Exercise Class 2

Contents

1 Integer division and modulo - *Blackboard*

Dividing an `int` or `unsigned int` variable by another one ignores the digits behind the decimal point like in the following examples.

Here the → is informally used as an "equal" sign. We chose this approach because in source code the `=` means assignment and the students have not yet seen the `==`. On the blackboard, you can use whatever you want, just be careful with the `=`.

```
7 / 3 → 2
15 / 4 → 3
16 / 4 → 4
```

The remainder can be calculated using the modulo division

```
7 % 3 → 1
15 % 4 → 3
16 % 4 → 0
```

The original number can be regained like this

```
(15 / 4) * 4  + 15 % 4 → 15
```

The modulo division is unsuitable for calculating with non-integer numbers (we will see floating point numbers later, which are finite bit representations of non-integer numbers). It is however very useful for utility, like asking if a number is even or odd. Another example is programming a calendar and wanting to add one day at the end of February every 4 years (leap year).

```
2012 % 4 → 0     // add a day
2011 % 4 → 3     // do not add a day
```

Questions?

Exercise 1) Mixed representation of a non-integer

Write a program which reads in two integers `a` and `b`, then calculates the quotient $\frac{a}{b}$ as a mixed expression and outputs it. For example, if $a = 17$ and $b = 6$, the output should be `2 5/6`.

Solution 1) *Blackboard OR Programming Environment*

Either show the most important code lines on the blackboard or develop the program in the environment together with the students.

```cpp
// Program: division.cpp

#include <iostream>

int main() {

  // input
  int a;
  std::cin >> a;
  int b;
  std::cin >> b;

  // computation
  int whole = a / b;
  int remainder = a % b;

  // output
  std::cout << a << " divided by " << b << " equals "
            << whole << " " << remainder << "/" << b << ".\n";

  return 0;
}
```

Questions?

2 Expressions - *Blackboard*

This is a good opportunity to repeat the difference between syntax and semantics. The ultimate syntax judge for code is the compiler.

Expressions involving multiplication and addition of numbers follow the same rules as they do in math. For example, we can add parentheses to the following expression and then evaluate it step by step (*mention that the `u` means `unsigned int`*).

```
5u+5*3u
5u + (5 * 3u)
5u + 15u
20u
```

However, there are also situations we have to get used to first. An example is that, in some cases, the compiler does not guarantee to evaluate the left one of two expressions first, if they both have the same precedence. Given `a=5`, the following line of code will either result in `b=2`, if `a++` is evaluated first, or in `b=0`, if `++a` is evaluated first.

This is a good opportunity to repeat the difference between associativity and evaluation order of C++ expressions.

```
b = ++a - a++;
```

Exercises 2 and 3 are well suited for letting the students solve them in groups of 2 or 3. If you have the time, explain the problem, give the students about 5 minutes to discuss it in small groups and then discuss the solution with the whole class.

Exercise 2) Evaluating expressions - Script Exercise 2,3,4

(a) Which of the following character sequences are not C++ expressions, and why not? Here, `a` and `b` are variables of type `int`.

(i) `1*(2*3)` (ii) `(a=1)` (iii) `(1` (iv) `(a*3)=(b*5)`

(b) For all of the expressions that you have identified in (a), decide whether these are lvalues or rvalues, and explain your decisions.

(c) Determine the values of the expressions and explain how these values are obtained. Which of these values are unspecified and can therefore not be determined uniquely?

Solution 2) *Blackboard*

(a)
(iii) Not an expression, since there is no closing parenthesis for the opening one.
(iv) Invalid, since `(a*3)` is an rvalue, but the left operand of the assignment operator must be an lvalue.

(b)
(i) An rvalue by definition of the multiplication operator `*`.
(ii) An lvalue by definition of the assignment operator `=`.

(c)
(i) The value is 6, obtained by evaluating the two multiplications.
(ii) The expression assigns the value of 1 to `a` and returns the lvalue `a`, thus the value of the expression is 1.

3 Binary Representation - *Blackboard*

First, we need to learn how to derive the binary representation of a decimal number. Divide the decimal number by two and keep the rest (just like a modulo division). Divide the remaining number again, and so on and so forth until you reach 0.

$$\begin{aligned} 61 &= 2 * 30 + 1 \\ 30 &= 2 * 15 + 0 \\ 15 &= 2 * 7 + 1 \\ 7 &= 2 * 3 + 1 \\ 3 &= 2 * 1 + 1 \\ 1 &= 2 * 0 + 1, \end{aligned}$$

where the blue digits read from the bottom to the top give us the binary representation 111101. Next, the other way around, calculating the decimal representation of a binary number. Add up the contributions from the non zero-digits to get the decimal number.

binary	1	1	1	1	0	1	
decimal	32	16	8	4	0	1	= 61

Now we know how the type `unsigned int` works. If you enter a positive decimal integer, the computer calculates the binary representation like we did above and then stores it. Be aware that the computer has a finite number of bits for every number. For example, the largest positive decimal integer a 4-bit `unsigned int` can store is 15 (which is 1111).

But how can we store negative numbers? A good motivation for the so-called 2's complement representation is that we wish to be able to add 1 to -1 and get zero (or in general add x to $-x$ and get 0), which, for example, in 4 bit binary looks like this

$$\begin{array}{r} 1111 \\ +0001 \\ \hline (1)0000 \end{array}$$

The (1) gets deleted because there are only 4 bits, and we get 0. So the 1111 is the -1 in the 2's complement representation. More generally, we can obtain the 2's complement representation for a given number $-x$ where $x > 0$ by 1.) determining the usual binary representation of $+x$, then 2.) flipping all bits and finally 3.) adding 1 to the number. For example -1 in 4-bit 2's complement representation:

$$+1 \quad \xrightarrow{1.)} \quad 0001 \quad \xrightarrow{2.)} \quad 1110 \quad \xrightarrow{3.)} \quad 1111$$

However, notice that this representation of a given number $-x$ just works properly as long as $-2^3 \leq -x < 0$. If the number is smaller than -2^3 it cannot be represented using

2's complement representation (it is then mapped to another number). Positive numbers x can be represented using 2's complement representation by mapping x to itself. Notice that also here we get problems as soon as x becomes too large, namely as soon as $x \geq 2^3$.

Now we can look at the full table of numbers for 4 bit `unsigned int` and `int` representation

A slide with the following table is part of the teaching material. Either construct the table slowly and interactively with the students on the blackboard and pay special attention to the negative numbers of the `int` representation, or show them the slide with the table on it and discuss it.

bin	uint	int	bin	uint	int
0000	0	0	1000	8	-8
0001	1	1	1001	9	-7
0010	2	2	1010	10	-6
0011	3	3	1011	11	-5
0100	4	4	1100	12	-4
0101	5	5	1101	13	-3
0110	6	6	1110	14	-2
0111	7	7	1111	15	-1

It is easy to see that in the 2's complement representation there will always be one more negative than positive number, because of the 0.

Optional: In case you want to discuss over- and underflows in more detail: Ask the students what would happen if you wanted to write the number 9 to a 4-bit `int`. The formal answer is that 9 is not representable, however, explain why the result is probably -7 (or 1001) on modern computers.

Questions?

3.1 (Optional) Hexadecimal representation - *Blackboard*

By themselves your students should be able to understand how conversions to the hexadecimal system work as soon as they understand how conversions too the binary system work. There is also an exercise about hexadecimal numbers on the exercise sheet. But if you think that it helps your students you can still cover this section and give them some background information.

There are again slides with the tables in this chapter. Make sure to use them as it takes a long time copying the tables onto the black board.

In the hexadecimal system we count like this

hexadec	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	...
decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...

Why are we taking a look at this strange, unintuitive representation? Since $16 = 2*2*2*2$, we can use hexadecimal numbers to express binary numbers in a short, more compressed fashion. Then one hexadecimal digit represents a 4 bit (4 bit is half a byte, sometimes called a nibble) binary number.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	Decimal	Hexadecimal
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

This allows us to write down a 32 bit binary number (which is one `unsigned int`) as a number between `0x00000000` and `0xffffffff`.

Questions?