allows us to overload the "operator" `[]`. The overload has one parameter, which can be of any type, and returns a value, which can be of any type. The syntax for calling the function is `a[i]`, in which `a` is an object and `i` is the argument passed to the function. Typically, we would use `operator[]` for a class that represented an array, vector, or similar kind of object. Figure 69 shows an inefficient and incomplete map class that associates names with telephone numbers, both represented as strings. The store is accessed by calling `operator[]` with a string argument. It makes use of the `pair` template defined in `<utility>`. The following code illustrates the use of this store (note the final statement). 199

```
Store myPhoneBook;
myPhoneBook.insert("Abe", "486-2849");
myPhoneBook.insert("Bo", "982-3847");
cout << myPhoneBook["Abe"];
```

200

In this example, `operator[]` has one parameter. This is the only possibility: you cannot declare `operator[]` with no parameters or with more than one parameter.

6.7.2 Calling (`operator()`)

C++ allows us to overload the "operator" `()`. The overload may have several parameters of any type, and returns a value, which can also be of any type. The syntax is `f(x, y, ...)`, in which `f` is an object and `x, y, ...` are the arguments passed to the function. Typically, we would use `operator()` in a situation where it makes sense to treat an instance as a function. Such objects are also known as ***function objects*** (or functors). function objects (functors)

Figure 70 on the following page shows a program that tests a rather simple class called `AddressBook`. The object `myFriends` is an `AddressBook` into which a few names have been entered. The output statement uses this object as if it was a function, providing two names as arguments. The effect, shown on the right, is to display the names within the given range. 201

The function that makes an instance of `AddressBook` behave like a function is `operator()` in the declaration shown in Figure 71 on page 123. In this example, the function takes two arguments of type `string` and returns a value of type `AddressBook`. 202 203 204 205 206

```
class Store
{
public:
    void insert(string key, string value)
    {
        data.push_back(pair<string, string>(key, value));
    }
    string operator[](string key)
    {
        for (   vector<pair<string, string> >::const_iterator it =
                                                  data.begin();
                it != data.end();
                ++it )
            if (it->first == key)
                return it->second;
        return "";
    }
private:
    vector<pair<string, string> > data;
};
```

Figure 69: A simple map class

```
int main()
{
   AddressBook myFriends;
   myFriends.add("Anne");
   myFriends.add("Bill");
   myFriends.add("Chun");
   myFriends.add("Dina");
   myFriends.add("Eddy");
   myFriends.add("Fred");
   myFriends.add("Geof");
   cout << myFriends("Bill", "Fred");
   return 0;
}
```

```
Bill
Chun
Dina
Eddy
```

Figure 70: Test program for class `AddressBook` (left) and results (right)

6.7.3 Explicit constructors

A constructor with a single parameter provides a form of type conversion. For example, suppose
207 class `Widget` has a constructor with a parameter of type `int`:

```
class Widget
{
public:
```

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class AddressBook
{
public:
   void add(string name)
   {
      names.push_back(name);
   }

   AddressBook operator()(string first, string last)
   {
      AddressBook result;
      bool ins = false;
      for (
         vector<string>::const_iterator it = names.begin();
         it != names.end();
         ++it )
      {
         if (*it == first)
            ins = true;
         if (*it == last)
            ins = false;
         if (ins)
            result.add(*it);
      }
      return result;
   }

   friend ostream & operator<<(ostream & os, const AddressBook & st)
   {
      for (
            vector<string>::const_iterator it = st.names.begin();
            it != st.names.end();
            ++it )
         os << *it << endl;
      return os;
   }

private:
   vector<string> names;
};
```

Figure 71: Class `AddressBook`

```
    Widget(int n);
    ....
```

This constructor gives the compiler permission to convert an integer to a `Widget` whenever it has an opportunity to do so. This behaviour might be appropriate, but it might also be an error.
208 For example, class `string` used to have a constructor with an integer argument[25] so that

```
string s(N);
```

could construct a string with N characters. A possible consequence was that

```
string s = 'a';
```

would construct a string with 97 characters – probably not what the programmer intended.

explicit
209 This kind of implicit conversion can be prevented by qualifying the constructor with `explicit`. To prevent `Widget`s being constructed from integers, rewrite the example above as

```
class Widget
{
public:
    explicit Widget(int n);
    ....
```

The keyword `explicit` does not prevent the constructor from being used at all, of course. It says that, in order to use this constructor, the call `Widget(N)` must actually appear in the program; the compiler will never call the constructor implicitly.

In C++98, the keyword `explicit` can be used with constructors only; it cannot be used with conversion operators such as `operator double` described in Section 6.4 on page 117.

`explicit` conversion operators in C++11

In C++11, you can now also use `explicit` with a conversion operator, to prevent implicit type conversions as discussed in Section 6.4 on page 117. For example, you can now declare

```
explicit operator double () const
```

and use it by explicitly calling

```
double x = double(r)
```

6.7.4 Friends

A function associated with a class can have three properties (Stroustrup 1997, page 278): 210

1. it can access private members of the class
2. it is in the scope of the class
3. it must be invoked on an object

Property 2 means that the function `f` associated with class `C` must be invoked in one of the following ways:

- `C::f()`
- `c.f()` where `c` is an instance of `C`
- `pc->f()` where `pc` is a pointer to an instance of `C`

Property 3 means that statements in the function can use the `this` pointer to the object for which the function is invoked.

The following table shows which kinds of function have these properties: 211

	Property		
Function kind	1	2	3
`friend`	√		
`static`	√	√	
member	√	√	√

The table shows that the three kinds of function form a hierarchy, with member functions having the most, and friend functions the least, access to the class.

Contrary to popular belief (especially amongst Java programmers), friends do not violate encapsulation in C++. The important point is that friends ***must be declared within the class***. This means that a class has complete control over its friends – enemies cannot use friends to subvert the class' protection mechanisms. Here is Stroustrup's opinion of friends (Stroustrup 1994, page 53): 212

> *A friendship declaration was seen as a mechanism similar to that of one protection domain granting a read-write capability to another. It is an explicit and specific part of a class declaration. Consequently, I have never been able to see the recurring assertions that a* `friend` *declaration "violates encapsulation" as anything but a combination of ignorance and confusion with non-C++ terminology.*

Here are some useful things to know about friends: 213

- A `friend` declaration can be placed anywhere in a class declaration. It is not affected by `public` or `private` attributes.
- A function or a class can be declared as a `friend`. In either case, friends of a class can access private members (data and functions) of the class.

[25] This constructor was eventually removed to avoid confusions like the one described here.

- Two or more classes can declare the same class or function as a friend.

Used correctly, friends actually provide ***better*** protection than other techniques. Here is an example (Stroustrup 1997, page 278): we have classes `Vector` and `Matrix` and we want to define functions that, for example, multiply a matrix by a vector. We could define the multiply function as a free function, external to both classes, but then we would have to expose the representations
214 of both `Vector` and `Matrix` for this function to use. Instead, we use friends:

```
class Vector
{
    friend Vector operator* (const Matrix & m,
                             const Vector & v);
    ....
}

class Matrix
{
    friend Vector operator* (const Matrix & m,
                             const Vector & v);
    ....
}

Vector operator* (const Matrix & m,
                  const Vector & v)
{
    ....
}
```

This provides a neat solution to the problem: the multiply function has access to private data in ***both*** classes but we have not "opened them up" to anyone else.

Another design consideration for choosing between a member function or a friend is conversion, as discussed in Section 6.4: in the call of the free (friend) function `f(x,y)`, the compiler may perform conversions to match the types of both the arguments `x` and `y` to the parameter types. The call to the member function `x.f(y)` can only invoke `X::f` where `x` is the class of `X` (although `y` may still be converted).

6.8 Summary

Here is a summary of functions and operators that might be included in a class declaration. We use `T` to stand for the name of the class being declared.

`T()` ***Default constructor.*** The compiler generates a default constructor, but only if the programmer declares ***no*** constructors. The compiler-generated default constructor calls the default constructors of the instance variables of the class.

The programmer may declare a default constructor. A default constructor does not need any parameters. Any parameters that it does have must have default values.

`T(...)` ***Constructor.*** Programmer may provide as many constructors as needed. The parameters of constructors must differ in number and type in accordance with normal overloading rules.

`explicit T(...)` ***Explicit constructor.*** If a constructor has a single parameter, the compiler is allowed to use it to perform conversions. For example, if class `T` has a constructor `T(int n)`, the following code implicitly invokes this constructor as `T(42)`:

```
T x;
x = 42;
```

Sometimes this behaviour is undesirable. To prevent implicit conversion, qualify the constructor with `explicit`. An `explicit` constructor is called only if its name appears in the source code. If `T(int)` was declared `explicit`, the code above would not compile; it would have to be rewritten as:

```
T x;
x = T(42);
```

We could also write just

```
T x(42);
```

because the compiler considers this to be an explicit call of the constructor.

`T(const T & x)` ***Copy constructor.*** The compiler generates a default copy constructor if the programmer does not declare a copy constructor. The default copy constructor performs a bitwise copy of each instance variable of the object. This is usually appropriate for value fields but incorrect for pointer fields.

The programmer may declare a copy constructor with the signature given above. The definition can perform any actions but should normally construct a semantic copy of the argument.

`~T()` ***Destructor.*** The compiler generates a default destructor if the programmer does not declare a destructor. The default destructor deallocates memory used by the object but does nothing else.

The programmer may declare a destructor with the signature given above. The destructor has no parameters and no return type. It may perform any actions but is normally used to destroy objects created dynamically during the object's lifetime.

`T & operator=(const T & x)` ***Assignment operator.*** The compiler generates a default assignment operator that performs a bitwise copy of all of the fields of the argument.

The programmer may declare an assignment operator with the signature given above. It may perform any actions but is normally used to construct a semantic copy of the argument and return a reference to it.

`T operator`⟨***unop***⟩`()` ***Unary operator.*** This is a unary operator. The implicit argument is `*this` and the function applies the unary operation to it and returns the result. The return type may be a reference (i.e., `T &`).

The unary operators that may be overloaded are:

```
!   &   ~   *   +   -   ++   --
```

For `++` and `--`, the compiler must be able to distinguish between prefix usage (e.g., `++n`) and postfix usage (e.g., `n++`). The appropriate declarations for class `T` are shown below. The parameter `int` enables the compiler to distinguish prefix and postfix; it must not be accessed inside the function definition. Note that the prefix form returns an Lvalue but the postfix form returns a Rvalue.

```
T& operator++();       // Prefix increment operator.
T operator++(int);     // Postfix increment operator.
T& operator--();       // Prefix decrement operator.
T operator--(int);     // Postfix decrement operator.
```

Unary operators in class `T` do not have to return a value of type `T`, although they normally do so.

Unary operators may also be declared as free functions, outside a class, with syntax

`T operator`⟨*unop*⟩`(const T & x)`

but then their connection with class `T` is through the parameter types only.

`T operator`⟨***binop***⟩`(const T & x)` ***Binary operator.*** This is a binary operator declared as a member function.

If `operator+`, for example, is defined as a member function, the expression `a+b`, with `a` an instance of `T`, is treated as `a.operator+(b)`. This means that `a+b` and `b+a` are treated differently, because the right operand may be converted but the left operand cannot be converted.

The binary operators that can be overloaded are:

```
,   !=   %   %=   &   &&   &=   *   *=   +   +=
-   -=   ->   ->*   /   /=   <   <<   <<=   <=
=   ==   >   >>   >>=   ^   ^=   |   |=   ||
```

The types in these examples are all the same (`T`), but this is not necessary. Code such as the following is permissible, although somewhat unusual:

```
class Money
{
   ...
   double operator+(int n);
};
```

This declaration introduces a member function that adds a `Money` value to an `int` and returns a `double` value.

Binary operators can also be overloaded in free functions. In this case, they have two parameters, corresponding to the left and right operands. For example, the following declaration permits multiplying a scalar by a vector:

```
Vector operator*(double scal, const Vector & vec);
```

`operator X ()` ***Conversion.*** In this form, `X` is a type not equal to the class type, `T`. The effect of providing a member function with this signature is to provide an automatic conversion from `T` to `X`. In other words, if the compiler is expecting a value of type `X`, but finds a value of type `T`, it will automatically insert a call to this function. A conversion operator declaration must not have parameters (the operand is the object itself).

The last line in the following example implicitly calls `operator Foo` to perform the conversion from `Bar` to `Foo`:

```
class Foo { ... };

class Bar
{
   ...
   operator Foo() { ... };
   ...
};

Bar b;
Foo f = b;
```

`X & operator[](...)` ***Indexing.*** If a class provides `operator[]`, instances of the class can be "indexed", as in `a[i]`.

The return type is usually not the class type. Typically, the class would be a template class representing a collection of some kind, and `X` would be the template parameter. For example:

```
template<typename Vehicle>
class ParkingLot
{
   ...
   Vehicle operator[](int i) const;
   Vehicle & operator[](int i);
   ...
};

class Car { ... };

Car myVeyron;
ParkingLot<Car> mall;
...
mall[76] = myVeyron;
```

It is usually necessary to provide two versions of `operator[]`, as in the example above. The first version returns an Rvalue that can be used with `const` objects but not on the left of `=`. The second version returns an Lvalue that can be used on the left of `=` but cannot be used with `const` objects.

A declaration of `operator[]` must have exactly one parameter. The parameter can have any type, and any type may be returned.

`X operator()(...)` ***Function.*** If a class provides `operator()`, instances of the class can be used as if they were functions, as in `f(x)`.

A declaration of `operator()` can have any number of parameters, including zero, and may return a value of any type.

Friends. Friends of a class are classes and functions that have access to the private data of the class. Their declarations must appear within the class declaration; their definitions may be elsewhere.

References

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.