4. Control Statements II

Visibility, Local Variables, While Statement, Do Statement, Jump Statements

Visibility

Declaration in a block is not “visible” outside of the block.

```c++
int main ()
{
    int i = 2;
    std::cout << i; // Error: undeclared name
    return 0;
}
```

Control Statement defines Block

In this respect, statements behave like blocks.

```c++
int main()
{
    for (unsigned int i = 0; i < 10; ++i)
    {
        s += i;
        std::cout << i; // Error: undeclared name
    }
    return 0;
}
```

Scope of a Declaration

Potential scope: from declaration until end of the part that contains the declaration.

```c++
in the block
{
    int i = 2;
    ...
}
in function body
int main() {
    int i = 2;
    ...
    return 0;
}
in control statement
for (int i = 0; i < 10; ++i) {
    s += i; ...
}
**Scope of a Declaration**

Real scope = potential scope minus potential scopes of declarations of symbols with the same name

```cpp
int main()
{
    int i = 2;
    for (int i = 0; i < 5; ++i)
        // outputs 0,1,2,3,4
        std::cout << i;
    // outputs 2
    std::cout << i;
    return 0;
}
```

**Automatic Storage Duration**

Local Variables (declaration in block)
- are (re-)created each time their declaration is reached
  - memory address is assigned (allocation)
  - potential initialization is executed
- are deallocated at the end of their declarative region (memory is released, address becomes invalid)

```cpp
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs 6, 7, 8, 9, 10
        int k = 2;
        std::cout << --k; // outputs 1, 1, 1, 1, 1
    }
    return 0;
}
```

**Local Variables**

Local variables (declaration in a block) have automatic storage duration.

**while Statement**

```cpp
while ( condition )
    statement
```

- **statement**: arbitrary statement, body of the while statement.
- **condition**: convertible to bool.
while Statement

while ( condition )
statement

is equivalent to

for ( ; condition ; )
statement

while-Statement: Semantics

while ( condition )
statement

condition is evaluated
true: iteration starts
statement is executed
false: while-statement ends.

while-statement: why?

In a for-statement, the expression often provides the progress
("counting loop")

for (unsigned int i = 1; i <= n; ++i)
    s += i;

If the progress is not as simple, while can be more readable.

Example: The Collatz-Sequence

( n ∈ N )

n0 = n
ni =
{ ni−1 \over 2 , if ni−1 even
3ni−1 + 1 , if ni−1 odd , i ≥ 1.

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, ... (repetition at 1)
The Collatz Sequence in C++

// Program: collatz.cpp
// Compute the Collatz sequence of a number n.
#include <iostream>

int main()
{
    // Input
    std::cout << "Compute the Collatz sequence for n =? ";
    unsigned int n;
    std::cin >> n;

    // Iteration
    while (n > 1) {
        if (n % 2 == 0)
            n = n / 2;
        else
            n = 3 * n + 1;
        std::cout << n << " ";
    }
    std::cout << "\n";
    return 0;
}

The Collatz Sequence

Does 1 occur for each $n$?

- It is conjectured, but nobody can prove it!
- If not, then the while-statement for computing the Collatz-sequence can theoretically be an endless loop for some $n$.

The Collatz-Sequence

n = 27:

do Statement

```
do
    statement
while ( expression );
```

- statement: arbitrary statement, body of the do statement.
- expression: convertible to bool.
do Statement

```c
do
  statement
while ( expression );
```

is equivalent to

```c
statement
while ( expression )
  statement
```

do-Statement: Semantics

- **Iteration starts**
  - *statement* is executed.
- **expression** is evaluated
  - true: iteration begins
  - false: do-statement ends.

do-Statement: Example Calculator

Sum up integers (if 0 then stop):

```c
int a;  // next input value
int s = 0;  // sum of values so far
do {
  std::cout << "next number =? ";
  std::cin >> a;
  s += a;
  std::cout << "sum = " << s << "\n";
} while (a != 0);
```

Conclusion

- Selection (conditional branches)
  - if and if-else-statement
- Iteration (conditional jumps)
  - for-statement
  - while-statement
  - do-statement
- Blocks and scope of declarations
Jump Statements

- break;
- continue;

break-Statement

- Immediately leave the enclosing iteration statement.
- useful in order to be able to break a loop “in the middle”

Calculator with break

Sum up integers (if 0 then stop)

```cpp
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    // irrelevant in last iteration:
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0);
```

Calculator with break

Suppress irrelevant addition of 0:

```cpp
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // stop loop in the middle
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0)
```
**Calculator with break**

Equivalent and yet more simple:

```cpp
int a;
int s = 0;
for (;;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // stop loop in the middle
    s += a;
    std::cout << "sum = " << s << "\n";
}
```

**Calculator with break**

Version without break evaluates `a` twice and requires an additional block.

```cpp
int a = 1;
int s = 0;
for (; a != 0;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a != 0) {
        s += a;
        std::cout << "sum = " << s << "\n";
    }
}
```

**continue-Statement**

- **Ignore negative input:**

```cpp
for (;;)
{
    std::cout << "next number =? ";
    std::cin >> a;
    if (a < 0) continue; // jump to }
    if (a == 0) break;
    s += a;
    std::cout << "sum = " << s << "\n";
}
```
Equivalence of Iteration Statements

We have seen:
- while and do can be simulated with for
- The three iteration statements provide the same "expressiveness" (lecture notes)

It even holds: Not so simple if a continue is used!

Control Flow

Order of the (repeated) execution of statements
- generally from top to bottom...
- ...except in selection and iteration statements

Control Flow if else

Control Flow for
Control Flow: the Good old Times?

Observation
Actually, we only need if and jumps to arbitrary places in the program (goto).

Models:
- Machine Language
- Assembler ("higher" machine language)
- BASIC, the first programming language for the general public (1964)

BASIC and home computers...

...allowed a whole generation of young adults to program.

Home-Computer Commodore C64 (1982)
Spaghetti-Code with goto

Output of all prime numbers with BASIC

```
10 N=2
20 D=1
30 IF N=D GOTO 100
40 IF N/D = INT(N/D) GOTO 70
50 GOTO 30
60 N=N+1
70 GOTO 20
100 PRINT N
110 GOTO 70
```

The “right” Iteration Statement

Goals: readability, conciseness, in particular
- few statements
- few lines of code
- simple control flow
- simple expressions

Often not all goals can be achieved simultaneously.

Odd Numbers in \{0, \ldots, 100\}

First (correct) attempt:
```
for (unsigned int i = 0; i < 100; ++i)
{
    if (i % 2 == 0)
        continue;
    std::cout << i << "\n";
}
```

Odd Numbers in \{0, \ldots, 100\}

Less statements, less lines:
```
for (unsigned int i = 0; i < 100; ++i)
{
    if (i % 2 != 0)
        std::cout << i << "\n";
}
```
Odd Numbers in \{0, \ldots, 100\}

*Less* statements, *simpler* control flow:

```cpp
for (unsigned int i = 1; i < 100; i += 2)
    std::cout << i << "\n";
```

This is the “right” iteration statement!

Jump Statements

- implement unconditional jumps.
- are useful, such as `while` and `do` but not indispensable
- should be used with care: only where the control flow is *simplified* instead of making it *more complicated*

The `switch`-Statement

```cpp
int Note;
...
switch (Note) {
    case 6:
        std::cout << "super!";
        break;
    case 5:
        std::cout << "cool!";
        break;
    case 4:
        std::cout << "ok.";
        break;
    default:
        std::cout << "hmm...";
}
```

Semantics of the `switch`-statement

- condition: Expression, convertible to integral type
- statement: arbitrary statement, in which case and default-lables are permitted, `break` has a special meaning.

- condition is evaluated.
- If statement contains a case-label with (constant) value of condition, then jump there
- otherwise jump to the default-label, if available. If not, jump over statement.
- The `break` statement ends the switch-statement.
Control Flow switch

switch

statement

break

case

default

Control Flow switch in general

If break is missing, continue with the next case.

7: ???
6: ok.
5: ok.
4: ok.
3: oops!
2: oops!
1: oooops!
0: ???

switch (Note) {
    case 6:
    case 5:
    case 4:
        std::cout << "ok."
        break;
    case 1:
        std::cout << "o";
    case 2:
        std::cout << "o";
    case 3:
        std::cout << "oops!"
        break;
    default:
        std::cout << "???";
}

“Proper Calculation”

// Program: fahrenheit_float.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    float celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
                << 9 * celsius / 5 + 32 << " degrees Fahrenheit.";
    return 0;
}

5. Floating-point Numbers I

Types float and double; Mixed Expressions and Conversion; Holes in the Value Range
Fixed-point numbers

- fixed number of integer places (e.g. 7)
- fixed number of decimal places (e.g. 3)

0.0824 = 0000000.082  third place truncated

Disadvantages

- Value range is getting even smaller than for integers.
- Representability depends on the position of the decimal point.

Floating-point numbers

- fixed number of significant places (e.g. 10)
- plus position of the decimal point

82.4 = 824 \cdot 10^{-1}

0.0824 = 824 \cdot 10^{-4}

- Number is $Mantissa \times 10^{Exponent}$

Types float and double

- are the fundamental C++ types for floating point numbers
- approximate the field of real numbers ($\mathbb{R}, +, \times$) from mathematics
- have a big value range, sufficient for many applications (double provides more places than float)
- are fast on many computers

Arithmetic Operators

Like with int, but ...

- Division operator / models a “proper” division (real-valued, not integer)
- No modulo operators such as % or %=
Literals are different from integers by providing

- decimal point
  - 1.0 : type double, value 1
  - 1.27f : type float, value 1.27
- and / or exponent.
  - 1e3 : type double, value 1000
  - 1.23e-7 : type double, value 1.23 \times 10^{-7}
  - 1.23e-7f : type float, value 1.23 \times 10^{-7}

Computing with float: Example

Approximating the Euler-Number

\[
e = \sum_{i=0}^{\infty} \frac{1}{i!} \approx 2.71828\ldots
\]

using the first 10 terms.

Computing with float: Euler Number

```
// Program: euler.cpp
// Approximate the Euler number e.
#include <iostream>
int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f;  // 1/i!
    float e = 1.0f;  // i-th approximation of e
    std::cout << "Approximating the Euler number...\n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
        t /= i;  // 1/(i-1)! -> 1/i!
        e += t;
        std::cout << "Value after term " << i << " : " << e << "\n";
    }
    return 0;
}
```

Value after term 1: 2
Value after term 2: 2.5
Value after term 3: 2.66667
Value after term 4: 2.70833
Value after term 5: 2.71667
Value after term 6: 2.71806
Value after term 7: 2.71825
Value after term 8: 2.71828
Value after term 9: 2.71828
Mixed Expressions, Conversion

- Floating point numbers are more general than integers.
- In mixed expressions integers are converted to floating point numbers.

\[
9 * \text{celsius} / 5 + 32
\]

Value range

Integer Types:
- Over- and Underflow relatively frequent, but ...
- the value range is contiguous (no "holes"): \( \mathbb{Z} \) is "discrete".

Floating point types:
- Overflow and Underflow seldom, but ...
- there are holes: \( \mathbb{R} \) is "continuous".

Holes in the value range

```cpp
float n1;
std::cout << "First number =? ";
std::cin >> n1;

float n2;
std::cout << "Second number =? ";
std::cin >> n2;

float d;
std::cout << "Their difference =? ";
std::cin >> d;

std::cout << "Computed difference - input difference = ">
<< n1 - n2 - d << "\n";
```

output 2.23517e-8

What is going on here?