Object Oriented Programming I

Topics today:

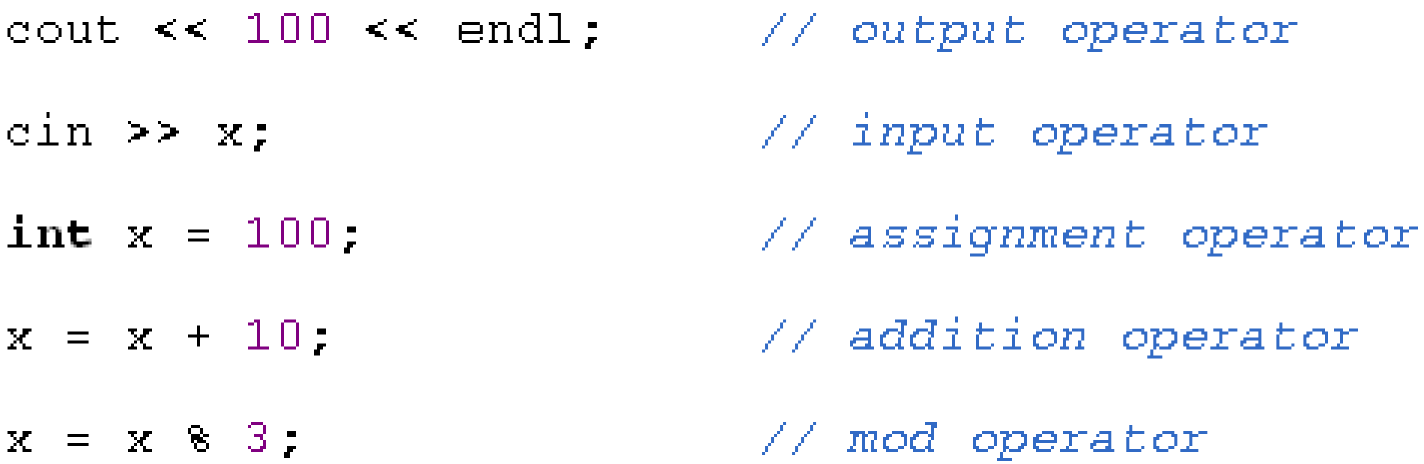
- Operators
- Literals and Expressions
- Loops
- Conditions

C++ Operators

Operators

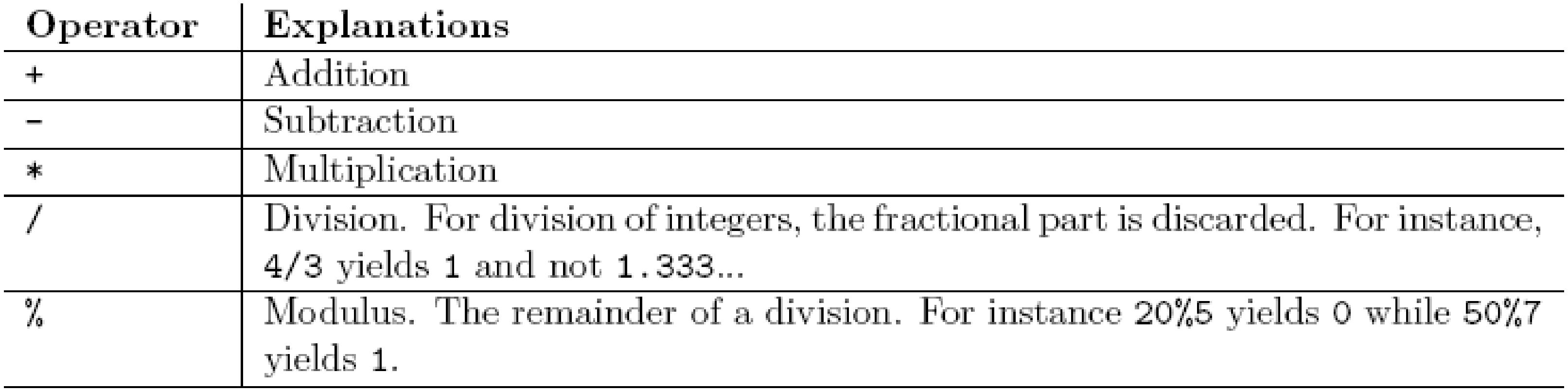
An operator is a symbol that performs an action

Examples of operators



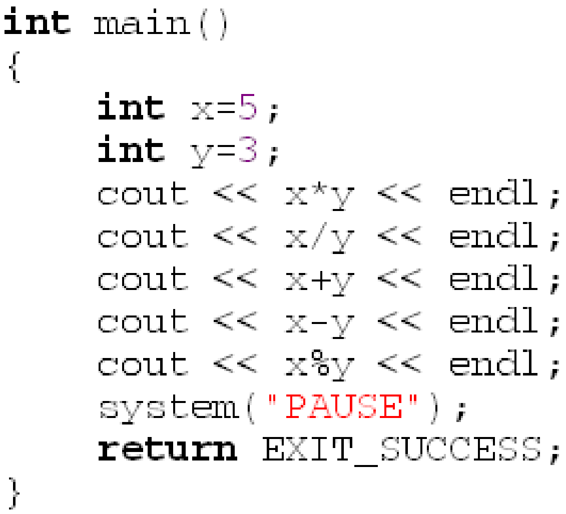
```
cout << 100 << endl;        // output operator

cin >> x;                   // input operator

int x = 100;                // assignment operator

x = x + 10;                 // addition operator

x = x % 3;                  // mod operator
```

Arithmetic C++ operators



Operator	Explanations
+	Addition
-	Subtraction
*	Multiplication
/	Division. For division of integers, the fractional part is discarded. For instance, 4/3 yields 1 and not 1.333...
%	Modulus. The remainder of a division. For instance 20%5 yields 0 while 50%7 yields 1.

Example:



```
int main()
{
    int x=5;
    int y=3;
    cout << x*y << endl;
    cout << x/y << endl;
    cout << x+y << endl;
    cout << x-y << endl;
    cout << x%y << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

mod Operator

- Let a,m be positive integers
- a mod m is the remainder of the division of a by m
- Examples: 10 mod 3=1, 21 mod 7=0
- Mod operator in C++ : **%**
- Examples: **10%3, 21%7**

Problem 1

a) Write and test a C++ program that creates two variables of type integer, initializes them to 24 and 10 and applies and five arithmetic operators to them: **+, -, /, *, %.**
Print the results to the screen with **cout**

b) Compute $100^{16} \bmod 73$

Problem 2

Write a program that does the following.

- Read a date from the keyboard (format DD MM YYYY)
- Output the weekday corresponding to this date

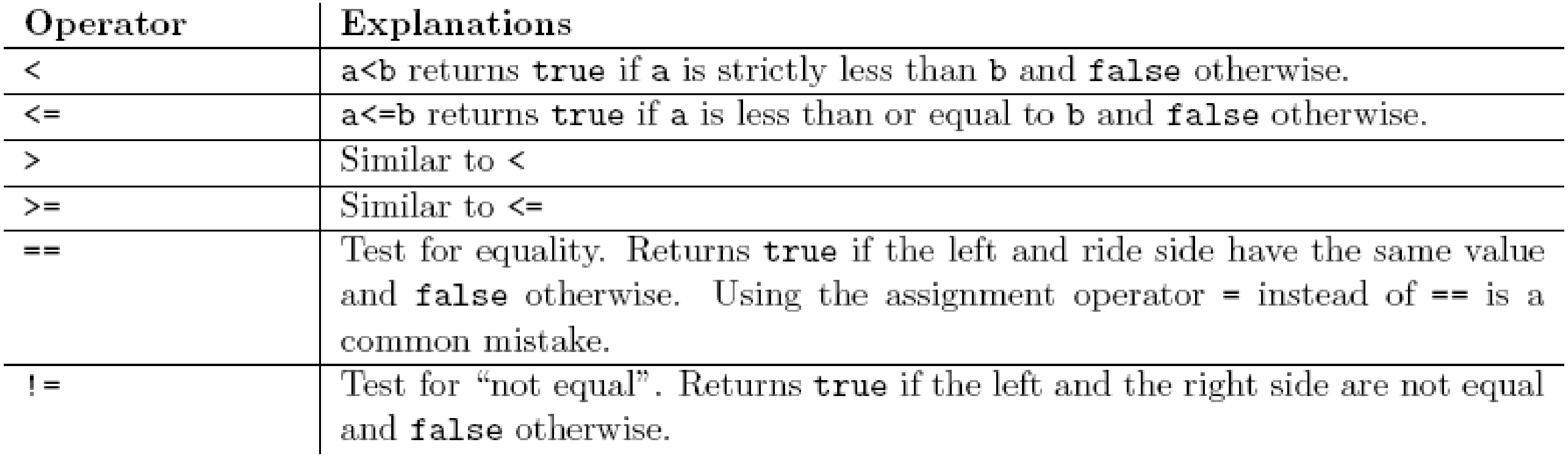
Hint: Compute *w* as follows

$$
\begin{aligned}
t &= \lfloor (12 - \texttt{month})/10 \rfloor \\
y &= \texttt{year} - t \\
m &= \texttt{month} + 12t \\
c &= \lfloor y/100 \rfloor \\
Y &= y \bmod 100 \\
w &= (\texttt{day} + Y + \lfloor Y/4 \rfloor + \lfloor c/4 \rfloor + 5c + \lfloor (26(m+1))/10 \rfloor) \bmod 7
\end{aligned}
$$

("$\lfloor\ \rfloor$"is the floor function)

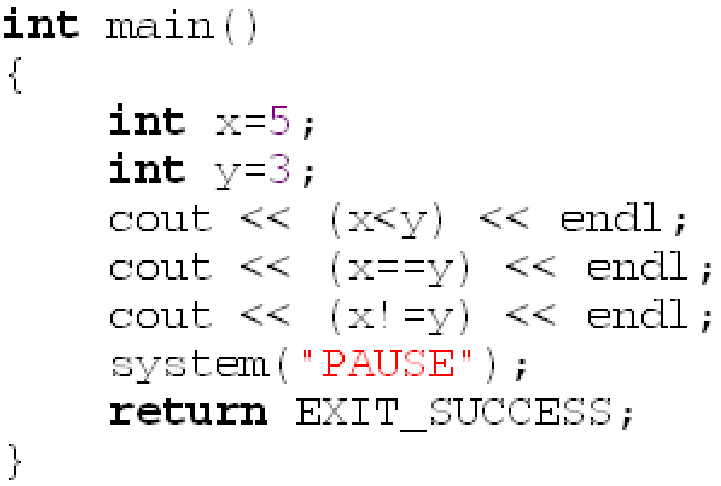
Then w=0 means Saturday, w=1 Sunday etc.

Basic Logical C++ operators



Operator	Explanations
`<`	`a<b` returns **`true`** if `a` is strictly less than `b` and **`false`** otherwise.
`<=`	`a<=b` returns **`true`** if `a` is less than or equal to `b` and **`false`** otherwise.
`>`	Similar to `<`
`>=`	Similar to `<=`
`==`	Test for equality. Returns **`true`** if the left and ride side have the same value and **`false`** otherwise. Using the assignment operator `=` instead of `==` is a common mistake.
`!=`	Test for "not equal". Returns **`true`** if the left and the right side are not equal and **`false`** otherwise.

Example:



```
int main()
{
    int x=5;
    int y=3;
    cout << (x<y) << endl;
    cout << (x==y) << endl;
    cout << (x!=y) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

&& (logical and) and || (logical or)

- Let **A** and **B** be boolean expressions (true or false)
- **A&&B** is true if and only if **A** and **B** are true
- **A||B** is true if and only if **A** or **B** is true (includes the case where both are true)
- **&&** has priority over **||**, i.e. , **&&** is evaluated first
- Recommendation: always put parentheses around boolean expressions (avoids mistakes and is easier to read)

&&, || Examples

`((5<7) && (4<5))` → true

`((5<7) && (4>5))` → false

`( (4<3) || (4!=5) )` → true

`( ((4<3) && (4==4)) || (4==5) )` → false

`( ((4<3) || (4==4)) && (4!=5) )` → true

Question

Is the following expression true or false?

```
((1<=1) || (0==0))  && ((5<5) || (5!=5))
```

Answer: False

What about the following?

```
(1<=1) || (0==0)  && ((5<5) || (5!=5))
```

Answer: True

Problem 3

- Read an integer x from the keyboard
- Write down a logical expression which is true if and only if x simultaneously satisfies the following conditions:
  (i) 1000 < x ≤ 10000
  (ii) x is odd
  (iii) x is divisible by 7
  (iv) x is divisible by 41 or 43
- Declare a bool variable y. Set the value of y to the value of the logical expression above
- Print y to the screen
- Find a value of x for which y becomes true

Literals and Expressions

Literals

A literal is a value we can type directly into C++ code.

Examples of literals:

```
23424    // integer literal
2.343    // double literal
"Hi!"    // string literal
'A'      // character literal
```

Literals (continued)

- Every literal has a type
- We can find out the type using the typeid-operator

Example:

```
cout << typeid(77+5.6).name() << endl;
```

The output will be “**d**” for double. Hence the expression **77+5.6** has type double

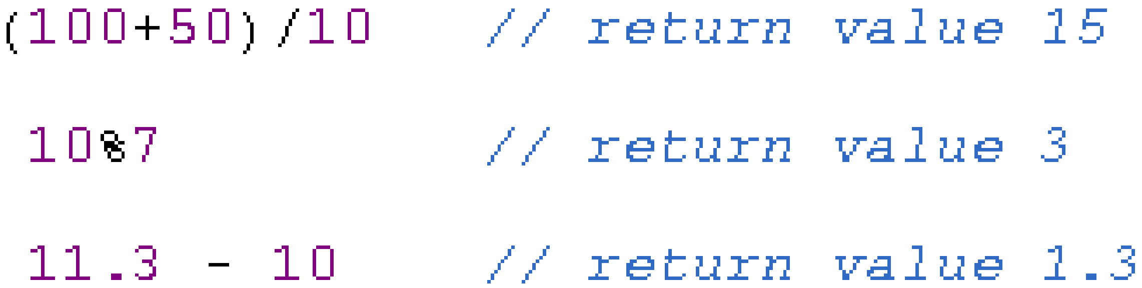
Expressions

- An expression is built from literals, variables, operators and parentheses and must follow the syntax rules
- Every expression is evaluated by the C++ program. The result is called the value of the expression
- The value of an expression can be outputted by **cout << expression;**
- An expression is not a complete C++ command. It can only be part of a C++ command

Expressions (continued)

- The type of an expression is the type of its value
- Rule of thumb: the type of an expression is the “most complicated type” occurring in it

Examples of Expressions



```
(100+50)/10     // return value 15

 10%7           // return value 3

 11.3 - 10      // return value 1.3
```

Question

Which of the following are expressions?

`3<4`

Yes, return value 1 (true)

`cout << 3;`

No, semicolon not allowed in expression (this example is a complete command)

`(((2<5)% 1000)>= 700)+50`

Yes, but don't use something like this (too confusing)

`cin >> x`

Yes, if x has been declared. The return value is 0 if and only if the input failed (useful for checking correctness of input)

Question

What is the return value of the expression **7/3** ?

Answer: **2** (integer division)

Question

What is the return value of the expression **'A'+1** ?

Answer: **66**

- **'A'** has type **char** and ascii-code 65
- **1** has type **int**
- The expression has type **int** since **int** is more complicated than **char**

Some Rules for Expression Values

- We can find out the value of an expression by **cout << expression**;
- The value of a literal is the literal itself
- The value of a variable name is the value of the variable
- The value of an assignment is the assigned value
- The value of an input operation like **cin >> x** is **true** (different from 0) if and only if the input was successful

Control Structures

Control structures

- To perform a task repeatedly in C++, we use loops: **for, while, do-while**
- To make the program execution dependent on conditions, we use conditional structures: **if-else, switch**
- To jump to another point of the program, we use jump statements: **break, continue, goto**

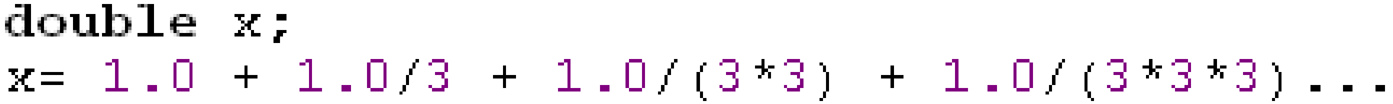
Loops

Doing Operations Repeatedly

- How to compute a sum like

$$\sum_{n=0}^{100}\frac{1}{3^n}=1+\frac{1}{3}+\frac{1}{9}+\frac{1}{27}\cdots\ \frac{1}{3^{100}}\quad \text{in C++?}$$

- Possible, but impractical:



```
double x;
x= 1.0 + 1.0/3 + 1.0/(3*3) + 1.0/(3*3*3)...
```

- Solution: loops

Loops

- If we know in advance, how many times an operations has to be repeated, we use a **for-loop**
- If we don't know in advance, we need to use a **while-loop**
- Warning: while-loops are very error prone

For-Loops

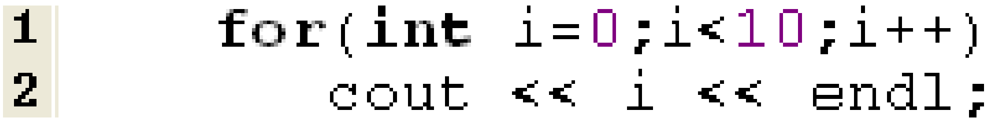
for-loop

```
for(initialization;condition;increase)
    statement
```

Explanations:

- The statement is executed as long as the condition is true
- initialization and increase are expressions used to control the values of variables occurring in the condition
- Each execution of the statement is called an iteration
- The statement can be a single command, or a statement block enclosed by braces {…}
- Everything after the statement does not belong to the for-loop
- Variables declared inside the statement are “out of scope” after the statement

for-loop example



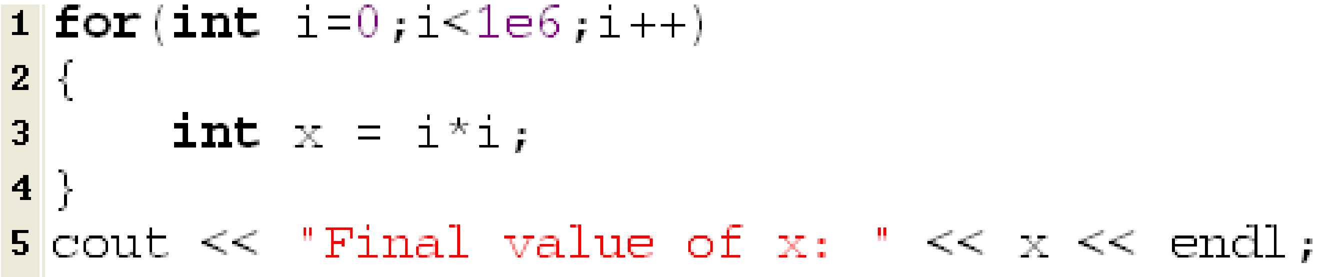
```
    for(int i=0;i<10;i++)
        cout << i << endl;
```

Explanations:

- The initialization here is **int i=0**
- **i** is called a **counter variable**
- The condition is **i<10**
- The increase expression is **i++**
- The semicolons and parenthesis are necessary
- The statement of the for-loop is **cout << i << endl**;
- Result: The numbers 0,1,…,9 will be printed to the screen

Question

What is wrong with the following?



```
1 for(int i=0;i<1e6;i++)
2 {
3       int x = i*i;
4 }
5 cout << "Final value of x: " << x << endl;
```

A lot!

- Line 5 does not belong to for-loop, `x` undeclared there
- Integer overflow as `i*i` will be much larger than 2 billions for big `i`
- Never declare variables (except counter variables) inside loops – bad style (but not syntax error)

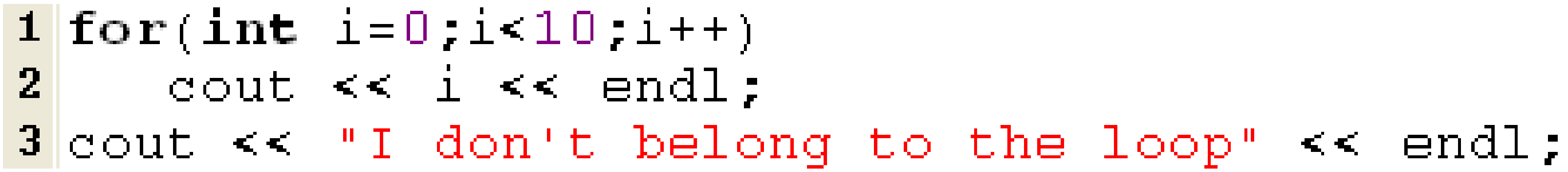
for-loop Examples

1. What belongs to the loop, what is not part of it?
2. How to use curly braces to include a command in a loop
3. Demonstrate that everything after closing curly brace does not belong to for-loop
4. Demonstrate that counter variable is out of scope after loop
5. How to avoid that counter variable gets out of scope

6. Compute $\sum_{n=1}^{100} n^2$

7. Compute $\prod_{\mathrm{n=2}}^{1000} \frac{n^3-1}{n^3+1} = \frac{7}{9} \cdot \frac{26}{28} \cdot \frac{63}{65} \cdots$

for-loop: Example 1

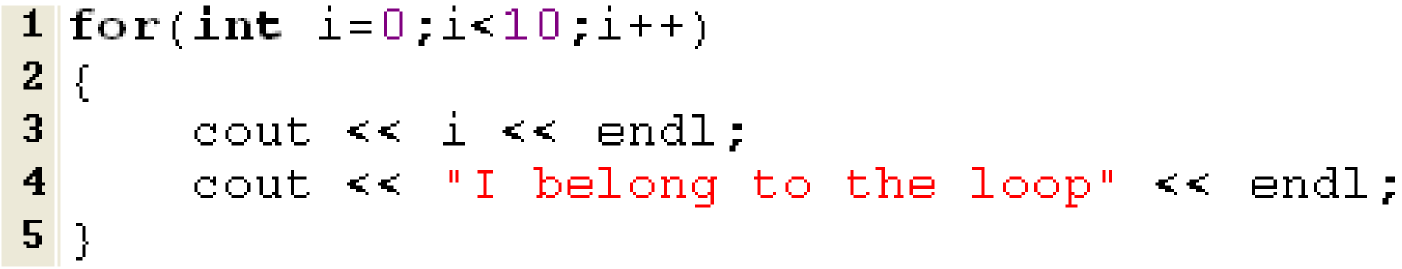


```
1 for(int i=0;i<10;i++)
2     cout << i << endl;
3 cout << "I don't belong to the loop" << endl;
```

Explanations:

- The command in line 3 does not belong to the for-loop
- To include two or more statements in a for-loop, we need to enclose them in curly braces

for-loop: Example 2



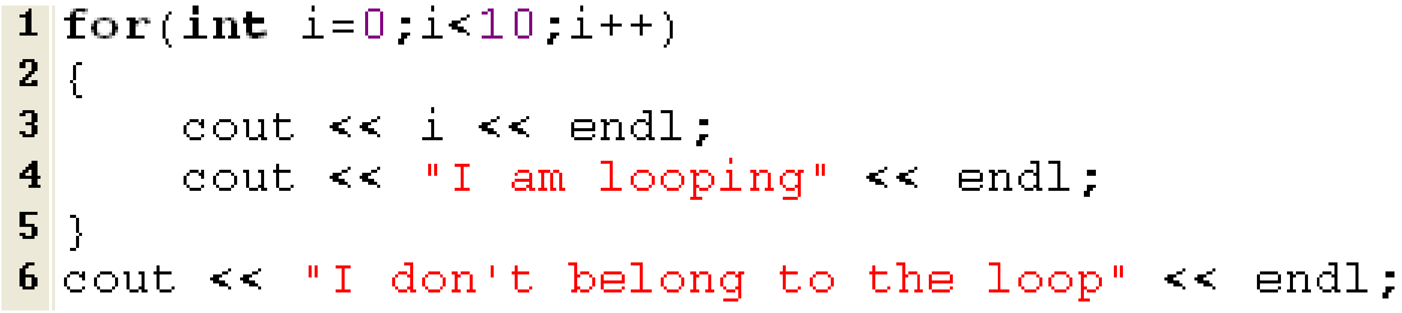
```
1 for(int i=0;i<10;i++)
2 {
3       cout << i << endl;
4       cout << "I belong to the loop" << endl;
5 }
```

Explanations:

•The commands in lines 3 and 4 both belong to the for-loop since they are enclosed in curly braces.

•“I belong to the loop” will be printed on the screen 10 times in total

for-loop: Example 3

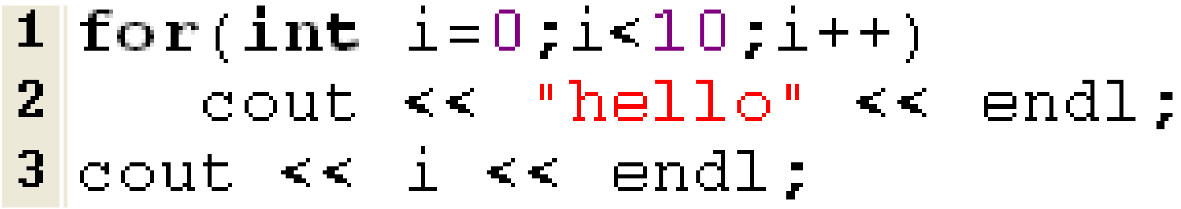


```
1 for(int i=0;i<10;i++)
2 {
3       cout << i << endl;
4       cout << "I am looping" << endl;
5 }
6 cout << "I don't belong to the loop" << endl;
```

Explanations:

- The commands in lines 3 and 4 both belong to the for-loop since they are enclosed in curly braces
- Everything after the closing curly brace does not belong the loop

for-loop: Example 4

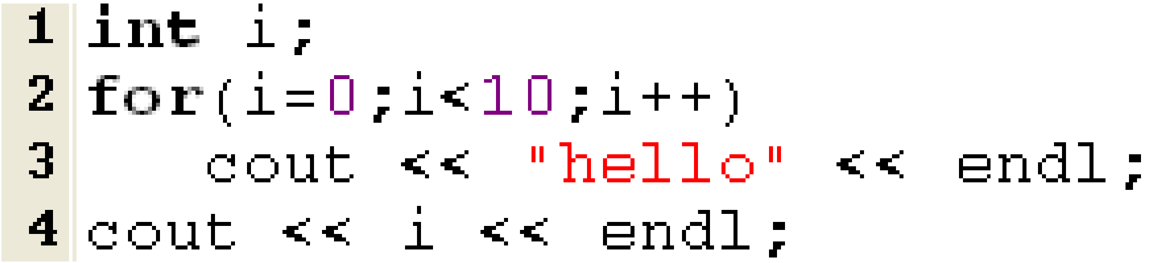


```
1 for(int i=0;i<10;i++)
2     cout << "hello" << endl;
3 cout << i << endl;
```

Explanations:

- Compiler error!
- If declared inside the loop, a variable (here i) is not valid outside the loop
- The “i” in line 3 is a syntax error

for-loop: Example 5



```
int i;
for(i=0;i<10;i++)
    cout << "hello" << endl;
cout << i << endl;
```

Explanations:

•No syntax error here!

•Here i is declared before the loop and hence also valid after the loop

•Question: what is the value of i after the execution of the loop?

•Answer: 10

Problem 4

a) Compute $\sum_{n=1}^{100} n^2$

b) Compute $\prod_{n=2}^{1000} \frac{n^3-1}{n^3+1} = \frac{7}{9} \cdot \frac{26}{28} \cdot \frac{63}{65} \cdots$

Results (for checking correctness)

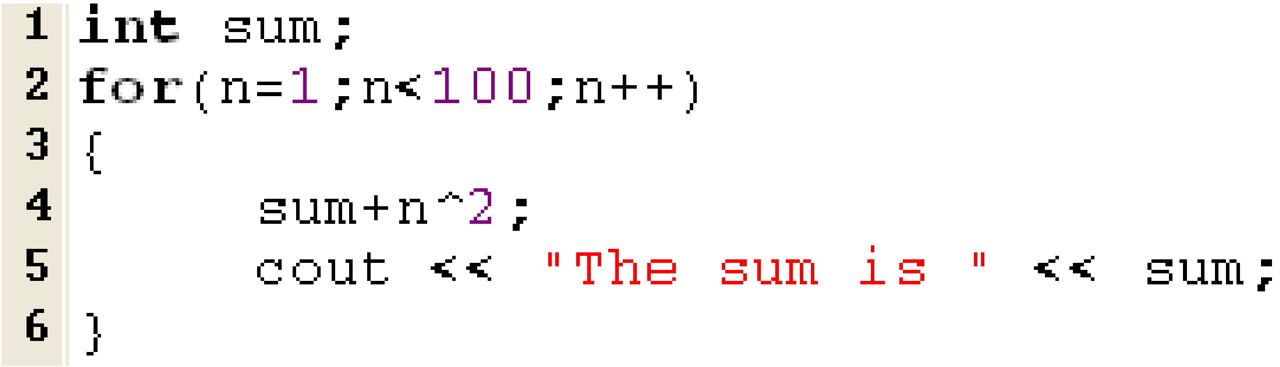
a) 338350

b) Something close to 2/3 (the infinite product is equal to 2/3)

How not to compute a sum

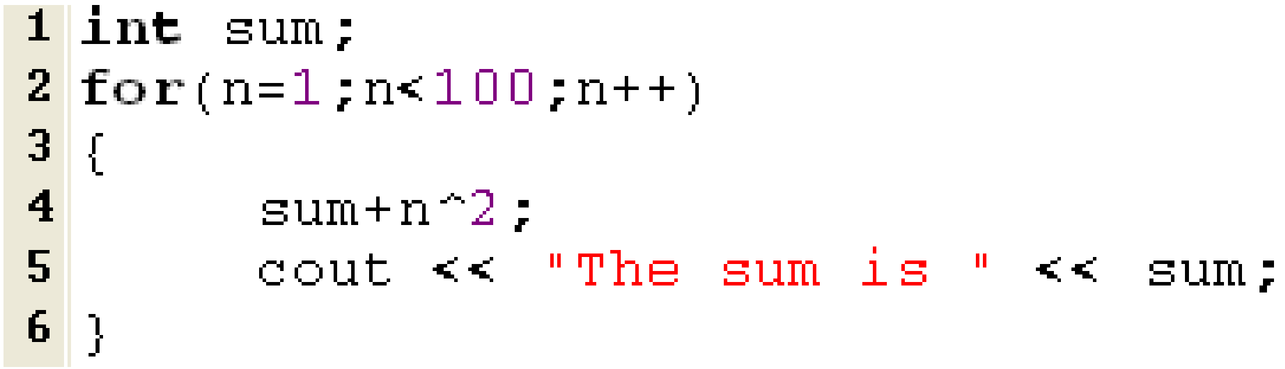
Goal: Compute $\sum_{n=1}^{100} n^2$

What is wrong with the following?



```
int sum;
for(n=1;n<100;n++)
{
        sum+n^2;
        cout << "The sum is " << sum;
}
```

Almost everything…



```
int sum;
for(n=1;n<100;n++)
{
        sum+n^2;
        cout << "The sum is " << sum;
}
```

Errors:

- `sum` is not initialized. It will have an unpredictable value
- `n` is not declared (compiler error)
- The condition should be `n<=100` or `n<101`
- `n^2` is incorrect. “^” is not a power operator. We can use `n*n`, for instance
- `sum+n^2;` has no effect. Correct is `sum=sum+n*n;` or `sum+=n*n;`
- The `cout`-statement should be after the curly braces

While-Loops

while-loop

```
while(condition)
    statement
```

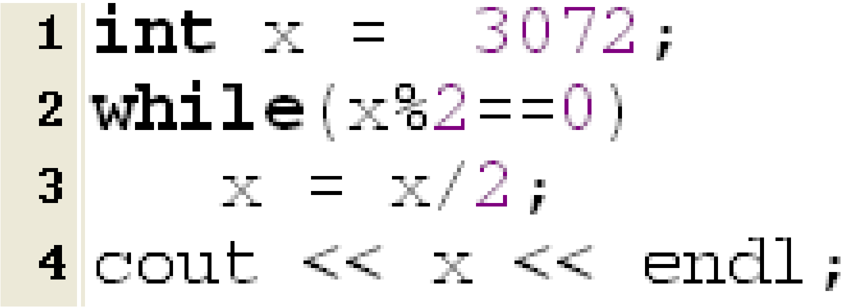
Explanations:

- The statement is executed as long as the condition is true
- Each execution of statement is called an iteration
- Make sure that the condition becomes false after finitely many iterations!

while-loop: Example 1

Divide 3072 successively by 2 until the result is odd. Print the final result to the screen

Solution



```
int x =  3072;
while(x%2==0)
   x = x/2;
cout << x << endl;
```

Problem 5

Set an integer to 1000. Use a while-loop to decrease the integer successively by 13 until it becomes negative. Print the final value of the integer to the screen.

Problem 6

Read integers from the keyboard until the user enters an integer divisible by 5.

Problem 7

- Read integers from the keyboard until the user enters an integer divisible by 5
- Count how many numbers were entered and print this number to the screen

Conditions

if-else-conditions

```
if(condition)
    statement1
else
    statement2
```

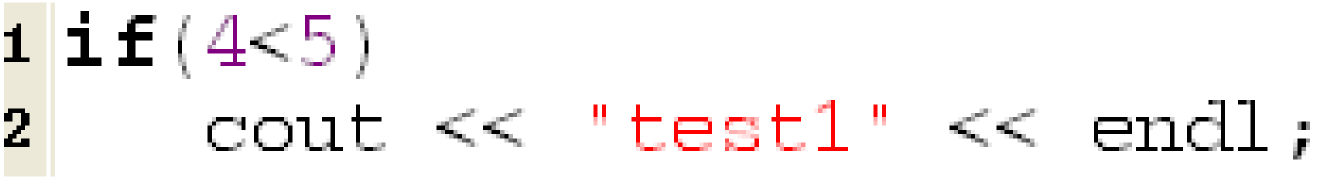
Explanations:

- If the condition is true, then statement1 is executed
- If the condition is false, then statement2 is executed
- The else part is optional
- If there is an else part, it must follow immediately after statement1
- if-else conditions can be nested
- Rule to find out which if belongs to which else: they behave like left and right parentheses

if - Examples

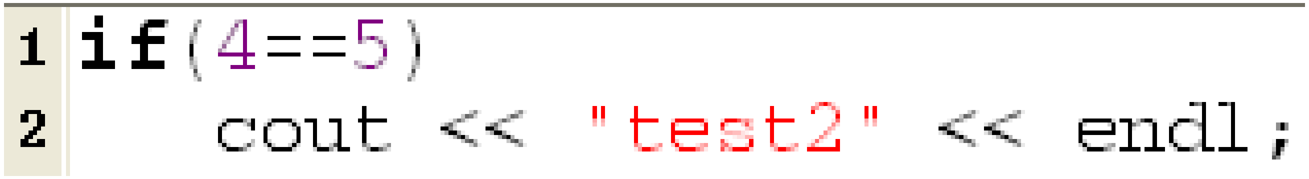
1. Simplest form: if-condition with just one command
2. Show that everything after the command is not controlled by the if-condition
3. How to control several commands by if-condition (use statement block)
4. Show that everything after the statement block is not controlled by the if-condition
5. Nested if-conditions

Example 1



```
if(4<5)
    cout << "test1" << endl;
```

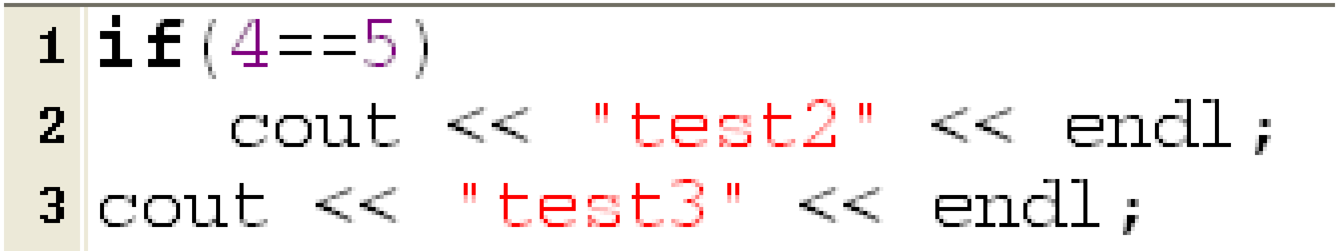
Since the condition “4<5” is true, the cout-command will be executed



```
if(4==5)
    cout << "test2" << endl;
```

Since the condition “4==5” is false, the cout-command will **not** be executed

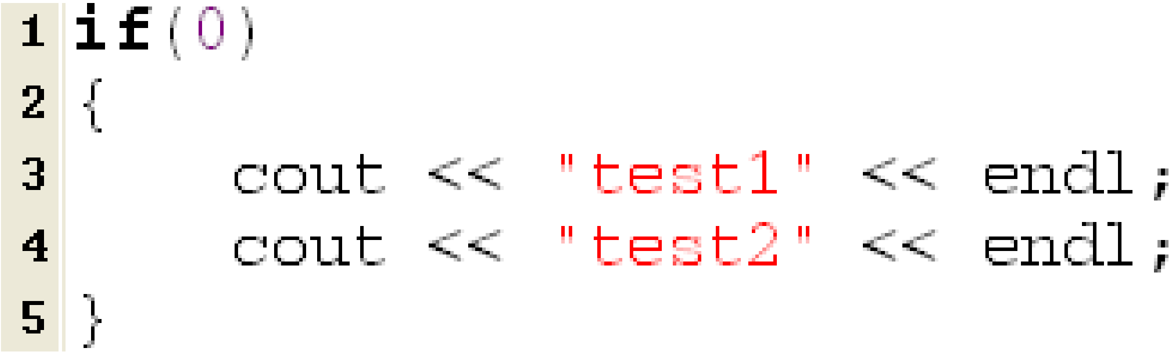
Example 2



```
1 if(4==5)
2     cout << "test2" << endl;
3 cout << "test3" << endl;
```

- Since the condition “4==5” is false, the cout-command in line 2 will **not** be executed
- Line 3 is **not controlled** by the if-condition, so it will be executed anyway

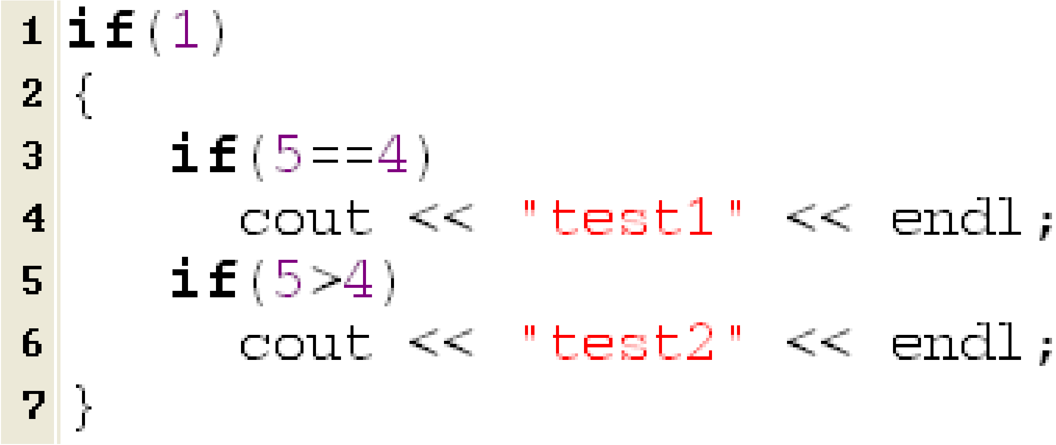
Example 3



```
if(0)
{
	cout << "test1" << endl;
	cout << "test2" << endl;
}
```

- The condition “0” is false (0 false, all other values true)
- Both cout-commands are controlled by the if-condition and both will not be executed

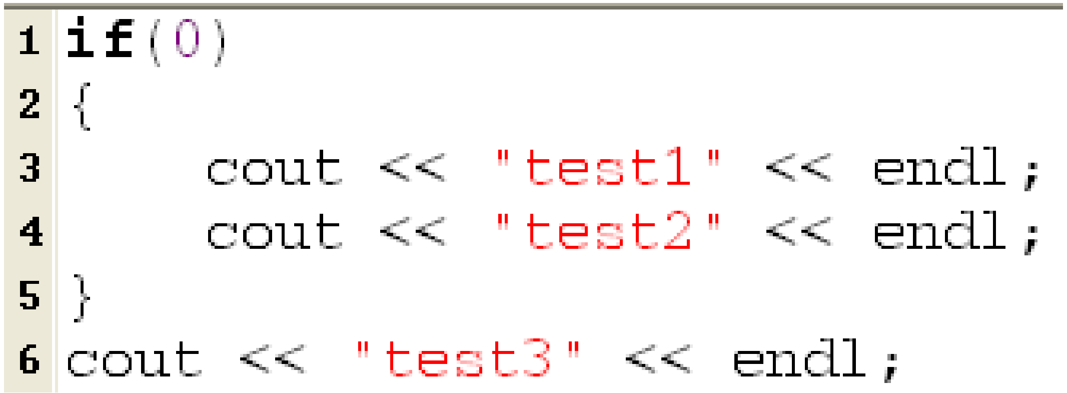
Example 4



```
1 if(1)
2 {
3     if(5==4)
4         cout << "test1" << endl;
5     if(5>4)
6         cout << "test2" << endl;
7 }
```

- Condition “1” is true, so lines 2-7 will be executed
- “5==4” is false => cout-command in line 4 not executed
- “5>4” is true => line 6 executed

Example 5



```
1 if(0)
2 {
3       cout << "test1" << endl;
4       cout << "test2" << endl;
5 }
6 cout << "test3" << endl;
```

Line 6 is not controlled by the if-condition and will be executed anyway

Problem 8

Write a program that generates 1,000,000 random numbers and counts how many of these numbers are in the range 500,501,…,1000 and are divisible by 163

Hint: a (pseudo) random number is returned by **rand()**

Problem 9

Write a program that reads an integer **x** from the keyboard and prints the sum of the digits of **x** to the screen (you can assume that **x** is nonnegative).

Problem 10

Write a program that reads a positive integer **x** from the keyboard and checks if **x** is a prime number.

Problem 11

- Write a program that attempts to read an integer **x** from the keyboard and repeats this until the user really enters an integer.
- Hints: If the user doesn’t enter an integer, the expression **cin** becomes false, so we can use a condition **if(!cin)**
- To restore cin so that it can read input again, use **cin.clear(); cin.ignore(10,’\n’);** (the ignore part makes sure the previous incorrect input is ignored)