16. Structs and Classes I

Rational Numbers, Struct Definition, Overlading Functions and Operators, Const-References, Encapsulation

Calculating with Rational Numbers

- Rational numbers ($\mathbb{Q}$) are of the form $\frac{n}{d}$ with $n$ and $d$ in $\mathbb{Z}$
- C++ does not provide a built-in type for rational numbers

Goal

We build a C++-type for rational numbers ourselves! 😊

Vision

How it could (will) look like

```cpp
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;
std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

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

A First Struct

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

Invariant: specifies valid value combinations (informal).

- struct defines a new type
- formal range of values: cartesian product of the value ranges of existing types
- real range of values: $\text{rational} \subseteq \mathbb{Z} \times \mathbb{Z}$.
Accessing Member Variables

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

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;
}
```

A First Struct: Functionality

A struct defines a new type, not a variable!

```cpp
// new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (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;
}
```

Meaning: every object of the new type is represented by two objects of type int the objects are called n and d.

Input

```cpp
// Input r
rational r;
std::cout << "Rational number r:\n";
std::cout << " numerator =? ";
std::cin >> r.n;
std::cout << " denominator =? ";
std::cin >> r.d;
// Input s the same way
rational s;
...
```

Vision comes within Reach ...

```cpp
// computation
const rational t = add (r, s);

// output
std::cout << "Sum is " << t.n << "/" << t.d << ".\n";
```
**Struct Definitions**

```c
struct T {
    T1 name1;
    T2 name2;
    ...
    Tn namen;
};
```

- **name of the new type (identifier)**
- **names of the underlying types**
- **names of the member variables**

Range of Values of $T$: $T_1 \times T_2 \times \ldots \times T_n$

---

**Struct Definitions: Examples**

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

- **underlying types can be fundamental or user defined**

```c
struct extended_int {
    // represents value if is_positive==true
    // and -value otherwise
    unsigned int value;
    bool is_positive;
};
```

- **the underlying types can be different**

---

**Structs: Accessing Members**

```c
expr.namek
```

- **expression of struct-type $T$**
- **name of a member-variable of type $T$**

- **expression of type $T_k$: value is the value of the object designated by $name_k$**
- **member access operator .**
### Structs: Initialization and Assignment

**Default Initialization:**

```cpp
rational t;
```

- Member variables of `t` are default-initialized
- for member variables of fundamental types nothing happens (values remain undefined)

**Initialization:**

```cpp
rational t = {5, 1};
```

- Member variables of `t` are initialized with the values of the list, according to the declaration order.

**Assignment:**

```cpp
rational s;
...

rational t = s;
```

- The values of the member variables of `s` are assigned to the member variables of `t`.

**Initialization:**

```cpp
rational t = add (r, s);
```

- `t` is initialized with the values of `add(r, s)`
**Structs: Initialization and Assignment**

**Assignment:**

```c++
rational t;
t = add (r, s);  
```

- t is default-initialized
- The value of add (r, s) is assigned to t

**Comparing Structs?**

For each fundamental type (int, double,...) there are comparison operators == and !=, not so for structs! Why?

- member-wise comparison does not make sense in general...
- ...otherwise we had, for example: \( \frac{2}{3} \neq \frac{4}{6} \)

**Structs as Function Arguments**

```c++
void increment(rational dest, const rational src)
{
    dest = add (dest, src); // modifies local copy only
}
```

Call by Value!

```c++
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment (b, a); // no effect!
std::cout << b.n << "/" << b.d; // 1 / 2
```
Structs as Function Arguments

```c
void increment(rational & dest, const rational src) {
    dest = add (dest, src);
}
```

Call by Reference

```c
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment (b, a);
std::cout << b.n << " / " << b.d; // 2 / 2
```

User Defined Operators

Instead of

```c
rational t = add(r, s);
```

we would rather like to write

```c
rational t = r + s;
```

This can be done with *Operator Overloading*.

Overloading Functions

- Functions can be addressed by name in a scope
- It is even possible to declare and to defined several functions with the same name
- the “correct” version is chosen according to the *signature* of the function.

Function Overloading

A function is defined by name, types, number and order of arguments

```c
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```

- the compiler automatically chooses the function that fits “best” for a function call (we do not go into details)

```c
std::cout << sq (3); // compiler chooses f2
std::cout << sq (1.414); // compiler chooses f1
std::cout << pow (2); // compiler chooses f4
std::cout << pow (3,3); // compiler chooses f3
```
Operator Overloading

- Operators are special functions and can be overloaded
- Name of the operator \( op \):
  \[
  \text{operator} \ op
  \]
- We already know that, for example, operator+ exists for different types

Adding rational Numbers – Before

```cpp
// 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;
}
...
const rational t = add (r, s);
```

Adding rational Numbers – After

```cpp
// 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;
}
...
const rational t = r + s;
```

Other Binary Operators for Rational Numbers

```cpp
// POST: return value is 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);
```
**Unary Minus**

has the same symbol as the binary minus but only one argument:

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

**Comparison Operators**

are not built in for structs, but can be defined

```cpp
// POST: returns true iff a == b
bool operator== (rational a, rational b) {
    return a.n * b.d == a.d * b.n;
}
```

\[
\frac{2}{3} = \frac{4}{6} \quad \checkmark
\]

**Arithmetic Assignment**

We want to 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; // 5/6
```

**Operator+=  First Trial**

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

does not work. Why?

- The expression `r += s` has the desired value, but because the arguments are R-values (call by value!) it does not have the desired effect of modifying `r`.
- The result of `r += s` is, against the convention of C++ no L-value.
**Operator +=**

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

The L-value `a` is increased by the value of `b` and returned as L-value:

```
r += s; now has the desired effect.
```

**In/Output Operators**

can also be overloaded.

- **Before:**
  ```cpp
  std::cout << "Sum is "
            << t.n << "/" << t.d << "\n";
  ```

- **After (desired):**
  ```cpp
  std::cout << "Sum is "
            << t << "\n";
  ```

**In/Output Operators**

can be overloaded as well:

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

writes `r` to the output stream and returns the stream as L-value.

**Input**

```cpp
// PRE: in starts with a rational number of the form "n/d"
// POST: r has been read from in
std::istream& operator>> (std::istream& in, rational& r) {
    char c; // separating character '/'
    return in >> r.n >> c >> r.d;
}
```

reads `r` from the input stream and returns the stream as L-value.
Goal Attained!

// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;

std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

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

Recall: Large Objects ...

struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};
void outputState (SimulatedCPU p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (unsigned int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
call by value: more than 256k get copied!

... are Better Passed as Const-Reference

struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};
void outputState (const SimulatedCPU& p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (unsigned int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
call by reference: only the address gets copied.

A new Type with Functionality...

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) {
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
…should be in a Library!

rational.h:
- Definition of a struct rational
- Function declarations

rational.cpp:
- arithmetic operators (operator+, operator+=, ...)
- relational operators (operator==, operator>, ...)
- in/output (operator >>, operator <<, ...)

Thought Experiment

The three core missions of ETH:
- research
- education
- technology transfer

We found a startup: RAT PACK®!
- Selling the rational library to customers
- ongoing development according to customer’s demands

The Customer is Happy

…and programs busily using rational.
- output as double-value \( \left( \frac{7}{5} \rightarrow 0.6 \right) \)

// POST: double approximation of r
double to_double (rational r)
{
    double result = r.n;
    return result / r.d;
}

The Customer Wants More

“Can we have rational numbers with an extended value range?”
- Sure, no problem, e.g.:

```
struct rational {
    int n;
    int d;
};
⇒
struct rational {
    unsigned int n;
    unsigned int d;
    bool is_positive;
};
```
**New Version of RAT PACK®**

*It sucks, nothing works any more!*
- What is the problem?
- $-\frac{3}{5}$ is sometimes 0.6, this cannot be true!
- That is your fault. Your conversion to double is the problem, our library is correct.

*Up to now it worked, therefore the new version is to blame!*

---

**Liability Discussion**

```c
// POST: double approximation of r
double to_double (rational r){
double result = r.n;
    return result / r.d;
}
```

correct using...

```c
struct rational {
    int n;
    int d;
};
r.is_positive and result.is_positive do not appear.
```

...not correct using

```c
struct rational {
    unsigned int n;
    unsigned int d;
    bool is_positive;
};
```

---

**We are to Blame!!**

- Customer sees and uses our representation of rational numbers (initially $r.n$, $r.d$)
- When we change it ($r.n$, $r.d$, $r.is_positive$), the customer’s programs do not work anymore.
- No customer is willing to adapt the programs when the version of the library changes.

⇒ RAT PACK® is history...

---

**Idea of Encapsulation (Information Hiding)**

- A type is uniquely defined by its *value range* and its *functionality*
- The *representation* should not be visible.
- ⇒ The customer is not provided with *representation* but with *functionality!*

```c
str.length(), v.push_back(1), ...
```
Classes

- provide the concept for encapsulation in C++
- are a variant of structs
- are provided in many object oriented programming languages

Encapsulation: public/private

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

*only* difference

- `struct`: by default *nothing* is hidden
- `class`: by default *everything* is hidden

*is used instead of struct if anything at all shall be “hidden”*