TU Munich, Department of Informatics
Chair XXV: Data Science and Engineering
Prof. Dr. Thomas Neumann

Exercise for the *Practical Course: Systems Programming in C++* Summer Term 2022

Michael Freitag, Maximilian Rieger, Moritz Sichert
({freitagm,riegerm,sichert}@in.tum.de)
https://db.in.tum.de/teaching/ss22/c++praktikum

Sheet Nr. 02 (due on 12.05.2022 at 23:59)

Create a fork of the Git repository at `https://gitlab.db.in.tum.de/cpplab22/sheet02`. The repository contains a subdirectory for each exercise on this sheet, into which you should put your answers. **Do not forget to `git push` your solution before the deadline expires.**

Once the deadline for this sheet expires, a signed tag will be created in your fork automatically. Your solutions will be graded based on the state of your repository at this tag. **Do not attempt to modify or remove this tag, as we cannot grade your solution otherwise.**

Exercise 1

(10 points)

In the lecture we introduced the C++ reference documentation at:

`https://en.cppreference.com/w/cpp`

This documentation should be the first place to go for any questions regarding the C++ language or standard library. Thus, being able to navigate and understand the reference documentation is an essential skill for a C++ programmer.

(a) In the lecture we introduced the value categories *lvalues* and *rvalues* but noted that this classification is inaccurate. Consult the reference documentation about value categories, and list the five value categories that C++ actually knows (no further explanation required).

(b) Consider the following code fragment.

```
for (unsigned i = 0, j = 0; i < 10; ++i, j *= 2) {
    /* do something */
}
```

Consult the reference documentation on `for` loops, and name the *syntactic* classification (e.g. statement, expression, ...) of the code fragments `unsigned i = 0, j = 0;` and `++i, j *= 2` in one sentence.

In one of these code fragments, the comma (`,`) is an operator. Explain in which fragment this is the case and briefly outline the semantics of the comma operator (3–5 sentences).

(c) Check the reference documentation on *implicit conversions* and briefly explain the key difference between numeric promotion and numeric conversion (3–5 sentences). Give two examples for each of these two conversions.

(d) Check the reference documentation on *I/O manipulators* and answer the following questions with a short example (1 sentence each):

- How can we print hexadecimal numbers?

- How can we print numbers with a fixed width and leading zeros?
- How can we right align numbers with a fixed width?

Exercise 2 (50 points)

In this task you will implement a simple VM (virtual machine) that supports four integer and two floating point registers and a few arithmetic operations on them.

The VM has exactly four integer registers called A, B, C, and D. Each register must be able to hold a 32 bit integer (i.e. `int32_t`). It also has exactly two 32 bit floating point (i.e. `float`) registers called X and Y.

Instructions are read by the VM from the standard input (`std::cin`) and executed immediately. Every input line contains exactly one instruction with the following format:

`<opcode> [<argument>...]`

`<opcode>` is an integer with a value between 0 and 100. It identifies the instruction that is executed next. Depending on the opcode there can be zero or more arguments. Each `<argument>` is one of the following: `<ireg>`, `<freg>`, `<iimm>`, or `<fimm>`.

`<ireg>` stands for an integer register and is one of the characters A, B, C, or D for the register A, B, C, or D respectively. Similarly, `<freg>` stands for a floating point register and is either the character X or Y. When the VM starts, the contents of all registers must be zero.

An argument can also be an integer or floating point value ("immediate"). `<iimm>` stands for an integer immediate with a value that fits into an integer register and `<fimm>` for a floating point immediate with a value that fits into a floating point register.

The instructions supported by the VM can be seen in Table 1. Your implementation must support all instructions described in that table. All arithmetic integer instructions (opcodes 50–54) should wrap their values on over- and underflow, e.g., adding one to the largest possible integer results in the smallest possible one. When a program tries to divide by zero with the division instructions (opcodes 54 and 63), the VM should print the message "`division by 0`" and then stop the execution.

Note: In C++ overflow of signed integers is undefined behavior, while this VM explicitly allows it! Your implementation must be able to handle this. *Hint*: Conversion between signed integer types of different sizes follows the rules of *integral conversions* and is always allowed.

A program that loads the values 123 and 456 into registers C and D, adds them, and stores the result in register C could look like this:

```
10 C 123
10 D 456
20 C
22
20 D
50
21 C
0
```

Your implementation does not have to deal with any unexpected input such as non-existing opcodes, wrong number of arguments, incorrect values for immediates, etc.

(a) In the subdirectory for this exercise you can find the file `simplevm.cpp`. Implement the virtual machine described above by completing the function `runVM()`. When the execution of the VM finishes (either by the `halt` instruction or by division by zero) this function should return the last value of the register A.

Usually, VM instructions are not written by hand. Instead, they are generated by programs that translate a higher-level language to VM instructions (e.g. the Java compiler generates code for the Java Virtual Machine). In the next task you will implement a simple function that directly generates VM instructions without a high-level language.

(b) Implement the function `fibonacciProgram()`. This function should print a VM program that calculates the n^{th} Fibonacci number. The n^{th} Fibonacci number f_n is defined as $f_n = f_{n-1} + f_{n-2}$ with $f_0 = 0$ and $f_1 = 1$. The printed program should not use any pre-computed constants but the values for f_0 and f_1.

Exercise 3

(20 points)

In the lecture we introduced the term *undefined behavior*. The lecture then continues to say that a C++ program must never contain undefined behavior.

(a) Many compilers fall back to some sensible default behavior when encountering undefined behavior such as signed integer overflows. Briefly explain why our C++ programs nevertheless must never contain undefined behavior (3–5 sentences).

(b) In particular, accessing an uninitialized local variable is undefined behavior. Use this knowledge to write a simple C++ function that produces different return values when it is compiled with `g++` and `-O1` as opposed to `-O0`. Your code should use a conditional statement that depends on the "value" of an uninitialized local variable. Hint: You can visit the compiler explorer at `https://compiler.db.in.tum.de/` to inspect the generated assembly code (see also the next subtask).

(c) Visit the compiler explorer at `https://compiler.db.in.tum.de/` and paste the C++ function that you wrote for the previous subtask into the C++ source text field. Have the compiler explorer compile your code once with the `-O0` flag, and once with the `-O1` flag. Write down the resulting assembly for each of the two flag settings in your solution, and briefly explain the key differences in the generated assembly (2–3 sentences).

Exercise 4

(20 points)

In the lecture we introduced the GNU Debugger (`gdb`) that is a very helpful tool to debug C++ programs. In this exercise you will debug a simple program. For this the documentation for `gdb` may be useful. It can be found here:

`https://sourceware.org/gdb/current/onlinedocs/gdb/`

Throughout this exercise you will work on the program that can be found in `program.cpp` in the subdirectory for this exercise. First, you have to compile the program with compiler flags that are useful for debugging.

(a) Compile the file `program.cpp` into an executable with `g++`. Use appropriate compiler flags that enable the C++20 standard, disable all optimizations, and include debug symbols into the generated file. The executable should be called `program`. Write down the command with all arguments that you used.

For the following tasks write down all `gdb` commands and their output (if any) that you used. Start a new `gdb` session by executing the command `gdb ./program` for each task.

(b) Set a breakpoint on the `main()` function and start the program. The execution should then immediately stop at the beginning of the `main()` function. Use `gdb` commands to answer the following questions:

- Which breakpoints are currently set?
- Which *local variables* does the `main()` function have?
- Which *threads* are currently running?
- What is the current value of the *instruction pointer register* (`rip` on x86)?

(c) Set a breakpoint on the function `foo()`. Run the program and input the number 1000 twice. When the program stops run exactly one command for each item in the following list:

- Print all function arguments
- Continue execution up to the next line
- Continue execution until line 11
- Print the value of `e`
- Continue execution until the function returns

(d) Set a breakpoint on the `main()` function and run the program. Call the function `foo()` with the arguments 12 and 34 for `a` and `b`, respectively, directly from the debugger. Try calling the function with other arguments. Explain in one sentence what the function `foo()` calculates.

(e) Run the program without setting breakpoints. Find a combination of inputs that makes the program crash and return to the debugger. Explain why the program crashed (*Hint*: Look at the *stack trace*).

Opcode + arguments	Name	Description
0	`halt`	End the execution of the VM.
10 `<ireg>` `<iimm>`	`movi`	Store the value from `<iimm>` in the register `<ireg>`.
11 `<freg>` `<fimm>`	`movf`	Store the value from `<fimm>` in the register `<freg>`.
20 `<ireg>`	`loada`	Copy the value from `<ireg>` into register A.
21 `<ireg>`	`storea`	Copy the value from register A into register `<ireg>`.
22	`swapab`	Swap the values of registers A and B.
30 `<freg>`	`loadx`	Copy the value from `<freg>` into register X.
31 `<freg>`	`storex`	Copy the value from register X into register `<freg>`.
32	`swapxy`	Swap the values of registers X and Y.
40	`itof`	Convert the value in register A to a floating point value. Store the result in register X.
41	`ftoi`	Convert the value in register X to an integer. Store the result in register A.
50	`addi`	Add the values from registers A and B. Store the result in register A.
51	`subi`	Subtract the value in register B from the value in register A. Store the result in register A.
52	`rsubi`	Subtract the value in register A from the value in register B. Store the result in register A.
53	`muli`	Multiply the values in registers A and B. Store the result in register A.
54	`divi`	Divide the value in register A by the value in register B using integer division. Store the quotient in register A and the remainder in register B.
60	`addf`	Add the values in registers X and Y. Store the result in register X.
61	`subf`	Subtract the value in register Y from the value in register X. Store the result in register X.
62	`mulf`	Multiply the values in registers X and Y. Store the result in register X.
63	`divf`	Divide the value in register X by the value in register Y. Store the result in register X.

Table 1: VM Instructions