# 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<sup>®</sup>!

- Selling the rational library to customers
- ongoing development according to customer's demands

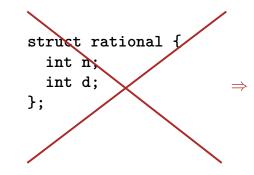
## The Customer is Happy

```
...and programs busily using rational.
■ output as double-value (<sup>3</sup>/<sub>5</sub> → 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 {
 unsigned int n;
 unsigned int d;
 bool is\_positive;
};

New Version of RAT PACK $^{\mathbb{R}}$ 



- 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;
                           r.is_positive and result.is_positive
  return result / r.d; do not appear.
}
                                 ... not correct using
 correct using...
                                 struct rational {
  struct rational {
                                   unsigned int n;
   int n;
                                   unsigned int d;
   int d;
                                   bool is_positive;
  };
                                 };
```

564

#### 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.
- $\Rightarrow$  RAT PACK<sup>®</sup> is history...

provide the concept for encapsulation in C++

are provided in many object oriented programming languages

Classes

are a variant of structs

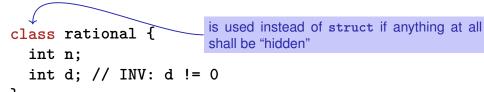
#### Idea of Encapsulation (Information Hiding)

- 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),...

569

## Encapsulation: public / private

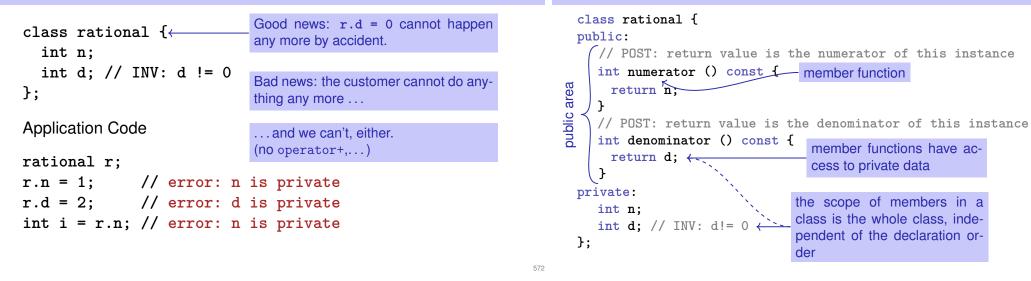


#### };

#### only difference

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

## Encapsulation: public / private



**Member Functions: Declaration** 

Member Functions: Call	Member Functions: Definition		
<pre>// Definition des Typs class rational {  }; // Variable des Typs rational r; member access</pre>	<pre>// 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</pre>		
<pre>int n = r.numerator(); // Zaehler</pre>	is the name of this implicit argument. this itself is a pointer to it.		
<pre>int d = r.denominator(); // Nenner</pre>	<ul> <li>const refers to the instance this, i.e., it promises that the value associated with the implicit argument cannot be changed</li> </ul>		
	<ul> <li>n is the shortcut in the member function for this-&gt;n (precise explanation of "-&gt;" next week)</li> </ul>		

### const and Member Functions

```
class rational {
public:
    int numerator () const
    { return n; }
    void set_numerator (int N)
    { n = N;}
...
}
rational x;
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 ...
                                  ... without member functions
class rational {
                                  struct bruch {
    int n:
                                      int n;
    . . .
                                       . . .
public:
                                  };
    int numerator () const
    {
                                  int numerator (const bruch& dieser)
        return this->n;
                                  {
    }
                                      return dieser.n;
                                  }
};
rational r;
                                  bruch r;
. . .
                                  . .
std::cout << r.numerator();</pre>
                                  std::cout << numerator(r);</pre>
```

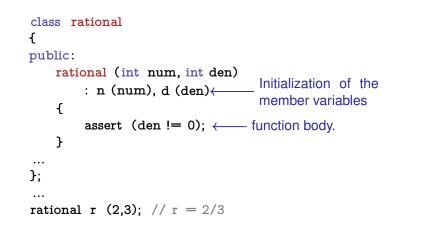
Member-Definition: In-Class vs. Out-of-Class				
<pre>class rational {     int n;  public:     int numerator () const     {         return n;     }</pre>	<pre>class rational {     int n;  public:     int numerator () const;  };</pre>			
<ul> <li></li> <li>Final Structure</li> <li>No separation between declaration and definition (bad for libraries)</li> </ul>	<pre>int rational::numerator () const {    return n; } This also works.</pre>	578		

#### **Constructors**

576

- 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!



## **Constructors: Call**

directly

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

■ indirectly (copy)

rational r = rational (1,2);

<b>Initialisation</b> "rational = int"?	User Defined Conversions		
class rational {	are defined via constructors with exactly one argument		
<pre>public: rational (int num) : n (num), d (1) {}</pre>	User defined conversion from int to rational (int num)		
<pre> }; rational r (2); // explicit initialization with 2 rational s = 2; // implicit conversion</pre>	<pre>rational r = 2; // implizite Konversion</pre>		

## **The Default Constructor**

 $\Rightarrow$  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 $^{\mathbb{R}}$ 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 <sup>®</sup> Reloaded		RAT PACK <sup>®</sup> Reloaded ?	
<pre>class rational {      private:         int n;         int d;     };</pre>	<pre>int numerator () const {    return n; }</pre>	<pre>class rational { private:     unsigned int n;     unsigned int d;     bool is_positive; };      value range of nominator     possible overflow in addiding </pre>	<pre>int numerator () const {     if (is_positive)         return n;     else {         int result = n;         return -result;     } } or and denominator like before ition</pre>
<pre>class rational { private:     unsigned int n;     unsigned int d;     bool is_positive; };</pre>	<pre>if (is_positive) private:</pre>		

**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 also etypes.

## 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  $\rightarrow$  rational::integer
  - function double to\_double (rational::integer)

589

# RAT PACK ${}^{\textcircled{R}}$ 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**

```
class rational {
public:
   rational (int num, int denum);
                                                    rational.h
   using integer = long int;
   integer numerator () const;
   . . .
private:
 • • •
};
rational::rational (int num, int den):
    n (num), d (den) {}
                                                    rational.cpp
rational::integer rational::numerator () const
                      \mathbf{\Lambda}
{
                  class name :: member name
    return n;
1
                                                                      593
```