ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecturer: Prof. B. Gärtner

# Informatik für Mathematiker und Physiker HS15

# Exercise Sheet 14

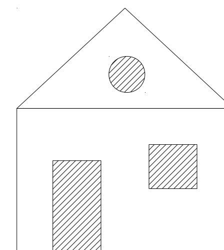
Submission deadline: (no submission)

Course URL: http://lec.inf.ethz.ch/ifmp/2015/

## Preface

This exercise sheet contains three exercises: one on inheritance, one with exam-style questions, and one challenge. The exercise with the exam-style questions consists of old exam questions, rewritten to give you an idea of how the questions on the actual exam might look like. In square-brackets [...] a reference to the initial exam question is given.

## House painting



Mr. Brush is hired to paint a house facade. To make an offer he needs to know the area that requires painting. He looks at the house facade and sees that it can be approximated by using different shapes: Triangle (width, height), Rectangle (width, height) and Circle (radius). Holes in shapes can be accounted for by subtracting the non-paintable areas, e.g. a door or a window.

Your task is to help Mr. Brush by implementing the three required shapes: `Rectangle`, `Triangle` and `Circle`. Each shape is implemented with its own class that inherits from class `Shape` and overrides the virtual member function `get_area` that returns the area defined by the shape.

We provide you the template `house_template.cpp` that is available from the course homepage. This template contains the definitions for the class `Shape`, and the `main` function that parses the test data, and calculates the area that requires painting. Note that in the `main` function the calls to the constructor are commented to make the template compile. Implement the three classes `Rectangle`, `Triangle`, and `Circle`, fill in the parameters for the constructor calls, and uncomment the constructor calls. The places where you have to add or modify code are marked with `TODO: IMPLEMENT ....`

Test data is provided as text in form of shape objects. The parser for the test data is included in the template to avoid a lengthy specification. You do not have to implement it, but you may want to take a look at it to understand how it works. Also you can use it for your own tests.

**Judge Examples** (Explanation: http://lec.inf.ethz.ch/ifmp/2015/judge_boxes.html)

```
triangle + 5.4 2.5
rectangle + 5.4 3.6
rectangle - 1.2 2.3
rectangle - 1.2 1.1
circle - 0.45
end
21.4738
```

**Submission:** https://challenge.inf.ethz.ch/team/websubmit.php?cid=5&problem=MP15141

## Exam-Style Questions

### a) [Winter 2014, Ex.1]

For each of the following 5 expressions, write C++ type and value in the corresponding gap! Assume that `x` has type `int` and value 3 *at the beginning of each subtask*.

| Expression | Type | Value |
|---|---|---|
| `0 < 1 && 1 != 1 \|\| ++x > 3` | | |
| `12.0 / 3u / 2u` | | |
| `2 + x++` | | |
| `2014 % 4 + x % 2` | | |
| `x / 2 + 9.0f / 6` | | |

### b) [Winter 2014, Ex.2]

For each of the following three code fragments, write down the sequence of numbers that it outputs!

a)
```
for (int i=0; i<100; i*=2)
     std::cout << ++i << " ";
```
Output: ______________________

b)
```
int i=0;
int j=5;
while (i != j)
     std::cout << (i+=2) - j++ << " ";
```
Output: ______________________

c)
```
void f(unsigned int x) {
     if (x == 0)
          std::cout << "-";
     else {
          std::cout << "*";
          f(x-1);
          std::cout << "*";
     }
}
f(5);
```
Output: ______________________

## c) [Winter 2014, Ex.4]

Your task is to provide an implementation of a function that computes a floating-point approximation of the exponential function

$$e^x := \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

for a given real number $x$. The implementation should be based on this definition of $e^x$ as an infinite series. To obtain an approximation of $e^x$, the function should sum up initial terms of this series (in double precision floating-point arithmetic) *until* adding the current term does not change the value obtained so far anymore. You are *not* allowed to use any external libraries. Per gap you are only allowed to write one programming line[1] (you may also leave gaps empty).

```
double exp (double x) {
    double t = 1.0;     // 1/i!
    double ex = 1.0;    // i-th approximation of e
    double xi = 1.0;    // x^i
    double ex_old = 0.0;
    ...                                 ________________________
    ...                                 ________________________
    ...                                 ________________________
    ...                                 ________________________
    ...                                 ________________________
    ...                                 ________________________
    ...                                 ________________________
}
```

## d) [Summer 2011, Ex.2]

The following questions shall be answered with either `yes` or `no`. In order to obtain points for this exercise you have to achieve at least 50% of the maximum number of points of this exercise. But for wrong answers you will *not* lose points.

a) There is no difference for the user between the following two declarations. ☐ `Yes`, ☐ `No`

```
class foo {                     struct foo {
    int i;                          int i;
};                              };
```

b) The body (`i = i - 0.6;`) of the following loop is executed less than $4$ times. ☐ `Yes`, ☐ `No`

```
int i = 3.7;
while (i > 0) {
    i = i - 0.6;
}
```

[1] By a programming line we mean here (i) *one* statement or (ii) the statements and expressions needed for *one* compound statement as e.g. the head-line of a for-loop.

c) The binary representation of the decimal number `1.625` is equal to `1.1001`. ☐ `Yes`, ☐ `No`

d) Given an arbitrary rational number `p/q`. There exists a base such that `p/q` admits a finite representation in the floating point system with this base. ☐ `Yes`, ☐ `No`

### e) [Summer 2012, Ex.6]

In this task we are going to write a class `Averager` which computes averages for given inputs. The following program illustrates the functionality by computing the average height of three humans. In general, the average of a sequence of $n$ elements is defined as the sum of these elements divided by $n$.

```
int main () {
    Averager m;
    m.add_value (1.85);                      // human 1
    m.add_value (1.79);                      // human 2
    m.add_value (1.64);                      // human 3
    std::cout << m.average_value() << "\n"; // Output: 1.76

    return 0;
}
```

In the following program code we are going to write this class `Averager` by defining suitable data members and member functions (Constructor, `add_value`, `average_value`). Also, pay attention to `const`-correctness. Each gap may contain at most one statement. PRE- and POST-conditions are not given here in order to not give you too many hints; however, you shall still `assert` for dangerous inputs. And you shall also make sure that your specified types do not reduce the precision of any results, but still do not waste memory. And finally, you may leave gaps empty (as long as this does not affect the correctness of the program).

```
// class for computing the average of n values
class Averager {
#1
    unsigned int n;  // number of values
    double s;        // sum of values

#2
    Averager () #3 {}

    #4 add_value (#5 v) #6 {
        #7
        #8
    }

    #9 average_value () #10 {
        #11
        #12
    }
};
```

#1: ______________________________ #2: ______________________________
#3: ______________________________ #4: ______________________________
#5: ______________________________ #6: ______________________________
#7: ______________________________ #8: ______________________________
#9: ______________________________ #10: ______________________________
#11: ______________________________ #12: ______________________________

## Challenge

In this exercise you are going to write your very own little implementation of C++.
You are given the base classes

```
// base class for all statements
struct statement {
  virtual void execute () {};
};

// base class for all expressions
template <typename T>
struct expression {
  virtual T get_value () const {}
};

// base class for all lvalues
template <typename T>
struct lvalue : public expression<T> {
    virtual void set_value (const T& v) {}
};
```

Derive at least the following classes from this base class:

```
// variable definition with initial value
struct variable;

// constant with initial value
struct constant;

// increment an lvalue by the value of an expression
struct increment;

// print the value of an expression
struct print;

// sequence of statements
struct block;

// repeat a block n times
struct repeat;

// execute one of two statements based on the result of a
// Boolean comparison
struct ifelse;
```

Use these classes to write meaningful programs of your choice. Here is an example (*Kleiner Gauss*) that also shows how the above classes are meant to be used.

```
// Kleiner Gauss:
// int s = 0;
// for (int i=1; i<=100; ++i)
//   s += i;

// Kleiner Gauss
variable<int> s = 0;
variable<int> i = 1;
increment<int> next_s (&s, &i);
increment<int> next_i (&i);
block update;
update.add (&next_s);
update.add (&next_i);
repeat kleiner_gauss (&update, 100);

// print result
print<int> print_result (&s);

// check result
constant<int> r = 5050;
constant<std::string> y = std::string("Richtig!");
constant<std::string> n = std::string("Falsch!");
print<std::string> say_yes (&y);
print<std::string> say_no (&n);

isequal<int> correct (&s, &r);
ifelse check_result (&correct, &say_yes, &say_no);

// execute statements
kleiner_gauss.execute();
print_result.execute(); std::cout << std::endl;
check_result.execute(); std::cout << std::endl;
```

There is a template `statements_template.cpp` available, which shows you a working example for `expression`, `lvalue`, `variable`, and `print`.