Chapter 4

Compound Types
4.1 Structs

In this section, you will learn three concepts for deriving new types from existing types. We show how structs are used to group data and to obtain new types with application-specific functionality. You will also see how operator overloading can help in making new types easy and intuitive to use.

Suppose we want to use rational numbers in a program, i.e., numbers of the form n/d, where both the numerator n and the denominator d are integers. C++ does not have a fundamental type for rational numbers, so we have to implement it ourselves.

We could of course represent a rational number simply by two values of type int, but this would not be in line with our perception of the rational numbers as a distinct mathematical concept. The two numbers n and d "belong together", and this is also reflected in mathematical notation: the symbol Q for the set of rational numbers indicates that we are dealing with a mathematical type, defined by its value range and its functionality (see Section 2.1.6). Ideally, we would like to get a C++ type that can be used like existing arithmetic types; the following piece of code (for adding two rational numbers) shows how this could look like.

```cpp
// input
std::cout << "Rational number r:\n";
Rational r;
std::cin >> r;

std::cout << "Rational number s:\n";
Rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

C++ offers several concepts for defining new types based on existing types. In this section, we introduce the concept of structs. A struct is used to aggregate several values of different types into one value of a new type. With this, we can easily model the mathematical type Q as a new type in C++. Here is a working program that makes a first step toward the desired piece of code above.

```cpp
1 // Program: userational.C
2 // Add two rational numbers.
3 #include <iostream>
4
5 // the new type Rational
6 struct Rational {
```
int n;
int d; // INV: d != 0
}
// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
int main ()
{
    // input
    std::cout << "Rational number r:\n";
    rational r;
    std::cout << " numerator =? "; std::cin >> r.n;
    std::cout << " denominator =? "; std::cin >> r.d;
    std::cout << "Rational number s:\n";
    rational s;
    std::cout << " numerator =? "; std::cin >> s.n;
    std::cout << " denominator =? "; std::cin >> s.d;
    
    // computation
    rational t = add (r, s);
    
    // output
    std::cout << "Sum is " << t.n << "/" << t.d << ".\n";
    
    return 0;
}  

Program 30: progs/userational.c

In C++, a struct defines a new type whose value range is the Cartesian product of a fixed number of types.\(^1\) In our case, we define a new type named rational whose value range is the Cartesian product int\times int, where we interpret a value \((n, d)\) as the quotient \(n/d\).

Since there is no type for the denominator with the appropriate value range \(\text{int}\backslash\{0\}\), we specify the requirement \(d \neq 0\) by an informal invariant, a condition that has to hold for all legal combinations of values. Such an invariant is indicated by a comment starting

\(^1\)Here and in the following, we identify a type with its value range to avoid clumsy formulations.
with

    // INV:

    Like pre- and postconditions of functions (see Section 3.1.1), invariants are an informal
    way of documenting the program; they are not standardized, and our way of writing them
    is one possible convention.

    The type rational is referred to as a struct, and it can be used like any other type;
    for example, it may appear as parameter type and return type in functions like add.

A struct defines a type, not variables. Let's get rid of one possible confusion right from the
beginning. The definition

    struct rational {
        int n;
        int d; // INV: d != 0
    };

does not define variables n and d of type int, although the two middle lines look like
variable declarations as we know them. Rather, all four lines together define a type of
the name rational, but at that point, neither a variable of that new type, nor variables
of type int have been defined. The two middle lines

    int n;
    int d; // INV: d != 0

specify that any actual object of the new type (i.e. any concrete rational number) "has"
is represented by two objects of type int that can be accessed through the names n and
d; see the member access below. This specification is important if we want to implement
operations on our new type like in the function add.

Here is an analogy for the situation. If the university administration wants to specify
how a student is represented in their files, they might come up with three pieces of data
that are necessary: a name, an identification number, and a program of study. This
defines the "type" of a student and allows functionality (registration, change of program
of study, etc.) to be realized, long before any students actually show up.

4.1.1 Struct definitions.

In general, a struct definition looks as follows.

```c
struct T {
    T1 name1;
    T2 name2;
    ...
    TN nameN;
};
```
4.1. **STRUCTS**

Here, $T$ is the name of the newly introduced struct (this name must be an identifier, Section 2.1.9), and $T_1, \ldots, T_N$ are names of existing types. These are called the *underlying types* of $T$. The identifiers $name1, name2, \ldots, nameN$ are are the *data members* of the new type $T$.

The value range of $T$ is $T_1 \times T_2 \times \ldots \times T_N$. This means, a value of type $T$ is an $N$-tuple $(t_1, t_2, \ldots, t_N)$ where $t_i \in T_i$.

"Existing types" might be fundamental types, but also user-defined types. For example, consider the vector space $\mathbb{Q}^3$ over the field $\mathbb{Q}$. Given the type rational as above, we could model $\mathbb{Q}^3$ as follows.

```c
struct rational_vector_3 {
    rational x;
    rational y;
    rational z;
};
```

Although it follows from the definition, let us make it explicit: the types $T_1, \ldots, T_N$ need not be the same. Here is an example: If $0, 1, \ldots, U$ is the value range of the type unsigned int, we can get a variant of the type int with value range

$$\{-U, -U+1, \ldots, -1, 0, 1, \ldots, U-1, U\}$$

as follows.

```c
struct extended_int {
    // represents $u$ if $n=\text{false}$ and $-u$ otherwise
    unsigned int u; // absolute value
    bool n; // sign bit
};
```

The value range of this type is $\{0, 1, \ldots, U\} \times \{\text{true}, \text{false}\}$, but like in the rational case, we interpret values differently: a value $(u, n)$ "means" $u$ if $n = \text{false}$ and $-u$ if $n = \text{true}$.

Even if two struct definitions have the same *member specification* (the part of the definition enclosed in curly braces), they define *different* types, and it is not possible to replace one for the other. Consider this trivial but instructive example with two apparently equal structs defined over an empty set of existing types.

```c
struct S {
};

struct T {
};

void foo (S s) {};

int main() {
    S s;
    T t;
}
4.1.2 Structs and scope

The scope of a struct is the part of the program in which it can be used (in a variable declaration, or as a formal function parameter type, for example). Structs behave similar to functions here: the scope of a struct is the union of the scopes of all its `declarations`, where a struct declaration has the form

```
struct T
```

The struct definition is a declaration as well, and usually one actually needs the definition in order to use a struct. This is easy to explain: in order to translate a variable declaration of struct type, or a function with formal parameters of struct type into machine language, the compiler needs to know the amount of memory required by an object of the struct. But this information is only obtainable from the definition of the struct; as long as the compiler has only seen a declaration of $T$, the struct $T$ is said to have `incomplete type`.

4.1.3 Member access

A struct is more than the Cartesian product of its underlying types—it offers some basic functionality on its own that we explain next. The most important (and also most visible) functionality of a struct is the access to the data members (the values $t_i$ in the N-tuple $t = (t_1, \ldots, t_N)$), and here is where the identifiers $\text{name}_1, \ldots, \text{name}_N$ come in. If $\text{expr}$ is an expression of type $T$ with value $(t_1, \ldots, t_N)$, then $t_k$—the K-th component of its value—can be accessed as

```
expr.nameK
```

Here, `.' is the `member access operator` (see Table ?? in the Appendix for its specifics). The composite expression $\text{expr}.\text{name}_K$ is an lvalue if $\text{expr}$ itself is an lvalue, and we say that the data member $\text{name}_K$ is `accessed for $\text{expr}$`.

Lines 25 and 26 of Program 30 assign values to the rational numbers $r$ through the member access operator, while line 37 employs the member access operator to output the
value of the rational number \( t \). The additional output of ‘/’ indicates that we interpret the 2-tuple \((n, d)\) as the quotient \( n/d \).

### 4.1.4 Initialization and assignment

We can initialize objects of struct type and assign values to them, just like we do it for fundamental types.

In line 34 of Program 30 for example, the variable \( t \) of type rational is initialized with the value of the expression \( \text{add} \ (r, \ s) \). In a struct, initialization is quite naturally done member-wise, i.e. for each data member separately. Under the hood, the declaration statement

\[
\text{rational} \ t = \text{add} \ (r, \ s);
\]

therefore has the effect of initializing \( t.n \) (with the first component of the value of \( \text{add} \ (r, \ s) \)) and \( t.d \) (with the second component). Interestingly, this also works with array members. Structs therefore provide a way of “faking” array initialization and assignment by wrapping the array into a struct. Here is an example to show what we mean.

```cpp
#include <iostream>

struct point {
    double coord[2];
};

int main()
{
    point p;
    p.coord[0] = 1;
    p.coord[1] = 2;

    point q = p;
    std::cout << q.coord[0] << " " << 1 << q.coord[1] << "\n"; // 2

    return 0;
}
```

This works since the data members of a struct object occupy a contiguous part of the main memory, and since (in contrast to array types) struct types “know” their memory requirements. From this, the compiler can figure out how many memory cells are to be copied for the initialization of \( q \) in point \( q = p \).

In the same way (memberwise initialization), the formal parameters \( a \) and \( b \) of the function \( \text{add} \) are initialized from the values of \( r \) and \( s \); the value of \( \text{add} \ (r, \ s) \) itself also results from an initialization of a (temporary) object when the return statement of the function \( \text{add} \) is executed.
Instead of the above declaration statement that initializes \( t \) we could also have written

\[
\begin{align*}
\text{rational } t; \\
t &= \text{add} (r, s);
\end{align*}
\]

Here, \( t \) is \textit{default-initialized} first, and this default-initializes the data members. In our case, they are of type \texttt{int}; for fundamental types, default-initialization does nothing, so the values of the data members are undefined after default-initialization (see also Section 2.1.8). In the next line, the value of \texttt{add (r, s)} is assigned to \( t \), and this assignment again happens member-wise.

**What about other operations?** For every fundamental type \( T \), two expressions of type \( T \) can be tested for equality, using the operators \texttt{==} and \texttt{!=}. It would therefore seem natural to have these two operators also for structs, implemented in such a way that they compare member-wise.

Formally, this would be correct: if \( t = (t_1, \ldots, t_N) \) and \( t' = (t'_1, \ldots, t'_N) \), then we have \( t = t' \) if and only if \( t_K = t'_K \) for \( K = 1, \ldots, N \).

But our type \texttt{rational} already shows that this won’t work: under member-wise equality, we would erroneously conclude that \( 2/3 \neq 4/6 \). The problem is that the \textit{syntactical value range int\times int} of the type \texttt{rational} does not coincide with the \textit{semantical value range} in which we identify pairs \((n, d)\) that define the same rational number \( n/d \).

The same happens with our type \texttt{extended\_int} from above: since both pairs \((0, \text{false})\) and \((0, \text{true})\) are interpreted as \( 0 \), member-wise equality would give us “\( 0 \neq 0 \)” in this case.

Only the implementor of a struct knows the semantical value range, and for this reason, C++ neither provides equality operators for structs, nor any other operations beyond the member access, initialization, and assignment discussed above. Operations that respect the semantical value range can be provided by the implementor, though, see next section.

You might argue that even member-wise initialization and assignment could be inconsistent with the semantics of the type. Later, we will indeed encounter such a situation, and we will show how it can be dealt with elegantly.

### 4.1.5 User-defined operators

New types require new operations, but when it comes to the naming of such operations, one less nice aspect of Program 30 shows in line 34. By defining the function \texttt{add}, we were able to perform the operation \( t := r + s \) through the statement

\[
\begin{align*}
\text{rational } t &= \text{add} (r, s);
\end{align*}
\]

Ideally, however, we would like to add rational numbers like we add integers or floating-point numbers, by simply writing (in our case)
4.1. STRUCTS

rational t = r + s;
The benefit of this might not be immediately obvious, in particular since the naming of
the function add seems to be quite reasonable; but consider the expression
rational t = subtract (multiply (p, q), multiply (r, s));
and its "natural" counterpart
rational t = p * q - r * s;
to get an idea what we mean.

The natural notation can indeed be achieved: a key feature of the C++ language is
that most of its operators (see Table ?? in the Appendix for the full list) can be overloaded
to work for other types as well. This means that we can use the same operator token to
implement various operators: we "overload" the token.

In principle, this is nothing new: we already know that the binary operator + is
available for several types, for example int and double. What is new is that we can add
even more overloads on our own, and simply let the compiler figure out from the call
parameter types which one is needed in a certain context.

In overloading an operator, we cannot change the operator’s arity, precedence or
associativity, but we can create versions of it with arbitrary formal parameter and return
types.

Operator overloading is simply a special case of function overloading. For example,
having the structs rational and extended_int available, we could declare the following
two functions in the same program, without creating a name clash: for any call to the
function square in the program, the compiler can find out from the call parameter type
which of the two functions we mean.

    // POST: returns a * a
    rational square (rational a);

    // POST: returns a * a
    extended_int square (extended_int a);

Function overloading in general is useful, but not nearly as useful as operator over-
loading. To define an overloaded operator, we have to use the functional operator
notation. In this notation, the name of the operator is obtained by appending its token
to the prefix operator. In case of the binary addition operator for the type rational,
this looks as follows and replaces the function add.

    // POST: return value is the sum of a and b
    rational operator+ (rational a, rational b)
    {
        rational result;
        result.n = a.n * b.d + a.d * b.n;
        result.d = a.d * b.d;
        return result;
    }
In Program 30, we can now replace line 34 by
\[
\text{rational } t = r + s; \quad \text{// equivalent to rational } t = \text{operator+}(r, s);
\]
Here, the comment refers to the fact that an operator can also be called in functional notation; in contrast, it appears in infix notation in \( r + s \). The call in functional notation can be useful for didactic purposes, since it emphasizes the fact that an operator is simply a special function; in an application, however, the point is to avoid functional notation and use the infix notation.

The other three basic arithmetic operations are similar, and here we only give their declarations.

\[
\begin{align*}
\text{// POST: return value is the difference of } a \text{ and } b \\
\text{rational operator- (rational } a, \text{ rational } b); \\
\text{// POST: return value is the product of } a \text{ and } b \\
\text{rational operator* (rational } a, \text{ rational } b); \\
\text{// POST: return value is the quotient of } a \text{ and } b \\
\text{// PRE: } b \neq 0 \\
\text{rational operator/ (rational } a, \text{ rational } b); \\
\end{align*}
\]

We can also overload the unary \(-\) operator; in functional operator notation, it has the same name as the binary version, but it has only one instead of two parameters. In the following implementation, we use the (modified) “local copy” of the call parameter \( a \) as the return value.

\[
\begin{align*}
\text{// POST: return value is } -a \\
rational \text{ operator- (rational } a) \\
\{ \\
\quad \text{a.n = } -\text{a.n; } \\
\quad \text{return } a; \\
\} \\
\end{align*}
\]

In order to compare rational numbers, we need the relational operators as well. Here is the equality operator as an example.

\[
\begin{align*}
\text{// POST: return value is true if and only if } a \text{ == } b \\
\text{bool operator== (rational } a, \text{ rational } b) \\
\{ \\
\quad \text{return a.n * b.d == a.d * b.n;} \\
\} \\
\end{align*}
\]

\subsection{4.1.6 Details}

Overloading resolution. If there are several functions or operators of the same name in a program, the compiler has to figure out which one is meant in a certain function call. This process is called overloading resolution and only depends on the types of the call
parameters. Overloading resolution is therefore done at compile time. There are two cases that we need to consider: we can either have an unqualified function call (like add \((r, s)\) in Program 30), or a qualified function call (like \(\text{std::sqrt}(2.0)\)). To process an unqualified function call of the form

\[
\text{fname} ( \text{expression1}, ..., \text{expressionN} )
\]

the compiler has to find a matching function declaration. Candidates are all functions \(f\) of name \(\text{fname}\) such that the function call is in the scope of some declaration of \(f\). In addition, the number of formal parameters must match the number of call parameters, and each call parameter must be of a type whose values can be converted to the corresponding formal parameter types.

In a qualified function call of the form

\[
\text{X::fname} ( \text{expression1}, ..., \text{expressionN} )
\]

where \(\text{X}\) is a namespace, only this namespace is searched for candidates.

**Argument-dependent name lookup (Koenig lookup).** There is one special rule that sometimes makes the list of candidates larger. If some call parameter type of an unqualified function call is defined in a namespace \(\text{X}\) (for example the namespace \(\text{std}\)), then the compiler also searches for candidates in \(\text{X}\). This is useful mainly for operators and allows them to be called unqualified in infix notation. The point of using operators in infix notation would be spoiled if we had to mention a namespace somewhere in the operator call.

**Resolution: Finding the best match.** For each candidate function and each call parameter, it is checked how well the call parameter type matches the corresponding formal parameter type. There are four quality levels, going from better to worse, given in the following list.

1. **Exact match.** The types of the call parameter and the formal parameter are the same.

2. **Promotion match.** There is a promotion from the call parameter type to the formal parameter type. We have seen some examples for promotions, like from \(\text{bool}\) to \(\text{int}\) and from \(\text{float}\) to \(\text{double}\).

3. **Standard conversion match.** There is a standard conversion from the call parameter type to the formal parameter type. We have seen that all fundamental arithmetic types can be converted into each other by standard conversions.

4. **User-defined conversion match.** There is a user-defined conversion from the call parameter type to the formal parameter type. We will get to user-defined conversions only later in this book.
A function \( f \) is called \emph{better} than \( g \) with respect to a parameter, if the match that \( f \) induces on that parameter is at least as good as the match induced by \( g \). If the match is really better, \( f \) is called \emph{strictly better} for the parameter.

A function \( f \) is called a \emph{best match} if it is better than any other candidate \( g \) in all parameters, and strictly better than \( g \) in at least one parameter.

Under this definition, there is at most one best match, but it may happen that there is no best match, in which case the function call is \emph{ambiguous}, and the compiler issues an error message.

Here is an example. Consider the two overloaded function declarations

\[
\begin{align*}
\text{void foo(double d);} \\
\text{void foo(unsigned int u);} \\
\end{align*}
\]

In the code fragment

\[
\begin{align*}
\text{float f = 1.0f;} \\
\text{foo(f);} \\
\end{align*}
\]

the first overload is chosen, since \texttt{float} can be promoted to \texttt{double}, but only standard-converted to \texttt{unsigned int}. In

\[
\begin{align*}
\text{int i = 1;} \\
\text{foo(i);} \\
\end{align*}
\]

the call is ambiguous, since \texttt{int} can be standard-converted to both \texttt{double} and \texttt{unsigned int}.

### 4.1.7 Goals

**Dispositional.** At this point, you should ...

1) know how structs can be used to aggregate several different types into one new type;
2) understand the difference between the syntactical and semantical value range of a struct;
3) know that C++ functions and operators can be overloaded.

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

(G1) define structs whose semantical value ranges correspond to that of given mathematical sets;
(G2) provide definitions of functions and overloaded operators on structs, according to given functionality;
(G3) write programs that define and use structs according to given functionality.
4.1.8 Exercises

Exercise 93 Define a type Tribool for three-valued logic; in three-valued logic, we have the truth values true, false, and unknown.

For the type Tribool, implement the logical operators

// POST: returns x AND y
Tribool operator&& (Tribool x, Tribool y);

// POST: returns x OR y
Tribool operator|| (Tribool x, Tribool y);

where AND (∧) and OR (∨) are defined according to the following two tables.

<table>
<thead>
<tr>
<th>∧</th>
<th>false</th>
<th>unknown</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>unknown</td>
<td>false</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>unknown</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>∨</th>
<th>false</th>
<th>unknown</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>unknown</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Test your type by writing a program that outputs these truth tables in some format of your choice.

Exercise 94 Define a type Z_7 for computing with integers modulo 7. Mathematically, this corresponds to the finite ring Z_7 = Z/7Z of residue classes modulo 7.

For the type Z_7, implement addition and subtraction operators

// POST: return value is the sum of a and b
Z_7 operator+ (Z_7 a, Z_7 b);

// POST: return value is the difference of a and b
Z_7 operator- (Z_7 a, Z_7 b);

according to the following table (this table also defines subtraction: x − y is the unique number z ∈ {0, . . . , 6} such that x = y + z).

<table>
<thead>
<tr>
<th>+</th>
<th>0 1 2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3 4 5 6</td>
</tr>
<tr>
<td>1</td>
<td>1 2 3 4 5 6 0</td>
</tr>
<tr>
<td>2</td>
<td>2 3 4 5 6 0 1</td>
</tr>
<tr>
<td>3</td>
<td>3 4 5 6 0 1 2</td>
</tr>
<tr>
<td>4</td>
<td>4 5 6 0 1 2 3</td>
</tr>
<tr>
<td>5</td>
<td>5 6 0 1 2 3 4</td>
</tr>
<tr>
<td>6</td>
<td>6 0 1 2 3 4 5</td>
</tr>
</tbody>
</table>

Exercise 95 Provide definitions for the following binary arithmetic operators on the type rational.
// POST: return value is the difference of a and b
rational operator - (rational a, rational b);

// POST: return value is the product of a and b
rational operator * (rational a, rational b);

// POST: return value is the quotient of a and b
// PRE: b != 0
rational operator / (rational a, rational b);

Exercise 96 Provide definitions for the following binary relational operators on the
type rational. In doing this, try to reuse operators that are already defined. (G2)(G3)

// POST: return value is true if and only if a != b
bool operator != (rational a, rational b);

// POST: return value is true if and only if a < b
bool operator < (rational a, rational b);

// POST: return value is true if and only if a <= b
bool operator <= (rational a, rational b);

// POST: return value is true if and only if a > b
bool operator > (rational a, rational b);

// POST: return value is true if and only if a >= b
bool operator >= (rational a, rational b);

Exercise 97 Provide definitions for the following binary arithmetic operators on the
type extended_int (Page 229), and test them in a program (for that it could be
helpful to provide an output operator for the type extended_int, and a function
that assigns to an extended_int value a value of type int). As in the previous
exercise, try to reuse code. (G2)(G3)

// POST: return value is the sum of a and b
extended_int operator + (extended_int a, extended_int b);

// POST: return value is the difference of a and b
extended_int operator - (extended_int a, extended_int b);

// POST: return value is the product of a and b
extended_int operator * (extended_int a, extended_int b);

// POST: return value is -a
extended_int operator - (extended_int a);
Exercise 98 Consider the following set of three functions.

```c
void foo(double, double) { ... } // function A
void foo(unsigned int, int) { ... } // function B
void foo(float, unsigned int) { ... } // function C
```

For each of the following function calls, decide to which of the functions \((A, B, C)\) it resolves to, or decide that the call is ambiguous. Explain your decisions! This exercise requires you to read the paragraph on overloading resolution in the Details section.

\(\begin{align*}
a) & \text{ foo(1, 1)} \\
b) & \text{ foo(1u, 1.0f)} \\
c) & \text{ foo(1.0, 1)} \\
d) & \text{ foo(1, 1u)} \\
e) & \text{ foo(1, 1.0f)} \\
f) & \text{ foo(1.0f, 1.0)}
\end{align*}\)
4.2 Type Variants

This section explains two ways of obtaining variants of a given type that have the same value range but differ in certain functionality aspects. Reference types enable functions to accept and return values and in particular change the values of their formal parameters. Const-types allow us to define values as being non-modifiable, in such a way that the compiler can detect illegal modifications. Reference types and const-types can be combined and naturally come up in implementing functionality for structs.

4.2.1 Reference types

Let us now try to implement the addition assignment operator += for the struct rational from Program 33. Here is an attempt:

```cpp
rational operator+= (rational a, rational b) {
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

With this, we can write

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

rational s;
s.n = 1; s.d = 3; // 1/3

r += s;
std::cout << r.n << "/" << r.d << "\n";
```

You may already see that the output of this will not be the desired 5/6. Recall from Section 3.1.3 what happens when r += s (equivalently, operator+= (r, s)) is evaluated: r and s are evaluated, and the resulting values are used to initialize the formal parameters a and b of the function operator+=. The values of r and s are not changed by the function call.

Hence, with the above implementation of operator+=, the value of the expression r += s is indeed 5/6, but the desired effect, the increment of r, does not happen. That’s why we get 1/2 as output in the above piece of code.

In order to implement operator+= properly, we must enable functions to change the values of their call parameters. Surprisingly, we do not need a new concept for that on the function side; we simply need a new category of types.
4.2. TYPE VARIANTS

Definition. If \( T \) is any type, then

\[
T&
\]

is the corresponding reference type (read \( T& \) as “\( T \) reference” or “reference to \( T \)”). In value range and functionality, \( T& \) is identical to \( T \). The difference is in the initialization and assignment semantics.

A variable of reference type \( T& \) (also called a reference) can be initialized only from an lvalue of type \( T \), or any type whose values can be converted to \( T \). The initialization makes it an alias of the lvalue: another name for the object behind the lvalue. We also say that the reference refers to that object. The following example shows this.

```cpp
int i = 5;
int& j = i; // j becomes an alias of i
j = 6; // changes the value of i
std::cout << i << "\n"; // outputs 6
```

A reference cannot be changed to refer to another object after initialization. If we later assign something to the reference, we in fact assign to the object referred to by it. In writing \( j = 6 \) in the above piece of code, we therefore change the value of \( i \) to 6, since \( j \) is an alias of \( i \).

Internally, a value of type \( T& \) is represented by the address of the object it refers to. This explains why we need an lvalue to initialize a reference type variable, and why things like

```cpp
int& j; // error: j must be an alias of something
int& k = 5; // error: the literal 5 has no address
```

don’t work. Any expression of reference type is an lvalue itself. We can therefore use a reference \( r \) to initialize another reference \( rr \), but then we don’t get a reference to \( r \), but another reference to the object referred to by \( r \):

```cpp
int i = 5;
int& j = i; // j becomes an alias of i
int& k = j; // k becomes another alias of i
```

### 4.2.2 Call by value and call by reference

When a function has a formal parameter of reference type, the corresponding call parameter must be an lvalue; when the function call is evaluated, the initialization of the formal parameter makes it an alias of the call parameter. In this way, we can implement functions that change the values of their call parameters. Here is an example.

```cpp
void increment (int& i)
{
    ++i;
```
4.2.3 Return by value and return by reference

If a formal parameter of a function has reference type, we have call-by-reference semantics with respect to that parameter. Equivalently, we say that we pass the parameter by reference.

If the formal parameter is not of reference type, we have call-by-value semantics: we pass the parameter by value. Under call by reference, the address of (or a reference to) the call parameter is used to initialize the formal parameter; under call-by-value semantics, it is the value of the call parameter that is used for initialization.

The basic rule is to pass a parameter by reference only if the function in question actually needs to change the call parameter value. If that is not the case, call by value is more flexible, since it allows a larger class of call parameters (lvalues and rvalues instead of lvalues only).

4.2.3 Return by value and return by reference

The return type of a function can be a reference type as well, in which case we have return-by-reference semantics (otherwise, we return by value). If the function returns a reference, the function call expression is an lvalue itself, and we can use it wherever lvalues are expected.

This means that the function itself chooses (by using reference types or not) whether its call parameters and return value are lvalues or rvalues. Section 2.1.13 and Section 2.2.4 document these choices for some of the operators on fundamental types, but only now we understand the mechanism that makes such choices possible.

As a concrete example, let us consider the following version of the function increment that exactly models the behavior of the pre-increment operator ++: it increments its lvalue parameter and returns it as an lvalue.

```cpp
int& increment (int& i)
{
    return ++i;
}
```

In general, we must make sure that an expression of reference type that we return refers to a non-temporary object. To understand what a temporary object is, let us consider the following function.
4.2. Type Variants

int& foo (int i)
{
    return i;
}

This is asking for trouble, since the formal parameter i runs out of scope when the
function call terminates. This means that the associated memory is freed and the address
expires (see Section 2.4.3). If we now write for example

```cpp
int i = 3;
int& j = foo(i);  // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

the reference j refers to an expired object, and the resulting behavior of the program is
undefined.

Reference Guideline: Whenever you create an alias for an object, ensure that the object
does not expire before the alias.

The compiler usually notices violations of the Reference Guideline and issues a warn-
ing.

4.2.4 More user-defined operators

Rational numbers: addition assignment. Let's get back to the addition assignment operator
for our new struct rational. In order to fix our failed attempt from the beginning of
this section, we need to add two characters only.

As in the previous function increment, the formal parameter a must be passed as a
reference, and to be compliant with the usual semantics of +=, we also return the result
as a reference:

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

The other arithmetic assignment operators are similar, and we don't list them here ex-
plicitly. Together with the arithmetic and relational operators discussed in Section 4.1.5,
we now have a useful set of operations on rational numbers.

Rational numbers: input and output. Let us look at Program 33 once more, with the
function name add replaced by operator+ and the function call add (r, s) replaced by
r + s. Still, we can spot potential improvements: instead of writing
\begin{verbatim}
std::cout << "Sum is " << t.n << "/" << t.d << "\n";
\end{verbatim}

in line 37, we'd rather write
\begin{verbatim}
std::cout << "Sum is " << t << "\n";
\end{verbatim}

just like we are doing it for fundamental types. After all, we want to think of a rational
number as a single value from the set \( \mathbb{Q} \) and not as two values from the set \( \mathbb{Z} \).

From what we have done above, you can guess that all we have to do is to overload
the output operator \( \ll \). In discussing the output operator in Section 2.1.13 we have
argued that the output stream passed to and returned by the output operator must be
an lvalue, since the output operator modifies the stream. Having reference types at our
disposal, this can easily be done: we simply pass and return the output stream (whose
type is \texttt{std::ostream}) as a reference:

\begin{verbatim}
// POST: a has been written to o
std::ostream& operator<<(std::ostream& o, rational r)
{
    return o << r.n << "/" << r.d;
}
\end{verbatim}

There is no reason to stop here: for the input, we would in the same fashion like
to replace the two input statements \texttt{std::cin \gg r.n; and std::cin \gg r.d;}
by the single statement
\begin{verbatim}
std::cin >> r;
\end{verbatim}

(and the same for the input of s). Again, we need to pass and return the input stream
(of type \texttt{std::istream}) as a reference. In addition, we must pass the rational
number that we want to read as a reference, since the input operator has to modify its value.

The operator first reads the numerator from the stream, followed by a separating
character, and finally the denominator. Thus, we can read a rational number in one go
by entering for example 1/2.

\begin{verbatim}
// POST: r has been read from i
// PRE: i starts with a rational number of the form "n/d"
std::istream& operator>>(std::istream& i, rational& r)
{
    char c; // separating character, e.g. '/'
    return i >> r.n >> c >> r.d;
}
\end{verbatim}

In contrast to \texttt{operator<<}, things can go wrong, e.g., if the user enters the character
sequence "A/B" when prompted for a rational number. Also, we probably don't want to
accept \( 3.4 \) as a rational number as our input operator does. There are mechanisms
to deal with such issues, but we won't discuss them here.

Let us conclude this section with a beautified version of Program 33. What makes
this version even nicer is the fact that in the main function, the new type is used exactly
like an "atomic" fundamental type such as \texttt{int}. 

In the spirit of Section 3.1.8 on modularization, we actually split the program into three files: a file rational.h that contains the definition of the struct rational, along with declarations of the overloaded operators; a file rational.C that contains the definitions of these operators; finally, a file userational2.C that contains the main program. At the same time, we put our new type rational and the operations on it into namespace ifm in order to avoid possible name clashes. Exercise 99 asks you to actually integrate the new rational number type into the math library that you have built in Exercise 79, so that Program 31 below can be compiled using this library.

```c
1 // Program: userational2.C
2 // Add two rational numbers.
3 #include <iostream>
4 #include <IFM/rational.h>
5
6 int main ()
7 {
8     // input
9     std::cout << "Rational number r:\n";
10    ifm::rational r;
11    std::cin >> r;
12
13    std::cout << "Rational number s:\n";
14    ifm::rational s;
15    std::cin >> s;
16
17     // computation and output
18    std::cout << "Sum is " << r + s << ".\n";
19
20    return 0;
21 }
```

Program 31: progs/userational2.C

```c
1 // Program: rational.h
2 // Define a type for rational numbers, and declare
3 // operations on it.
4 #include <iostream>
5
6 namespace ifm {
7
8     // the new type rational
```

---

2This might make you wonder why we can write the expression r + s in Program 31, without mentioning the namespace in which the operator+ in question is defined. The Details of Section 4.1 explains this in the paragraph on argument-dependent name lookup.
```c
struct rational {
    int n;
    int d; // INV: d != 0
};

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

// POST: a has been written to o
std::ostream & operator<< (std::ostream & o, rational a);

// POST: a has been read from i
// PRE: i starts with a rational number of the form "n/d"
std::istream & operator>> (std::istream & i, rational & a);

Program 32: progs/rational.h

#include <IFM/rational.h>

namespace ifm {

    // POST: return value is the sum of a and b
    rational operator+ (rational a, rational b) {
        rational result;
        result.n = a.n * b.d + a.d * b.n;
        result.d = a.d * b.d;
        return result;
    }

    // POST: a has been written to o
    std::ostream & operator<< (std::ostream & o, rational a) {
        return o << a.n << "/" << a.d;
    }

    // POST: a has been read from i
    // PRE: i starts with a rational number of the form "n/d"
    std::istream & operator>> (std::istream & i, rational & a) {
        char c; // separating character, e.g. '/'
    }
}
4.2.5 Const-types

Let us come back to the addition operator for rational numbers from Program 33. Although this operator does not intend to change the values of its call parameters, the efficiency fanatic in you might suggest to speed up this operator by using reference types anyway:

```c
// POST: return value is the sum of a and b
rational operator+ (rational& a, rational& b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

Indeed, this version is correct and potentially faster than the previous one, since the initialization of a formal parameter is done by copying just one address, rather than two int values as in the member-wise copy that takes place under the call-by-value semantics. Even if the saving is small in this example, you can imagine that member-wise copy can be pretty expensive in structs that are more elaborate than rational; in contrast, call by reference is fast for all types, even the most complicated ones.

Unfortunately, the call parameters must be lvalues under call by reference, so we can't write the expression `a + b + c`, for example, even if `a`, `b`, `c` are variables of type rational (why?). Still, the faster version might work in our application; it does so in Program 31, for example, since in this program, we call `operator+` with lvalue operands only.

One less obvious (and much more dangerous) problem remains, though: in passing the parameters as references, we allow the operator to change the values of its call parameters in the first place, even if that happens unintentionally. In functions that are
larger than the above operator+, it can easily happen that we modify some of the call parameters simply by mistake.

Not making such mistakes is the prime responsibility of the programmer, of course, but the clever programmer calls the programming language for help whenever possible. In this spirit, the above "efficiency fix" for operator+ is a bad move, since it introduces a new possible source of errors.

If this sounds too abstract for you, here is an example where it is simply wrong to move to call-by-reference semantics; the compiler has no chance to detect this error since it is purely semantical. Consider the unary subtraction operator for the type rational from Section 4.1.5.

```cpp
// POST: return value is -a
rational operator- (rational a)
{
    a.n = -a.n;
    return a;
}
```

Changing this to

```cpp
rational operator- (rational& a)
{
    a.n = -a.n;
    return a;
}
```

has a drastic (and undesired) consequence: the expression -a will still have the same value as before, but it will have the additional effect of changing the value of a. We have "accidentally" created a completely different operator.

As many other high-level programming languages, C++ offers a mechanism that—if properly used—allows the compiler to detect undesired changes of values as in the previous example. The idea is to promise that a certain value will not be changed, and then let the compiler check whether we keep our promise. In the call-by-reference version of the unary subtraction operator, the (false) promise can be given as follows, using the keyword const.

```cpp
rational operator- (const rational& a)
{
    a.n = -a.n; // error: a was promised to be constant
    return a;
}
```

In compiling this variant of the operator, the compiler will issue an error message, pointing out the mistake. We can then fix it by either going back to call-by-value semantics, or by introducing a result variable like in operator+ above.

From the strictly functional point of view, this promise mechanism is superfluous, and there are programming languages in use that don't have it (C used to be such a
language, until the \texttt{const} keyword was added in 1999, motivated by its success in C++.\footnote{Indeed, “Real programmers” (as described in the classic article \textit{Real programmers don’t use PASCAL}) would leave it to the “Quiche eaters” to use such a mechanism.} Also, nobody forces us to make use of the promise mechanism.\footnote{An rvalue of the type (or of any type, for that matter), has constant value anyway.} But the whole point of high-level programming languages is to make the programmer’s life easier; the compiler is our friend and can help us to avoid many time-consuming errors. The \texttt{const} mechanism is like a check digit: by providing additional redundant data (the \texttt{const} keyword), we make sure that inconsistencies in the whole data set (the program) are automatically detected.

\textbf{Definition.} If $T$ is any type, then

\begin{center}
\texttt{const }$T$
\end{center}

is the \textit{const-qualified} type (const-type for short) of $T$, and $T$ itself is the \textit{underlying type}. The const-qualified version of $T$ has exactly the same value range and functionality as $T$. The only difference is that an expression of const-type is not allowed to change its value (in other words, it is constant); this is our promise, and the compiler checks whether we keep that promise.

If we write for example

\begin{verbatim}
const int n = 5;
int n = 6;
\end{verbatim}

the compiler will issue an error message concerning the assignment $n = 6$, since $n$ has the const-type \texttt{const int}.

Values of const-type must always be initialized. Writing

\begin{verbatim}
const int n; // error: uninitialized constant
\end{verbatim}

is illegal (and makes no sense, since we can never assign a value to $n$ later).

\subsection{What exactly is constant?}

Let us consider some lvalue of type \texttt{const }$T$. If the underlying type $T$ is not a reference type, then the lvalue is associated with a constant \texttt{object}. For example, the declaration

\begin{verbatim}
const int n = 5;
int ik = n; // error: const-qualification is discarded
i = 6;
\end{verbatim}

\begin{verbatim}
\end{verbatim}
We cannot use an expression of type `const T` to initialize (or assign to) an expression of type `T&`, since that would create a modifiable alias for an object that was promised to be constant.

On the other hand, an lvalue (actually, any expression) of type `const T&` is the alias of an object, but that object is not necessarily constant itself. The const-qualification in this case is merely a promise that the object’s value will not be modified through the alias in question. Here is an example that illustrates this point.

```cpp
int n = 5;
const int & i = n; // i becomes a non-modifiable alias of n
int& j = n; // j becomes a modifiable alias of n
i = 6; // error: n is modified through const-reference
j = 6; // ok: n receives value 6
```

Here, we do not have a constant `object`, but a constant `expression` (namely `i`). An expression of type `const T&` is also called a `const-reference`.

### 4.2.7 Const-references

First of all, the typename `const T&` is paranthesized as `const (T&)`, i.e. we get constant values of reference type. Const-references are very useful and often appear in real-life code. Let us come back to our faster version of `operator+` for rational numbers. Its “safe” version is this:

```cpp
// POST: return value is the sum of a and b
rational operator+ (const rational& a, const rational& b) {
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

The fact that this compiles confirms that we are not changing the values of the formal parameters `a` or `b` within the body of this function. But there was another problem that we apparently didn’t solve yet: passing parameters by reference requires lvalues as call parameters, and this severely restricts the applicability of the operator. Fortunately, this is a non-issue: const-references (in particular, formal parameters of const-reference-type) can be initialized from rvalues as well. This means that we can write

```cpp
const int & i = 3;
```

Behind the scenes, the compiler creates a temporary object that holds the value 3, and the address of this temporary object is used to initialize the const-reference `i`. The compiler makes sure that the temporary object does not expire before the const-reference that refers to it (see the Reference Guideline on Page 243).

The same happens when a formal function parameter of const-reference-type is initialized from an rvalue.
A parameter of type `const T&` is therefore the all-in-one device suitable for every purpose: if the call parameter is an lvalue, the initialization is very efficient (only its address needs to be copied), and otherwise, we essentially fall back to call-by-value semantics.

Despite this, there are still situations where `T` is preferrable over `const T&` as a parameter type. If `T` is a fundamental type or a struct with small memory requirements, it does not pay off to move to `const T&`, since the saving in handling lvalue parameters is so small (or even nonexistent) that it won't compensate for the (slightly) more costly access to the formal function parameter in the function body. Indeed, call by reference adds one indirection: to look up the value of a formal function parameter under call-by-reference semantics, we first have to look up its address and then look up the actual value at that address. Under call-by-value semantics, the address of the value is “hardwired” (and refers to some object on the call stack, see Section 3.2.2).

Also, it is often convenient to use the formal parameter as a local variable and modify its value (see `operator+` above); for that, its type must not be a `const-type`.

### 4.2.8 Const-types as return types.

Const-types may also appear as return types of functions, just like any other types. In that case, the `const` promises that the function call expression itself is constant.

If the return type is not a reference type, the function call expression is an lvalue and hence not modifiable anyway. In this case, the `const` keyword is legal but has no effect. Const-types therefore only make a difference if the function returns a reference.

Note that it is not generally valid to replace return type `T` by `const T&`; while this syntactically works for the formal parameter types, it can for the return type result in syntactically correct but semantically wrong code.

As an example, let's replace `rational` by `const rational&` as the return type of `operator+`:

```cpp
class rational { /*...*/ };

class rational& operator+ (const rational& a, const rational& b) {  
    rational result;  
    result.n = a.n * b.d + a.d * b.n;  
    result.d = a.d * b.d;  
    return result;  
}
```

In executing the `return` statement, the return value (in this case a const-reference) to be passed to the caller of the function is initialized with the expression `result`. Now recall that the initialization of a (const-)reference from an lvalue simply makes it an alias of the lvalue. But the lvalue in question (namely `result`) is a local variable whose memory is freed and whose address becomes invalid when the function call terminates (see Section 2.4.3 and Section 4.2.1). The consequence is that the returned reference will be the alias of an expired object, and using this reference results in undefined behavior of the program.
CHAPTER 4. COMPOUND TYPES

Errors like this are very hard to find (and we cannot reliably count on compiler warnings here), since the program may work as intended, for example if the memory that was associated to the expired object is not immediately reused. But on another platform, the program may behave differently or even crash.

4.2.9 When to use const?

Whenever you think about the appropriate type of a variable, a formal function parameter, or a function’s return value, it is good practice to think about const-qualification at the same time. After all, you should know what you want to do with the variable, parameter, or return value (if you don’t, this paragraph is even more important), so you also know whether the program needs to change its value at some point.

The basic rule to follow is this:

**Const Guideline:** Use const-types whenever this is possible and makes a difference. It always makes a difference in connection with reference types.

Indeed, it is more than the promise of constant value that distinguishes the type `const T&` from `T&`: while we need lvalues to initialize and assign to objects of type `T&`, rvalues suffice for `const T&`. We have also argued that `const T&` is preferable to `T` in many situations, simply for efficiency reasons. You cannot ignore these facts, even if you don’t care about the promise mechanism otherwise.

If `T` is not a reference type, then the question whether `const T` makes a difference from `T` has usually not such a clear answer, with one exception: in return types of functions that do not return references, the `const` keyword really makes no difference and should therefore be omitted.

In the same spirit, the `const` keyword is typically omitted for formal function parameters that are not references. In this situation, `const` is *not* redundant, though: if a formal parameter is of `const`-type, we promise not to use the formal parameter as a modifiable local variable. But this promise is neither necessary to prevent accidental modification of the call parameter (call by value already takes care of this), nor does it influence the outside behavior of the function in any way. In fact, if you write functions for a library (see Section 3.1.8), you better refrain from such `const`-type usage, as it unnecessarily restricts you: if you later decide to change the function definition, you are committed to the `const`-type parameter (even if this turns out to be impractical), unless you also change the header file that contains the function declaration.

Also, not all variables that could be declared `const` in a program are typically done so, simply because it makes (or appears to make) no difference in the context of the declaration. As an example, consider line 34 in Program 30: it is possible to declare the variable `t` as being of `const`-type `const rational`, but it doesn’t make a difference, since this variable occurs once only afterwards, and this occurrence is just three lines below.

For concreteness, let us stipulate that a variable that is meant to have constant value should definitely get `const`-type if its scope spans more than 10 lines of code.
4.2. TYPE VARIANTS

4.2.10 Goals

Dispositional. At this point, you should ...

1) understand the alias concept behind reference types and the Reference Guideline;
2) understand the difference between call by value and call by reference semantics for
   function parameters;
3) understand const-types and the Const Guideline.

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

(G1) state exact pre-and postconditions for functions involving formal parameter types
   or return types of reference and/or const-type;
(G2) write functions that modify (some of) their call parameters;
(G3) find syntactical and semantical errors in programs that are due to improper han-
   dling of reference types;
(G4) find syntactical and semantical errors in programs that are due to improper han-
   dling of const-types;
(G5) find the declarations in a given program whose types should be const-according
   to the Const Guideline.

4.2.11 Exercises

Exercise 99 Build an object code file rational.o from Program 33 and integrate it
into the library libmath.a that you have created in Exercise 79. Compile the main
program Program 31 using this library.

Exercise 100 Consider the following family of functions:

```c
T foo (S i)
{
    return ++i;
}
```

with T being one of the types int, intk and const intk, and S being one of the
types int, const int, intk and const intk. (This defines 12 different functions).

a) Find the combinations of T and S for which the resulting function definition
   is syntactically valid, and explain your answer.

b) Among the combinations found in a), find the combinations of T and S for
   which the resulting function definition is also semantically valid, meaning that
   function calls always have well-defined value and effect; explain your answer.
c) For all combinations found in b), give precise postconditions for the corresponding function foo.

(G1)(G3)(G4)

Exercise 101 Write a function that swaps the values of two int-variables. (G2)
For example,

```cpp
int a = 5;
int b = 6;
// here comes your function call
std::cout << a << "\n"; // outputs 6
std::cout << b << "\n"; // outputs 5
```

Exercise 102 We want to have a function that normalizes a rational number, i.e. transforms it into the unique representation in which numerator and denominator are relatively prime, and the denominator is positive. For example,

\[
\frac{21}{-14}
\]

is normalized to

\[
\frac{-3}{2}.
\]

There are two natural versions of this function:

// POST: r is normalized
void normalize (rational& r);

// POST: return value is the normalization of r
rational normalize (const rational& r);

Implement one of them, and argue why you have chosen it over the other one.

Hint: you may want to use the function gcd from Section 3.2, modified for parameters of type int (how does this modification look like?). (G2)(G2)

Exercise 103 Provide a definition of the following function.

// POST: return value indicates whether the linear equation
// a * x + b = 0 has a real solution x ; if true is
// returned, the value s satisfies a * s + b = 0
bool solve (double a, double b, double& s);

Test your function in a program for at least the pairs \((a, b)\) from the set

\[\{(2, 1), (0, 2), (0, 0), (3, -4)\}\].

(G2)
Exercise 104 Reconsider the following programs and identify the declarations (of variables or formal parameters) in which you could replace a type $T$ by its const-version $\text{const } T$.

a) Program 1 (Page 18)
b) Program 6 (Page 73)
c) Program 26 (Page 183)
d) Program 27 (Page 213)
e) Program 30 (Page 226)

Exercise 105 Find all mistakes (if any) in the following programs, and explain why these are mistakes. All programs share the following two function definitions and only differ in their main functions.

```c
int foo (int & i) {
    return i += 2;
}

const int & bar (int & i) {
    return i += 2;
}
```

a) int main()
   {
      const int i = 5;
      int & j = foo (i);
   }

b) int main()
   {
      int i = 5;
      const int & j = foo (i);
   }

c) int main()
   {
      int i = 5;
      const int & j = bar (foo (i));
   }
d) int main()
   {
       int i = 5;
       const int& j = foo( bar(i));
   }

e) int main()
   {
       int i = 5;
       const int j = bar(++,i);
   }

4.2.12 Challenges

Exercise 106 Implement an integral type bigint whose values have an arbitrary (but fixed) number of digits, for example 1,000 (recall that int values have 32 binary digits on many platforms). The type should have the operators +, -, *, /, \% along with their assignment versions, and an output operator.

Write a program to test your number type. For example, given a and b, it should always hold that a = (a / b) * b + a \% b.

Use the type bigint to finally answer all the nagging questions that have haunted you for so long: what is the twentieth power of 3? What is F_{50}, the fiftieth Fibonacci number, etc.?