C++ Programming

Exercise Sheet 3
Secure Software Engineering Group
Philipp Schubert
philipp.schubert@upb.de

May 07, 2021

Solutions to this sheet are due on 14.05.2021 at 16:00. Please hand-in a digital version of your answers via PANDA at https://panda.uni-paderborn.de/course/view.php?id=22691.
Note: If you copy text or code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited.

This exercise sheet will help you to get some more experience in general C++ programming. In addition, you will define your first own data type. You can achieve 16 points in total.

Exercise 1.
In this exercise, you will define your own data type.

a) Define a data type called **MyType** using the **struct** keyword. (1 P.)

b) Implement all special member functions for **MyType** such that they print a message to the command-line when they are called (e.g. ”ctor called”). The special member functions are:

- **MyType(); // constructor**
- **∼MyType(); // destructor**
- **MyType(const MyType &t); // copy constructor**
- **MyType& operator= (const MyType &t); // copy assignment operator**
- **MyType(MyType &&t); // move constructor**
- **MyType& operator= (MyType &&t); // move assignment operator**

(Hint: Since both of the assignment operators must return something, just return their formal parameter. We will introduce the special **this** pointer in detail later on.) (3 P.)

c) Now write a small **main** function and declare a few variables of type **MyType**. Experiment with your novel type and carefully observe what special member function is called in which situation. Craft your **main** function in such a way that each of those special member functions is called at least once! What is the strangest special member function in your opinion? (Hint: remember the **std::move** function.) (3 P.)

d) Finally, do as you would do in professional software development and separate your implementation of MyType into header and implementation file. That is, put the type declaration (including the member functions' signatures) in a header file (`.h`) and put the implementations of the member functions in an implementation file (`.cpp`). (It makes sense to name both files `MyType` followed by the specific file ending. If you would like to use the MyType type in another file, you now have to include the corresponding header file for MyType. The linker will "plug-in" the respective definitions in the linking step.) When providing the implementations for the member functions in a separate `.cpp` file, you have to prefix the member functions' names with the corresponding type name, e.g. `MyType::MyType() { /* insert implementation here */ }` (2 P.)

Exercise 2.
So far we used `int main();` as the signature for the main function. C++ allows this other signature: `int main(int argc, char **argv);`. The contents of argc and argv are provided by the operating system. If you call your program

- argc contains the number of command-line arguments provided by the user and
- argv contains a pointer to character arrays (built-in/C-style strings) that store the arguments.

If a user executes a program like `./program Hello`, the value of argc is 2 and the value of argv[0] is "program" and argv[1]'s value is "Hello".

Write a small (and not very clever) calculator program, that reads three arguments from the command-line and performs some basic arithmetic. The following program call `./calculator value1 op value2` would perform the specified operation op on value1 and value2. Use '+' for addition, '-' for subtraction, 'x' for multiplication and '/' for division. If op is none of the above, the program should provide an error message. (Hint: use the std::stod function to convert a floating point number in string representation obtained from argv to its double representation. It is probably a good idea to check the number of arguments provided by the user **before** you try to read from argv—recall undefined behavior). (2 P.)

Exercise 3.
Consider the following function that performs a transformation from polar- to cartesian coordinates.

```
#include <cmath> // header is needed for sin() and cos()

void pol2cart(double r, double phi) {
  double x = r * cos(phi);
  double y = r * sin(phi);
}
```

In C++, a function can only return a single value. Of course it would be possible to wrap x and y into std::pair or a similar wrapper type, but this is not allowed in this exercise! Can you find a way to get the values of x and y out of this function without using return? Adjust the above function as necessary but do not use return. (2 P.)

Exercise 4.
You should be familiar with Pascal's triangle from mathematics. Pascal's triangle looks like follows:

```
1
1 1
1 2 1
```

```
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
...
```

Write a program that uses the variable **std::vector<std::vector<unsigned>> pascal(depth, std::vector<unsigned>(depth,1));** to calculate Pascal's triangle up to an arbitrary depth provided as a command-line parameter and prints the result to the command-line. As shown in the above, only print the interesting parts. Do not use the binomial coefficient. Instead, calculate each row in terms of its predecessor row! (Hint: Do not be afraid of the nested std::vector variable **pascal**—as defined in the above it is already initialized to a $depth \times depth$ matrix and all elements are set to 1; you can access its elements using **pascal[i][j]**.) (3 P.)

Exercise 5.

This is an optional exercise: You may have noticed that we wasted memory calculating Pascal's triangle by using an $n \times n$ matix. We only need a lower triangular matrix with $\binom{n}{2} = \frac{n(n-1)}{2}$ elements for the computation.

A neat way of doing so is to find a mapping $m : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ that maps 2-dimensional coordinates of a lower triangular matrix to a 1-dimensional coordinate of a **std::vector**. That way we can store the elements in one dimension:

$$T = (t)_{ij} = \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ 1 & 2 & 1 & \\ \vdots & \dots & & \ddots \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 1 & \dots \end{pmatrix}$$

You can then calculate Pascal's triangle using a variable of type **std::vector<unsigned>**. Do so! (0 P.)