TP 3 - Problem 4 - I

```cpp
#include <cmath>
#include <iostream>

#define SHOW(arg) std::cout << "Macro SHOW “" #arg "”: " << (arg) << '\n';

class Complex {
public:
  // Default ctor: the fields are initialised with default value.
  // It is not the semantic of built-in types.
  Complex() : x_{}, y_{} {}
  // Parametrized ctor.
  Complex(double x, double y) : x_{x}, y_{y} {}
  // Copy ctor.
  Complex(Complex const &other) : x_{other.x_}, y_{other.y_} {}
  // Dtor.
  ~Complex() {}
  double abs() const {
    if (y_ == 0)
      return std::abs(x_);
```

TP 3 - Problem 4 - II

```
20        return std::sqrt(x_ * x_ + y_ * y_);
21      }
22      Complex conjugate() const { return {x_, -y_}; }
23      // Operator=: returns a reference on the assigned object.
24      Complex &operator=(Complex const &other) {
25        // Protection over c = c.
26        if (this != &other)
27          x_ = other.x_, y_ = other.y_;
28        return *this;
29      }
30      // Operator+=: returns a reference on the assigned object.
31      Complex &operator+=(Complex const &other) {
32        x_ += other.x_, y_ += other.y_;
33        return *this;
34      }
35      // Operator+: returns a new object.
36      Complex operator+(Complex const &other) {
37        // It is clever to use first the copy ctor to duplicate the left
38        // operand and, second, to reuse the += operator. The standard ensures
39        // the copy elision on return.
```

TP 3 - Problem 4 - III

```
40        Complex result{*this};
41        return result += other;
42      }
43      // Operator-=: returns a reference on the assigned object.
44      Complex &operator-=(Complex const &other) {
45        x_ -= other.x_, y_ -= other.y_;
46        return *this;
47      }
48      // Operator-: returns a new object.
49      Complex operator-(Complex const &other) {
50        Complex result{*this};
51        return result -= other;
52      }
53      // Operator*: returns a new object.
54      Complex operator*(Complex const &other) const {
55        // Create an uninitialized object.
56        Complex result{UnitialisedTag{}};
57        // Compute the complex multiplication minimising the number of real
58        // multiplications.
59        double const x_t_ox{x_ * other.x_};
```

TP 3 - Problem 4 - IV

```
60        double const y_t_oy{y_ * other.y_};
61        result.x_ = x_t_ox - y_t_oy;
62        result.y_ = (x_ + y_) * (other.x_ + other.y_) - x_t_ox - y_t_oy;
63        return result;
64      }
65      // Operator==.
66      bool operator==(Complex const &other) const {
67        return (x_ == other.x_) && (y_ == other.y_);
68      }
69      // Operator!=.: reuse the operator ==.
70      bool operator!=(Complex const &other) const { return !(*this == other); }
71
72    private:
73      // Class used as a tag. This tag flags the ctor with uninitialized
74      // fields.
75      struct UnitialisedTag {};
76      // Ctor with uninitialized fields.
77      Complex(UnitialisedTag) {}
78      // Real part.
79      double x_;
```

TP 3 - Problem 4 - V

```
80      // Imaginary part.
81      double y_;
82      // The free operator << must be a friend of my user defined class so it
83      // have access to the private data member of this class. It is preferable
84      // to declare the operator as a private one: it is only found via the
85      // argument-dependant lookup.
86      friend std::ostream &operator<<(std::ostream &os, Complex const &);
87    };
88
89    // Extends the free operator << with the user defined class. This operator
90    // returns a reference to the stream object so you can chain stream
91    // operations together.
92    std::ostream &operator<<(std::ostream &os, Complex const &c) {
93      return os << '(' << c.x_ << ", " << c.y_ << ')';
94    }
95
96    int main() {
97      SHOW(Complex{})
98      // The macro-processor is stupid: additional parentheses are required to
99      // prevent the comma from being interpreted as an argument separator.
```

TP 3 - Problem 4 - VI

```
100     SHOW((Complex{1, 2}))
101     SHOW((Complex{1, 0}.abs()))
102     SHOW((Complex{1, 1}.abs()))
103     Complex c;
104     SHOW((c = Complex{1, 2}))
105     SHOW((c += Complex{1, 2}))
106     SHOW((Complex{1, 2} + Complex{1, 2}))
107     SHOW((Complex{1, 0} * Complex{2, 0}))
108     SHOW((Complex{0, 1} * Complex{0, 2}))
109     SHOW((Complex{1, 1} * Complex{1, 1}))
110     SHOW((Complex{1, 1} == Complex{1, 1}))
111     SHOW((Complex{1, 1} == Complex{0, 1}))
112     SHOW((Complex{1, 1} == Complex{1, 0}))
113     SHOW((Complex{1, 1} != Complex{1, 1}))
114     return 0;
115 }
```

TP 3 - Problem 4 - VII

Output:

```
1  Macro SHOW “Complex{}”: (0, 0)
2  Macro SHOW “(Complex{1, 2})”: (1, 2)
3  Macro SHOW “(Complex{1, 0}.abs())”: 1
4  Macro SHOW “(Complex{1, 1}.abs())”: 1.41421
5  Macro SHOW “(c = Complex{1, 2})”: (1, 2)
6  Macro SHOW “(c += Complex{1, 2})”: (2, 4)
7  Macro SHOW “(Complex{1, 2} + Complex{1, 2})”: (2, 4)
8  Macro SHOW “(Complex{1, 0} * Complex{2, 0})”: (2, 0)
9  Macro SHOW “(Complex{0, 1} * Complex{0, 2})”: (-2, 0)
10 Macro SHOW “(Complex{1, 1} * Complex{1, 1})”: (0, 2)
11 Macro SHOW “(Complex{1, 1} == Complex{1, 1})”: 1
12 Macro SHOW “(Complex{1, 1} == Complex{0, 1})”: 0
13 Macro SHOW “(Complex{1, 1} == Complex{1, 0})”: 0
14 Macro SHOW “(Complex{1, 1} != Complex{1, 1})”: 0
```