19. Classes

Overloading Functions and Operators, Encapsulation, Classes, Member Functions, Constructors

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

- 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)

- 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:

  operator op

- we already know that, for example, operator+ exists for different types
Adding rational Numbers – Before

// 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

// 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

// 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:

// 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;
}
```

2/3 = 4/6 ✓

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;
}
```

this works

- 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";
```

writes \( r \) to the output stream and returns the stream as L-value.

Input

// PRE: in starts with a rational number
// of the form "n/d"
// POST: r has been read from in
```cpp
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
```cpp
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;
```

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

\( r \) and \( s \) are added and output.
A new Type with Functionality...

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

// POST: return value is the sum of a and b
```c
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!

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

```c
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 (\( \frac{3}{5} \rightarrow 0.6 \))

```c
// 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.:

  ```c
  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

// 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;
  };
  ```

- ...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!

Encapsulation: public/private

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

only difference

struct: by default nothing is hidden
class: by default everything is hidden

Application Code

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

Classes

provide the concept for encapsulation in C++
are a variant of structs
are provided in many object oriented programming languages
**Member Functions: Declaration**

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

**Member Functions: Call**

```cpp
// Definition des Typs
class rational {
    // ...
};
...
// Variable des Typs
rational r;
int n = r.numerator(); // Zaehler
int d = r.denominator(); // Nenner
```

**Member Functions: Definition**

```cpp
// POST: returns numerator of this instance
int numerator () const { return n; }
```

**const and Member Functions**

```cpp
class rational {
public:
    int numerator () const { return n; }
    void set_numerator (int N) { n = N; }
    rational x;
    x.set_numerator(10); // ok;
    const rational y = x;
    int n = y.numerator(); // ok;
y.set_numerator(10); // error;
...
}
```

The `const` at a member function is to promise that an instance cannot be changed via this function.

`const` items can only call `const` member functions.
Comparison

Roughly like this it were ...

class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return this->n;
    }
};
rational r;
...
std::cout << r.numerator();

without member functions

struct bruch {
    int n;
    ...
};
int numerator (const bruch& dieser)
{
    return dieser.n;
}
bruch r;
...
std::cout << numerator(r);

Member-Definition: In-Class vs. Out-of-Class

class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return n;
    }
};

this also works.

No separation between declaration and definition (bad for libraries)

Initialisation? Constructors!

class rational {
    int n;
    ...
public:
    rational (int num, int den)
    : n (num), d (den)
    {
        assert (den != 0);
    }
};
rational r (2,3); // r = 2/3

Constructors

■ are special member functions of a class that are named like the class
■ can be overloaded like functions, i.e. can occur multiple times with varying signature
■ are called like a function when a variable is declared. The compiler chooses the “closest” matching function.
■ if there is no matching constructor, the compiler emits an error message.
Constructors: Call

- directly
  ```
  rational r (1,2); // initialisiert r mit 1/2
  ```
- indirectly (copy)
  ```
  rational r = rational (1,2);
  ```

Initialisation “rational = int”?

```cpp
class rational
{
public:
    rational (int num)
    : n (num), d (1)
    {}
    ... 
};
...
rational r (2); // explicit initialization with 2
rational s = 2; // implicit conversion
```

User Defined Conversions

are defined via constructors with exactly one argument

```cpp
rational (int num)
    : n (num), d (1)
    {}
```

```cpp
rational r = 2; // implizite Konversion
```

The Default Constructor

```cpp
class rational
{
public:
    empty list of arguments
    ... 
    rational ()
    : n (0), d (1)
    {}
    ...
    ... 
    rational r; // r = 0
    
    ⇒ There are no uninitialzed variables of type rational any more!
```
**Alterantively: Deleting a Default Constructor**

```cpp
class rational
{
public:
...
    rational () = delete;
...}
```

```cpp
rational r; // error: use of deleted function 'rational::rational()
⇒ There are no uninitialized variables of type rational any more!
```

**RAT PACK® Reloaded ...**

Customer’s program now looks like this:

```cpp
// POST: double approximation of r
double to_double (const rational r)
{
    double result = r.numerator();
    return result / r.denominator();
}
```

We can adapt the member functions together with the representation ✓

```cpp
int numerator () const
{
    return n;
}
```

**The Default Constructor**

- is automatically called for declarations of the form `rational r;`
- is the unique constructor with empty argument list (if existing)
- must exist, if `rational r;` is meant to compile
- if in a struct there are no constructors at all, the default constructor is automatically generated
RAT PACK® Reloaded?

class rational {
  ...
private:
  unsigned int n;
  unsigned int d;
  bool is_positive;
};

int numerator () const {
  if (is_positive)
    return n;
  else {
    int result = n;
    return −result;
  }
}

- value range of nominator and denominator like before
- possible overflow in addition

Encapsulation still Incompletee

Customer’s point of view (rational.h):
class rational {
public:
  // POST: returns numerator of *this
  int numerator () const;
 ... 
private:
  // none of my business
};

- We determined denominator and nominator type to be int
- Solution: encapsulate not only data but also types.

Fix: “our” type rational::integer

Customer’s point of view (rational.h):
public:
  using integer = long int; // might change
  // POST: returns numerator of *this
  integer numerator () const;

- We provide an additional type!
- Determine only Functionality, e.g:
  - implicit conversion int → rational::integer
  - function double to_double (rational::integer)

RAT PACK® Revolutions

Finally, a customer program that remains stable

// POST: double approximation of r
double to_double (const rational r) {
  rational::integer n = r.numerator();
  rational::integer d = r.denominator();
  return to_double (n) / to_double (d);
}
class rational {
private:
...
};
rational::rational (int num, int den):
    n (num), d (den) {}