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*:

```
operatorop
```

- 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;
```

$\uparrow$

infix notation

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

```
// 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

```
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

```
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 $\mathrm{C}{+}{+}$ no L-value.

Operator +=

```
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:

```
std::cout << "Sum is " << t.n << "/" << t.d << "\n";
```

- After (desired):

```
std::cout << "Sum is " << t << "\n";
```

In/Output Operators

can be overloaded as well:

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

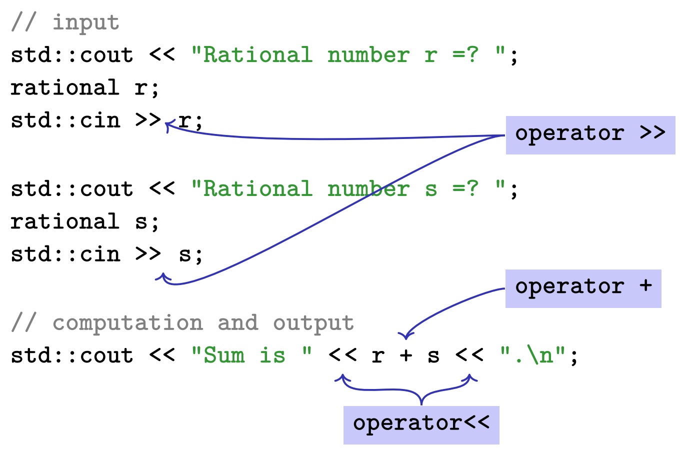
writes `r` to the output stream
and returns the stream as L-value.

Input

```
// 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";
```

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 <<`, ...)

Invariants and Representation

- We want to guarantee that invariants hold for our data structure.

  Example 1: For each `Rational r` it always holds that $r.d \neq 0$.
  Example 2: Each `Rational r` always is reduced.

- Provides to the user the “what” and not the “how”.

  `Vector v`: `v.size()`, `v[3]`

- It should be possible to change the internal representation without having to rewrit user code.

  `Complex`: polar coordinates vs. cartesian coordinates

Idea of Encapsulation (Information Hiding)

- A type is uniquely defined by its *value range* and its *functionality*
- The *representation* should *not be visible*.
- $\Rightarrow$ Not *representation* but functionality is offered!

$\uparrow$

`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
};
```

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: we cannot do anything any more . . .

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
```

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;  member access

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

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.

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)

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);
```

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 = rational(2,3); // r = 2/3
```

Constructors: Call

- directly

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

- indirectly (copy)

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

The Default Constructor

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

$\Rightarrow$ 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

Object Initialisation in C++

- For technical and historical reasons, C++ provides different forms of object initialisation, which differ in syntax and semantics. To initialise a new object t of type T, you may thus see any of the following:
  - T t`(...);`
  - T t`{...};`
  - T t `=` T`(...);`
  - T t `=` T`{...};`
  - T t `= {...};`
- Important: All shown forms are initialisations, not assignments, and T t `= ...;` is not equivalent to T t`;` t `= ...;`
- The variants with = can be used together with dynamic memory allocation (`new`)
- If you are interested, see also:
  - `https://en.cppreference.com/w/cpp/language/initialization`
  - `https://isocpp.org/wiki/faq/cpp11-language#uniform-init`
  - `https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Res-list`
  - `https://quuxplusone.github.io/blog/2019/02/18/knightmare-of-initialization`

Object Initialisation in This Course: Background

- It's not the goal of this course to train you to become expert C++ developers, but to teach generally relevant concepts and skills that can be transferred – ideally most directly – to other programming languages
- We'll therefore use three syntactic variants, each in a specific situation
- Our decisions aim at achieving the following, partly contradicting, goals:
  1. Portability: the syntax should resemble initialisation as done in other mainstream languages, e.g. Python, Java, Go, JavaScript
  2. Uniformity: using fewer syntaxes reduces the risk of confusion
  3. Stability: small changes (e.g. adding explicit constructors) should not easily break code (e.g. because compiler no longer generates certain constructors)
  4. Convenience: in specific situations, certain forms are particularly intuitive & easy to use
  5. Idiomatic: resulting code should be realistic C++ code

Object Initialisation in This Course: Conventions

1. For primitive data types (`int`, `bool`, `double`, ...) and initialisation with literals, we use *copy initialisation*. Examples:

```
int i = 5;
std::string s = "I am literally a string, folks!";
```

2. For *component-wise* initialisation of containers (e.g. `vector`; more soon) and simple datatypes with only public member variables (e.g. `struct rational` on slide 506), we use *list initialisation* and *aggregate initialisation*, respectively. Examples:

```
std::vector<char> vowels = {'A', 'E', 'I', 'O', 'U'};
rational half = {1, 2};
```

Object Initialisation in This Course: Conventions

3. In the remaining cases, e.g. for `struct rational` with private member variables from slide 554, we'll use *copy initialisation* with explicit constructor calls in function syntax. Examples:

```
rational half = rational(1,2);
std::vector<int> empty = std::vector<int>(7, 0); // vector with seven zeroes

// 'auto' fits in nicely, and avoids having to repeat types
auto half = rational(1,2);
auto empty = std::vector<int>(7, 0);
```

Object Initialisation in This Course: Conventions

4. Exception: initialisation of member variables in constructor definitions (*member initialiser lists*). Example:

```
class rational {
    ...
    rational(int n, int d): num(n), den(d) {...}
}
```

None of the syntactic shapes allowed here – t`(...)` and t`{...}` – is ideal, and we somewhat arbitrarily decided in favour of parentheses.

Object Initialisation: Final Words

Don't panic!

- You don't need to understand the reasons for, nor the differences between, the individual ways of initialising objects
- You also don't need to know the names of the different forms of object initialisation
- It suffices to understand object initialisation on the level of lecture examples and exercises
- You do not need to follow our conventions, and are allowed to use other forms of object initialisation

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!

Initialisation “rational = int”?

```
class rational
{
public:
    rational (int num)
       : n (num), d (1)
    {}  ⟵ empty function body
...
};
...
rational r = rational(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)
  {}

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

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

Benutzerdefinierte Konversionen

Wie kann man implizite Konversion von `rational` nach `double` realisieren?

- Problem: `double` ist kein Struct (keine Klasse), wir können dem Typ keinen Konstruktor „verpassen“ (gilt auch für alle anderen Zieltypen, die nicht „uns“ gehören)
- Lösung: wir bringen unserem Typ `rational` die Konversion nach `double` bei (als Member-Funktion):

```
struct rational{
    ...
    operator double () ⟵ impliziter Rückgabetyp double
    {
        return double (n)/d;
    }
};

rational a(1,2);
double b = a;  // implizite Konversion
```

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.