References

- Definition
- Difference between references and pointers
- Direct call by reference
- References as function output

- type&

What are references?

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
  int var = 5;
  int& ref = var;

  cout << "var = " << var << endl;
  cout << "ref = " << ref << endl;
  ref = 7;
  cout << "var = " << var << endl;
  cout << "ref = " << ref << endl;

  return 0;
}
```

- References are alias names for objects/variables
- `type& ref = var;`
  - Definition of a reference `ref` to `var`
  - `var` must be of type `type`
  - The reference must be initialized in its definition
- Avoid confusion with address-of operator
  - `type&` is a reference
  - `&var` is the memory address of `var`
- Output:

```
var = 5
ref = 5
var = 7
ref = 7
```


Address-of operator

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4
5  int main() {
6    int var = 5;
7    int& ref = var;
8
9    cout << "var = " << var << endl;
10   cout << "ref = " << ref << endl;
11   cout << "Address of var = " << &var << endl;
12   cout << "Address of ref = " << &ref << endl;
13
14   return 0;
15 }
```

- Declaration and initialization of a reference (line 7)
  - `ref` is a reference (alias name) to the variable `var`
  - i.e., `ref` and `var` have the same address
- Output:

```
var = 5
ref = 5
Address of var = 0x7fff532e8b48
Address of ref = 0x7fff532e8b48
```


Pointers as function arguments

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4
5  void swap(int* px, int* py) {
6    int tmp = *px;
7    *px = *py;
8    *py = tmp;
9  }
10
11 int main() {
12   int x = 5;
13   int y = 10;
14   cout << "x = " << x << ", y = " << y << endl;
15   swap(&x, &y);
16   cout << "x = " << x << ", y = " << y << endl;
17   return 0;
18 }
```

- Output:

```
x = 5, y = 10
x = 10, y = 5
```

- Code already seen in C:
  - The addresses `&x` and `&y` are passed to the function via call by value
  - Local variables `px` and `py` of type `int*`
  - Access to memory location of `x` via `*px` (dereferencing)
  - The same for `*py`
- Lines 6–8: Contents of `*px` and `*py` are swapped

References as function arguments

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 void swap(int& rx, int& ry) {
6   int tmp = rx;
7   rx = ry;
8   ry = tmp;
9 }
10
11 int main() {
12   int x = 5;
13   int y = 10;
14   cout << "x = " << x << ", y = " << y << endl;
15   swap(x, y);
16   cout << "x = " << x << ", y = " << y << endl;
17   return 0;
18 }
```

- Output:

  `x = 5, y = 10`

  `x = 10, y = 5`

- Call by reference in C++ (literally!)
  - References are passed as input to the function
  - Syntax: `type fctName( ..., type& ref, ... )`
    - This input is passed as a reference
- `rx` is a local name (lines 5–9) for the memory location of `x` (lines 12–17)
- The same for `ry` and `y`

References vs. pointers

- Pointers are variables containing memory addresses of other variables
- References are alias names for existing variables
  - Mandatory initialization with declaration
  - References cannot be assigned afterwards
- Roughly speaking:
  References = *Constant pointers with automatic dereferencing* (i.e., * applied by compiler)
- References are no complete alternative to pointers
  - No multiple assignment / re-assignment
  - No dynamically allocated memory
  - No arrays of references
  - References cannot be `NULL`
  - Only one level of indirection
    - Pointer to pointer OK
    - No reference to reference
- The syntax hides the program structure
  - From a function call it is not clear if call by value or call by reference is performed
  - Possible source of runtime errors
- **When is call by reference meaningful?**
  - When the input is large!
    - Call by value makes a copy of the data
  - Then, the function call becomes cheaper

References as function output 1/3

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 int& fct() {
6   int x = 4711;
7   return x;
8 }
9
10 int main() {
11   int var = fct();
12   cout << "var = " << var << endl;
13
14   return 0;
15 }
```

- References can be return values of functions
  - Meaningful for objects (see later!)
- Like for pointers, be careful with lifetime!
  - Return value is a reference (line 7)
  - But the corresponding memory is deallocated (end of block)
- The compiler prints a warning

  `reference_output.cpp:7: warning: reference to stack memory associated with local variable ’x’ returned`

References as function output 2/3

```
#include <iostream>
using std::cout;
using std::endl;

class Demo {
private:
  int val;
public:
  Demo(int input) {
    val = input;
  }
  int getContent() {
    return val;
  }
};

int main() {
  Demo var(10);
  int x = var.getContent();
  x = 1;
  cout << "x = " << x << ", ";
  cout << "val = " << var.getContent() << endl;
  return 0;
}
```

- Output:

  `x = 1, val = 10`

- This code will be compared to the one shown on the following slide

References as function output 3/3

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 class Demo {
6 private:
7   int val;
8 public:
9   Demo(int input) {
10    val = input;
11  }
12  int& getContent() {
13    return val;
14  }
15 };
16
17 int main() {
18  Demo var(10);
19  int& x = var.getContent();
20  x = 1;
21  cout << "x = " << x << ", ";
22  cout << "val = " << var.getContent() << endl;
23  return 0;
24 }
```

- Output:
  `x = 1, val = 1`
- Be careful: A `private` member has been changed!
  - This is actually an undesired behavior
  - Possible source of runtime error
- The implementation of `getContent` is not changed
  - Only modified signature
  - Changes in lines 12 and 19

Keyword `const`

- Definition of constants
- Read-only references
- Overloading & `const` for variables
- Overloading & `const` for references
- Overloading & `const` for methods

- `const`
- `const int*`, `int const*`, `int* const`
- `const int&`

Basic constants

- Definition via `#define CONST value`
  - Simple text replacement of `CONST` with `value`
  - Error-prone & cryptic error messages
    - If `value` causes a syntax error
  - Recall convention: Constants names uppercase
- Better approach: Use 'constant variables'
  - e.g., `const int var = value;`
  - e.g., `int const var = value;`
    - Both variants are equivalent
  - It is a variable, but the compiler prevents users from changing its value
  - Mandatory initialization with declaration
- **Be careful** with pointers
  - `const int* ptr` is a pointer to `const int`
  - `int const* ptr` is a pointer to `const int`
    - Both variants are equivalent
  - `int* const ptr` is a constant pointer to `int`

Example 1/2

```
int main() {
  const double var = 5;
  var = 7;
  return 0;
}
```

- Compilation leads to a syntax error:
  `const.cpp:3: error: read-only variable is not assignable`

```
int main() {
  const double var = 5;
  double tmp  = 0;
  const double* ptr = &var;
  ptr = &tmp;
  *ptr = 7;
  return 0;
}
```

- Compilation leads to a syntax error:
  `const_pointer.cpp:6: error: read-only variable is not assignable`

Example 2/2

```
1 int main() {
2   const double var = 5;
3   double tmp  = 0;
4   double* const ptr = &var;
5   ptr = &tmp;
6   *ptr = 7;
7   return 0;
8 }
```

- Compilation leads to a syntax error:

  `const_pointer2.cpp:4: error: cannot initialize a variable of type 'double *const' with an rvalue of type 'const double *'`

  - The pointer `ptr` has a wrong type (line 4)

```
1 int main() {
2   const double var = 5;
3   double tmp  = 0;
4   const double* const ptr = &var;
5   ptr = &tmp;
6   *ptr = 7;
7   return 0;
8 }
```

- Compilation leads to two syntax errors:

  `const_pointer3.cpp:5: error: read-only variable is not assignable`

  `const_pointer3.cpp:6: error: read-only variable is not assignable`

  - Assignement for pointer `ptr` (line 5)
  - Dereferencing and writing (line 6)

Read-only references

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
  double var = 5;
  double& ref = var;
  const double& cref = var;
  cout << "var = " << var << ", ";
  cout << "ref = " << ref << ", ";
  cout << "cref = " << cref << endl;
  ref = 7;
  cout << "var = " << var << ", ";
  cout << "ref = " << ref << ", ";
  cout << "cref = " << cref << endl;
  // cref = 9;
  return 0;
}
```

- `const type& cref`
  - Declaration of a constant reference to `type`
    - Alternative syntax: `type const& cref`
  - i.e., `cref` is like a variable of type `const type`
  - Access to reference possible only to **read**
- Output:

  `var = 5, ref = 5, cref = 5`

  `var = 7, ref = 7, cref = 7`

- Typing `cref = 9;` would lead to a syntax error

  `error: read-only variable is not assignable`

Read-only references as output 1/2

```
#include <iostream>
using std::cout;
using std::endl;

class Demo {
private:
  int val;
public:
  Demo(int input) {
    val = input;
  }
  int& getContent() {
    return val;
  }
};

int main() {
  Demo var(10);
  int& x = var.getContent();
  x = 1;
  cout << "x = " << x << ", ";
  cout << "val = " << var.getContent() << endl;
  return 0;
}
```

- Output:

  `x = 1, val = 1`

- Be careful: A `private` member has been changed!
- Code already presented in slide 246

Read-only references as output 2/2

```
#include <iostream>
using std::cout;
using std::endl;

class Demo {
private:
  int val;
public:
  Demo(int input) { val = input; }
  const int& getContent() { return val; }
};

int main() {
  Demo var(10);
  const int& x = var.getContent();
  // x = 1;
  cout << "x = " << x << ", ";
  cout << "val = " << var.getContent() << endl;
  return 0;
}
```

- Output:

  `x = 10, val = 10`

- Assignment `x = 1;` would lead to a syntax error

  `error: read-only variable is not assignable`

- Declaration `int& x = var.getContent();` would lead to a syntax error

  `error: binding of reference to type 'int' to a value of type 'const int' drops qualifiers`

- Meaningful, if read-only return value is large
  - e.g., vector, large string, etc.

Type casting

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 double square(double& x) {
 6   return x*x;
 7 }
 8
 9 int main() {
10   const double var = 5;
11   cout << "var = " << var << ", ";
12   cout << "var*var = " << square(var) << endl;
13   return 0;
14 }
```

- `const type` is stronger than `type`
  - No type casting from `const type` to `type`
- Compilation leads to a syntax error:

  `const_typecasting.cpp:12 error: no matching function for call to 'square'`
  `const_typecasting.cpp:5: note: candidate function not viable: 1st argument ('const double') would lose const qualifier`
- Type casting from `type` to `const type` is fine!
- Possible workaround: Change signature to
  - `double square(const double& x)`

Read-only references as input 1/5

```
 1 #include "vector_first.hpp"
 2 #include <iostream>
 3 #include <cassert>
 4
 5 using std::cout;
 6
 7 double product(const Vector& x, const Vector& y){
 8   double sum = 0;
 9   assert( x.size() == y.size() );
10   for (int j=0; j<x.size(); ++j) {
11     sum = sum + x.get(j)*y.get(j);
12   }
13   return sum;
14 }
15
16 int main() {
17   Vector x(100,1);
18   Vector y(100,2);
19   cout << "norm(x) = " << x.norm() << "\n";
20   cout << "norm(y) = " << y.norm() << "\n";
21   cout << "x.y = " << product(x,y) << "\n";
22   return 0;
23 }
```

- Advantage: Quicker data input without copy!
  - Data cannot be changed!
- Problem: Compilation leads to a syntax error

  `const_vector.cpp:9: error: member function 'size' not viable: 'this' argument has type 'const Vector', but function is not marked const`
  - i.e., problem with the method `size`

Read-only references as input 2/5

```
 1 #ifndef _VECTOR_NEW_
 2 #define _VECTOR_NEW_
 3
 4 #include <cmath>
 5 #include <cstdlib>
 6 #include <cassert>
 7
 8 // The class Vector stores vectors in Rd
 9
10 class Vector {
11 private:
12   // dimension of the vector
13   int dim;
14   // dynamic coefficient vector
15   double* coeff;
16
17 public:
18   // constructors and destructor
19   Vector();
20   Vector(int, double = 0);
21   ~Vector();
22
23   // return vector dimension
24   int size() const;
25
26   // read and write vector coefficients
27   void set(int k, double value);
28   double get(int k) const;
29
30   // compute Euclidean norm
31   double norm() const;
32 };
33
34 #endif
```

- Read-only methods are marked with `const`
  - `className::fct(... input ...) const { ... }`
  - Possible for methods, not for all functions
- New syntax in lines 24, 28, and 31

Read-only references as input 3/5

```
#include "vector_new.hpp"
#include <iostream>
using std::cout;

Vector::Vector() {
  dim = 0;
  coeff = (double*) 0;
  cout << "new empty vector" << "\n";
}

Vector::Vector(int dim, double init) {
  assert(dim > 0);
  this->dim = dim;
  coeff = (double*) malloc(dim*sizeof(double));
  assert( coeff != (double*) 0);
  for (int j=0; j<dim; ++j) {
    coeff[j] = init;
  }
  cout << "new vector, length " << dim << "\n";
}

Vector::~Vector() {
  if (dim > 0) {
    free(coeff);
  }
  cout << "free vector, length " << dim << "\n";
}
```

- Here no changes!

Read-only references as input 4/5

```
29 int Vector::size() const {
30   return dim;
31 }
32
33 void Vector::set(int k, double value) {
34   assert(k>=0 && k<dim);
35   coeff[k] = value;
36 }
37
38 double Vector::get(int k) const {
39   assert(k>=0 && k<dim);
40   return coeff[k];
41 }
42
43 double Vector::norm() const {
44   double norm = 0;
45   for (int j=0; j<dim; ++j) {
46     norm = norm + coeff[j]*coeff[j];
47   }
48   return sqrt(norm);
49 }
```

- New syntax in lines 29, 38, and 43

Read-only references as input 5/5

```
#include "vector_new.hpp"
#include <iostream>
#include <cassert>

using std::cout;

double product(const Vector& x, const Vector& y){
  double sum = 0;
  assert( x.size() == y.size() );
  for (int j=0; j<x.size(); ++j) {
    sum = sum + x.get(j)*y.get(j);
  }
  return sum;
}

int main() {
  Vector x(100,1);
  Vector y(100,2);
  cout << "norm(x) = " << x.norm() << "\n";
  cout << "norm(y) = " << y.norm() << "\n";
  cout << "x.y = " << product(x,y) << "\n";
  return 0;
}
```

- Advantage: Quicker data input without copy!
  - Data cannot be changed!
- Output:

```
new vector, length 100
new vector, length 100
norm(x) = 10
norm(y) = 20
x.y = 200
free vector, length 100
free vector, length 100
```


Syntax summary

- For 'normal' data types (no pointers, no references)
  - `const int var`
  - `int const var`
    - Same meaning = Integer constant
- For references
  - `const int& ref` = Reference to `const int`
  - `int const& ref` = Reference to `const int`
- For pointers
  - `const int* ptr` = Pointer to `const int`
  - `int const* ptr` = Pointer to `const int`
  - `int* const ptr` = Constant pointer to `int`
- For methods with read-only rights
  - `className::fct(... input ...) const`
  - Otherwise methods cannot work with `const`-references
- Meaningful, if the return value is a reference
  - `const int& fct(... input ...)`
  - Reasonable only for *large* read-only return values
  - **Be careful:** Return value must exist, otherwise runtime error!

Overloading and `const` 1/2

```
#include <iostream>
using std::cout;

void f(int x) { cout << "int\n"; };
void f(const int x) { cout << "const int\n"; };

int main() {
    int x = 0;
    const int c = 0;
    f(x);
    f(c);
    return 0;
}
```

- `const` is not considered for input variables
  - Compilation leads to a syntax error:

    `overload_const.cpp:5: error: redefinition of 'f'`

```
#include <iostream>
using std::cout;

void f(int& x) { cout << "int\n"; };
void f(const int& x) { cout << "const int\n"; };

int main() {
    int x = 0;
    const int c = 0;
    f(x);
    f(c);
    return 0;
}
```

- `const` is considered for input references
  - Compilation fine and output:

```
int
const int
```

Overloading and const 2/2

```
#include <iostream>
using std::cout;

class Demo {
private:
    int content;
public:
    Demo() { content = 0; }
    void f() { cout << "normal object\n"; };
    void f() const { cout << "const object\n"; };
};

int main() {
    Demo x;
    const Demo y;
    x.f();
    y.f();
    return 0;
}
```

- Methods can be overloaded with `const`-methods
  - `const`-methods are used only for `const`-objects
  - Otherwise the ‘normal’ methods are used
- Output:

  ```
  normal object
  const object
  ```


Operator overloading

- Copy constructor
- Type casting
- Assignment operator
- Unary and binary operators

- `operator`

Class for complex numbers

```
1 #include <iostream>
2 #include <cmath>
3
4 class Complex {
5 private:
6   double re;
7   double im;
8 public:
9   Complex(double=0, double=0);
10  double real() const;
11  double imag() const;
12  double abs() const;
13  void print() const;
14 };
15
16 Complex::Complex(double re, double im) {
17   this->re = re;
18   this->im = im;
19 }
20
21 double Complex::real() const {
22   return re;
23 }
24
25 double Complex::imag() const {
26   return im;
27 }
28
29 double Complex::abs() const {
30     return sqrt(re*re + im*im);
31 }
32
33 void Complex::print() const {
34   std::cout << re << " + " << im << " * i";
35 }
```

- Default parameters in the first declaration
  - Line 9: Forward declaration of the constructor
  - Lines 16–19: Code for the constructor

Copy constructor

```
Complex::Complex(const Complex& rhs) {
  re = rhs.re;
  im = rhs.im;
}
```

- `className::className(const className& rhs)`
- Special constructor with call
  - `Complex lhs = rhs;`
  - or `Complex lhs(rhs);`
- Creation of a new object `lhs` containing the data of `rhs`
  - Input as constant reference (read-only)
- It is automatically created (shallow copy), if not explicitly implemented
  - Here formally not necessary, as all data are static
  - Important, if the class contains dynamic data

Assignment operator

```
Complex& Complex::operator=(const Complex& rhs) {
  if (this != &rhs) {
    re = rhs.re;
    im = rhs.im;
  }
  return *this;
}
```

- `className& className::operator=(const className& rhs)`
- If `Complex lhs, rhs;` already declared
  - Assignment `lhs = rhs;`
  - Input as constant reference (read-only)
  - Output as reference to allow 'assignment chains'
    - e.g., `a = b = c = d;`
    - `=` assignment from left to right
    - `a = ...` requires evaluation of `b = c = d;`
- Functionality:
  - Data of `lhs` are overwritten with those of `rhs`
  - Possible dynamic data of `lhs` should be deallocated before
- `this` is a pointer to the object under consideration
  - i.e., `*this` is the object (dereferencing)
- `if` avoids conflicts in self-assignments `z = z;`
  - Here formally not necessary, only static data
- It is automatically created (shallow copy), if not explicitly implemented
  - Here formally not necessary, only static data
  - Important, if the class contains dynamic data

Type casting

- $\mathbb{R} \subset \mathbb{C}$, i.e., $x \in \mathbb{R} \Rightarrow x \in \mathbb{C}$

```
Complex::Complex(double re = 0, double im = 0) {
  this->re = re;
  this->im = im;
}
```

- Constructor provides type casting from `double` to `Complex`
  - i.e., $x \in \mathbb{R}$ entspricht $x + 0i \in \mathbb{C}$

```
Complex::operator double() const {
  return re;
}
```

- Type casting from `Complex` to `double`
  - Formally: `ClassName::operator type() const`
    - Implicit via type of return value
- Note standard type castings
  - Implicit from `int` to `double`
  - Implicit from `double` to `int`

Unary operators

- Unary operators = Operators with one argument

```
const Complex Complex::operator-() const {
  return Complex(-re,-im);
}
```

- Change of sign `-` (minus)
  - `const Complex Complex::operator-() const`
    - The output is of type `const Complex`
    - The method works only with the current members
    - The method is read-only for the current data
  - It is a method of the class
- Callable with `-x`

```
const Complex Complex::operator~() const {
  return Complex(re,-im);
}
```

- Conjugation `~` (tilde)
  - `const Complex Complex::operator~() const`
    - The output is of type `const Complex`
    - The method works only with the current members
    - The method is read-only for the current data
  - It is a method of the class
- Callable with `~x`

complex_part.hpp

```
1  #ifndef _COMPLEX_PART_
2  #define _COMPLEX_PART_
3
4  #include <iostream>
5  #include <cmath>
6
7  class Complex {
8  private:
9    double re;
10   double im;
11 public:
12   Complex(double=0, double=0);
13   Complex(const Complex& rhs);
14   ~Complex();
15   Complex& operator=(const Complex& rhs);
16
17   double real() const;
18   double imag() const;
19   double abs() const;
20   void print() const;
21
22   operator double() const;
23
24   const Complex operator~() const;
25   const Complex operator-() const;
26 };
27
28 #endif
```

- Line 12: Forward declaration with default input
- Lines 12 and 22: Type casting `Complex` vs. `double`

complex_part.cpp 1/2

```
#include "complex_part.hpp"

using std::cout;

Complex::Complex(double re, double im) {
  this->re = re;
  this->im = im;
  cout << "Constructor\n";
}

Complex::Complex(const Complex& rhs) {
  re = rhs.re;
  im = rhs.im;
  cout << "Copy constructor\n";
}

Complex::~Complex() {
  cout << "Destructor\n";
}

Complex& Complex::operator=(const Complex& rhs) {
  if (this != &rhs) {
    re = rhs.re;
    im = rhs.im;
    cout << "Assignment\n";
  }
  else {
    cout << "Self-assignment\n";
  }
  return *this;
}
```


complex_part.cpp 2/2

```
double Complex::real() const {
  return re;
}

double Complex::imag() const {
  return im;
}

double Complex::abs() const {
    return sqrt(re*re + im*im);
}

void Complex::print() const {
  cout << re << " + " << im << "*i";
}

Complex::operator double() const {
  cout << "Complex -> double\n";
  return re;
}

const Complex Complex::operator-() const {
  return Complex(-re,-im);
}

const Complex Complex::operator~() const {
  return Complex(re,-im);
}
```


Beispiel

```
#include <iostream>
#include "complex_part.hpp"
using std::cout;

int main() {
  Complex w(1);
  Complex x;
  Complex y(1,1);
  Complex z = y;
  x = x;
  x = ~y;
  w.print(); cout << "\n";
  x.print(); cout << "\n";
  y.print(); cout << "\n";
  z.print(); cout << "\n";
  return 0;
}
```

- Output:

```
Constructor
Constructor
Constructor
Copy constructor
Self-assignment
Constructor
Assignment
Destructor
1 + 0*i
1 + -1*i
1 + 1*i
1 + 1*i
Destructor
Destructor
Destructor
Destructor
```


Example: Type casting

```
1  #include <iostream>
2  #include "complex_part.hpp"
3  using std::cout;
4
5  int main() {
6    Complex z((int) 2.3, (int) 1);
7    double x = z;
8    z.print(); cout << "\n";
9    cout << x << "\n";
10   return 0;
11 }
```

- Constructor requires `double` as input (Line 6)
  - First explicit type casting of `2.3` to `int`
  - Then implicit type casting to `double`
- Output:

```
Constructor
Complex -> double
2 + 1*i
2
Destructor
```

Binary operators

```
 1 const Complex operator+(const Complex& x,const Complex& y){
 2   double xr = x.real();
 3   double xi = x.imag();
 4   double yr = y.real();
 5   double yi = y.imag();
 6   return Complex(xr + yr, xi + yi);
 7 }
 8 const Complex operator-(const Complex& x,const Complex& y){
 9   double xr = x.real();
10   double xi = x.imag();
11   double yr = y.real();
12   double yi = y.imag();
13   return Complex(xr - yr, xi - yi);
14 }
15 const Complex operator*(const Complex& x,const Complex& y){
16   double xr = x.real();
17   double xi = x.imag();
18   double yr = y.real();
19   double yi = y.imag();
20   return Complex(xr*yr - xi*yi, xr*yi + xi*yr);
21 }
22 const Complex operator/(const Complex& x,const double y){
23   assert(y != 0)
24   return Complex(x.real()/y, x.imag()/y);
25 }
26 const Complex operator/(const Complex& x,const Complex& y){
27   double norm = y.abs();
28   assert(norm > 0);
29   return x*~y / (norm*norm);
30 }
```

- Binary operators = Operators with two arguments
  - e.g., `+`, `-`, `*`, `/`
- Outside the class definition as function
  - Formally: `const type operator+(const type& rhs1, const type& rhs2)`
  - **Be careful:** No `type::` as no part of the class!
- Lines 22 and 26: Note $x/y = (x\overline{y})/(y\overline{y}) = x\overline{y}/|y|^2$

Operator <<

```
std::ostream& operator<<(std::ostream& output,
                         const Complex& x) {
  if (x.imag() == 0) {
    return output << x.real();
  }
  else if (x.real() == 0) {
    return output << x.imag() << "i";
  }
  else {
    return output << x.real() << " + " << x.imag() << "i";
  }
}
```

- Printing via `cout` is part of the class `std::ostream`
- Successive printing via repeated `<<`
  - Implementation might require `for` loops, e.g., to print vectors / matrices with `cout`

complex.hpp

```
#ifndef _COMPLEX_
#define _COMPLEX_

#include <iostream>
#include <cmath>
#include <cassert>

class Complex {
private:
  double re;
  double im;
public:
  Complex(double=0, double=0);
  Complex(const Complex&);
  ~Complex();
  Complex& operator=(const Complex&);

  double real() const;
  double imag() const;
  double abs() const;

  operator double() const;

  const Complex operator~() const;
  const Complex operator-() const;

};

std::ostream& operator<<(std::ostream& output,
                          const Complex& x);
const Complex operator+(const Complex&, const Complex&);
const Complex operator-(const Complex&, const Complex&);
const Complex operator*(const Complex&, const Complex&);
const Complex operator/(const Complex&, const double);
const Complex operator/(const Complex&, const Complex&);

#endif
```

- “Full library” without unnecessary `cout` in the following `cpp` source code

complex.cpp 1/3

```
#include "complex.hpp"
using std::ostream;

Complex::Complex(double re, double im) {
  this->re = re;
  this->im = im;
}

Complex::Complex(const Complex& rhs) {
  re = rhs.re;
  im = rhs.im;
}

Complex::~Complex() {
}

Complex& Complex::operator=(const Complex& rhs) {
  if (this != &rhs) {
    re = rhs.re;
    im = rhs.im;
  }
  return *this;
}

double Complex::real() const {
  return re;
}

double Complex::imag() const {
  return im;
}

double Complex::abs() const {
  return sqrt(re*re + im*im);
}

Complex::operator double() const {
    return re;
}
```

complex.cpp 2/3

```
const Complex Complex::operator-() const {
  return Complex(-re,-im);
}

const Complex Complex::operator~() const {
  return Complex(re,-im);
}

const Complex operator+(const Complex& x,const Complex& y){
  double xr = x.real();
  double xi = x.imag();
  double yr = y.real();
  double yi = y.imag();
  return Complex(xr + yr, xi + yi);
}

const Complex operator-(const Complex& x,const Complex& y){
  double xr = x.real();
  double xi = x.imag();
  double yr = y.real();
  double yi = y.imag();
  return Complex(xr - yr, xi - yi);
}

const Complex operator*(const Complex& x,const Complex& y){
  double xr = x.real();
  double xi = x.imag();
  double yr = y.real();
  double yi = y.imag();
  return Complex(xr*yr - xi*yi, xr*yi + xi*yr);
}
```


complex.cpp 3/3

```
const Complex operator/(const Complex& x, const double y){
  assert(y != 0);
  return Complex(x.real()/y, x.imag()/y);
}

const Complex operator/(const Complex& x,const Complex& y){
  double norm = y.abs();
  assert(norm > 0);
  return x*~y / (norm*norm);
}

std::ostream& operator<<(std::ostream& output,
                         const Complex& x) {
  if (x.imag() == 0) {
    return output << x.real();
  }
  else if (x.real() == 0) {
    return output << x.imag() << "i";
  }
  else {
    return output << x.real() << " + " << x.imag() << "i";
  }
}
```


complex_main.cpp

```
#include "complex.hpp"
#include <iostream>
using std::cout;

int main() {
  Complex w;
  Complex x(1,0);
  Complex y(0,1);
  Complex z(3,4);

  w = x + y;
  cout << w << "\n";

  w = x*y;
  cout << w << "\n";

  w = x/y;
  cout << w << "\n";

  w = z/(x + y);
  cout << w << "\n";

  w = w.abs();
  cout << w << "\n";

  return 0;
}
```

- Output:

```
1 + 1i
1i
-1i
3.5 + 0.5i
3.53553
```


Function call & Copy constructor 1/2

```
#include <iostream>
using std::cout;

class Demo {
private:
  int data;
public:
  Demo(int data = 0) {
    cout << "Standard constructor\n";
    this->data = data;
  }

  Demo(const Demo& rhs) {
    cout << "Copy constructor\n";
    data = rhs.data;
  }

  Demo& operator=(const Demo& rhs) {
    cout << "Assignment operator\n";
    data = rhs.data;
    return *this;
  }

  ~Demo() {
    cout << "Destructor\n";

  }
};
```

- When the function is called, the input data are passed to the function via copy constructor

Function call & Copy constructor 2/2

```
void function(Demo input) {
  cout << "Function with call by value\n";
}

void function2(Demo& input) {
  cout << "Function with call by reference\n";
}

int main() {
  Demo x;
  Demo y = x;
  cout << "*** Function call (call by value)\n";
  function(y);
  cout << "*** Function call (call by reference)\n";
  function2(x);
  cout << "*** Program end\n";
  return 0;
}
```

- When the function is called, the input data are passed to the function via copy constructor
- Output:

```
Standard constructor
Copy constructor
*** Function call (call by value)
Copy constructor
Function with call by value
Destructor
*** Function call (call by reference)
Function with call by reference
*** Program end
Destructor
Destructor
```


Syntax summary

- Constructor (= Type casting to `Class`)
  `Class::Class( ... input ... )`
- Destructor
  `Class::~Class()`
- Type casting from `Class` to `type`
  `Class::operator type() const`
  - Explicit via prepended `(type)`
  - Implicit via assignment to variable of type `type`
- Copy constructor (Declaration with initialization)
  `Class::Class(const Class&)`
  - Explicit call via `Class var(rhs);`
    - or `Class var = rhs;`
  - Implicit for function calls (call by value)
- Assignment operator
  `Class& Class::operator=(const Class&)`
- Unary operators, e.g., tilde `~` or minus `-`
  `const Class Class::operator-() const`
- Binary operators, e.g., `+`, `-`, `*`, `/`
  `const Class operator+(const Class&, const Class&)`
  - Outside the class as function
- Printing via `cout`
  ```
  std::ostream& operator<<(std::ostream& output,
                           const Class& object)
  ```


Binary operators inside the class

```
1  #include <iostream>
2  using std::cout;
3
4  class Complex {
5  private:
6    double re;
7    double im;
8  public:
9    Complex(double re=0, double im=0) {
10     this->re = re;
11     this->im = im;
12   }
13   const Complex operator-() const {
14     return Complex(-re,-im);
15   }
16   const Complex operator-(const Complex& y) {
17     return Complex(re-y.re, im-y.im);
18   }
19   void print() const {
20     cout << re << " + " << im << "\n";
21   }
22 };
23
24 int main() {
25   Complex x(1,0);
26   Complex y(0,1);
27   Complex w = x-y;
28   (-y).print();
29   w.print();
30 }
```

- Binary operators `+`, `-`, `*`, `/` possible as methods
  - Sign (unary operator): Lines 13–15
  - Subtraction (binary operator): Lines 16–18
    - Then first argument = Current object
- Instead of outside the class as function
  `const Complex operator-(const Complex& x, const Complex& y)`

Which operators can be overloaded?

+	-	*	/	&	^	%
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

- As unary operators, prepended `++var`
  `const Class Class::operator++()`
- As unary operators, postpended `var++`
  `const Class Class::operator++(int)`
- As binary operators
  `const Class operator+(const Class&, const Class&)`
- As both unary/binary operators
  - e.g., division `Complex/double` vs. `Complex/Complex`
  - e.g., unary and binary (negative sign vs. minus)
  - Note the different signatures!
- Definition of new operators not possible!
- `.`, `:`, `::`, `sizeof`, `.*` cannot be overloaded!
- In the final test operator signatures will be given!
  - Exception: constructor, destructor!
- https://www.c-plusplus.net/forum/232010-full
- https://en.wikipedia.org/wiki/Operators_in_C_and_C++

Dynamic memory allocation

- Dynamic memory allocation in C++
- Rule of three

- `new`, `new ... []`
- `delete`, `delete[]`

`new` vs. `malloc`

- `malloc` allocates only memory
  - **Disadvantage:** Constructors are not called
    - i.e., manual initialization
- For a dynamic object
  `type* var = (type*) malloc(sizeof(type));`
  `*var = ...;`
- For a dynamic vector of objects of length `N`
  `type* vec = (type*) malloc(N*sizeof(type));`
  `vec[j] = ...;`
- In C++ type casting is mandatory for `malloc`!

- `new` allocates memory and calls constructors
- For a dynamic object (with standard constructor)
  `type* var = new type;`
- For a dynamic object (with constructor)
  `type* var = new type(... input ... );`
- For a dynamic vector of objects of length `N` (with standard constructor)
  `type* vec = new type[N];`
    - Standard constructor for each coefficient

- **Convention:** Use always `new`!
- In C++, there is no equivalent to `realloc`

`delete` vs. `free`

- `free` deallocates memory allocated with `malloc`
  `type* vec = (type*) malloc(N*sizeof(type));`
  `free(vec);`
  - Independent of object / Vector of objects
  - Use it only for the output of `malloc`!

- `delete` calls destructor and deallocates the memory previously allocated with `new`
  `type* var = new type(... input ... );`
  `delete var;`
  - Use it for a dynamically generated object
  - Use it only for the output of `new`!
- `delete[]` calls destructor for each coefficient and deallocates the memory previously allocated with `new ...[N]`
  `type* vec = new type[N];`
  `delete[] vec;`
  - Use it for a dynamic vector of objects
  - Use it only for the output of `new ...[N]`!

- **Convention:** If a pointer does not point to any dynamic memory, it should manually be directed to `NULL`
  - i.e., after `free`, `delete`, `delete[]` it follows
    - `vec = (type*) NULL;`
    - In C++ more often: `vec = (type*) 0;`

Rule of three

- Also: Law of the big three

- If either destructor or copy constructor or assignment operator are implemented, then all three of them should be implemented!

- Necessary, if a class contains dynamic arrays
  - Otherwise automatically done by the compiler as shallow copy (OK for basic data type!)
  - Shallow copy leads to runtime error for dynamic arrays

Desregarding the rule of three 1/2

```
#include <iostream>
using std::cout;

class Demo {
private:
  int n;
  double* data;
public:
  Demo(int n, double input);
  ~Demo();
  int getN() const;
  const double* getData() const;
  void set(double input);
};

Demo::Demo(int n, double input) {
  cout << "constructor, length " << n << "\n";
  this->n = n;
  data = new double[n];
  for (int j=0; j<n; ++j) {
    data[j] = input;
  }
}

Demo::~Demo() {
  cout << "destructor, length " << n << "\n";
  delete[] data;
}

int Demo::getN() const {
  return n;
}

const double* Demo::getData() const {
  return data;
}
```

- Destructor is available (dynam. allocated memory)
- Copy constructor and assignment operator are missing

Desregarding the rule of three 2/2

```
void Demo::set(double input) {
  for (int j=0; j<n; ++j) {
    data[j] = input;
  }
}

std::ostream& operator<<(std::ostream& output,
                         const Demo& object) {
  const double* data = object.getData();
  for(int j=0; j<object.getN(); ++j) {
    output << data[j] << " ";
  }
  return output;
}

void print(Demo z) {
  cout << "print: " << z << "\n";
}

int main() {
  Demo x(4,2);
  Demo y = x;
  cout << "x = " << x << ", y = " << y << "\n";
  y.set(3);
  cout << "x = " << x << ", y = " << y << "\n";
  print(x);
  x.set(5);
  cout << "x = " << x << ", y = " << y << "\n";
  return 0;
}
```

- Output:

```
x = 2 2 2 2 , y = 2 2 2 2
x = 3 3 3 3 , y = 3 3 3 3
print:  3 3 3 3
destructor, length 4
x = 5 5 5 5 , y = 5 5 5 5
destructor, length 4
Memory access error
```


vector.hpp

```
1 #ifndef _VECTOR_
2 #define _VECTOR_
3 #include <cmath>
4 #include <cassert>
5
6 // The class Vector stores vectors in Rd
7 class Vector {
8 private:
9   int dim;
10  double* coeff;
11
12 public:
13  // constructors, destructor, assignment
14  Vector();
15  Vector(int dim, double init=0);
16  Vector(const Vector&);
17  ~Vector();
18  Vector& operator=(const Vector&);
19  // return length of vector
20  int size() const;
21  // read and write entries
22  const double& operator[](int k) const;
23  double& operator[](int k);
24  // compute Euclidean norm
25  double norm() const;
26 };
27
28 // addition of vectors
29 const Vector operator+(const Vector&, const Vector&);
30 // scalar multiplication
31 const Vector operator*(const double, const Vector&);
32 const Vector operator*(const Vector&, const double);
33 // scalar product
34 const double operator*(const Vector&, const Vector&);
35
36 #endif
```

- Overloading of [ ]
  - For constant objects, method from line 22
  - For ‘normal objects’, method from line 23

vector.cpp 1/4

```
#include "vector.hpp"
#include <iostream>
using std::cout;

Vector::Vector() {
  dim = 0;
  coeff = (double*) 0;
  // just for demonstration purposes
  cout << "constructor, empty\n";
}

Vector::Vector(int dim, double init) {
  assert(dim >= 0);
  this->dim = dim;
  if (dim == 0) {
    coeff = (double*) 0;
  }
  else {
    coeff = new double[dim];
    for (int j=0; j<dim; ++j) {
      coeff[j] = init;
    }
  }
  // just for demonstration purposes
  cout << "constructor, length " << dim << "\n";
}

Vector::Vector(const Vector& rhs) {
  dim = rhs.dim;
  if (dim == 0) {
    coeff = (double*) 0;
  }
  else {
    coeff = new double[dim];
    for (int j=0; j<dim; ++j) {
      coeff[j] = rhs[j];
    }
  }
  // just for demonstration purposes
  cout << "copy constructor, length " << dim << "\n";
}
```

vector.cpp 2/4

```
Vector::~Vector() {
  if (dim > 0) {
    delete[] coeff;
  }
  // just for demonstration purposes
  cout << "free vector, length " << dim << "\n";
}

Vector& Vector::operator=(const Vector& rhs) {
  if (this != &rhs) {
    if (dim != rhs.dim) {
      if (dim > 0) {
        delete[] coeff;
      }
      dim = rhs.dim;
      if (dim > 0) {
        coeff = new double[dim];
      }
      else {
        coeff = (double*) 0;
      }
    }
    for (int j=0; j<dim; ++j) {
      coeff[j] = rhs[j];
    }
  }
  // just for demonstration purposes
  cout << "deep copy, length " << dim << "\n";
  return *this;
}

int Vector::size() const {
  return dim;
}
```


vector.cpp 3/4

```
 78 const double& Vector::operator[](int k) const {
 79   assert(k>=0 && k<dim);
 80   return coeff[k];
 81 }
 82
 83 double& Vector::operator[](int k) {
 84   assert(k>=0 && k<dim);
 85   return coeff[k];
 86 }
 87
 88 double Vector::norm() const {
 89   double sum = 0;
 90   for (int j=0; j<dim; ++j) {
 91     sum = sum + coeff[j]*coeff[j];
 92   }
 93   return sqrt(sum);
 94 }
 95
 96 const Vector operator+(const Vector& rhs1,
 97                        const Vector& rhs2) {
 98   assert(rhs1.size() == rhs2.size());
 99   Vector result(rhs1);
100   for (int j=0; j<result.size(); ++j) {
101     result[j] += rhs2[j];
102   }
103   return result;
104 }
```

- Access to vector coefficients via `[ ]` (lines 78 and 83)

vector.cpp 4/4

```
106 const Vector operator*(const double scalar,
107                        const Vector& input) {
108   Vector result(input);
109   for (int j=0; j<result.size(); ++j) {
110     result[j] *= scalar;
111   }
112   return result;
113 }
114
115 const Vector operator*(const Vector& input,
116                        const double scalar) {
117   return scalar*input;
118 }
119
120 const double operator*(const Vector& rhs1,
121                        const Vector& rhs2) {
122   double scalarproduct = 0;
123   assert(rhs1.size() == rhs2.size());
124   for (int j=0; j<rhs1.size(); ++j) {
125     scalarproduct += rhs1[j]*rhs2[j];
126   }
127   return scalarproduct;
128 }
```

- Line 115: If `Vector * double` is not implemented, one gets a cryptic runtime error:
  - Implicit type casting from `double` to `int`
  - Call of constructor with an `int`-input
  - Probably termination due to `assert` in line 123
- Operator `*` overloaded three times:
  - `Vector * double` scalar multiplication
  - `double * Vector` scalar multiplication
  - `Vector * Vector` scalar product

Example

```
#include "vector.hpp"
#include <iostream>
using std::cout;

int main() {
  Vector vector1;
  Vector vector2(100,4);
  Vector vector3 = 4*vector2;
  cout << "*** Addition\n";
  vector1 = vector2 + vector2;
  cout << "Norm1 = " << vector1.norm() << "\n";
  cout << "Norm2 = " << vector2.norm() << "\n";
  cout << "Norm3 = " << vector3.norm() << "\n";
  cout << "Skalarprodukt = " << vector2*vector3 << "\n";
  cout << "Norm " << (4*vector3).norm() << "\n";
  return 0;
}
```

- Output:

```
constructor, empty
constructor, length 100
copy constructor, length 100
*** Addition
copy constructor, length 100
deep copy, length 100
free vector, length 100
Norm1 = 80
Norm2 = 40
Norm3 = 160
Scalarproduct = 6400
Norm copy constructor, length 100
640
free vector, length 100
free vector, length 100
free vector, length 100
free vector, length 100
```