Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger

LS 11 Algorithm Engineering

Lecture 3 • Winter 2011/12 • Oct 25

technische universität dortmund

department of computer science

Visual C++: Problems and Solutions

- New section on web page (scroll down)
- Some typical problems we experienced with VC++ and solutions to fix them
- Will be extended if necessary

Loops Continued

- **do**-**while**-loops: Similar as **while**-loops, but the condition is evaluated after each iteration.
- The general form is:

```
do
    statement;
while ( condition );
```

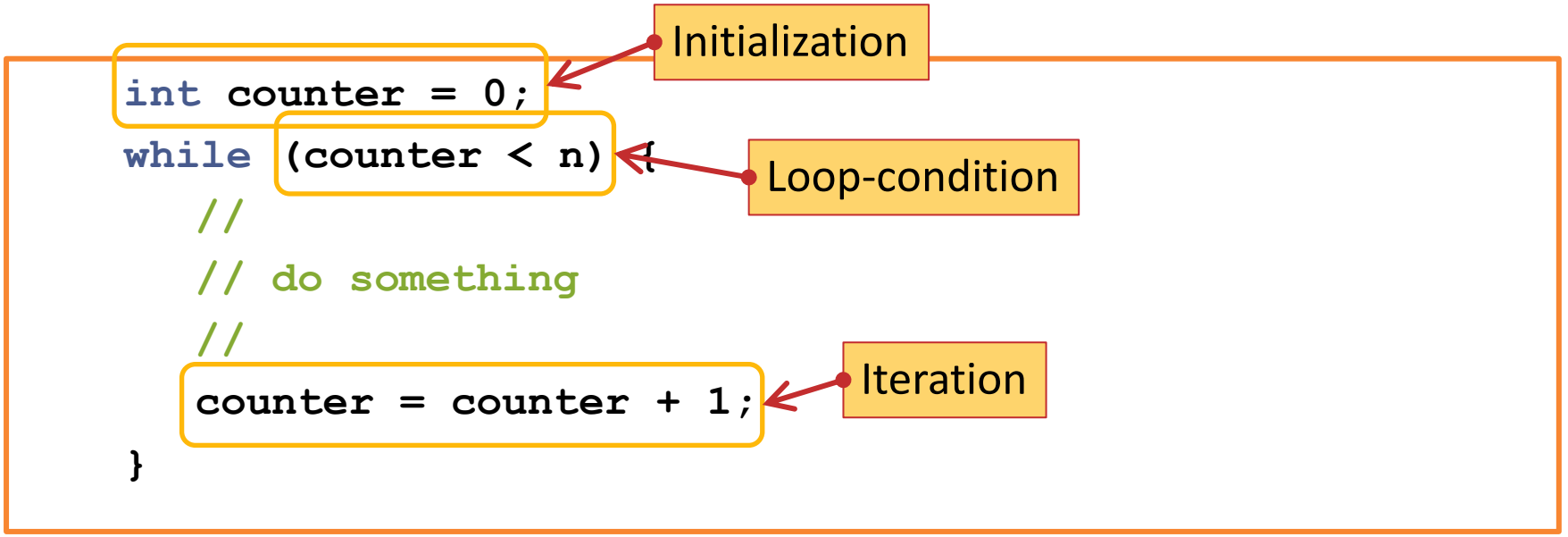
- *statement* is executed before *condition* is evaluated.
- If *condition* evaluates to false, the loop is terminated.
- *statement* is executed at least once!

Example: Sum up numbers until 0 is entered

```
#include <iostream>
using namespace std;

int main() {
    int number, sum = 0;

    do {
       cin >> number;
       sum = sum + number;
    } while (number != 0);

    cout << "sum = " << sum << endl;

    return 0;
}
```

A typical pattern for while-loops

- We often use **while**-loops of the following form:



- There is a special syntax for writing such kinds of **while**-loops!

for-Loops

- The general form is:

```
for ( init-statement;  condition;  iter-statement )
    body-statement;
```

- Semantics:

1. First *init-statement* is executed (only once!)
2. Then *condition* is evaluated: false → terminate loop
3. Then *body-statement* is executed.
4. Then *iter-statement* is executed.
5. Go to step 2.

Example: Print first n square numbers

```
int n = 10; /* for example */

for (int i = 1; i <= n; i = i + 1)
   cout << i * i << endl;
```

This is equivalent to:

```
int n = 10; /* for example */

int i = 1;
while (i <= n) {
   cout << i * i << endl;
   i = i + 1;
}
```

Transforming for-loops into while-loops

```
for(init-stat; cond; iter-stat)
{
    body-statement-1;
    …
    body-statement-n;
}
```

```
{
    init-stat;
    while( cond )
    {
        body-statement-1;
        …
        body-statement-n;
        iter-stat;
    }
}
```

We will understand the block around the **while**-statement next time when discussing scope and lifetime of variables!

The break-Statement

- Used to terminate a loop at some place in the body of the loop.
- Syntax:

```
break;
```

- Semantics: Terminates the nearest enclosing **while**-, **do-while**-, or **for**-loop statement.
- Example:

```
for (int i = 1; i <= 100; i = i + 1) {
   int x; cin >> x;
   if (x == 0)
      break;
   cout << 100 / x << endl;
}
```

The continue-Statement

- Used to terminate an iteration of a loop.
- Syntax:

```
continue;
```

- Semantics: Terminates the current iteration of the nearest enclosing **while**-, **do-while**-, or **for**-loop statement.
- Example:

```
for (int i = 1; i <= 100; i = i + 1) {
   int x; cin >> x;
   if (x == 0)
      continue;
   cout << 100 / x << endl;
}
```

Strings

- Strings (texts) can be stored in variables of type std::string.
- std::string is part of the C++ standard library, not the C++ language itself.
- We have to include this additional functionality first:

```
#include <string>
```

- String literals have to be enclosed in quotation marks "…"

```
std::string name;
name = "Carsten";
```

Using Strings

- We can use std::string almost like a built-in type:
  - Initialization: `std::string name = "Carsten"`
  - Comparing for equality with `==`
  - Assignment using `=`
  - Concatenation using `+`
- Example:

```
string name = "Gutwenger";
string firstName = "Carsten";

name = firstName + " " + name;
cout << "The name is " << name << endl;

if(name == "Carsten Gutwenger")
   cout << "That's me!" << endl;
```

Warning

- Don’t compare string literals!

```
if ("abc" == "abc")
    cout << "Undefined if we get here!" << endl;
```

Special Characters

- Within string literals, you can use the following special characters:
  - `\n` newline character
  - `\t` tab stop character
  - `\"` quotation mark
  - `\\` backslash character
- Example:

```
"This \"string\"\ngoes over two lines"
```

Output formatting

- C++ supports various manipulators for nicely formatting output.
- You need to include this special functionality:

```
#include <iomanip>
```

- Manipulators are inserted into the output stream (with the output operator <<) just as you print data.
- The following manipulators are useful for integers:
  - setw(*n*): sets the number n of characters to be used as field width for the next insert operation.
  - left: output is left-aligned in the output field.
  - right: output is right-aligned in the output field.

Example: Formatting Output

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int a = 7, b = 12345;

    cout << left;
    cout << setw(10) << a << "***" << endl;
    cout << setw(10) << b << "***" << endl;
    cout << right;
    cout << setw(10) << a << "***" << endl;
    cout << setw(10) << b << "***" << endl;

    return 0;
}
```

Output:

```
7          ***
12345      ***
         7***
     12345***
```

10 characters

The bool Data Type

- Variables of type **bool** store one of two possible values: **true** and **false**
- Example:

```
bool found = true;
bool isGreater = ( a > b ); // a and b are variables
```

- Operators for Boolean expressions:
  - logical AND: `&&`
  - logical OR: `||`
  - logical NOT: `!`
- Example:

```
!found || ( a > b && a < 2*b )
```

Printing bool values

- By default, bool values are printed as 0 or 1 (corresponding to false or true).
- You can change this behavior with manipulators!
  - **boolalpha**: print false or true
  - **noboolalpha**: print 0 or 1
- Example:

```
bool b = true;

cout << b << endl;
cout << boolalpha << b << endl;
cout << noboolalpha << b << endl;
```

Output:

```
1
true
1
```

Preparations for next week

- Floating-point numbers (`float`, `double`)
- Increment and Decrement (e.g. ++, += operators)
- Scope and lifetime of variables
- C++-Vectors (std::vector)