17. Classes

Encapsulation, Classes, Member Functions, Constructors

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 $(\frac{3}{5} \to 0.6)$

```
// 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;
};
```

(crossed out) $\Rightarrow$

```
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;
}
```

`r.is_positive` and `result.is_positive` do not appear.

correct using…

```
struct rational {
  int n;
  int d;
};
```

…not correct using

```
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!

`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`

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

`class` is used instead of `struct` if anything at all shall be "hidden"

only difference

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

Encapsulation: `public` / `private`

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

Good news: `r.d = 0` cannot happen any more by accident.

Bad news: the customer cannot do anything any more . . .

Application Code

. . . and we can't, either. (no `operator+`,. . .)

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


Member Functions: Declaration

```
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
};
```

public area

member function

member functions have access to private data

the scope of members in a class is the whole class, independent of the declaration order

Member Functions: Call

```
// Definition des Typs
class rational {
    ...
};
...
// Variable des Typs
rational r;

int n = r.numerator();    // Zaehler
int d = r.denominator(); // Nenner
```

member access

Member Functions: Definition

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

- A member function is called for an expression of the class. in the function, `this` is the name of this implicit argument. `this` itself is a pointer to it.
- const refers to the instance `this`, i.e., it promises that the value associated with the implicit argument cannot be changed
- `n` is the shortcut in the member function for `this->n` (precise explanation of "`->`" next week)

`const` and Member Functions

```
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;
    }
    ....
};
```

- No separation between declaration and definition (bad for libraries)

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

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

- This also works.

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*.

Initialisation? Constructors!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den)  ←—— Initialization of the member variables
    {
        assert (den != 0);  ←—— function body.
    }
...
};
...
rational r (2,3); // r = 2/3
```


Constructors: Call

- directly

```
rational r (1,2); // initialisiert r mit 1/2
```

- indirectly (copy)

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


Initialisation "rational = int"?

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


User Defined Conversions

are defined via constructors with exactly *one* argument

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

User defined conversion from `int` to `rational`. values of type `int` can now be converted to `rational`.

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

The Default Constructor

```
class rational
{
public:
    ...
    rational ()          // empty list of arguments
      : n (0), d (1)
    {}
...
};
...
rational r;     // r == 0
```

⇒ There are no uninitiatlized variables of type rational any more!

Alterantively: Deleting a Default Constructor

```
class rational
{
public:
    ...
    rational () = delete;
...
};
...
rational r;     // error: use of deleted function 'rational::rational()
```

⇒ There are no uninitiatlized variables of type rational any more!

The Default Constructor

- is automatically called for declarations of the form `rational r;`
- is the unique constructor with empty argmument 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 . . .

Customer's program now looks like this:

```
// 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 ✓

RAT PACK® Reloaded ...

before

```
class rational {
...
private:
  int n;
  int d;
};
```

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

after

```
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;
  }
}
```


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 Incompleete

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 alsoe 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);
}
```


Separate Declaration and Definition

rational.h

```
class rational {
public:
   rational (int num, int denum);
   using integer = long int;
   integer numerator () const;
   ...
private:
 ...
};
```

rational.cpp

```
rational::rational (int num, int den):
    n (num), d (den) {}
rational::integer rational::numerator () const
{
    return n;
}
```

class name :: member name