Exercise Class 4

1 Expressions

So far in the lectures we have seen three groups of types:

1. logical variables: `bool`;
2. integers: `int` and `unsigned int`;
3. floating point numbers: `float` and `double`.

We have already seen earlier that expressions may contain subexpressions of different types. To evaluate expressions which contain different types, we need rules for converting between them. The rule is converting to the more general type of the types involved. Below is the order from the least general type `bool` to the left and the most general type `double` to the right:

`bool` $<$ `int` $<$ `unsigned int` $<$ `float` $<$ `double`

For example, when evaluating the expression `5.0 / 2`, the compiler converts the integer `2` to a `double` `2.0` and then uses the floating point division (and not the integer division) which results in `2.5`.

1.1 Exercise: Expressions

1.1.1 Task

1. Which of the following character sequences are not C++ expressions, and why not? Here, `x` and `y` are variables of type `int`.
   a) `(y++ && y)+ 2.0`
   b) `(y++ * y)+ 2.0`
   c) `y = (x++ = 3)`
   d) `3.0 + 3 - 4 + 5`
   e) `5 % 4 * 3.0 + true * x++`
2. For all of the valid expressions that you have identified in 1, decide whether these are lvalues or rvalues, and explain your decisions.
3. Determine the values of the expressions and explain how these values are obtained. Assume that `x == 1` and `y == -1`.

1.1.2 Solution

1. 1c is invalid because `x++` is rvalue and on the left hand side of the assignment there must be a lvalue. All other cases are valid C++ expressions.

2. - 1a – rvalue.
   - 1b – rvalue.
   - 1d – rvalue.
   - 1e – rvalue.

3. - 1a: value is `2.0`. Step by step computation:

     ```
     (y++ && y) + 2.0
     (-1 && y) + 2.0   // Short circuiting.
                  // y = 0
     (true && y) + 2.0 // Short circuiting.
     (true && 0) + 2.0
     (true && false) + 2.0
     (false) + 2.0
     0.0 + 2.0
     2.0
     ```

     Note: this expression is well defined just because `&&` is short-circuiting.

   - 1b: execution of this expression exhibits undefined behaviour because execution of reading value of `y` and `y++` are not ordered. Another example that causes undefined behaviour would be:

     ```
     int a = 5;
     std::cout << a << " " << ++a << std::endl;
     ```

     "Undefined behaviour" is a specific term used in the C++ standard that means that the compiled program can behave in a completely different way than one would expect. For example, the statement above that should print two numbers to standard output, may instead reboot the computer.

   - 1d: value is `7.0`. Step by step computation:

     ```
     ((3.0 + 3) - 4) + 5
     ((3.0 + 3.0) - 4) + 5
     (6.0 - 4) + 5
     (6.0 - 4.0) + 5
     2.0 + 5
     2.0 + 5.0
     7.0
     ```

   - 1e: value is `4.0`. Step by step computation:

     ```
     ((5 % 4) * 3.0) + (true * (x++))
     (1 * 3.0) + (true * (x++))
     (1.0 * 3.0) + (true * (x++))
     3.0 + (true * (x++))
     ```

```
3.0 + (true * 1)
3.0 + (1 * 1)
3.0 + 1
3.0 + 1.0
4.0
```

2 Scopes

Scopes define the code segments of our program in which a variable (lvalue) exists. The scope of a variable starts at the point of its definition and ends at the end of the block where it was defined. For example:

```
if (x < 7) {
    int a = 8;
    std::cout << a;  // Fine, prints 8.
}
std::cout << a;  // Compiler error, a does not exist.
```

When a variable scope ends, the variable gets deallocated. Already in two weeks you will see a construct for whose correct use it is very important to know how long each variable lives.

One way to make the example above to compile, would be to declare another variable `a` in the surrounding scope:

```
int a = 2;
if (x < 7) {
    int a = 8;
    std::cout << a;     // Prints 8.
}
std::cout << a;         // Prints 2.
```

While this compiles, it is a bad programming style. First, it is harder to understand for the (human) reader because there are two variables with the same name `a`. Second, in this example, most likely the programmer had an intention to change the value of the outer `a` inside the `if` statement instead of just printing `8`. This intention would be written as:

```
int a = 2;
if (x < 7) {
    a=8;
    std::cout << a;     // Prints 8.
}
std::cout << a;         // Prints 8.
```

One important thing to remember regarding scopes and loops is that variables declared in the initialisation of the `for`-loop last for the entire loop while variables declared in the loop body last only for one loop iteration. For example, in the following code snippet:

```
int sum = 0;
for (int i = 0; i < 5; ++i) {
    int a;
```

```
    std::cin >> a;
    sum += a;
}
```

`sum` is available inside a loop and after it, `i` is available only in the loop, and `a` is available only in one loop iteration.

3 Loops

3.1 Exercise: Loop Correctness

3.1.1 Task

Can a user of the program observe the difference between the output produced by these three loops? If yes, how?

```
// Program: output_till_n.cpp

#include <iostream>

int main () {
  std::cout << "Enter a number: ";
  int n;
  std::cin >> n;


  // loop 1
  for (int i = 1; i <= n; ++i)
    std::cout << i << "\n";


  // loop 2
  int i = 0;
  while (i < n)
    std::cout << ++i << "\n";


  // loop 3
  i = 1;
  do
    std::cout << i++ << "\n";
  while (i <= n);

  return 0;
}
```

3.1.2 Solution

There are the following differences:

- Unlike loops 1 and 2, loop 3 does output `1` for input $n \leq 0$ because the statement in a `do`-loop is always executed once, before the condition is checked.

- If n is the largest possible integer, then the loops 1 and 3 may be infinite because the condition `i <= n` is going to be true for all possible `i`. **Note:** overflow of a signed integer is undefined behaviour, which means that loops 1 and 3 may also have some completely unexpected behaviour.

3.2 Exercise: Representing Loops in Terms of Other Loops

3.2.1 Task

1. Convert the following `for`-loop into an equivalent `while`-loop:

```
for (int i = 0; i < n; ++i)
    BODY
```

2. Convert the following `while`-loop into an equivalent `for`-loop:

```
while (condition)
    BODY
```

3. Convert the following `do`-loop into an equivalent `for`-loop:

```
do
    BODY
while (condition);
```

3.2.2 Solution

1. A possible way to convert a `for`-loop into an equivalent `while`-loop:

```
{   // This additional block restricts the scope of i.
    int i = 0;
    while (i < n) {
      BODY
      ++i;
    }
}
```

2. A possible way to convert a `while`-loop into an equivalent `for`-loop:

```
for ( ;condition; )
    BODY
```

3. A possible way to convert a `do`-loop into an equivalent `for`-loop:

```
BODY
for ( ;condition; )
    BODY
```