# C++ Programming

Exercise Sheet 11

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

July 09, 2021

Solutions to this sheet are due on 16.07.2021 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 familiarize yourself with the very basics of static program analysis. You can achieve 16 points in total.

**Exercise 1.**
Draw the control-flow graphs for the following functions:

```
int foo(bool b) {
  int x;
  if (b) {
    x = −1;
  } else {
    x = 0;
  }
  try {
    x = getSensorValue();
  } catch (runtime_error("bad!")) {
    std::cout << "could not read sensor value\n";
  }
  return x;
}
```

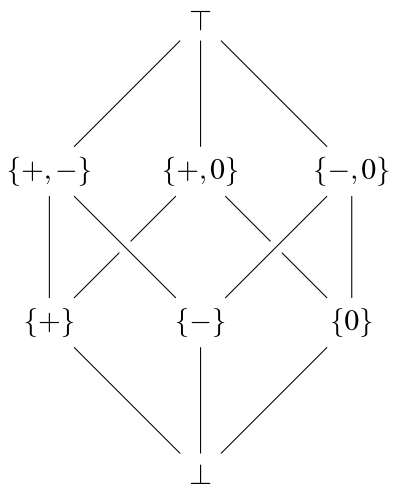
(3 P.)

```
int bar(int s, int e) {
  int x = s;
  for (int i = 1; i <= e; ++i) {
    x += i;
  }
  return x;
}
```

(3 P.)

**Important:** In the following, you are free to choose whether you would like to work on exercise 2 or exercise 3. You are also free to solve both exercises, of course. Solving both exercises might be especially interesting for those of you who would like to get a better idea of what we are doing in our Secure Software Engineering research group at Paderborn University's Heinz Nixdorf Institute.

**Exercise 2.**
Performing a sign analysis – that is an analysis that determines the sign of a variable, which can be positive, negative or zero. We define this analysis to use tuples $\langle x, S\rangle$ as data-flow facts, where $x$ is the identifier of a variable and $S \in \mathscr{L}$, a set drawn from the power set lattice over the domain $D = \{\bot, +, -, 0, \top\}$:



The special symbol $\bot = \{\}$ denotes "no information", whereas the special symbol $\top = \{+, -, 0\}$ denotes the "most imprecise element" in the lattice. The direction of this analysis is forward and the merge operator is $\cup$ (set union). An example: say at some program node $n$ you are left with the set $n_{out} = \{\langle a, \bot\rangle, \langle b, \{+\}\rangle, \langle c, \top\rangle, \langle d, \{+, -\}\rangle\}$. That means that the four data-flow facts in $n_{out}$ hold at statement $n$ and state there is a variable ...

- $a$ for which you have no information about its sign
- $b$ which is positive at this very statement
- $c$ for which you cannot tell anything of use because it can have any sign
- $d$ which may be positive or negative

Observe the following code:

```
void foo(bool b) {
  int a = 5;
  int c = 2;
  if (b) {
    a = a − 7;
  } else {
    c = c + 3;
  }
  std::cout << "a's value is: " << a << '\n';
  std::cout << "c's value is: " << c << '\n';
}
```

a) Draw **foo**'s control-flow graph. (3 P.)

b) Annotate each edge in the control-flow graph with the set of data-flow facts that holds after its respective outgoing node. If two different control-flow edges lead to some common successor node, their data-flow information must be merged before you can continue propagating the information. In this analysis, the merge operator is set union, meaning we go upwards in $\mathscr{L}$ (and therefore gain imprecision). Report on the analysis's results at the very end of **foo**? (7 P.)

Optional material(s): if you still cannot get enough of static program analysis, check out Prof. Bodden's Designing Code Analysis I (DECA I) course at
`https://youtube.com/playlist?list=PLamk8lFsMyPXrUIQm5naAQ08aK2ctv6gE`.

**Exercise 3.**
Observe the LLVM compiler's intermediate representation (LLVM IR) of the following function that performs some arithmetic computation.

```
define dso_local i32 @_Z11my_functionii(i32 %x, i32 %y) #4 {
  entry:
    %x.addr = alloca i32, align 4
    %y.addr = alloca i32, align 4
    %result = alloca i32, align 4
    %i = alloca i32, align 4
    store i32 %x, i32* %x.addr, align 4
    store i32 %y, i32* %y.addr, align 4
    %0 = load i32, i32* %x.addr, align 4
    store i32 %0, i32* %result, align 4
    store i32 1, i32* %i, align 4
    br label %for.cond

  for.cond:                               ; preds = %for.inc, %entry
    %1 = load i32, i32* %i, align 4
    %2 = load i32, i32* %y.addr, align 4
    %cmp = icmp slt i32 %1, %2
    br i1 %cmp, label %for.body, label %for.end

  for.body:                               ; preds = %for.cond
    %3 = load i32, i32* %x.addr, align 4
    %4 = load i32, i32* %result, align 4
    %mul = mul nsw i32 %4, %3
    store i32 %mul, i32* %result, align 4
    br label %for.inc

  for.inc:                              ; preds = %for.body
    %5 = load i32, i32* %i, align 4
    %inc = add nsw i32 %5, 1
    store i32 %inc, i32* %i, align 4
    br label %for.cond

  for.end:                               ; preds = %for.cond
    %6 = load i32, i32* %result, align 4
    ret i32 %6
  }
```

The LLVM language reference can be found at `llvm.org/docs/LangRef.html`. Here is a short summary of the most relevant parts:

- *%X*, is a local variables
- *i32* denotes a 32-bit integer type
- The *alloca* instruction allocates a variable on the stack (creates a local variable)
- The *store* instruction stores a value to a memory location (variable)
- The *br* instruction branches to the specified label (or labels depending on the first parameter)
- The *load* instruction loads a value from a memory location (variable)
- The *icmp slt* instruction performs a signed less than integer comparison
- The *ret* instruction returns a variable from a function

a) What does this function compute and what would be a better function name? (7 P.)

b) Translate the program (in LLVM IR representation) into a semantically equivalent piece of C++ code! (3 P.)

Optional material(s): if you still cannot get enough of LLVM, feel free to check out the talk "Introduction to LLVM" that has been given at LLVM Developer's Meeting in 2019 by Eric Christopher and Johannes Doerfert at `https://www.youtube.com/watch?v=J5xExRGaIIY`.