Informatik I (D-ITET)
Übungsstunde 4, 17.10.2016

Hossein Shafagh, shafagh@inf.ethz.ch
people.inf.ethz.ch/mshafagh/info1-2016/
Problem 3.3. From Decimal to Binary Representation

// method: use descending powers of two and subtraction determine the smallest power of two
// that is greater or equal than the number to be converted (remove leading zeros).
unsigned int power;
for(power = 1; power <= number / 2; power *= 2);

// for all powers of two: if possible to subtract, then this
// power of two is part of the binary representation of a.
std::cout << "For " << number << " the binary representation is: ";
for(; power != 0; power /= 2) {
    if (number >= power) {
        std::cout << "1";
        number -= power;
    } else {
        std::cout << "0";
    }
}
Problem 3.3. From Decimal to Binary Representation

unsigned int n;
string decimal;
do {
    if(n%2==1) {
        decimal = "1" + decimal;
    }
    if(n%2==0) {
        decimal = "0" + decimal;
        n = n/2;
    }
} while (n!=0);
Problem 3.4. Fibonacci Numbers (primes)

// try all fibonacci numbers
unsigned int prev_n = 1;
unsigned int count = 0;

for(unsigned int n = 2; n <= m;)
{
    // test if n is prime
    unsigned int d;
    for(d = 2; n % d != 0; ++d);

    if(d == n)
    {
        // found fibonacci prime
        std::cout << n << "\n";
        ++count;
    }

    // compute next fibonacci number
    unsigned int next_n = n + prev_n;
    prev_n = n;
    n = next_n;
}
Problem 3.4. Fibonacci Numbers (Overflow)

```cpp
#include <limits>  // wichtig fuer max()

// try all fibonacci numbers
unsigned int prev_n = 1;
unsigned int n = 0;
bool overflow = false;

unsigned int i;
for(i = 0; i < number && !overflow; ++i) {
    std::cout << n << "\n";

    // check for overflow
    overflow = n > std::numeric_limits<unsigned int>::max() - prev_n;

    // compute next fibonacci number
    if (!overflow) {
        const unsigned int next_n = n + prev_n;
        prev_n = n;
        n = next_n;
    }
}
```
Problem 3.4. Fibonacci Numbers (Overflow)

// check for overflow (aber zu spät!)
overflow = n < prev_n;

#include <math.h>
// check for overflow
overflow = n > pow(2, 32) - prev_n;

// hardcoded → nicht gut!
while(number < 48){
    ...
}

while- and do-loops

while (condition) {
    statement;
}

do {
    statement;
} while (condition);
while- and do-loops (examples)

```cpp
int digits = 0;
int number;
std::cin >> number;

// count number of digits
while (number != 0) {
    number /= 10;
    digits++;
}
```
Exercise 1) Loop Correctness

Does any of the following three loops not always write all positive numbers up to n to the standard output?

```cpp
#include <iostream>

int main () {
    std::cout << "Enter a number: ";
    int n, i;
    std::cin >> n;
    // loop 1
    for (i = 1; i <= n; ++i)
        std::cout << i << "\n";
    // loop 2
    i = 0;
    while (i < n)
        std::cout << ++i << "\n";
    // loop 3
    i = 1;
    do
        std::cout << i++ << "\n";
    while (i <= n);
    return 0;
}
```
for-, while- and do-loops

Goals
- as little code as possible
- easy-to-understand code

Hints
- for: Some counting is done, but the counter is not needed after the loop. e.g. repeat something n times
- while: the loop condition depends on variables which already exist before the loop. e.g. decrease x until it’s a power of 5
- do: the loop condition depends on variables which are obtained in the loop body. e.g. execute std::cin >> x until x > 3
Break & Continue!
Scope

- Scopes define the code segments of our program in which a variable (lvalue) exists.
- To understand the concept of scopes we first have to understand that a fundamental part of a program in C++ are blocks. The boundaries of such a block are marked by { and }.

```cpp
if (x < 7) {
    int a=8;
}

std::cout << a; // Compiler error, a does not exist.
```
Scope

- for-loops require special attention, a variable defined in the init statement does only exist in the scope of the for-loop.

```cpp
int sum = 0;
for (int i = 0; i < 5; ++i) {
    sum += i;
    std::cout << i << "\n"; // Outputs i
}
std::cout << sum << "\n"; // Outputs 10
std::cout << i << "\n"; // Compiler error, i does not exist
```
Scope

- **for-loops**: variables defined in the loop body exist just for the current iteration.

```cpp
unsigned int x = 2;
int i = 5;

if (x > 1) {
    int i = 3;
    std::cout << i; // outputs 3
}

std::cout << i; // outputs 5
```
The types float and double

- **float**
  - Datentyp für Zahlen mit Nachkommastellen (32 Bit)
  - Literal:
  - ohne Exponent: 288.18f,
  - mit Exponent: 0.28818e3f (100 * 0.28818)

- **double**
  - Grösserer Datentyp für Zahlen mit Nachkommastellen (64 Bit)
  - Literal:
  - ohne Exponent: 288.18,
  - mit Exponent: 0.28818e3 (100 * 0.28818)
  - Unterschied zu float: double ist genauer (grössere Präzision und grösseres Exponenten-Spektrum), braucht aber mehr Platz im Speicher (float: 32 Bit, double: 64 Bit).
Type Conversion

- The rule is converting to the more general type of the two types involved.
- The order from the least general type bool to the left and the most general type double to the right.

```
char/bool < int < unsigned int < float < double
```

- **int vs. unsigned int**: with u
- **double vs. float**: with f

- Example: 5.0/2 → 5.0/2.0 → 2.5
Exercise 2) Evaluating Expressions

- Assume that \( x \) is a variable of type \texttt{int} with value 1.

- (a) \( 3.0 + 3 - 4 + 5 \)

- (b) \( 5 \% 4 * 3.0 + \text{true} * x++ \)

- (c) \( -3 - 4u + 8.0 \)
Exercise 2) Evaluating Expressions

- Assume that x is a variable of type int with value 1.

- (a) 3.0+ 3 – 4 + 5
  (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
Exercise 2) Evaluating Expressions

- Assume that x is a variable of type int with value 1.

- (b) $5 \ % \ 4 \ * \ 3.0 \ + \ true \ * \ x++$
  
  $((5 \ % \ 4) \ * \ 3.0) \ + \ (true \ * \ (x++))$
  
  $(1 \ * \ 3.0) \ + \ (true \ * \ (x++))$
  
  $(1.0 \ * \ 3.0) \ + \ (true \ * \ (x++))$
  
  $3.0 \ + \ (true \ * \ (x++))$
  
  $3.0 \ + \ (true \ * \ 1)$
  
  $3.0 \ + \ (1 \ * \ 1)$
  
  $3.0 \ + \ 1$
  
  $3.0 \ + \ 1.0$
  
  $4.0$
Exercise 2) Evaluating Expressions

- Assume that \( x \) is a variable of type \( \text{int} \) with value 1.

- This exercise is tricky since the value to which -3 is converted depends on the number of bits the computer uses to represent \( \text{int} \) and \( \text{unsigned int} \). In the following solution we assume a 32bit representation.

\[
\begin{align*}
(c) \quad & -3 - 4u + 8.0 \\
& (-3 - 4u) + 8.0 \\
& (4294967293u - 4u) + 8.0 \\
& 4294967289u + 8.0 \\
& 4294967289.0 + 8.0 \\
& 4294967297.0
\end{align*}
\]
Debugging with Assert
Übungsblatt 4

- Problem 4.1. Loop mix-up
- Problem 4.2. Loop Analysis
- Problem 4.3. Floats and Conversions
- Problem 4.4. Approximation of π
  - The number π can be defined through various infinite sums.
Problem 4.3. Floats and Conversions

Determine type and value for each of the following expressions:

(a) $2e2 - 3e3f > -23.0$
   - false (boolean)

(b) $-7 + 7.5$
   - 0.5 (double)

(c) $1.0f / 2 + 1/3 + 1/4$
   - 0.5 (float)

(d) $1u - 2u < 0$
   - false (boolean)

(e) $1 + 2 * 3 + 4$
   - 11 (int)

(f) $\text{int}(8.5) - \text{int}(7.6)$
   - 1 (int)