Abstract Data Types

(from http://www.princeton.edu/ achaney/tmve/wiki100k/docs/Abstraction_%28computer_science%29.html)
In computer science, the mechanism and practice of abstraction reduces and factors out details so that one can focus on a few concepts at a time.
The following English definition of abstraction helps to understand how this term applies to computer science, IT and objects:
From

> Abstraction in mathematics is the process of extracting the underlying essence of a mathematical concept, removing any dependence on real world objects with which it might originally have been connected, and generalizing it so that it has wider applications or matching among other abstract descriptions of equivalent phenomena.[1][2][3] Two of the most highly abstract areas of modern mathematics are category theory and model theory.

The concept originated by analogy with abstraction in mathematics. The mathematical technique of abstraction begins with mathematical definitions; this has the fortunate effect of finessing some of the vexing philosophical issues of abstraction. For example, in both computing and in mathematics, numbers are concepts in the programming languages, as founded in mathematics. Implementation details depend on the hardware and software, but this is not a restriction because the computing concept of number is still based on the mathematical concept.
In computer programming, abstraction can apply to control or to data: Control abstraction is the abstraction of actions while data abstraction is that of data structures.
Control abstraction involves the use of subprograms and related concepts control flows Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind datatype.
One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code.
The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the abstraction principle. The requirement that a programming language provide suitable abstractions is also called *the abstraction principle.*
see more at http://www.d.umn.edu/ tcolburn/papers/Abstraction.pdf

0.1 Procedural Abstraction

In C++ procedural abstractions are called functions. A function encapsulates and names an algorithm to solve a particular problem. That algorithm can be used in a program simply by calling the function by name, generally supplying information and receiving a result. `sqrt`, `pow`, and `setprecision` familiar are built-in functions. We can design and implement our own functions as we need them.
The term *procedural abstractions* is used to describe the use of functions as fundamental actions without regard to the details of their implementation. For example, using a `sort()` function on an array is a procedural abstraction. When we use the function in a program, we assume that the function has already been written and thoroughly tested. It's like putting thumbdrive in a USB port: we know what it's supposed to do, and we assume that it will do that, but we don't need to think about the details of how it works. Instead, our history of using thumbdrives leads us to be confident in its performance and we can focus on the task of transferring files.
Structured programming involves decomposing programs into reliable functions that work like that thumbdrive: they do what is expected without misbehaving.
To structure procedural abstraction, we often separate a function into two parts: its interface and its definition. These are often separated in two different files, or into to different sections of a file. For example in an Ubuntu system (Linux) you might store the interface for a `sort()` function in a file named `sort.h` (a *header file*) and the definition (implementation) may be stored in a file named `sort.cpp` (the *source code file*).

The *interface* is the part the user (called the *client*) sees. It contains all the information that the compiler needs to compile the program that calls the function. That includes the declaration of the function, called the *prototype*, and possibly other definitions and declaration. It also contains the documentation that the client need to use the function.
The *implementation* is the complete definition of the function: the actual algorithm with detailed instructions that the computer performs to take input and produce the output. It is analogous to the inner workings of a thumbdrive. It is essential for the thing to work but irrelevant to the client who uses it. Consequently, the implementation file is often compiled separately and provided to the user only as an *object module* in machine language. This is called *information hiding*. Preventing the the client from viewing the function's implementation ensures that the programs that use the function will be independent of its algorithm. This is advantageous because it guarantees that, like replacing one battery with another, a newer version of the function could replace the current one without having to make any changes in the algorithm. Such modularaization is essential to structured programming.
EXAMPLE 1 Abstracting a `sort()` Function
Here are four files:

`program.cpp`

```
#include "sort.h"

int main() {
   int a, b, c;
   //...
   sort(a,b,c);
   //...
   return 0;
}
```

`sort.h`

```
void sort(int& x, int& y, int& z)
// POSTCONDITION: x <= y <= z
```

`sort.cpp`

```
void swap(int& x, int& y) {
   int t = x;
   x = y;
   y = t;
}

void sort(int& x, int& y, int& z) {
   if(y < x) swap(x, y);
   if(z < y) swap(y, z);
   if(y < x) swap(x, y);
}
```

`test_sort.cpp`

```
#include <iostream>
#include "sort.h"

void print(int x, int y, int z) {
   std::cout << x << ", " << y << ", "
            << z << endl;
}

int main() {
   int a = 77;
   int b = 33;
   int c = 99;
   sort(a,b,c);
   print(a,b,c);
   sort(a,c,b);
   print(a,c,b);
   sort(b,a,c);
   print(b,a,c);
   sort(b,c,a);
   print(b,c,a);
   sort(c,a,b);
   print(c,a,b);
   sort(c,b,a);
   print(c,b,a);
   return 0;
}
```

The program named `program.cpp` contains the client program which calls the `sort()` function. To compile successfully, it must include the precompiler directive `#include "sort.h"`.

The file named `sort.h` contains the function interface. this is the function prototype together with a comment that specifies what the function does.
The function named `sort.cpp` contains the function implementation. This is the complete definition of the function. In this case, that includes an auxiliary `swap()` function.
The function named `test_sort.cpp` contains the function test driver. It is a temporary program whose only purpose is to thoroughly test the function.

The implementors of the function write the function interface, implementation, and the test driver. They could then compile the function on a `Linux` system like this:

```
$ g++ -c sort.cpp
$ ls sort*
sort.cpp sort.h  sort.o
```

Here the dollar sign is taken to be the Linux prompt and `g++` is the C++ compile command. The `Linux -c` option on the compile command tells the compiler to compile but not link the source code. This is necessary because the `sort.cpp` file contains no `main()` function (it is not a complete C++ program). The `Linux ls` command then shows the result of the compile-but-don't-link command: it produced the object module named `sort.o` which contains the machine language translation of the C++ code in `sort.cpp`. this is where information hiding takes place: the function implementors can supply their customers with the function interface `sort.h` and this binary file without revealing the implentation algorithm.
The customers write the client program `program.cpp` and then they could compile and run it in a `Linux` environment like this:

```
$ g++ -o program program.cpp sort.o
$ ls program*
program program.cpp
program
```

The C++ compiler compiles the client program in the file `program.cpp` and then links it with the object module `sort.o` to produce the executable file named `program`. This program is then run simply by using that file name as a `Linux` command. (The `Linux` option `-o program` tells the compiler to use the name `program` for the executable file.
The implementors can compile and run their test driver the same way:

```
$ g++ -o text_sort text_sort.cpp sort.o
$ ls text_sort*
text_sort text_sort.cpp
text_sort
```

This sort of "check-expect" testing procedure is an essential component in software development.

EXAMPLE 2 Interchanging Modules

Suppose that some time after delivering their first version of `sort()` the implementors develop the following improved version: After testing this version, they can produce an object module for it and ship it out to their customers.

`sort.cpp`

```
void sort(int& x, int& y) {
   if(x > y) {
      int t = x;
      x = y;
      y = t;
   }
}

void sort(int& x, int& y, int& z) {
   sort(x, y);
   sort(y, z);
   sort(x, y);
}
```

Note the use of overloading here: the two definitions of `sort()` *have different parameter lists, which version is called depends on how many parameters are passed.*
Now, if a customer had compiled her application program separately, like this:

```
   $ g++ -c program program.cpp
   $ ls program*
   program program.cpp program.o
```

then she could upgrade her application without having to recompile it, like this:

```
   $ g++ -o program program.o aoer.o
```

The big advantage here is that the program is improved without changing any of its source code.
Implementing a function is like fulfilling a contract which specifies exactly what the function is required to do. The specifications then become the documentation for the function's specification file. In practice, the specifications may be written by the customer.
Function specifications are often written in terms of preconditions and postconditions. A *precondition* is a statement about the function's parameters that is assumed to be true before the function is called. A *postcondition* is a statement about the function's output (*i.e. its return value and any arguments passed by reference*) that is guaranteed to be true after the function returns.

EXAMPLE 3 Preconditions and Postconditions

```
double geometric_mean(double x, double y);
//  PRECONDITION: x >= 0.0 && y >= 0.0
// POSTCONDITION: r*r == x*y, where r is the value returned
```

FUNCTION TEMPLATES

The `swap()` function (see Example 1) is widely used in sorting and other essential algorithms. The two values that it interchanges might have type `int` in some applications and type `float` or `string` in others. But the algorithm itself is the same regardless of the type of its two parameters. C++ provides a mechanism that saves the programmer from having to write separate definitions for different type versions of the same function. Called a *function template*, it provides the minimal information needed by the compiler to generate its own definitions of the function whenever it needs them.

EXAMPLE 4 A `swap()` Function Template

Here is a template for `swap()` functions, followed by a program that will use three instantiations of it:

```
template<class T>
void swap(T& x, T& y) {
   T t = x;
   x = y;
   y = t;
}
int main() {
   int a = 44;
   int b = 66;
   swap(a, b); //compiler generates void swap(int, int);
   float s = 4.4;
   float t = 6.6;
   swap(s, t);  // compiler generates void swap(float, float);
   string sr = "Juan";
   string sra = "Maria";
   swap(sr, sra); //compiler generates void swap(string, string);
   return 0;
}
```

When the compiler reads `swap(a,b)` it recognizes that `a` and `b` have the same type (`int`) and uses the template to generate the function by substituting `int` for `T` in the three places where it occurs in the template. It does the same again when it reads `swap(s,t)` except that it substitutes `float` for `T` in the same three places. And likewise, when it reads `swap(sr, sra)` it substitutes `string` for `T`.
The only difference between a function definition and a function template definition is that the latter is preceded by the code `template <class T>` and the symbol `T` is used in place of a type. The symbol is called a *template parameter*.

DATA ABSTRACTION

Data abstraction is a generalization of procedural abstraction. The latter (see above) facilitates the development of software by separating the interfaces of user-defined functions from their implementations. The former does the same for user-defined data types. C++ extends this facility by allowing the user to incorporate functions (operations) within the data types which are called *classes*.

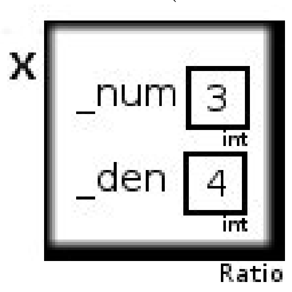
EXAMPLE 5 A `Ratio` Class Here is a C++ class definition for a type whose objects represent ratios (fractions):

```
class Ratio {
public:
   Ratio(int num, int den) { _num = num; _den = den; }
   void print() { std::cout << _num << '/' << _den; }
private:
   int _num;  // numerator
   int _den;  // denominator
};
```

The definition block contains two parts, one labeled `public` and one labeled `private`. Data objects and functions declared in the `public` section may be used anywhere in the program: in `main()` or in any other function. But data and functions declared in the `private` section may only be used within the class itself. In this case, we have two function members (`Ratio()` and `print()`) in the `public` section and two data members (`_num` and `_den`) of an object are "hidden" from the outside world. In most programmer-defined classes like this, most of the class functions are in the `public` section and most of the class data are in the `private` section.
Most C++ programmers follow the convention of capitalizing the name of a new class; *e.g.* `Ratio`; this distinguishes it as the name of a programmer-defined type. Another recommended method is to prefix an underscore to the name of each `private` member. This allows the same names without the underscore to be used in other places (*e.g.* `_num = num` in the `Ratio()` function) to make the code easier to understand.
Note that the bodies of the class functions (`Ratio()` and `print()`) are set on the same lines as their heads. This is normally done only with very simple functions. More often, the function bodies will be defined elsewhere, even in a separate file, leaving only the function prototype within the class definition. Also note that the class data (`_num` and `_den`) are declared on separate lines, with a comment for each. That form is recommended, instead of simply declaring `int _num, _den;`.
Finally, note that the first member function has the same name as the class itself (`Ratio`) and that it has no return type. Such class functions are called constructors. A *constructor* is a special member function that is automatically invoked whenever an object of the class is declared (*i.e.* "constructed").
Here is a little program that uses this `Ratio` class:

```
int main() {
   Ratio x(3,4);  // constructs the object x for 3/4
   x.print();  //calss the print() function for object x
   cout << endl;
   return 0;
}
```

The first statement in `main()`, `Ratio x(3,4)` invokes this constructor, which creates the object `x` and passes the arguments 3 and 4 to the parameters `num` and `den`.
At this point, the object `x` can be visualized as shown below. The name of the object is `x`; its type is `Ratio`; it has two fields (data members) named `_num` and `_den`, both of type `int`; and their values are 3 and 4, respectively.



The call `x.print()` produces output `3/4`.
To summarize, a class defines a new type in C++. The class contains data and functions, with the data (also called *fields*) usually in the `private` section and the functions (also called *methods* or *operations*) usually in the `public` section of the class declaration, like this:

```
class ObjectType {
public:
   ObjectType() { /* def. of constructor */ }
   void f(short k) { /* def. of function f() */ }
   int g() { /* definition of function g() */ }
private:
   float a;
   char c;
};
```

In external functions (such as `main()`), class objects (also called *instances*) are declared as any other objects are declared, like this:

    ObjectType u, v;

The member functions of the class can only be called when “bound” to a specific object of the class, like this:

    u.f(44); .. the call f() is bound to the object u
    int n = v.g(); .. the call g() is bound to the object v

The class object to which a member function call is bound is called the *implicit argument* for the call. For example, the call `u.f(44)` actually has two arguments: the explicit argument 44 and the implicit argument `u`.
A class constructor has the same name as the class itself and it has no return type. It is invoked automatically when the class is *instiated* (*i.e.*, an object of that class type is declared), and it cannot be called explicitly.

EXAMPLE 6 Defining the Addition Operation for the `Ratio` Class

Like all numeric types, the `Ratio` type should “know” how to add. We can give it this ability by including a `sum()` member function:

```
class Ratio {
public:
   Ratio(int num, int den { _num = num; _den = den; }
   void print() const { cout << num << '/' << _den; }
   Ratio sum(Ratio y) {
      int num = _num*y._den + _den*y._num;
      int den = _den*y._den;
      return temp;
   }
private:
   int _num;   //numerator
   int _den;   //denominator
};
int main() {
   Ratio r(3,4), s(2,3), t(0,1);   //constructs objects r,s and t
   t = r.sum(s);    // assigns to t the sum or r and s
   t.print();
   cout << endl;
   return 0;
}
```

The first line of `main()` constructs the three `Ratio` objects `r, s` and `t`, which represent 3.4, 2/3 and 0/1, respectively. The second line calls the `sum()` member function, passing the argument `s` and using `r` as the implicit argument (we say that `r` “owns” the call). This assigns $3 \cdot 3 + 4 \cdot 2 = 17$ to the local variable `num` and $4 \cdot 3 = 12$ to the local variable `den`. Then it constructs the local `Ratio` object `temp` using 17 and 12 to represent the ratio 17/12, and returns that object’s value to where the function was called in `main()`. Then that value (the pair $\{17, 12\}$) is assigned to `t` and then `t` is printed on the fourth line. the resulting output to the console is 17/12.
Notice that the member data values of the implicit argument `r` are references without the “dot” notation: `int den = _den*y._den;`. That is because the private data members (`_num` and `_den`) of whatever object “owns” the function call are directly accessible. When the call is `r.sum(s)`, the line `int den = _den*y._den;` means `int den = r._den*s,_den;`. If the call were `a.sum(s);`, then the same line would mean `int den = a._den*s._den;`
Also note here that an object (`s`) of type `Ratio` is passed to a function, an object (`temp`) of type `Ratio` is used as a local variable within a function, and an (anonymous) object of type `Ratio` is assigned to another `Ratio` object (`t`). The point is that objects of programmer-defined types can be used the same way as objects of fundamental types (*e.g.* `int, float,` *etc*).
The `sum()` function in Example 6 is a bit awkward because it represents the addition operation + which takes two operands, but it must be defined with only one (explicit) parameter. That is a result of the function being a member of the Ratio class, which requires it to have one implicit argument (the object that owns the call: `r` in `r.sum(s)`). The function would be more natural if it could be called using both operands (`r` and `s`) as explicit parameters like this:

```
    t = sum(r,s);
```

That would require the function to be declared as a non-member function, like this:

```
Ratio sum(Ratio x, Ratio y) {
    int num = x._num*y._den + x._den*y._num;
    int den = x._den*y._den;
    Ratio temp(num, den);
    return temp;
}
```

But that won’t work because the function needs access to the private data members `_num` and `_den` of the objects `x` and `y`; only member functions are granted that access. Fortunately, C++ provides an exception to that rule, precisely for situations like this. The solution is to make the `sum()` function a “friend” of the `Ratio` class.

C++ `friend` Functions

A `friend` function of a class is a non-member function that has the privileges of a member function of the class,m namely access to the class’s private members. One reason for making a function a `friend` of a class instead of an actual member of the class is to eliminate the requirement of the implicit argument, making instead all arguments passed to the function explicit. This simplifies the syntax and makes the code more readable.
A function is declared to be a `friend` of a class simply by declaring the function within the class and preceding its declaration with the keyword `friend`:

EXAMPLE 7 Using `friend` Functions

This works the same as Example 6:
```
class Ratio {
public:
     Ratio(int num, int den { _num = num; _den = den; }
     void print() const { cout << _num << '/' << _den; }
     friend Ratio sum(Ratio x, Ratio y) {
        int num = x._num*y._den + x._den*y._num;
        int den = x._den*y._den;
        return temp;
     }
private:
     int _num; //numerator
     int _den; //denominator
};
int main() {
     Ratio r(3,4), s(2,3), t(0,1); //construct objects r,s,t
     t = sum(r, s); // assigns to t the sum or r and s
     t.print();
     cout << endl;
     return 0;
}
```
The form `t = sum(r,s)` is a notational improvement over the form `t = r.sum(s)` that was used in example 6. As mentioned earlier, the actual definitions of a class's member functions are usually specified separately. This is illustrated in the next example.

EXAMPLE 8 Separating the Definitions of Member Functions from Their declarations

The same class as in Example 7:

```
class Ratio {
   friend Ratio sum(Ratio, Ratio);
public:
   Ratio(int, int); //constructor
   void print() const;
private:
   int _num;  //numerator
   int _den;  //denominator
};

Ratio sum(Ratio x, Ratio y) {
   int num = x._num*y._den + x._den*y._num;
   int den = x._den*y._den;
   Ratio temp(num, den);
   return temp;
}

Ratio::Ratio(int num, int den) {
   _num = num;
   _den = den;
}
void Ratio::print() {
   cout << _num << '/' << _den;
}
```

The only significant change here is that the bodies of the `friend` function `sum()` and of both member functions `Ratio()` and `print()` have been removed from the class, thereby separating the function definitions from their declarations. We have also shifted the `friend` function up to the beginning of the class definition to emphasize that it is not a member function. Its placement outside of the `public` and `private` declaration sections of the

class is irrelevant since it is not a member of the class.
Notice that the necessity of the*scope resolution operator* in the prefix `Ratio::` beforethe names of the member functions `Ratio()` and `print()` in their definitions. A class is like a namespace, restricting the scope of the names declared within. Of course, the definition of the `sum()` function does not need any scope resolution because it is not declared to be the member of any class; it is only a `friend` of the `Ratio` class.
The definitions of class functions are usually separated from their declarations that are given inside the class, like this:

```
class ObjectType {
public:
   ObjectType(); // constructor
   void f(short);
   int g();
private:
   float a;
   char c;  //comment describing it
};
// the implementation for the class ObjectType:
ObjectType::ObjectType() {
   // definition of constructor ObjectType() goes here
}

void ObjectType::f(short k) {
   //definition of function f() goes here
}

int ObjectType::g() {
   //definition of function g() goes here
}
```

The first part is called the *class interface*, and the second part is called the *class implementation.* This separation makes it easier to use the class because all the information that the programmer needs to use the class is given in its interface, leaving its implementation details hidden typically in a separate file. It also allows the class's creator to change the class implementation without affecting its interface or the programs that use it.

OVERLOADING OPERATORS An *operator* is a function that can be called with an alternative *infix* syntax form. For example, the *addition operator* + is usually used like this:

```
c = a + b
```

But it can also be used in its formal form, like this:

```
c = operator+(a, b);
```

As a function, its name is `operator+`.
Recall that any function in C++ can be *overloaded.* That means that the same name can be used for different functions, as long as their parameter lists are different. For example, the following two distinct functions can be defined in the same scope:

```
void swap(int x, int y) { int temp=x; x=y y=temp; }
void swap(float x, float y) { float temp=x; x=y y=temp; }
```

Then the following code will execute as intended:

```
int m = 33, n = 66;
swap(m,n); // swaps the integers m and n
float a=4.44, b=8.88;
swap(a,b); // swaps the floats a and b
```

The calls are made to the spearate functions, distinguished by their different parameter lists.
A class defines a new type. To give the new type functionality comparable to the operations that exist for fundamental types, we usually overload the operators used for those operations. For example, the addition operator defined for numeric types can be overloaded for our new `Ratio` type:

EXAMPLE 9 Overloading the Addition Operator

This class is equivalent to that defined in Example 8.

```
class Ratio {
   friend Ratio operator+(Ratio, Ratio);
public:
   Ratio(int, int); //constructor
   void print() const;
private:
   int _num;   //numerator
   int _den;   //denominator
};

Ratio operator+(Ratio x, Ratio y) {
   int num = x._num*y._den + x._den*y._num;
   int den = x._den*y._den;
   Ratio temp(num, den);
   return temp;
}
```

The implementation of the `Ratio` and `print()` functions are omitted since they are identical to those in Example 8.
The advantage of naming the function `operator+()` instead of `sum()` is that the operator name allows the function to be called like this:

```
    Ratio r(3,4), s(2,3), t(0,1);
    t = r + s;
```

This is a significant improvement over the form `t = sum(r,s)` used in Example 7.
The other arithmetic operators, `operator-()`, `operator*()` and `operator/()`, can be overloaded the same way.
In addition to overloading arithmetic operators, it is also convenient to overload the input and output operators `operator<<()` and `operator>>()`.

EXAMPLE 10 Overloading the Input and Output Operators

This extends Example 9:

```
class Ratio {
   friend Ratio operator+(Ratio, Ratio);
   friend istream& operator>>(istream& istr, Ratio& r);
   friend ostream& operator<<(ostream& ostr, const Ratio& r);
public:
   Ratio(int, int); //constructor
private:
   int _num;   //numerator
   int _den;   //denominator
};

istream& operator>>(istream& istr, Ratio& x) {
   char ch; // used to eat the slash character '/'
   istr >> x._num;
   if(istr.peek() == '/') istr >> ch >> x._den;
   else x._den = 1;
   return istr;
}

ostream& operator<<(ostream& ostr, const Ratio& x) {
   if(x._den==1) ostr << x._num;
   else ostr << x._num << '/' << x._den;
   return ostr;
}
```

The implementation of the `Ratio()` constructor is omitted since it is the same as in Example 8.

Note that the output operator `operator<<` replaces the `print()` function.
With these overloaded operators, we can write the following more natural looking code:

```
   Ratio r(3,4), s(2,3), t(0,1);
   cout << "\nEnter two ratios:";
   cin >> r >> s; // calls operator>>(istream&,Ratio&) twice
   t = r + s;
   cout << t << endl; //calls operator<<(ostream&,const Ratio&) once}
```

The third line first makes the call `operator>>(cin,r)`, passing the arguments `cin` and `r` to the parameters `ostr` and `x`, respectively, both by reference. If the input is 22/7, then the function reads 22 into `r._num` and 7 into `r._den` (ignoring the slash character). Since `r` was passed by reference, the resulting ration $\{22, 7\}$ is stored in `r`. Note how the `operator>>()` function uses the `peek()` function to determine whether the next character in the input stream is the slash character. If it is (as in the input 22/7), then that character is read into `ch` and the next integer is read into `r._den`. This code allows integer input also to be accepted by the input function (*e.g.* 119 would be read and stored as $\{119, 1\}$.).
The input function returns the input stream as a reference. This allows it to be passed along to the next call to the input function when the operator `>>` is chained as in

```
      cin >> r >> s;
```

In fact, these two calls are actually implemented as

```
      operator>>(operator>>(cin, r), s );
```

so a second ratio can be read into the object `s`.

CLASS INVARIANTS

One problem that is likely to occur with ratios (fractions) is that they are likely to be stored in non-reduced form such as 24/18 and -2/-23. Even if they are always in reduced form when they are created, non-reduced forms can still result, as with 1/6 + 3.8 = 26/48. This can be confusing because a single ratio can have an unlimited number of different forms (*e.g*, $(-5)/17 = (-15)/51 = 20/(-68) = \cdots$). this problem can be solved by specifying a class invariant.
A *class invariant* is a condition that is forced upon all instances of the class. We shall specify the following two class invariants for the `Ratio` class:

1 The data members `_num` and `_den` have no common factors greater than 1.

2 The data member `_den`> 0.

These two conditions ensure that no two distinct `Ratio` objects are numerically equal unless they have the same values stored in their `_num` and `_den` fields; *i.e.*, that every fraction that can be represented as a `Ratio` object has a unique representation.
Class invariants are typically enforced by means of `private` utility functions.

EXAMPLE 11 Enforcing Class Invariants

This extends Example 10:

```
class Ratio {
   friend Ratio operator+(Ratio, Ratio);
   friend istream& operator>>(istream&, Ratio&);
   friend ostream& operator<<(ostream&, const Ratio&);
public:
   Ratio(int, int); //constructor
private:
   void _reduce(); //enforce class invariants
   int _num;  //numerator
   int _den;  //denominator
};
```

```
Ratio operator+(Ratio x, Ratio y) {
   int num = x._num*y._den + x._den*y._num;
   int den = x._den*y._den;
   Ratio temp(num, den);
   temp._reduce();
   return temp;
}

istream& operator>>(istream& istr, Ratio& x) {
   char ch; // used to eat the slash character '/'
   istr >> x._num;
   if(istr.peek() == '/') istr >> ch >> x._den;
   else x._den = 1;
   x._reduce();
   return istr;
}

Ratio::Ratio(int num,int den) {
   _num = num;
   _den = den;
   _reduce();
}

int gce(int m ,int n) {
   //assert(m > 0 && n > 0);
   while(m > 0) {
      if(m < n) swap(m,n);
      m -= n;
   }
   return n;
}

void Ratio::_reduce() {
   if (_num == 0 || _den == 0) {
      _num = 0;
      _den = 1;
      return;
   }
   if (_den < 0) {
      _num *= -1;
      _den *= -1;
   }
   int abs_num = (_num < 0 ? -_num : _num);
   int g - gcd(abs_num, _den);
   _num /= g;  //enforces lowest terms invariance
   _den /= g;
}
```

The `_reduce()` function is a `private` member function of the `Ratio` class that is used by the addition operator `operator+()`, the input operator `operator>>()`, and the constructor `Ratio()`. The `gcd()` fuction does not need to be a member function because it does not access any `private` data and is used only by the `_reduce()` function. No`gcd()` function does use the `swap()` function which would be defined just above it in the class implementation file.

The `_reduce()` function first checks for division by 0; if the denominator is 0, the ratio is change to the default 0/1. Next it checks for a negative denominator to enforce the second class invariant. Then it uses the conditional expression operator to set up the absolute value of the numerator: if the condition `(_num < 0)` is true it assigns `-_num` to `abs_num`; otherwise it assigns `_num` to it. then it uses the `gcd()` function to obtain the greatest common divisor `g` of `abs_num` and `_den`. Since the greatest common divisor is a multiple of all other common divisors, dividing `_num` and `_den` by `g` guarantees that the resulting pair will have no common divisors greater than 1.

The `assert()` function used in the `gcd()` function is assumed to be defined elsewhere. Its single parameter accepts a condition (*i.e.* and `int`). If the condition is false (*i.e.* evaluates to 0), then the program aborts at that point. If the condition is true (*i.e.* evaluates to non-zero), then the program continues. Such a function is defined in the old C header file `assert.h`, which is named `cassert` in Standard C++.

CONSTRUCTORS AND DESTRUCTORS

There is still room for improvement of the `Ratio` class. For example, we can use an *initialization list* for the constructor:

EXAMPLE 12 Initialization Lists for Constructors

```
class Ratio {
   friend Ratio operator+(Ratio, Ratio);
   friend istream& operator>>(istream&, Ratio&);
   friend ostream& operator<<(ostream&, const Ratio&);
public:
   Ratio(int, int); //constructor
private:
   void _reduce(); //enforce class invariants
   int _num;  //numerator
   int _den;  //denominator
};

Ratio::Ratio(int num, int den) : _num(num), _den(_den) {
   _reduce();
}
```

The only change here over Example 11 is the initialization list in the implementation of the constructor `Ratio`:
    : _num(num), _den(den)
This is simply an alternative for the assignment statements
    _num = num;
    _den = den;
which were in the body of the function.
Initialization lists are very specialized: they can be used only in constructors and only for initializing member data. But they are convenient and are widely used.
A more significant improvement is to use *default values* for the constructor parameters:

EXAMPLE 13 Default Values for Parameters

```
class Ratio {
   friend Ratio operator+(Ratio, Ratio);
   friend istream& operator>>(istream&, Ratio&);
   friend ostream& operator<<(ostream&, const Ratio&);
public:
   Ratio(int=0, int=1); //constructor
private:
   void _reduce(); //enforce class invariants
   int _num;  //numerator
   int _den;  //denominator
};
```

No change is needed here in the implementation. We have simply added (in **boldface** "=0" and "=1" to the parameter list of the constructor's declaration. The effect is to use the value 0 for the parameter `num` and 1 for the parameter `den` if values are not passed in as arguments. For example:

```
  Ratio x(22,7); //invokes the default constructor
  Ratio y(x);  // invokes the copy constructor x->y
  Ratio z=x;   // invokes the copy constructor x->z
```

The copy constructor looks like the default constructor:

```
class Ratio {
   friend operator+(Ratio, Ratio);
   friend istream& operator>>(istream&, Ratio&);
   friend ostream& operator<<(ostream&, const Ratio&);
public:
   Ratio(int=0,int=1); //default contructor
   Ratio(const Ratio&); // copy constructor
private:
   _reduce();  // enforces class invariants
   int _num; //numerator
   int _den; //denominator
};
Ratio::Ratio(const Ratio& r) : _num(r._num), _den(r._den) {}
```

The copy constructor's single parameter is passed by *constant reference*. This means that the function can access the argument passed but it cannot change it. In this implementation, it simply copies the values of the argument's `_num` and `_den` fields into the corresponding fields of the new object being constructed.

THE FOUR AUTOMATIC MEMBER FUNCTIONS

Every class in C++ must have four special member functions: default constructor, a copy constructor, an assignment operator, and a destructor. Since these member functions are required, the compiler will create them automatically if they are not defined explicitly in the class.

The *default constructor* is the constructor that can be called with no arguments. It is invoked whenever a declaration without parameters is executed, like this:

```
ObjectType x; //creates object x of type ObjectType
```

The *copy constructor* is the constructor whose header has the special form:

```
ObjectType(const ObjectType&);
```

It is invoked automatically in three special cases:

(1) whenever a newly declared object is initialized:

```
ObjectType y=x; //the copy constrcutor x->y
```

(2) whenever an object is passed by value to a function:

```
void f(ObjectType x) {
  //..}
f(y);
```

(3) whenever an object is returned by value from a function:

```
ObjectType f() {
  ObjectType y;
  // ...
  return y;
}
x=f(); //copy constructor y->x
```

The *assignment operator* is the overloaded operator whose header has the special form

```
ObjectType& oepartor=(const ObjectType&);
```

It is invoked whenever one object is assigned to another:

```
    ObjectType x, y;
    //...
    x = y; // the assignment operator is called to copy y into x
```

The *destructor* is the function whose header has the special form:

```
~ObjectType();
```

It is invoked automatically whenever the object goes "out of scope":

```
if(t > 0) {
  ObjectType x;
} // the destructor is invoked for x
```

EXAMPLE 15 The Assignment Operator

Here are an assignment operator for the `Ratio` class:

```
Ratio& Ratio::operator=(const Ratio& r) {
  _num = r._num;
  _den = r._den;
  return *this;
}
```

Although it is not a constructor, the assignment operator is almost the same as the copy constructor (see Example 14): it copies the values of the argument's `_num` and `_den` fields into the corresponding fields of the implicit argument (the object that "owns" the call). Then the function returns that implicit argument with the statement:

```
  return *this;
```

The C++ keyword `this` can be used only within class member functions; it always refers to the implicit argument. For example, when the statement

```
y = x;
```

executes for the existing `Ratio` objects `x` and `y`, it is equivalent to the call

```
y.operator=(x);
```

which passes the explicit argument to `x` to the parameter `r` and uses `y` as the implicit argument. In that case, `this` is a pointer to (*i.e.*, contains the memory address of) `y`, and `*this` is a synonym for `y` itself. The complete effect of this call to the assignment operator is

```
y._num = x._num;
y._den = x._den;
```

Remember that the assignment operator, like the default constructor, the copy constructor, and the destructor, will be created automatically by the compiler if it is not explicitly specified in the class definition. In that case, the compiler implements it in the simplest way possible: by copying the fields of the object on the right of the assignment into the fields of object to the left. Since that is precisely what this implemented version does, it really does no need to be included at all in the class definition. In fact, the only times that the assignment operator, the default constructor, the copy constructor, or the destructor really need to be explicitly implemented is when they would do more than the obvious. In the `Ratio` class, that applies only to the default constructor which is implement to work with optional `int` parameters. Example 18 illustrates a class where the destructor needs to be implemented explicitly.

0.2 ABSTRACT DATA TYPES

An abstraction is an idealization in the mathematical sense: it is an imaginary idea that usually can be only approximated in the "real world." One of the best examples of this is the set of all integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. This set is essential to most mathematics. But it does not really exist outside of the mind's conception of it. We can imagine it, but there is no way to represent it in the real world, by computer or other means. The best we can do is approximate it. That is what the type `long` does. On a standard 32-bit workstation, a `long` object may assume any one of the the 4,294,967,296 elements of the set `long`$= \{-2147483648, \cdots, -2, -1, 0, 1, 2, \cdots, 2147483647\}$.

That is adequate for most programs, but tiny compared to $\mathbb{Z}$.
The point here is that the real set `long` is far different from the ideal set $\mathbb{Z}$. The successful programmer will keep that difference in mind. In practice, the difference becaome painfully appernt when integer overflow occurs.

EXAMPLE 16 Integer Overflow

This program will produce erroneous output on mush standard 32-bit PCs and workstations:

```
int main() {
   long n=1;
   for (int i=0; i<20; i++) {
      n *= 4;
      std::cout << n << std::endl;
   }
   return 0;
}
```

Assumking that the larges value of type `long` is 2147483647 ($= 2^{31} - 1$), this program will suffer integer overflow as soon as `i` reaches 16. The acutal output looks like this:

```
4
16
64
256
1024
4096
16384
65536
262144
1048576
4194304
16777216
67108864
268435456
1073741824
0
0
0
0
0
```

On another workstation, the last 5 numbers output are negative! In both cases, those values are obviously incorrect. The problem simply is that the next number after 1073741824 ($= 2^{30}$) would ideally be 4294967296 ($= 2^{32}$), but that is greater than the larges value that a `long` can have.

In computer science, we use the term `abstract data type` ("ADT") to describe the ideal which the real implemented data type represents. $\mathbb{Z}$ is the ADT which type `long` represents. Of course, there may be several different representations of the same ADT. For example, in C++ the integer types (`bool, char, short, long unsigned char, unsigned short, unsigned, unsigned long,` and `long long`) are all representations of $\mathbb{Z}$.

An *ADT* is a description of an ideal type which could be implemented in different ways in a program. The description includes the set of all values that the ideal type could have, the set of operations that could be performed on objects of that type, and any other information (*e.g.*, class invariants) that could be performed on objects of that type. It also serves as a specification that can be used when deriving algorithms, so that they can be implemented after the ADT has been implemented.

An ADT may be based upon a mathematical abstraction (as $\mathbb{Z}$ is) or upon more practical considerations. For example, we may want to define an ADT fro graphic images in order to build a type for the ADT named `image` that could be used in computer graphics programs. Our choice of operation to define for the ADT would likely be based upon previous experience, both with computers and with real visual images.

EXAMPLE 17 Am ADT for Stacks

A *stack* is a container that uses the "last-in-first-out" (LIFO) method for insertions and removals. Imagine a stack of tray in a lunch room: when "push" a tray onto the stack, it goes on top; when you "pop" a tray off the stack, you get the one that was on the top.

Here is a formal ADT specification for stack:

ADT: Stack

Represents:

A sequence of elements, all of the same type

Access

A stack allows access only at one end of the sequence, called the *top* of the stack. Both insertions and removals must be made at the top.

Constructors and Destructors:

`create` Creates an empty stack of a given maximum size.

`destroy` De-allocates the member used for the stack.

Access Functions:

`top` Return the last element put on the the stack.

`is_empty` Returns `true` if the stack is empty, otherwise `false`.

`is_full` Returns `true` if the stack is full, otherwise `false`.

Mutator Functions:

`push` Inserts a new element at the top of the stack.

`pop` Removes and returns the element from the top of the stack.

EXAMPLE 18 An Implementation of the ADT Stack with Element Type `char`

```
class Stack {
public:
   Stack(int s=100); // sets the default maximum number at 100
   ~Stack();
   cjar top() const;
   bool is_empy() const;
   void push(const char);
   char pop();
private:
   char* _a;  //The stack itself: a dynamic array of char
   int _max; // the maximum number of elements on the stack
   int _count;  // the actual number of elements on the stack
};

Stack::Stack(int m) : _max(m), _count(0) {
   _a = new char[_max];
   assert(_a != 0);
}

Stack::~Stack() {
   delete [] _a;
}

char Stack::top() const {
   assert(_count > 0);
   return _a[-count-1];
}

char Stack::is_empty() const {
   return bool(_count == 0];
}

char Stack::is_full() const {
   return bool(_count == _max];
}

char Stack::push(const char x) {
   assert(_count < _max);
   _a[_count++] = x;
}

char Stack::pop() {
   assert(_count > 0);
   return _a[--_count];
}
```

The class has three private data members: a dynamic array named `_a`, and two integers names `_max` and `_count`. A *dynamic array* in C++ is a pointer(*i.e.*, an address in memory) that can be used like an ordinary array (*i.e.*, with the subscript operator, as in `_a[4]`. But unlike an ordinary array, the size of a dynamic array may be declared at run-time by means of the `new` operator: `_a = new char[_max]`, and whose memory allocation can be de-allocated (*i.e.*, returned to the "heap" of memory available for use by other dynamic objects) by means of the `delete` operator: `delete [] _a`.

The integer `_max` holds the maximum number of elements that can be pushed onto the stack, and the integer `_count` contains the actual number of elements that are currently on the stack. Note the class invariant 0 ≤`_count`≤`q_max`.

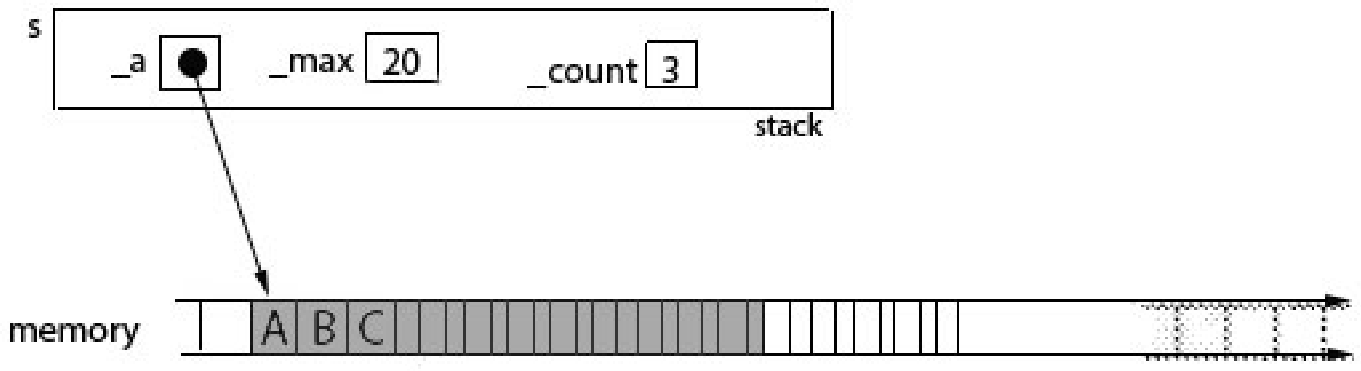
Note the use of the `assert()` function defined in the header `<cassert>`. This function will abort the program if the condition passes to it is false. In the constructor, the condition (`_a!=0`) means that the `new` operator was successful and there was enough dynamic memory available to allocate `_max` elements. The other conditions used in the calls to `assert()` will prevent the array index `_count` from going out of range; *i.e.*. they enforce the class invariant: 0 ≤`_count`≤`_max`.

Also note the effective use of the postincrement and predecrement operators in the `push()` and `pop()` functions. Since `_count` is always the location of the next element to be pushed onto the stack (the top of the stack is always `_a[_count-1]`), it has to be incremented after a new item is pushed onto the stack and it has to be decremented before the top element is popped off the stuck.

After this code executes:

```
    Stack s(20);
    s.push('A');
    s.push('B');
    std::cout << s.pop() << std::endl;
    s.push('C');
    s.push('D');
    std::cout << s.pop() << std::endl;
    s.push('E');
```

the stack object named s would look like this:



The shaded part of memory represents those bytes that have been allocated for the dynamic array _a. Its abstraction should be imagined as a stack exposing only the top (A) as available to push/pop operations.

1. What is the difference between procedural abstraction and data abstraction?

2. What is an *implicit argument*?

3. What is a `friend` function.

4. What is the purpose of separating a class implementation from its interface?

5. Show how the line
```
cin >> r >> s >> t;
```
is actually implemented for the `Ratio` objects. (See example 10.)

6. Implement the multiplication operator * for the `Ratio` class.

7. A *queue* is a container that uses the "first-in-first-out" ("FIFO") method for insertions and removals. Imagine a line of people waiting to buy tickets to a show: People enter the queue at the back and leave the queue at the front.

   Implement the following ADT for queues, with element type `char`:
   ADT: Queue
   Represents:
   A sequence of elements, all of the same type.
   Access:
   A queue allows access only at the ends of the sequence, called the *back* and the *front* of the queue. Insertions are allowed only at the back, and removals are allowed only at the front.
   Constructors and Destructors:

`create`	Creates an empty queue of a given maximum size.
`destroy`	De-allocates the memory used for the queue.

   Access Functions:

`back`	Returns the last element of the queue.
`front`	Returns the first element of the queue.
`is_empty`	Returns `true` if the queue is empty.
`is_full`	Returns `true`if the queue is full.

   Mutator Functions:

`enter`	Inserts a new element at the back of the queue.
`leave`	Removes and returns the first element from the front of the queue.

8. Write a function that uses a stack to reverse a queue.

9. Clock arithmetic is based on the notion of a 12-hour clock which "wraps around" the time every 12 hours. The arithmetic operations are the same as with ordinary integers, except for the wraparound property that keeps all values within the finite range $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$. Mathematicians call this *modulo 12 arithmetic*. (It is equivalent to the set named $\mathbb{Z}_{12}$ with the slight diference that "0" is used in place of "12".) For example, $7 + 9 = 4, 7 - 9 = 10$, and $7 * 9 = 3$. (Division is omitted.)
   Implement the following ADT for $\mathbb{Z}_{12}$:
   ADT: Hour
   Represents
   An element from the finite set $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$.
   Constructors and Destructors:
   `create` Creates an object whose value is between 0 and 11, inclusive.
   Arithmetic Operators:
   `sum` Returns the sum of two given hours.
   `difference` Returns the difference of two given hours.
   `product` Returns the product of two given hours.
   Input/output operators:
   `input` Reads a value for an hour from the standard input.
   `output` Prints the value for an hour from the standard output.

10. Implement the following ADT for a random number:
   ADT: Random
   Represents
   An object represents a device that generates random numbers. It uses a "seed" integer that generates the random numbers and is changed after each generation.
   Constructors and Destructors:
   `create` Creates a random number generator. If no seed is passed to it, it accesses the system clock to initialize the seed.
   Mutator Function:
   `reset` Changes the seed for the existing object. If no seed is passed to it, it accesses the system clock to initialize the seed.
   Generator Functions:
   `integer` Generates an integer selected at random from a uniform distribution in the range lo $\leq n \leq$ hi, where the default value for hi is `INT_MAX` and for lo is 1.
   `real` Generates a real selected at random from a uniform distribution in the range $0.0 \leq x < 1.0$.

11. Implement the following ADT for a pair of dice:
   ADT: Dice
   Represents
   An object represents two dice in terms of the sum of their values shown.
   Domain:
   The finite set $\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$.
   Constructor:
   `create` Creates an object whose value is between 2 and 12, inclusive.
   Access Function:
   `Toss` Simulates the tossing of the two dice. The value of each die is selected from a uniform distribution of integers from 1 to 6, and then their sum is returned.

12. Implement the following ADT for an address:
    ADT: Address
    Represents
    An object represents a mailing address.
    Constructor:

`create`	Creates address with optional given `string` values for the fields.

    Access Functions:

`street`	Returns a `string` with the street component of the address.
`city`	Returns a `string` with the city component of the address.
`state`	Returns a `string` with the state component of the address.
`code`	Returns a `string` with the postal code (*e.g.* the ZIP Code) of the address.
`country`	Returns a `string` with the country component of the address.

    Mutator Function:

`set_street`	Sets the street field to a given `string` value.
`set_city`	Sets the city field to a given `string` value.
`set_state`	Sets the state field to a given `string` value.
`set_code`	Sets the code field to a given `string` value.
`set_country`	Sets the country field to a given `string` value.

    Output Operator:
    Prints the complete mailing address.

13. Implement the subtraction operator for the `Ratio Class`. (see Example 9).

14. Implement the division operator `/` for the `Ratio Class`.)

15. Implement the following member function for the `Queue` class (Problem 7):
    ```
    int size()        // returns the number of elements in the queue
    ```

16. The mathematical set $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ is used in abstract algebra with many important practical applications in coding theory and other sciences. The simplest version is the case where $n = 2; \mathbb{Z}_2 = \{0, 1\}$, which is equivalent to the type `bool`. Another familiar version is the case where $n = 12$: $\mathbb{Z}_{12}$ is similar to the `Hour` class in Problem 7. Write up a complete ADT specification for $\mathbb{Z}_7$. Then implement it. Call it `ModN`. Use a `oconst int N = 7`. Include all the arithmetic operation, including the *division* (`/`) and *remainder* (`%`) operators. (It is a mathematical fact that division works in $\mathbb{Z}_n$ only if n is a prime number. For example, it does not work in $\mathbb{Z}_{12}$ because $6/2$ has more than one answer: $2*3 = 6$ but $2*9 = 18 = 6$ om $\mathbb{Z}_{12}$,so $6/2$ could be either 3 or 9!)

17. Implement the following ADT for a coin purse: **ADT: Purse**
    Description
    An object represents a coin changer or coin purse that can contain any number of pennies (1¢), nickels(5¢), dimes (10¢), and quarters (25¢).
    Invariant:
    The number of coins is minimal for the given monetary unit.
    Constructor:

`create`	Creates a purse with a given number of pennies, nickels, dimes and quarters.

    Access Functions:

`pennies`	Returns the number of pennies in the purse.
`nickels`	Returns the number of nickels in the purse.
`dimes`	Returns the number of dimes in the purse.
`quarters`	Returns the number of quarters in the purse.
`value`	Returns the total value of the coins in the purse..

Mutator Functions:

`insert`	Adds a given monetary value to the purse.
`remove`	Removes a given monetary value from the purse.
`empty`	Empties to the purse.

18. Implement the following ADT for a measured distance. **ADT: Distance**

Description

`create` Creates an object that represents a single measured distance in a given number of meters.

Constructors and Destructors:

`cm`	Returns the distance measured in centimeters.
`km`	Returns the distance measured in kilometers.
`in`	Returns the distance measured in inches.
`ft`	Returns the distance measured in feet.
`mi`	Returns the distance measured in miles.

Mutator Functions:

`set_cm`	Resets the distance measured in centimeters.
`set_km`	Resets the distance measured in kilometers.
`set_in`	Resets the distance measured in inches.
`set_ft`	Resets the distance measured in feet.
`set_mi`	Resets the distance measured in miles.
`add_cm`	Adds a distance measured in centimeters.
`add_km`	Adds a distance measured in kilometers.
`add_in`	Adds a distance measured in inches.
`add_ft`	Adds a distance measured in feet.
`add_mi`	Adds a distance measured in miles.
`subtract_cm`	Subtracts a distance measured in centimeters.
`subtract_km`	Subtracts a distance measured in kilometers.
`subtract_in`	Subtracts a distance measured in inches.
`subtract_ft`	Subtracts a distance measured in feet.
`subtract_mi`	Subtracts a distance measured in miles.

Operators:

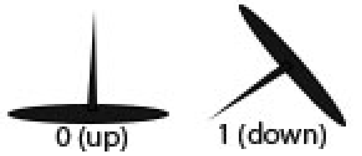
`assignment`	Assign another distance to `*this`.
`multiply`	Multiplies `*this` by a non-negative real number.
`divide`	Divides `*this` by a positive real number.

Friend Operators:

`multiply`	Multiplies one distance by another.
`divide`	Divides one distance by another.

19. Write and implement a complete ADT similar to that in Problem 11 for a class named Dice that represents the sum of three four-sided tossed. (Each die is a regular tetrahedron.)

20. Write and implement a complete ADT similar to that in Problem 11 for a class named `Tack` that uses the `Random` class (Problem 10) to represent the state of a tossed thumbtack. Assume that the thumbtack lands with its point up 60% of the time. The two outcomes are 0 (for point up) and 1 (for point down.).



21. Implement the following ADT for Date:

ADT: Date

Description
An object represents a specific date in history).
Invariants:
There was no year 0; the day after Dec 31 1 B.C. as Jan 1, 1 AD.
The month field can have only 12 different values.
The day field must be a positive integer that does not exceed the number of days in its month.
Constructor:

`create`	Creates an object representing a date given its era, year, month, and day.

Access Functions:

`era`	Returns the date's era, either BC or AD.
`year`	Returns the date's year; *e.g.*, 1969.
`month`	Returns the date's month; *e.g.* August.
`day`	Returns the date's dya; *e.g.*31.

Arithmetic Functions:

`add`	Adds a given number of days to the date.
`subtract`	Subtracts a given number of days from the date.

Output Operator:

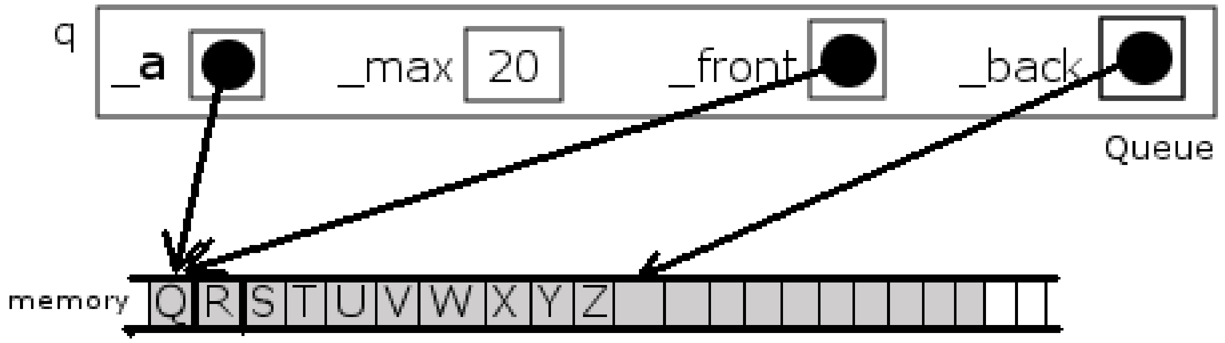
`output`	Prints the date; *e.g.*, "August 15, 1958 A.D."

Hint: Use a `private` utility function named `is_leap()` that determines whether the year is a leap year.

22. Write and implement a complete ADT for a class named `Person` whose objects represent people. Then implement it. Include fields for `Address` (see Problem 10), date of birth, date of death, identification number (*e.g.* Social Security Number in the U.S.A.), sex, telephone number, email address, and web page URL. Use the `Date` class (Problem 11) for DOB and DOD. Include an access function that returns the person's age in years on a given date.

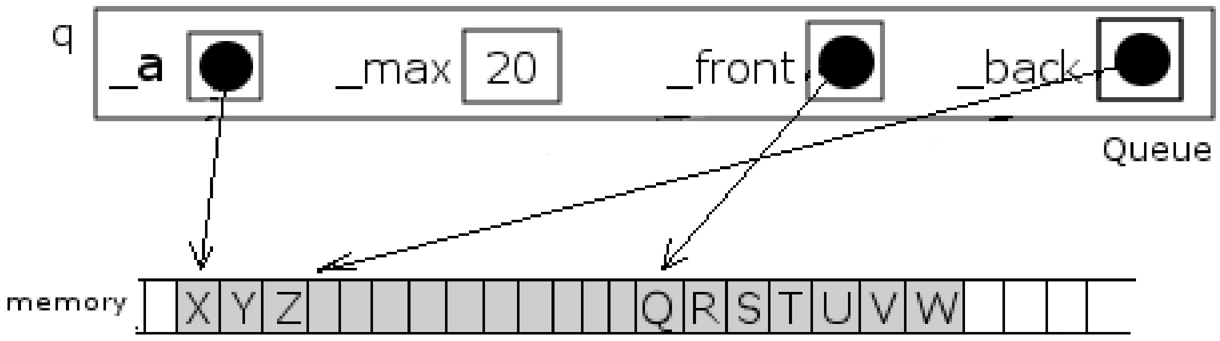
23. We can imagine a queue as a row of seats, where the first person to arrive sits in the left-most seat, the second person sits to the right of him, *etc.*, each new arrival sitting to the right of the previous arrival. This is the same as a stack, except that the departure algorithm is different. If it were a stack, the first to leave would be the one at the right end (*i.e.*, the last one who arrived). But since it is a queue the first to leave is the one on the left end (*i.e.*, the first one who arrived). Dynamically, there is another distinction between a stack and a queue: When one leaves a stack, he or she is the only one who moves, but when one leaves a queue, all the others in the queue shift one seat to the left. That shifting requires much movement of data in the array implementation.

    (a) In the implementation in Problem 7, we chose not to shift the elements in the `leave()` function. This allows it to run faster, but it makes inefficient use of the allocated space. (Each seat is used only once!) Modify the `leave()` function so that after each departure, everyone remaining in the queue shifts one seat to the left. For example a `char` queue implemented as in Problem 7 with `max = 20` would look like this after 18 arrivals, 16 departures, and then 8 more arrivals:



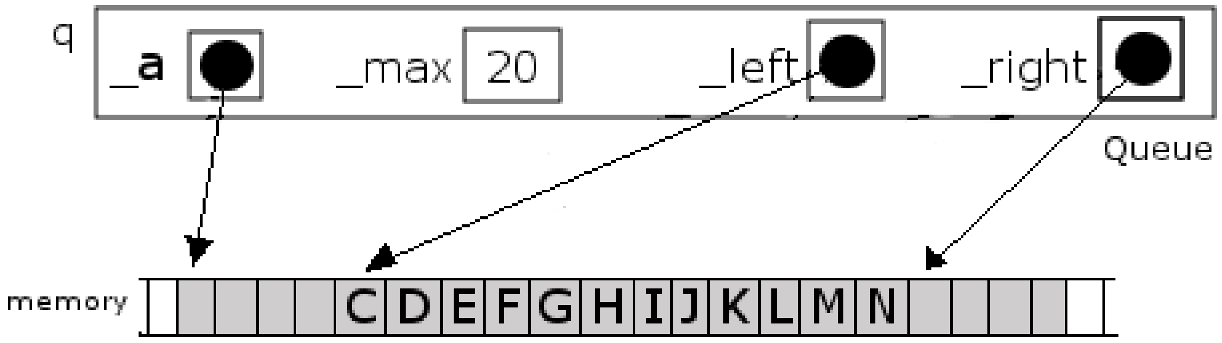
    Note that in this implementation, the `_front` pointer is redundant and could be ommitted.

    (b) Instead of shifting everyone to the left after each departure, we could simply "wrap around" the end whenever there are no more seats on the left. For example a `char` queue implemented as in Problem 7 with `max = 20` would look like this after 18 arrivals, 16 departures, and then 8 more arrivals:



Implement this algorithm for the `leave()` function. This is the most efficient implementation of the `leave()` function when an array is used. It is called a *circular array.*

24. A *deque* (rhymes with "heck") is a container that allows insertions and removals at both ends. The word is a contraction for "double-ended queue." Here is how a deque could be implemented using dynamic arrays:



Array subscripts are shown here even though the actual memory addresses would be large hexadecimal numbers instead. The deque is allowed to grow and shrink on both the left and the right. Growth would start in the middle at `a[max/2]`. The _left pointer points to the last element inserted on the left, and the `right` pointer points to the next insertion on the right. (This asymmetry is a result of maintaining the conventions used in all the other container classes that the difference between the two pointers equals the number of elements in the container.) Write a formal ADT for the deque data structure, and then implement it for elements of type `char`.

25. Write and implement a complete ADT for a class named `Angle` whose objects represent a specific plane angle measurement. This class will be similar to the `Distance` class in Problem 18. Include access functions that return the equivalent measure in degrees, radian and grads. Also include mutator functions for increasing or decreasing the measure.

26. Write and implement a complete ADT for a class named `Numeral` whose objects represent specific positive integers This class will be similar to the `Distance` class in Problem 18. Include access functions that return the Roman numeral equivalent (*e.g.* MMXIV) and the Hindu-Arabic equivalent (*e.g.* 2014). Also include mutator functions for increasing or decreasing the value.

27. Wrote and o,[;e,emt a cp,[;ete ADT for a class named `Money` whose objects represent a specific amount of money. This class will be similar to the `Distance` class in Problem 18. Include access functions that return the equivalent amount in primary currencies, such as dollars, pounds, marks, and yen. Also include mutator functions for increasing or decreasing the amount in dollars.

28. Write and implement a complete ADT for a class named `Book` whose objects represent published books. Include fields for author, title, publisher, year, and ISBN.

29. Write and implement a complete ADT for a class named Complex whose objects represent complex numbers (*e.g.*, $2.91 - 74.03\imath$). Also member functions like those for the `Ratio` class.

30. Write and implement a complete ADT for a class named `Time` whose object represent specific times of day (*e.g.*, 7:52:29 pm). Also include mutator functions for increasing or decreasing the time by a given number of seconds.