4.3 Classes

This section introduces the concept of classes as an extension of the struct concept from Section 4.1. You will learn about data encapsulation as a distinguishing feature of classes. This feature makes type implementations more safe and flexible. You will first learn classes feature by feature for rational numbers, and then see two complete classes in connection with random number generation.

4.3.1 Encapsulation

In the previous two sections, we have defined a new struct type `rational` whose value range models the mathematical type $\mathbb{Q}$ (the set of rational numbers), and we have shown how it can be equipped with some useful functionality (arithmetic and relational operators, input and output).

To motivate the transition from structs to classes in this section (and in particular the aspect of encapsulation), let us start off with a thought experiment. Suppose you have put the struct `rational` and all the functionality that we have developed into a nice library. In Exercise 99 you have actually done this, for the very basic version of the type `rational` from Program 32 and Program 33. Now you have sold the library to a customer; let's call it RAT (*Rational Thinking Inc.*). RAT is initially happy with the functionality that the library provides, and starts working with it. But then some unpleasant issues come up.

Issue 1: Initialization is cumbersome. Some code developed at RAT needs to initialize a new variable `r` with the rational number $1/2$; for this, the programmer in charge must write

```
rational r; // default-initialization of r
r.n = 1;    // assignment to data member
r.d = 2;    // assignment to data member
```

The declaration `rational r` default-initializes `r`, but the actual value of `r` must be provided through *two* assignments later. RAT tell you that they would prefer to initialize `r` from the numerator and denominator in one go, and you realize that they have a point here. Indeed, if the programmer at RAT forgets one of the assignments, `r` has undefined value (and you get to handle the bug reports). If the struct is larger (consider the example of `rational_vector_3` on page 229), the problem is amplified.

Issue 2: Invariants cannot be guaranteed. Any legal value of the type `rational` must have a nonzero denominator. You have stipulated this as an invariant in Program 32, but there is no way of enforcing this invariant. It is possible for anyone to write


```
rational r;
r.n = 1;
r.d = 0;
```

and thus violate the *integrity* of the type, the correctness of the internal representation.

You might argue that it would be quite stupid to write `r.d = 0`, and even the programmer at RAT can't be that stupid. But in RAT's application, the values of rational numbers arise from complicated computations somewhere else in the program; these computations may result in a zero denominator simply by mistake, and in allowing value `0` to be assigned to `r.d`, the mistake further propagates instead of being withdrawn from circulation (again, you get to handle the bug reports).

You think about how both issues could be addressed in the next release of the rational numbers library, and you come up with the following solution: As another piece of functionality on the type `rational`, you define a function that creates a value of type `rational` from two values of type `int`.

```
// PRE:  d != 0
// POST: return value is n/d
rational create_rational (int n, int d) {
  // somehow check here that d != 0
  rational result;
  result.n = n;
  result.d = d;
  return result;
}
```

You then advise RAT to use this function whenever they want to initialize or assign to a rational number. For example,

```
rational r = create_rational (1, 2);
```

would initialize `r` with $1/2$ in one go, and at the same time make sure that the denominator is nonzero.

Such a creation function certainly makes sense for structs in general, but the two issues above don't really go away. The reason is that this safe creation can be circumvented by not using it. In fact, your advice might not have reached the programmer at RAT, and even if it did, the programmer might be too lazy (or too stubborn) to follow it. It is therefore still possible to write `rational r;` and forget about data member assignment, and it is still possible to assign `0` to `r.d`. Behind this lies in fact a much larger poblem, as you discover next.

Issue 3: The internal representation cannot be changed. After having used the rational numbers library for some time, RAT approaches you with a request for a version with a larger value range, since they have observed that intermediate values sometimes overflow.

You recall the type `extended_int` from Page 229 and realize that one thing you could easily do is to change the type of numerator and denominator from `int` to `unsigned int`

and store the sign of the rational number separately as a data member of type `bool`. for example like this:

```
struct rational {
  unsigned int n;    // absolute value of numerator
  unsigned int d;    // absolute value of denominator
  bool is_negative; // sign of the rational number
};
```

It is also not too hard to rewrite the library files `rational.h` and `rational.C` to reflect this change in representation.

But shortly after you have shipped the new version of your library to RAT (you have even included the safe creation function `create_rational` from above in the hope to resolve issues 1 and 2 above), you receive an angry phone call from the management of RAT: the programmer reports that although the application code still compiles with the new version of the library, *nothing* works anymore!

After taking a quick look at the application code, you suddenly realize what the problem is: the code is cluttered up with expressions of the form *expr*`.n` and *expr*`.d`, as in

```
rational r;
r.n = 1;
r.d = 2;
```

Already this particular piece of code does not work anymore: a rational number is now represented by *three* data members, but the (old) application code obviously does not initialize the (new) member of type `bool`. Now you regret not to have provided the `create_rational` function in the first place; indeed, the statement

```
rational r = create_rational (1, 2);
```

would still work, assuming that you have correctly adapted the definition of the function `create_rational` to deal with the new representation. But the problem is much more far-reaching and manifests itself in each and every occurence of *expr*`.n` or *expr*`.d` in the application code, since the data members have changed their meaning (they might even have changed their names): in letting RAT access numerator and denominator through data members that are specific to a certain representation, you are now committed to that representation, and you can't change it without asking RAT to change its application code as well (which they will refuse, of course).

When the RAT management realizes that the new rational numbers with extended value range are useless for them, they terminate the contract with you. Disappointed as you are, you still realize that what you need to avoid such troubles in the future is *encapsulation*: a mechanism that *hides* the actual representation of the type `rational` from the customer, and at the same time offers the customer *representation-independent* ways of working with rational numbers.

In C++, encapsulation is available through the use of classes, and we start by explaing

how to hide the representation of a type from the customer.[5]

4.3.2 Public and private

Here is a preliminary `class` version of `struct rational` that takes care of data hiding.

```
class rational {
private:
   int n;
   int d; // INV: d != 0
};
```

In the same way as a `struct`, a `class` aggregates several different types into a new type, but the `class` keyword indicates that *access restriction* may occur, realized through the keywords `public` and `private`.

A data member is *public* if and only if its declaration appears somewhere after a `public:` specifier, and with no `private:` specifier in between. It is *private* otherwise. In particular, if the class definition (see Section 4.3.9 below for the precise meaning of this term) contains no `public:` specifier, all data members are private by default. In contrast, a struct is a class where all data members are public by default.[6]

If a data member is private, it cannot be accessed by customers through the member access operator. If a data member is public, there are no such restrictions. Under our above definition of `class rational`, the following will therefore not compile:

```
rational r;
r.n = 1;      // error: n is private
r.d = 2;      // error: d is private
int i = r.n; // error: n is private
```

In particular, the assignment `r.d = 0` becomes impossible (which is good), but at a (too) high price: now your customer cannot do anything with a rational number, and even you cannot implement `operator+`, say, as you used to do it in Program 33. What we are still lacking is some way of accessing the encapsulated representation. This functionality is provided by a second category of class members, namely *member functions*.

4.3.3 Member functions

Let us now add the missing functionality to `class rational` through member functions. It would seem natural to start with safe creation, but since there are specific member functions reserved for this purpose, let us first show two "general" member functions that grant safe access to the numerator and denominator of a rational number (we'll discuss below what `*this` and `const` mean here; and if you wonder why we can use

[5] In the following, the term "customer" is used in a broader sense for all programs that use the class.

[6] In fact, access specifiers may also occur in structs, but we will use a class whenever there are any access restrictions.

n and d before they are declared: this is a special feature of class scope, explained in Section 4.3.9).

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const
   {
     return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const
   {
     return d;
   }
private:
   int n;
   int d; // INV: d!= 0
};
```

If r is a variable of type rational, for example, the customer can then write

```
int n = r.numerator();     // get numerator of r
int d = r.denominator();   // get denominator of r
```

using the member access operator as for data members. The customer can call these two functions, since they are declared public. Access specifiers have the same meaning for member functions as for data members: a private member function cannot be called by the customer. This kind of access to the representation is flexible, since the corresponding member functions can easily be adapted to a new representation; it is also safe, since it is not possible to change the values of the data members through the functions numerator and denominator. As a general rule of thumb, all data members of a class should be private (otherwise, you encourage the customer to access the data members, with the ugly consequences mentioned in Issue 3 above).

The implicit call argument and *this. In order to call a member function, we need an expression of the class type for which we *access* the function, and this expression (appearing before the .) is an *implicit call argument* whose value may or may not be modified by the function call.

Within each member function, the lvalue *this refers to this implicit call argument and explains the appearance of *this in the postconditions of the two member funtions above. It does not explain why an asterisk appears in *this, but we will get to this later.

Const member functions. A const keyword after the formal argument list of a member function refers to the implicit argument *this and therefore promises that the member function call does not change the value (represented by the values of the data members) of *this. We call such a member function a *const member function.*

Member function call. The general syntax of a member function call is

expression.fname (*expression1*, ..., *expressionN*)

Here, *expression* is an expression of a class type for which a member function called *fname* is declared, *expression1*,..., *expressionN* are the call arguments, and . is the member access operator. In most cases, *expression* is an lvalue of the class type, typically a variable.

Access to members within member functions. Within the body of a member function f of a class, any member (data member of member function) of the same class can be accessed without a prefix *expr*.; in this case, we implicitly access it for *this. In our example, the expression n in the return statement of the member function numerator refers to the data member n of *this. The call r.numerator() therefore does what we expect: it returns the numerator of the rational number r.

Within member functions, we can also access members for other expressions of the same class type through the member access operator (like a customer would do it). All accesses to class members within member functions of the same class are *unrestricted*, regardless of whether the member in question is public or private. The public: and private: specifiers are only relevant for the customer, but not for member functions of the class itself.

Member functions are sometimes also referred to as *methods* of the class.

Member functions and modularization. In the spririt of Section 3.1.8, it would be useful to source out the member function definitions, in order to allow separate compilation. This works like for ordinary functions, except that in a member function definition outside of the class definition, the function name must be qualified with the class name. In the header file rational.h we would then write only the declarations (as usual within namespace ifm):

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const;
   // POST: return value is the denominator of *this
   int denominator () const;
private:
   int n;
   int d; // INV: d!= 0
};
```

The matching definitions would then appear in the source code file `rational.C` (again within namespace `ifm`, and after including `rational.h`) as follows.

```
int rational::numerator () const
{
  return n;
}
int rational::denominator () const
{
  return d;
}
```

4.3.4 Constructors

A constructor is a special member function that provides safe initialization of class values. The name of a constructor coincides with the name of the class, and—this distinguishes constructors from other functions—it does not have a return type, and consequently no return value. A class usually has several constructors, and the compiler figures out which one is meant in a given context (using the rules of overloading resolution, see the Details of Section 4.1).

The syntax of a constructor definition for a class *T* is as follows.

T (*T1 pname1*, *T2 pname2*, ..., *TN pnameN*)
  : *name1* (*expression1*), ..., *nameM* (*expressionM*)
block

Here, *pname1*,..., *pnameN* are the formal arguments of the constructor. In the *initializer*

: *name1* (*expression1*), ..., *nameM* (*expressionM*)

name1,..., *nameM* are data members, and *expression1*,...,*expressionM* are expressions of types whose values can be converted to the respective data member types. These values are used to initialize the data members, before *block* is executed, and in the order in which the members are declared in the class. In other words, the order in the initializer is ignored, but it is good practice to use the declaration order here as well. If a data member is not listed in the initializer, it is default-initialized. In the constructor body *block*, we can still set or change the values of some of the data members.

For the type `rational`, here is a constructor that initializes a rational number from two integers.

```
// PRE: d != 0
// POST: *this is initialized with numerator / denominator
rational (int numerator, int denominator)
```



```
  : n (numerator), d (denominator)
{
  // somehow check that d != 0
}
```

To use this constructor in a variable declaration, we would for example write

```
rational r (1,2); // initializes r with value 1/2
```

In general, the declaration

T x (*expression1*, ..., *expressionN*)

defines a variable *x* of type *T* and at the same time initializes it by calling the appropriate constructor with call arguments *expression1*,..., *expressionN*.

The constructor can also be called explicitly as in

```
rational r = rational (1, 2);
```

This initializes `r` not directly from two integers, but from an expression of type `rational` that is constructed by the explicit constructor call `rational(1,2)` (which is of type `rational`).

4.3.5 Default constructor

In Section 4.1.4, we have introduced the term *default-initialization* for the kind of initialization that takes place in declarations like

```
rational r;
```

For fundamental types, default-initialization leaves the value in question undefined, but for class types, the *default constructor* is automatically called to initialize the value. If present, the default constructor is the unique constructor with an empty formal argument list.

By providing a default constructor, we can thus make sure that class type values are *always* properly initialized. In case of the class `rational` (or any arithmetic type), default-initialization with value `0` seems to be the canonical choice, and here is the corresponding default constructor.

```
// POST: *this is initialized with 0
rational ()
  : n (0), d (1)
{}
```

In fact, we *must* provide a default constructor if we want the compiler to accept the declaration `rational r`. This makes class types safer than fundamental types, since it is not possible to circumvent a constructor call in declaring a variable.

The careful reader will notice that there must be an excpetion to this rule: Program 30 in Section 4.1 contains the declaration statement `rational r;` although in that program,

the type `rational` is a struct without any constructors. This is in fact the only exception: for a class without any constructors, the default constructor is implicitly provided by the compiler, and it simply default-initializes the data members; if a data member is of class type, this in turn calls the default constructor of the corresponding class. This exception has been made so that structs (which C++ has inherited from its precursor C) fit into the class concept of C++.

4.3.6 User-defined conversions

Constructors with one argument play a special role: they are *user-defined conversions*. For the class `rational`, the constructor

```
// POST: *this is initialized with value i
rational (int i)
  : n (i), d (1)
{}
```

is a user-defined conversion from `int` to `rational`. Under this constructor, `int` becomes a "type whose values can be converted to `rational`". This for example means that we can provide a call argument of type `int` whenever a formal function argument of type `rational` is expected; in the implicit conversion that takes place, the converting constructor is called. With user-defined conversions, we go beyond the set of *standard conversions* that are built-in (like the one from `int` to `double`), but in contrast to the (sometimes incomplete) standard conversion rules stipulated by the C++ standard, we make the rules ourselves.

There are meaningful user-defined conversions that can't be realized by constructors. For example, if we want a conversion from `rational` to `double`, we can't add a corresponding constructor to the type `double`, since `double` is not a class type. Even conversions to some class type T might not be possible in this way: if T is not "our" type (but comes from a library, say), we cannot simply add a constructor to T. In such situations, we simply tell *our* type how its values should be converted to the target type. The conversion from `rational` to `double`, for example, could be done through a member function named `operator double` like this.

```
// POST: return value is double-approximation of *this
operator double ()
{
  return double(n)/d;
}
```

In general, the member function `operator` S has implicit return type S and induces a user-defined conversion to the type S that is automatically invoked whenever this is necessary.

4.3.7 Member operators

All functionality of rational numbers that we have previously provided through "global" functions (`operator+`, `operator+=`,...) must now be reconsidered, since directly accessing the data members is no longer possible. Instead, we will use the member functions `numerator` and `denominator` for non-modifying access to the representation, and a constructor for returning a result. Addition for example then works like this (and becomes a bit lengthy):

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
  int rn = a.numerator() * b.denominator() +
           a.denominator() * b.numerator();
  int rd = a.denominator() * b.denominator();
  return rational (rn, rd);
}
```

But under access restrictions, there are some things that we cannot do properly through global functions. As an example, consider `operator+=`. This operator needs to change the value of a rational number, but there is no specific member function that allows us to do this. We can only simulate the change through the addition and an assignment, like this.

```
// POST: b has been added to a; return value is the new value of a
rational& operator+= (rational& a, rational b)
{
  return a = a + b;
}
```

This works but is inefficient (consider larger structs), since we first construct an intermediate result `a + b` which is subsequently copied back into `a`. In fact, `operator+=` was designed to avoid exactly this detour that we need to take now.

A better way to go is to realize `operator+` as a public member function (a *member operator*), having only one formal argument (for `b`), and `*this` taking the role of `a`. This looks as follows.

```
// POST: b has been added to *this; return value is
//       the new value of *this
rational& operator+= (rational b)
{
  n = n * b.d + d * b.n;
  d *= b.d;
  return *this;
}
```

Within this member function, there is no problem in accessing the data members directly, since the access restrictions do not apply to member functions. This version of `operator+`

is as efficient as the one previously used for `struct rational`, and it can in turn even serve as a basis for a more succint implementation of `operator+`:

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
  return a += b;
}
```

Prefer nonmember operators over member operators. You might argue that even `operator+` should become a member function of `class rational`, and indeed, this would probably allow a slightly more efficient implementation. There is one important reason to keep this operator global, though, and this has to do with user-defined conversions.

Having the conversion from `int` to `rational` that we get through the constructor

```
// POST: *this is initialized with value i
rational (int i);
```

we can for example write expressions like `r + 2` or `2 + r`, where `r` is of type `rational`. In compiling this, the compiler automatically inserts a converting constructor call. Now, having `operator+` as a member would remove the second possibility of writing `2 + r`. Why? Let's first see what happens when `r + 2` is compiled. If `operator+` is a member function, then `r + 2` "means"

```
r.operator+ (2)
```

In compiling this, the compiler inserts the conversion from the call argument type `int` to the formal argument type `rational` of `operator+`, and everything works as expected. `2 + r`, however, would mean

```
2.operator+ (r)
```

which makes no sense whatsoever. If we write a binary operator as a member function, then the first call argument *must* be of the respective class type. Implicit conversions do not work here: they only adapt call arguments to formal argument types of concrete functions, but they cannot be expected to "find" the class whose `operator+` has to be applied.

4.3.8 Nested types

There is a third category of class members, and these are *nested types*. To motivate these, let us come back to Issue 3 above, the one concerning the internal representation of rational numbers. If you think about consequently hiding the representation of a rational number from the customer, then you probably also want to hide the numerator and denominator type. As indicated in the example, these types might internally change, but in the member functions `numerator` and `denominator`, you still promise to return `int`-values.

A better solution would be to promise only a type with certain properties, by saying for example that the functions `numerator` and `denominator` return an integral type (Section 2.2.9). Then you can internally change from one integral type to a different one without annoying the customer. Technically, this can be done as follows.

```
class rational {
public:
   // nested type for numerator and denominator
   typedef int rat_int;
   ...
   // realize all functionality in terms of rat_int
   // instead of int, e.g.
   rational (rat_int numerator, rat_int denominator); // constructor
   rat_int numerator() const;                         // numerator
   ...
private:
   rat_int n;
   rat_int d; // INV: d!= 0
};
```

In customer code, this can be used for example like this.

```
typedef rational::rat_int rat_int;
rational r (1,2);
rat_int numerator = r.numerator();       // 1
rat_int denominator = r.denominator();   // 2
```

We already see one of the properties that the nested type `rational::rat_int` must have in order for this to work. For example, values of type `int` must be convertible to it.

Typedef declarations. A *typedef declaration* introduces a new name for an existing type into its scope. It does *not* introduce a new type. In fact, the new name can be used synonymously with the old name in all contexts. In the above code, we see this twice: within the class `rational`, the typedef declaration introduces a nested type `rat_int`, a new name for the type `int`. In the customer code, the class's nested type (that can be accessed using the scope operator, if the nested type declaration is public) receives a new (shorter) name.

In real-life C++ code, there are nested types of nested types of nested types,..., and typenames tend to get very long due to this. The typedef mechanism allows us to keep our code readable.

4.3.9 Class definitions

We now have seen the major ingredients of a class. Formally, a class definition has the form

```
class T{
  class−element ... class−element
};
```

where T is an identifier. The sequence of *class-element*'s may be empty. Each *class-element* is an *access specifier* (`public:` or `private:`), or a *member declaration*. A member declaration is a declaration statement that typically declares a member function, a data member, or a nested type. Collectively, these are called *members* of the class, and their names must be identifiers. A class definition introduces a new type, and this type is called a *class type*, as opposed to a fundamental type.

A member function definition is a declaration as well, but if the class definition does not contain the definition of a member function, this function must have a matching definition somewhere else (see Section 4.3.3). All member function definitions together form the *class implementation*.

Class Scope. Any member declaration of a class is said to have *class scope*. Its declarative region is the class definition. Class scope differs from local scope (Section 2.4.3) in one aspect. The potential scope of a member declaration is not only the part of the class definition "below" the declaration, but it spans the *whole* class definition, *and* the formal argument lists and bodies of all member function definitions. In short, a class member can be used "everywhere" in the class.

If two class definitions form disjoint declarative regions, there is no problem in using the same name for members of both classes.

4.3.10 Random numbers

We now have all the means to put together a complete and useful implementation of the type `rational` as a class in C++; but since we have already seen most of the necessary code in Section 4.1 and in this section, we leave the full class as Exercise 107 and continue here with a fresh class that has a little more entertainment in store.

Playing games on the computer would be pretty boring without some unpredictability: a chess program should not always come up with the same old moves in reaction to your same old moves, and in an action game, the enemies should not always pop up at the same time and location. In order to achieve unpredictability, the program typically uses a *random number generator*. This term is misleading, though, since the numbers are in reality generated according to some fixed rule, in such a way that they *appear* to be random. But for many purposes (including games), this is completely sufficient, and we call such numbers *pseudorandom*.

Linear congruential generators. A simple and widely-used technique of getting a sequence of pseudorandom numbers is the *linear congruential method*. Given a *multiplier* $a \in \mathbb{N}$,

an *offset* $c \in \mathbb{N}$, a *modulus* $m \in \mathbb{N}$ and a *seed* $x_0 \in \mathbb{N}$, let us consider the sequence $x_1, x_2, \ldots$ of natural numbers defined by the rule

$$x_i = (ax_{i-1} + c) \bmod m, \quad i > 0.$$

A small example is the pseudorandom number generator `knuth8`, defined by the following parameters.

$$a = 137, \quad c = 187, \quad m = 2^8 = 256, \quad x_0 = 0.$$

The sequence $x_1, x_2, \ldots$ of numbers that we get from this is

187, 206, 249, 252, 151, 138, 149, 120, 243, 198, 177, 116, 207, 130, 77, 240, 43, 190, 105, 236, 7, 122, 5, 104, 99, 182, 33, 100, 63, 114, 189, 224, 155, 174, 217, 220, 119, 106, 117, 88, 211, 166, 145, 84, 175, 98, 45, 208, 11, 158, 73, 204, 231, 90, 229, 72, 67, 150, 1, 68, 31, 82, 157, 192, 123, 142, 185, 188, 87, 74, 85, 56, 179, 134, 113, 52, 143, 66, 13, 176, 235, 126, 41, 172, 199, 58, 197, 40, 35, 118, 225, 36, 255, 50, 125, 160, 91, 110, 153, 156, 55, 42, 53, 24, 147, 102, 81, 20, 111, 34, 237, 144, 203, 94, 9, 140, 167, 26, 165, 8, 3, 86, 193, 4, 223, 18, 93, 128, 59, 78, 121, 124, 23, 10, 21, 248, 115, 70, 49, 244, 79, 2, 205, 112, 171, 62, 233, 108, 135, 250, 133, 232, 227, 54, 161, 228, 191, 242, 61, 96, 27, 46, 89, 92, 247, 234, 245, 216, 83, 38, 17, 212, 47, 226, 173, 80, 139, 30, 201, 76, 103, 218, 101, 200, 195, 22, 129, 196, 159, 210, 29, 64, 251, 14, 57, 60, 215, 202, 213, 184, 51, 6, 241, 180, 15, 194, 141, 48, 107, 254, 169, 44, 71, 186, 69, 168, 163, 246, 97, 164, 127, 178, 253, 32, 219, 238, 25, 28, 183, 170, 181, 152, 19, 230, 209, 148, 239, 162, 109, 16, 75, 222, 137, 12, 39, 154, 37, 136, 131, 214, 65, 132, 95, 146, 221, 0, 187, ...

From here on, the sequence repeats itself (in general, the period can never be longer than m). But until this point, it appears to be pretty random (although a closer look reveals that it is not random at all; do you discover a striking sign of nonrandomness?).

In order to make the magnitude of the random numbers independent from the modulus, it is common practice to *normalize* the numbers so that they are real numbers in the interval $[0, 1)$.

Program 34 below contains the definition of a class `random` in namspace `ifm` for generating normalized pseudorandom numbers acoording to the linear congruential method. There is a constructor that allows the customer to provide the parameters a, c, m, x_0, and a member function `operator()` to get the respective next element in the sequence of the x_i.

```
1 // Prog: random.h
2 // define a class for pseudorandom numbers.
3
4 namespace ifm {
5   // class random: definition
```

```
6   class random {
7   public:
8     // POST: *this is initialized with the linear congruential
9     //       random number generator
10    //           x_i = ( a * x_{i-1} + c) mod m
11    //       with seed x0.
12    random(unsigned int a, unsigned int c,
13           unsigned int m, unsigned int x0);
14
15    // POST: return value is the next pseudorandom number
16    //       in the sequence of the x_i, divided by m
17    double operator()();
18
19  private:
20    const unsigned int a_; // multiplier
21    const unsigned int c_; // offset
22    const unsigned int m_; // modulus
23    unsigned int xi_;      // current sequence element
24  };
25 } // end namespace ifm
```

Program 34: *progs/random.h*

The function `operator()` has no arguments in our case (that's why its declaration is `operator()()`, which admittedly looks a bit funny), and it overloads the *function call operator*, see Table 9 in the Appendix. In general, if *expr* is an expression of some class type which has the member function

`operator`()(*T1 name1*,..., *TN nameN*)

then *expr* can be used like a function: the expression

expr (*expr1*,..., *exprN*)

is equivalent to a call of the member function `operator()` with arguments *expr1*,...,*exprN* for the expression *expr*. We will see such calls in Program 37 and Program 38 below.

Here is the implementation of the class `random` in which we see how `operator()` updates the value of the data member `x_i` to be the respective next element in the sequence of the x_i.

```
1  // Prog: random.C
2  // implement a class for pseudorandom numbers.
3
4  #include <IFM/random.h>
5
```



```
6  namespace ifm {
7    // class random: implementation
8    random::random(unsigned int a, unsigned int c,
9                   unsigned int m, unsigned int x0)
10     : a_(a), c_(c), m_(m), xi_(x0)
11   {}
12
13   double random::operator()()
14   {
15     // update xi acording to formula,...
16     xi_ = (a_ * xi_ + c_) % m_;
17     // ...normalize it to [0,1), and return it
18     return double(xi_) / m_;
19   }
20 } // end namespace ifm
```

Program 35: *progs/random.C*

Many commonly used random number generators are obtained in exactly this way. For example, the well-known generator `drand48` returns pseudorandom numbers in $[0, 1)$ according to the parameters

$$a = 25214903917, \quad c = 11, \quad m = 2^{48},$$

and a seed chosen by the customer.[7] It is clear that we need a large modulus to obtain a useful generator, since m is an upper bound for the number of different numbers that we can possibly get from the generator. This means that `knuth8` from above is rather a toy generator.

The game of choosing numbers. Here is a game that you could play with your friend while waiting for a delayed train. Each of you independently writes down an integer between 1 and 6. Then the numbers are compared. If they are equal, the game is a draw. If the numbers differ by one, the player with the smaller number gets CHF 2 from the one with the larger number. If the two numbers differ by two or more, the player with the larger number gets CHF 1 from the one with the smaller number. You can repeat this until the train arrives (or until one of you runs out of cash, and hopefully it's your friend).

If you think about how to play this game, it's not obvious what to do. One thing *is* obvious, though: you should not write down the same number in every round, since then your friend quickly learns to exploit this by writing down a number that beats your number (by design of the game, this is always possible).

You should therefore add some unpredictability to your choices. You could, for example, secretly roll a dice in every round and write down the number that it shows.

[7] Assuming that the value range of `unsigned int` is $\{0, \ldots, 2^{32}-1\}$, we can't realize this generator using our class `random`. Doing all the computations over the type `double` and simulating the modulo operator in a suitable way is the way to go here.

But Exercise 112 reveals that your friend can exploit this as well.

You must somehow finetune your random choices, but how? In order to experiment with different distributions, you decide to define and implement a class `loaded_dice` that rolls the dice in such a way that the probability for number i to come up is equal to a prespecified value p_i (a fair dice has $p_i = 1/6$ for all $i \in \{1, \ldots, 6\}$). Then you could let different loaded dices play against each other, and in this way discover suitable probabilities to use against your friend (who is by the way not studying computer science).

Program 36 shows a suitable class definition (that in turn relies on the class `random` from above, with the normalization to the interval $[0, 1)$). We will get to the class implementation (and the meaning of the data members) in Program 37 below.

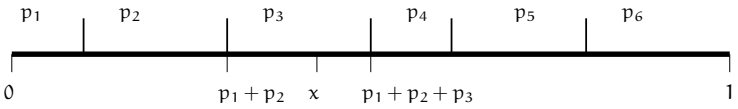
```
1  // Prog: loaded_dice.h
2  // define a class for rolling a loaded dice.
3
4  #include <IFM/random.h>
5
6  namespace ifm {
7    // class loaded_dice: definition
8    class loaded_dice {
9    public:
10     // PRE: p1 + p2 + p3 + p4 + p5 <= 1
11     // POST: *this is initialized to choose the number
12     //       i in {1,...,6} with probability pi, according
13     //       to the provided random number generator; here,
14     //       p6 = 1 - p1 - p2 - p3 - p4 - p5
15     loaded_dice (double p1, double p2, double p3, double p4,
16                  double p5, ifm::random& generator);
17
18     // POST: return value is the outcome of rolling a loaded
19     //       dice, according to the probability distribution
20     //       induced by p1,...,p6
21     unsigned int operator()();
22
23   private:
24     // p_upto_i is p1 + ... + pi
25     const double p_upto_1;
26     const double p_upto_2;
27     const double p_upto_3;
28     const double p_upto_4;
29     const double p_upto_5;
30     // the generator (we store an alias in order to allow
31     // several instances to share the same generator)
32     ifm::random& g;
```



```
33   };
34 } // end namespace ifm
```

Program 36: *progs/loaded_dice.h*

To initialize the loaded dice, we have to provide the probabilities $p_1, \ldots, p_5$ ($p_6 = 1 - \sum_{i=1}^{5} p_i$), and the random number generator that is being used to actually roll the dice. Again, we overload `operator()` to realize the functionality of rolling the dice once. How do we implement this functionality? We partition the interval $[0, 1)$ into 6 right-open intervals, where interval i has length p_i:



Then we draw a number x at random from $[0, 1)$, using our generator. If the number that we get were truly random, then it would end up in interval i with probability exactly p_i. Under the assumption that our pseudorandom numbers behave like random numbers in a suitable way, we therefore declare i as the outcome of rolling the dice if and only if x ends up in interval i. This is the case if and only if

$$p_1 + \ldots + p_{i-1} \leq x < p_1 + \ldots + p_i.$$

This explains the data members `p_upto_1`,..., `p_upto_5` (we don't need `p_upto_0` ($= 0$) and `p_upto_6` ($= 1$)). The constructor in Program 37 simply sets these members from the data provided, and the implementation of `operator()` uses them in exactly the way that was envisioned by the previous equation.

```
1  // Prog: loaded_dice.C
2  // implement a class for rolling a loaded dice.
3
4  #include <IFM/loaded_dice.h>
5
6  namespace ifm {
7    // class loaded_dice: implementation
8    loaded_dice::loaded_dice
9    (double p1, double p2, double p3, double p4, double p5,
10    ifm::random& generator)
11     : p_upto_1 (p1),
12       p_upto_2 (p_upto_1 + p2),
13       p_upto_3 (p_upto_2 + p3),
14       p_upto_4 (p_upto_3 + p4),
15       p_upto_5 (p_upto_4 + p5),
16       g (generator)
```

```
17   {}
18
19   unsigned int loaded_dice::operator()()
20   {
21     double x = g();
22     if (x <= p_upto_1) return 1;
23     if (x <= p_upto_2) return 2;
24     if (x <= p_upto_3) return 3;
25     if (x <= p_upto_4) return 4;
26     if (x <= p_upto_5) return 5;
27     return 6;
28   }
29 } // end namespace ifm
```

Program 37: *progs/loaded_dice.C*

Now you can compare two different loaded dices to find out which one is better in the game of choosing numbers. Program 38 does this, assuming that you are using a loaded dice that prefers larger numbers, and your friend uses a loaded dice that stays more in the middle. It turns out that in this setting, you win in the long run, but not by much (CHF 0.12 on average per round). Exercise 113 challenges you to find the best loaded dice that you could possibly use in this game.

```
1  // Prog: choosing_numbers.C
2  // let your loaded dice play against your friend's dice
3  // in the game of choosing numbers.
4
5  #include <iostream>
6  #include <IFM/loaded_dice.h>
7
8  // POST: return value is the payoff to you (possibly negative),
9  //       given the numbers of you and your friend
10 int your_payoff (unsigned int you, unsigned int your_friend)
11 {
12   if (you == your_friend) return 0;       // draw
13   if (you < your_friend) {
14     if (you + 1 == your_friend) return 2; // you win 2
15     return -1;                            // you lose 1
16   } // now we have your_friend < you
17   if (your_friend + 1 == you) return -2;  // you lose 2
18   return 1;                               // you win 1
19 }
20
21 int main() {
22   // the random number generator; let us use the generator
23   // ANSIC instead of the toy generator knuth8; m = 2^31;
```



```
24   ifm::random ansic (1103515245u, 12345u, 2147483648u, 12345u);
25
26   // your strategy may be to prefer larger numbers and use
27   // the distribution (1/21, 2/21, 3/21, 4/21, 5/21, 6/21)
28   double p = 1.0/21.0;
29   ifm::loaded_dice you (p, 2*p, 3*p, 4*p, 5*p, ansic);
30
31   // your friend's strategy may be to stay more in the middle
32   // and use the distribution (1/12, 2/12, 3/12, 3/12, 2/12, 1/12)
33   double q = 1.0/12.0;
34   ifm::loaded_dice your_friend (q, 2*q, 3*q, 3*q, 2*q, ansic);
35
36   // now simulate 1 million rounds (the train may be very late...)
37   int your_total_payoff = 0;
38   for (unsigned int round = 0; round < 1000000; round++) {
39     your_total_payoff += your_payoff (you(), your_friend());
40   }
41
42   // output the result:
43   std::cout << "Your total payoff is "
44             << your_total_payoff << "\n";
45
46   return 0;
47 }
```

Program 38: *progs/choosing_numbers.C*

4.3.11 Details

Friend functions. Sometimes, we want to grant nonmember functions access to the internal representation of a class. Typical functions for which this makes sense are the in- and output operators `operator<<` and `operator>>`. Indeed, writing out or reading into the internal representation often requires some knowledge of this representation that goes beyond what other functions need.

We cannot reasonably write `operator<<` and `operator>>` as members (why not?), but we can make these functions *friends* of the class. As a friend, a function has unrestricted access to the private class members. It is clear that the *class* must declare a function to be its friend, and not the other way around, since it's the class that has to protect its privacy, and not the function. Formally, a *friend declaration* is a member declaration of the form

`friend` *function–declaration*;

This declaration makes the respective function a friend of the class and grants access to

all data members, whether they are public or private. For the class `rational`, we could rewrite the `private` section as follows to declare in- and output operators to be friends of the class.

```
class rational {
private:
  friend std::ostream& operator<< (std::ostream& o, rational r);
  friend std::istream& operator>> (std::istream& i, rational& r);
  int n;
  int d; // INV: d!= 0
};
```

In the definition of these operators, we can then access the numerator and denominator through `.n` and `.d` as we used to do it in Section 4.2.4. If possible, friend declarations should be avoided, since they compromise encapsulation; but sometimes, they are useful in order to save unnecessary member functions.

4.3.12 Goals

Dispositional. At this point, you should ...

1) be able to explain the purpose of a class in C++;
2) understand the new syntactical and semantical terms associated with C++ classes, in particular *access specifiers*, *member functions*, and *constructors*;
3) understand the classes `ifm::random` and `ifm::loaded_dice` in detail.

Operational. In particular, you should be able to ...

(G1) find syntactical and semantical errors in a given class definition and implementation;

(G2) describe value range and functionality of a type given by a class definition;

(G3) add functionality to a given class through member functions;

(G4) write simple classes on your own;

(G5) work with and argue about pseudorandom numbers.

4.3.13 Exercises

Exercise 107 *Provide a full implementation of rational numbers as a class type, and test it. The type should offer all arithmetic operators (including in- and decrement, and the arithemtic assignments), relational operators, as well as in- and output and user-defined conversions (from* `int` *and to* `double`*). As an invariant, it should hold that the internal representation is normalized (see also Exercise 102). For all the functionality you provide, decide whether it should be realized by member functions,*

or by nonmember functions. The class should also have a nested numerator and denominator type to achieve more flexibility, and there should be a conversion function from values of this type. (G3)(G4)

Exercise 108 *Rewrite the struct* `Tribool` *that you have developed in Exercise 93 into a class, by*

a) making the data members private,

b) adding corresponding access functions,

c) adding an access function `is_bool() const` *that returns true if and only if the value is not unknown, and*

d) adding user-defined conversions from and to the type `bool`*.*

(G4)

Exercise 109

a) Find all errors in the following program. Fix them and describe the functionality of the type `Clock`*, by providing pre-and postconditions for the member functions.* (G1)(G2)

```
1  #include <iostream>
2
3  class Clock {
4    Clock(unsigned int h, unsigned int m, unsigned int s);
5    void tick();
6    void time(unsigned int h, unsigned int m,
7              unsigned int s);
8  private:
9    unsigned int h_;
10   unsigned int m_;
11   unsigned int s_;
12 };
13
14 Clock::Clock(unsigned int& h,
15              unsigned int& m,
16              unsigned int& s)
17     : h_(h), m_(m), s_(s)
18 {}
19
20 void Clock::tick()
21 {
22   h_ += (m_ += (s_ += 1) / 60) / 60;
23   h_ %= 24; m_ %= 60; s_ %= 60;
24 }
25
26 void Clock::time(unsigned int& h,
27                  unsigned int& m,
28                  unsigned int& s)
29 {
30   h = h_;
31   m = m_;
```

```
32    s = s_;
33  }
34
35  int main() {
36    Clock c1 (23, 59, 58);
37    tick();
38
39    unsigned int h;
40    unsigned int m;
41    unsigned int s;
42    time(h, m, s);
43
44    std::cout << h << ":" << m << ":" << s << "\n";
45
46    return 0;
47  }
```

b) Implement an output operator for the class `Clock`. (G3)

Exercise 110 *Write a program* `random_triangle.C` *to simulate the following random process graphically. Consider a fixed triangle* `t` *and choose an arbitrary vertex of* `t` *as a starting point. In each step, choose as a next point the midpoint between the current point and a (uniformly) randomly selected vertex of* `t`.

The simulation at each step draws the current point into a Window. Use the window object `ifm::wio` *defined in* `<IFM/window>` *for graphical output, and choose the triangle with vertices* $(0,0)$, $(512,0)$, *and* $(256,512)$. *Use the random number generator* `ansic` *from Program 38. At begin, the program should read in a seed for the random number generator and the number of simulation steps to perform. For testing purposes, let the simulation run for about 100,000 steps.* (G5)

Exercise 111 *Consider the generator* `ansic` *used in Program 38. Since the modulus is* $m = 2^{31}$, *the internal computations of the generator will certainly overflow if 32 bits are used to represent* `unsigned int` *values. Despite this, the sequence of pseudorandom numbers computed by the generator is correct and coincides with its mathematical definition. Explain this!* (G5)

Exercise 112 *Find a loaded dice that beats the fair dice in the game of choosing numbers. (This is a theory exercise.)* (G5)

4.3.14 Challenges

Exercise 113 *What is the best loaded dice for playing the game of choosing numbers? Give its distribution! You could try to approximate the distribution experimentally, or somehow compute it. (***Hint***: in order to find a suitable theoretical model, search for the term "zero-sum games", or directly go to the corresponding chapter in* `http://www.inf.ethz.ch/personal/gaertner/cv/lecturenotes/ra.pdf`. *Once you have formulated the problem as a zero-sum game, you can solve it using for example the web-interface* `http://banach.lse.ac.uk/form.html` (G5)