## Summary

- Operators
- Literals and Expressions
- Control Flows

# C++ Operators

**See:**

**http://www.cplusplus.com/doc/tutorial/operators/**

## Operators

- An operator is a symbol that performs an action

  Examples of operators

```
int x = 10;    // assignment
int x = 2 + 3; // addition
double x = 3.2 / 2.3;  // division

cout << 100;   // stream out
cin >> x;      // stream in

x = 3 << 2     // shift left
```

# 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 = 10;
    int y = 3;
    cout << x * y << endl;
    cout << x + y << endl;
    cout << x / y << endl;
    cout << x - y << endl;
    cout << x % y << endl;
    return 0;
}
```

What happens if we change the type of x to double?

## 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

## Operator Overload

- There are many version of each operator
- For instance the operator plus (+) is defined for all possible numeric types

```
double + double => double
int + int => int
float + float => float
...
```

## Problem

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.

# Comparison 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. |

```
int main()
{
    int x = 10;
    int y = 3;
    cout << (x > y) << endl;
    cout << (x < y) << endl;
    cout << (x >= y) << endl;
    cout << (x <= y) << endl;
    cout << (x != y) << endl;
    cout << (x == y) << endl;
    return 0;
}
```

Note the use of parenthesis, to clarify that **>** must be performed before **<<**

## && (logical and), || (logical or), ! (logical not)

- 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)
- !A is true if A if false, it is false if A is true
- && has priority over ||, i.e. , && is evaluated first
- Recommendation: always put parentheses around boolean expressions (avoids mistakes and is easier to read)

## &&, ||   Examples

```
int main()
{
    bool t = 10>3;
    bool f = 10<3;
    cout << (t && f) << endl;
    cout << (t || f) << endl;
    cout << !t << endl;
    cout << !f << endl;
    cout << (t && !f) << endl;
    return 0;
}
```

## 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

## Short Circuit Boolean Evaluation (built in types only)

- In (2 > 3) && (4 < 5) the sub-expression (4<5) is not evaluated. This is because after we evaluated (2>3) we already know that the && expression is false.
- This is important when the sub-expression are not pure

```
 x && foo()
```

- Here the function foo, which in addition to returning a value could have side effects, will no be executed if x is false

## Problem

- 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) $1000 < x \leq 10000$
  (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 (e.g. x=2009)

## Bit Operators

- &, |, ^, <<, >>, ~

- Interpret a n-bit integer variable as a mere sequence of bits.
- Examples:

```
3 & 9 = 0011 & 1001 = 0001 = 1
3 | 9 = 0011 | 1001 = 1011 = 11
3 << 1 = 0011 << 1 = 0110 = 6
3 >> 1 = 0011 >> 1 = 0001 = 1
```

## Precedence

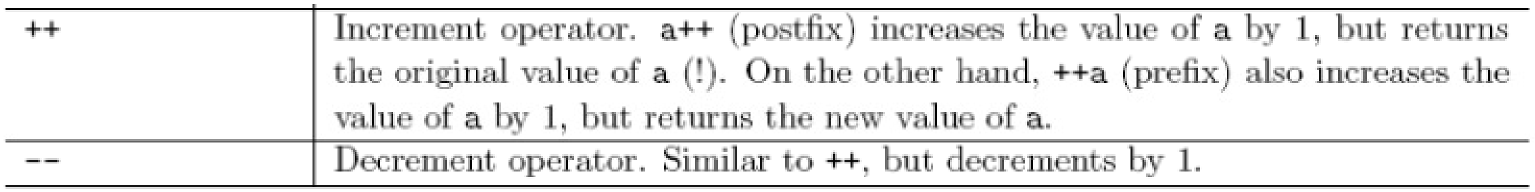
- Some operators have higher precedence than others
- In 2+3*5 the multiplication is carried out before the addition. This is because the multiplication has higher precedence than the addition
- All operators (boolean, comparison, arithmetic, bitwise, ...) are ranked in terms of precedence: http://en.cppreference.com/w/cpp/language/operator_precedence

## Unary vs Binary Plus / Minus

```
int x = 2 - 3;    // binary minus operator
int x = -3;       // unary minus operator
```

An operator **arity** is defined by the number of arguments the operator requires

## Increment / Decrement



| | |
|---|---|
| ++ | Increment operator. a++ (postfix) increases the value of a by 1, but returns the original value of a (!). On the other hand, ++a (prefix) also increases the value of a by 1, but returns the new value of a. |
| -- | Decrement operator. Similar to ++, but decrements by 1. |

```
int main()
{
    int x = 1;
    cout << x << endl;
    cout << x++ << endl;
    cout << x << endl;
    cout << ++x << endl;
    return 0;
}
```

## Problem

❑ Declare and initialize a variable x of type integer.

❑ Use a cout command to check what the return values of (x++) and (++x) are.

❑ Check what happens to the value of x after the command ++x; respectively x++; has been executed.

## Assignment, Compund Assignment, Conditional Assignment

Example:

```
int x = 2;    // assignment operator

x += 3;   // addition and assignment operator combined

x *= 3;   // multiplication and assignment operator combined
```

See: **http://www.cplusplus.com/doc/tutorial/operators/**

## Conditional Assignment

Condition **?** Expression1 **:** Expression2

- A ternary operator
- If the condition is true the expression evaluates to Expression1, otherwise to Expression2

Examples:

```
int x = (1>2) ? 1 : 3;  // x takes the value 3
int y = (3>2) ? 1 : 3;  // x takes the value 1
```

# 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

- Every literal has a type.
- We can find out the type using the typeid-operator (see http://en.cppreference.com/w/cpp/language/typeid)

Example:

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

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

## Scientific Format (e-Format) for `float` and `double` values

$$a\mathtt{e}b \text{ or } a\mathtt{E}b \text{ means } a \cdot 10^b$$

Examples

| e-format | meaning |
|---|---|
| 1e6 | 1000000 |
| 1.33E-1 | 0.133 |
| 0.314e1 | 3.14 |
| 4e-5 | 0.00004 |

Note these are used for input / output or literals, but they do not correspond to the internal representation, which is IEEE754

## Example

```
int main()
{
    double x = 1.2e2;  // same as 1.2 * 10^2
    cout << x << “\n”;
}
```

# 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 **return value** of the expression
- The return value of an expression can be outputted by
  `cout << expresssion;`
- The type of the return value is called the **type** of the expression.
- Rule of thumb: the type of an expression is the “most complex type” occurring in it (remember sub-typing and overloading).
- An expression is not a complete C++ command. It can only be part of  a C++ command.

## 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)

## Ascii codes

- The size of **char** type (or **unsigned char**) is 1 byte, i.e. 8 bits
- 8 bits can be mapped to the range 0-255
- Special characters are associated to each of these values via this table: http://www.asciitable.com/
- Such special characters have effects when sent to the console:
  - 65 means ‘A’, if sent to the console it will display the character ‘A’

## Question

What does the below code print?

```
char c1 = 72, c2 = 'e', c3 =, c4 = 111;
cout << c1 << c2 << c3 << c3 << c4 << "\n";
```

## 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`

## Literals modifiers

http://www.cplusplus.com/doc/tutorial/constants/

# Escape Characters

- To print a “new line” we have sometime used **std::endl** and sometimes “**\n**”
- “**\n**” is a special escape character
- Escape characters are prefixed by \
- See: [https://en.wikipedia.org/wiki/Escape_sequences_in_C](https://en.wikipedia.org/wiki/Escape_sequences_in_C)
- They can be used to print to the console actions (e.g. new line, insert a tab) or to include in string literals special characters of or unicode characters not available on the keyboard
- E.g., if I want to include **"** in a string literal, there is a problem: **"** is also the delimiter for string literals.

  ```
  std::cout << "My name is \"Fabio\"" << std::endl;
  ```

  will print:

  My name is “Fabio”

# Some Rules for Return Values

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

## Flow Control Structures

## Flow Control Structures

- The instructions in a program are executed sequentially. This is not usually very useful by itself, because it does not allow:
  - execution to adapt to inputs
  - to avoid code repetition
- There are however special instructions which allow to change the flow of a program
- At low level they are all implemented via the primitive
  - *if some condition is true then, instead of going to the next instruction, jump to some other instruction*

## Flow Control structures

- To make the program execution dependent on conditions, we use conditional structures:

  `if-else, switch`
- To perform a task repeatedly in C++, we use loops:

  `for, while, do-while`
- To jump to another point of the program, we use jump statements:

  `break, continue, goto (`*`goto`*` usually not necessary and not for beginners)`

## Conditions

## Conditional Operations

- Ask a question and then select the next operation to be executed on the basis of the answer to that question
- In most languages this is represented by the instruction if-then-else
- In C++ ‘then’ is omitted

## if – else

- This is the simplest possible structure

```
if (condition)
   statementA;
else
  statementB;
```

- Example

```
if (patientTemperature>37)
   cout << “I recommend paracetamol”;
else
  cout << “You are fine, go home”;
```

## `if-else-`conditions

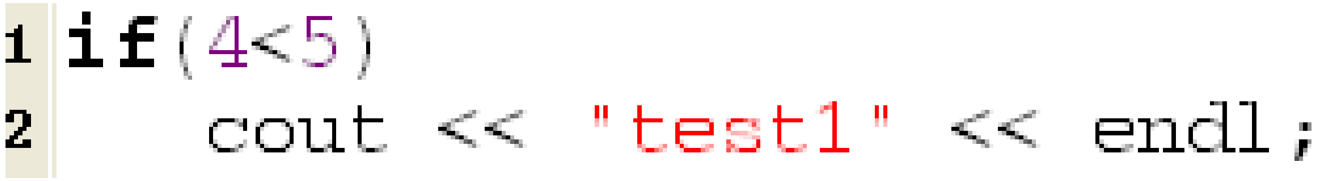
```
if(condition)
     statement1
else
     statement2
```

- Remember a statement can be either a single command or a statement block
- 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

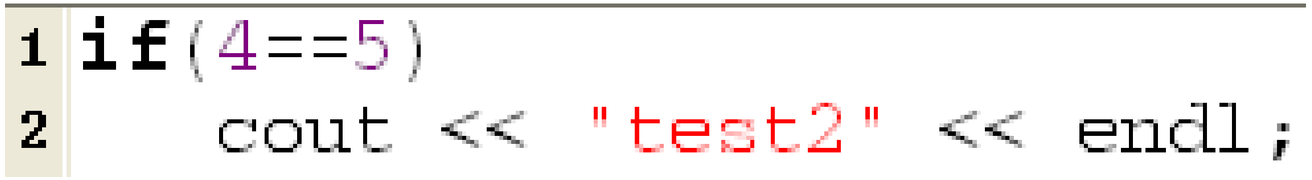
1. Simplest form: if-condition with just one command
2. Show that everything after the command is not controlled by the if-condition
3. How do control several commands by if-condition (use statement block)
4. Show that everything after the statement block is not controlled by the if-condition
5. Nested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

# Example 1



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

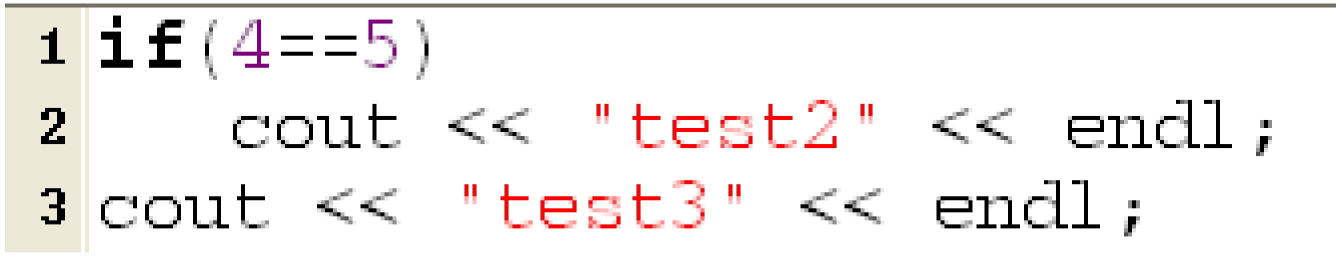
- Since the condition “4<5” is true, the cout-command will be executed



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

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

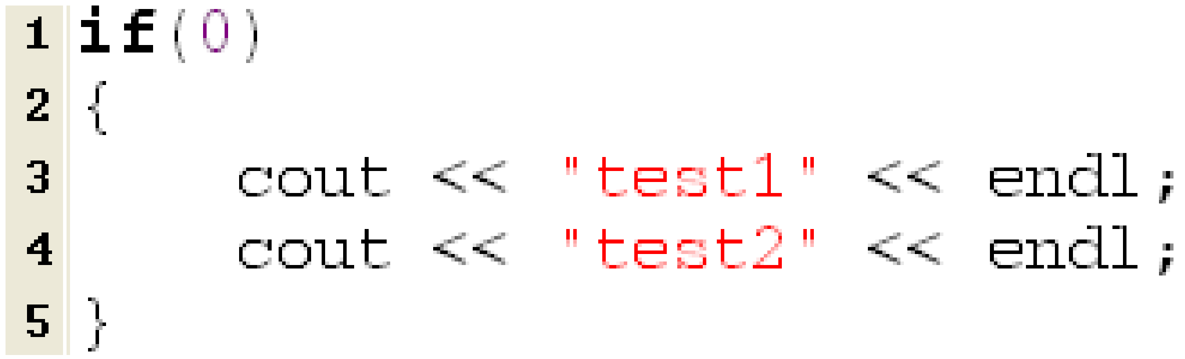
## Example 2



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

- Since the condition “4==5” is false, the 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

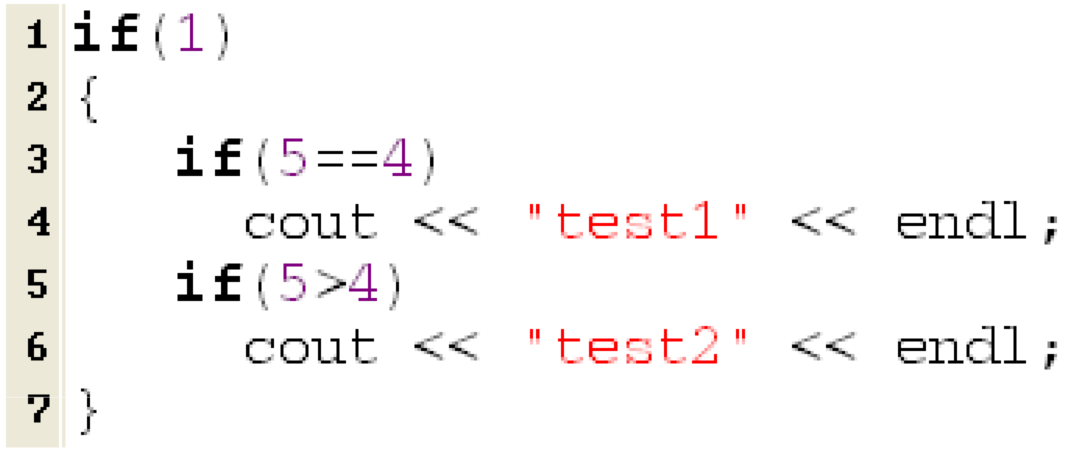
# Example 3



```
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

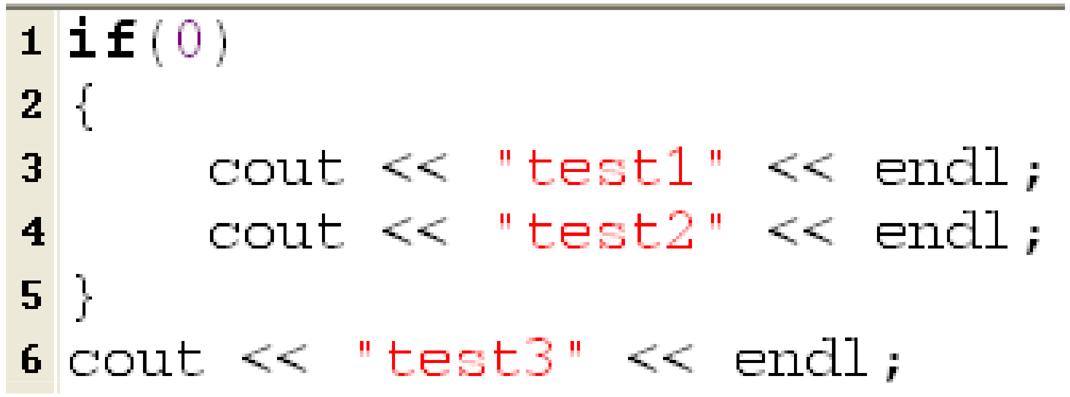
# Example 4



```
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

## Indentation

- In the previous slide we used indentation
- In most languages indentation is just optional
- It is good practice to make extensive use of it, as it makes source code easier to read

## Loops

## Loops

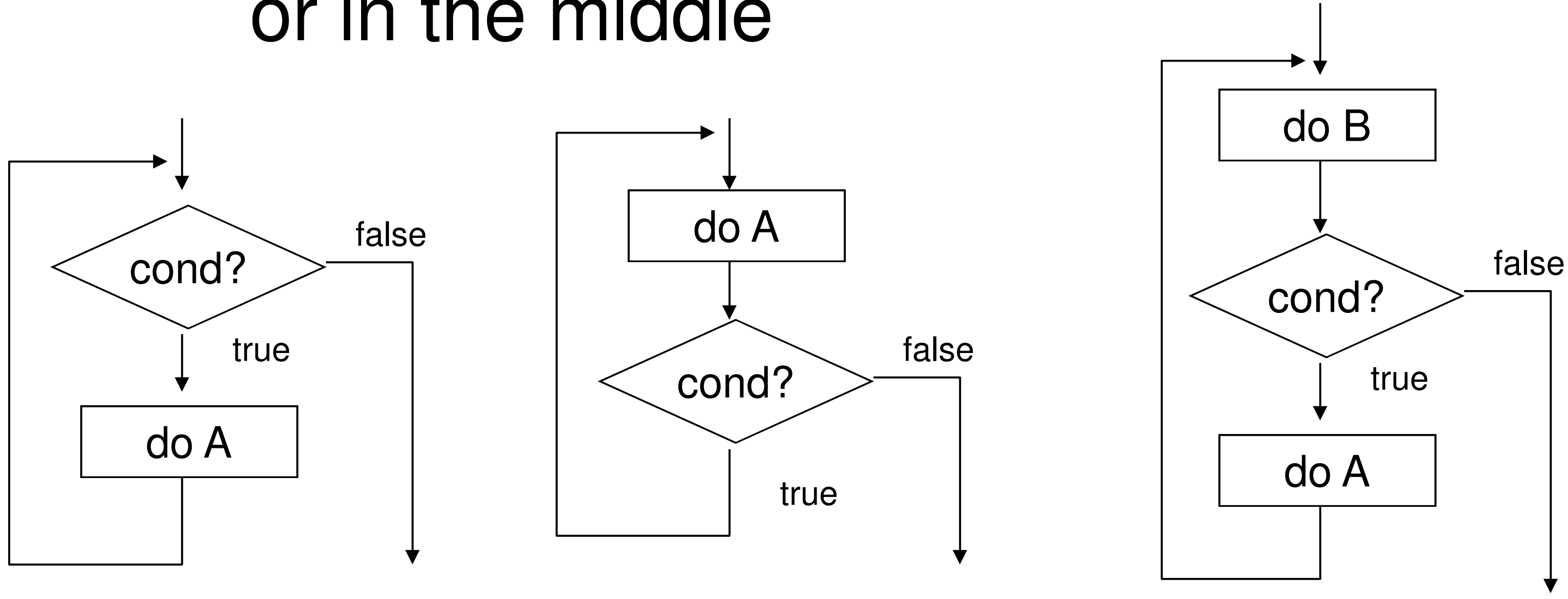
- Tell us to go back and repeat the execution of a previous block of instructions

```
check tyre pressure
if tyre pressure is too low
    connect the pump
    repeat until tyre pressure is ok
        pump 10 times
        check tyre pressure
    disconnect the pump
ride your bycicle
```

note I used indentation to define blocks of code

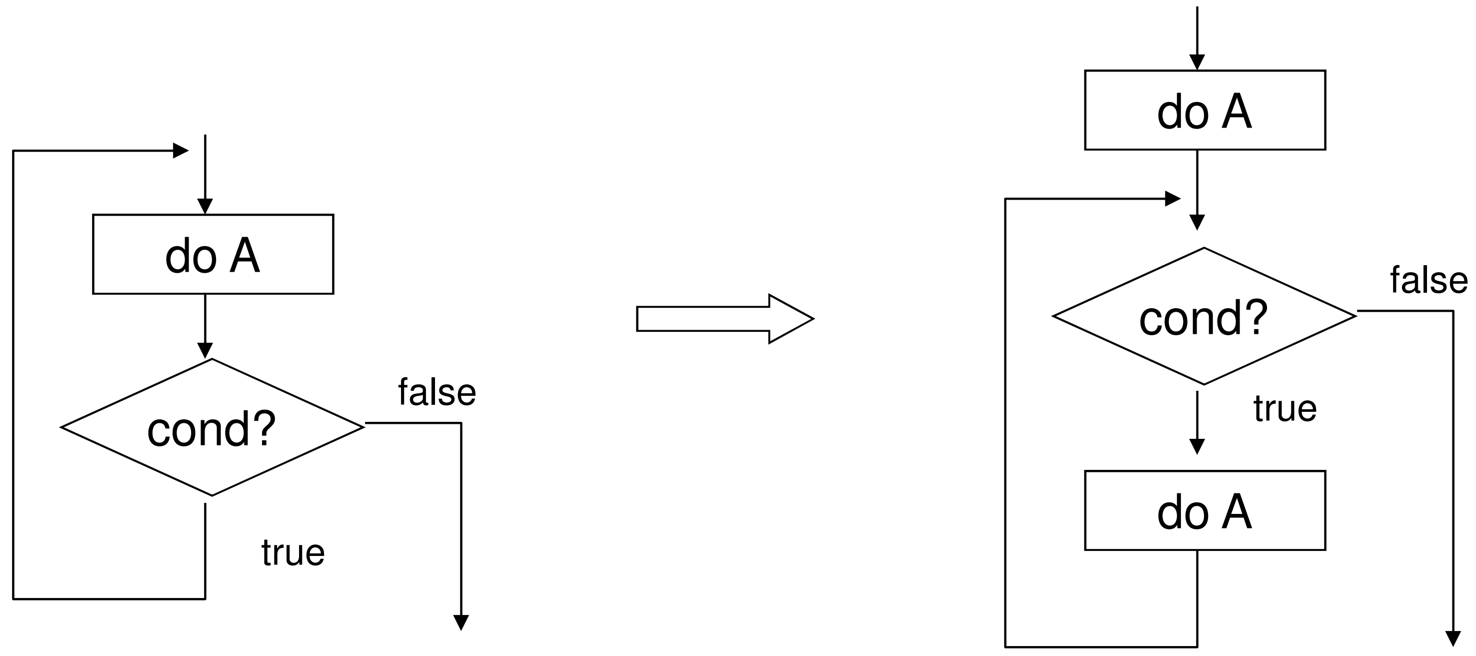
# Loops

- Three types, depending if the condition is checked at the beginning, at the end or in the middle



## Loop Equivalence

- All type of loops can be transformed into another type at the cost of code repetition



## 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

## For-Loops

## `for`-loop

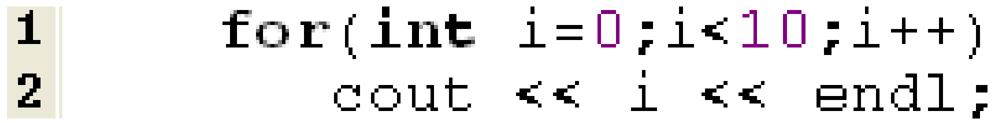
```
for(initialization; condition; increment)
    statement
```

- The ***statement*** is executed as long as the ***condition*** is true.
- ***initialization*** and ***increment*** 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
- In the initialization we can declare variables local to the loop

## for loop – Order of Operations

1. initialization
2. if condition is false, exit from the loop
3. execute statement
4. increment
5. goto step 2

# `for`-loop example

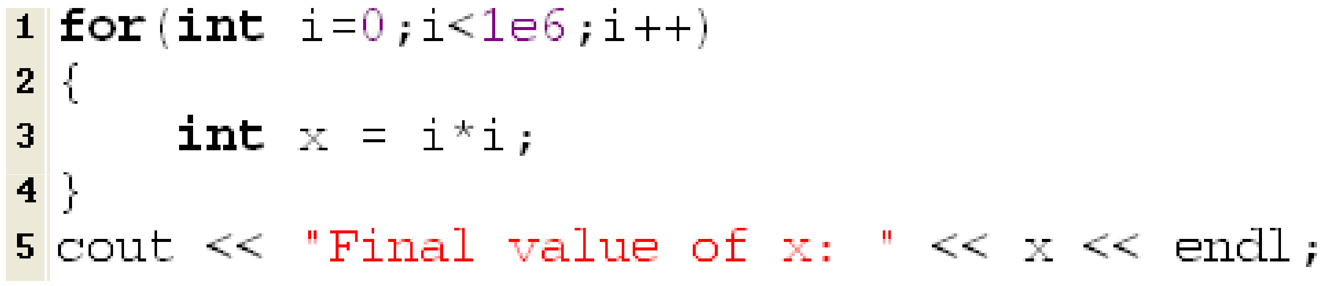


```
for(int i=0;i<10;i++)
    cout << i << endl;
```

- The initialization here is `int i=0`
- `i` is called a **counter variable**
- In this example `i` is declared in the initialization, i.e. it is **local to the loop**
- The condition is `i<10`
- The increment 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;
```

- ❑ 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`

## for-loop - Problem

$$\bullet\ \text{Implement:}$$

$$\sum_{n=0}^{100} \frac{1}{3^n} = 1 + \frac{1}{3} + \frac{1}{9} + \frac{1}{27} \cdots \ \frac{1}{3^{100}}$$

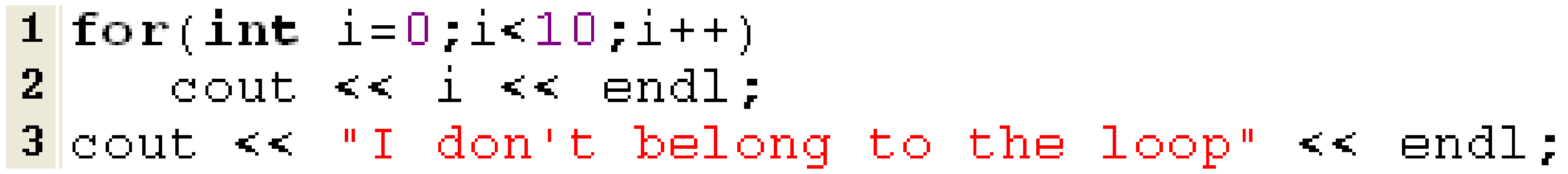
$$\text{hint: rewrite as:}$$

$$= 1 + \sum_{n=1}^{100} \underbrace{\frac{1}{3^n}}_{A_n} = 1 + \sum_{n=1}^{100} A_{n-1} \cdot \frac{1}{3}$$

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

# `for`-loop: Example 1



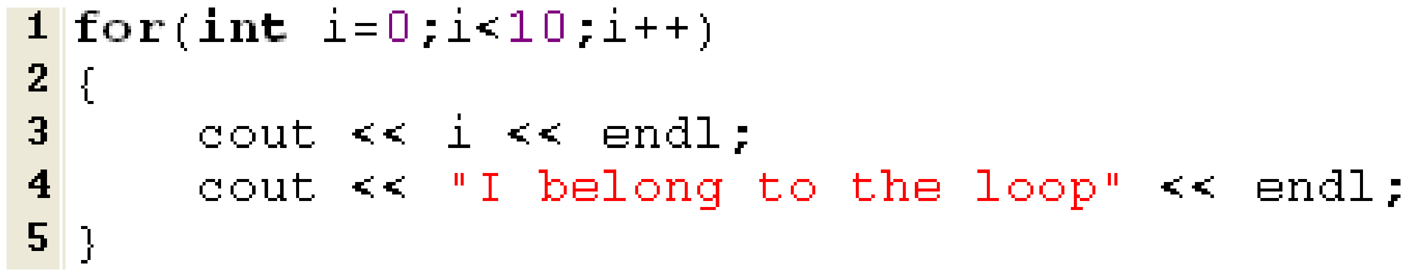
```
1 for(int i=0;i<10;i++)
2     cout << i << endl;
3 cout << "I don't belong to the loop" << endl;
```

**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



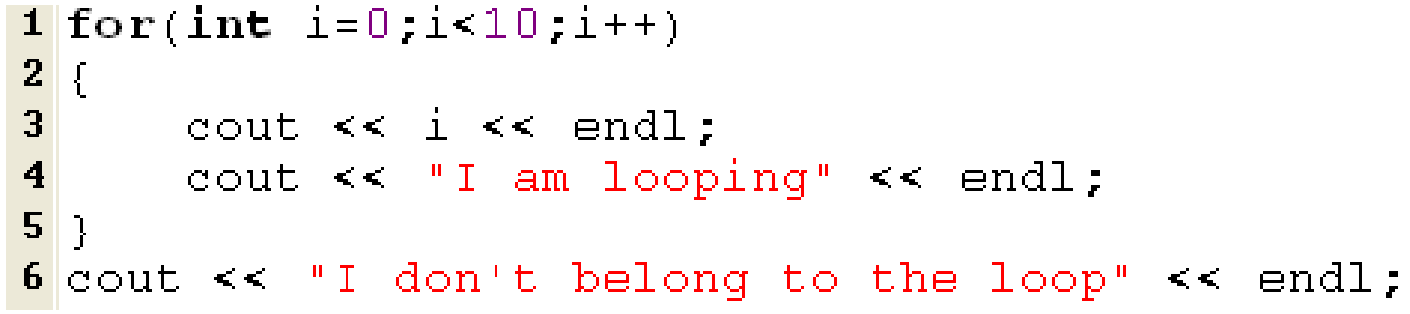
```
1 for(int i=0;i<10;i++)
2 {
3       cout << i << endl;
4       cout << "I belong to the loop" << endl;
5 }
```

**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



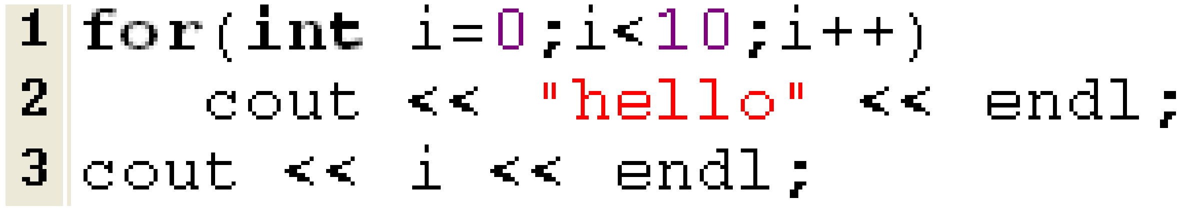
```
1 for(int i=0;i<10;i++)
2 {
3       cout << i << endl;
4       cout << "I am looping" << endl;
5 }
6 cout << "I don't belong to the loop" << endl;
```

**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

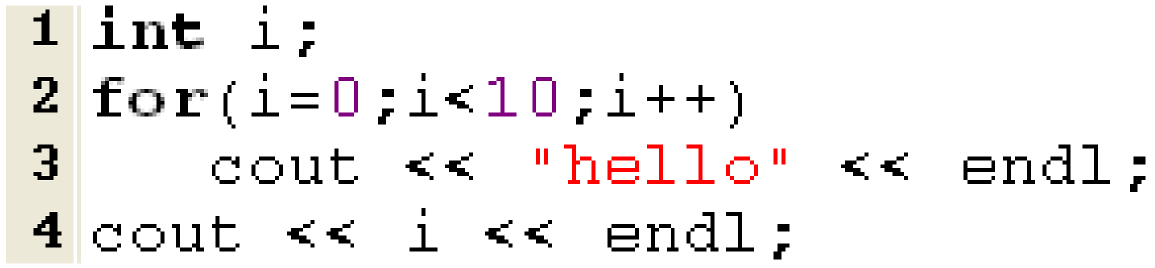


```
1 for(int i=0;i<10;i++)
2     cout << "hello" << endl;
3 cout << i << endl;
```

**Explanations:**

- Compiler error!
- If declared inside the loop, a variable (here `i`) is not valid outside the loop
- So 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

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

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

Results:

a) 338350

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

# Computing the sum of the first N integer numbers

```
int main()
{
    int n;
    cout << "enter N: ";
    cin >> n;

    int sum = 0;
    for (int i = 1; i <= n; ++i)
        sum += i;

    cout << sum << endl;

    return 0;
}
```

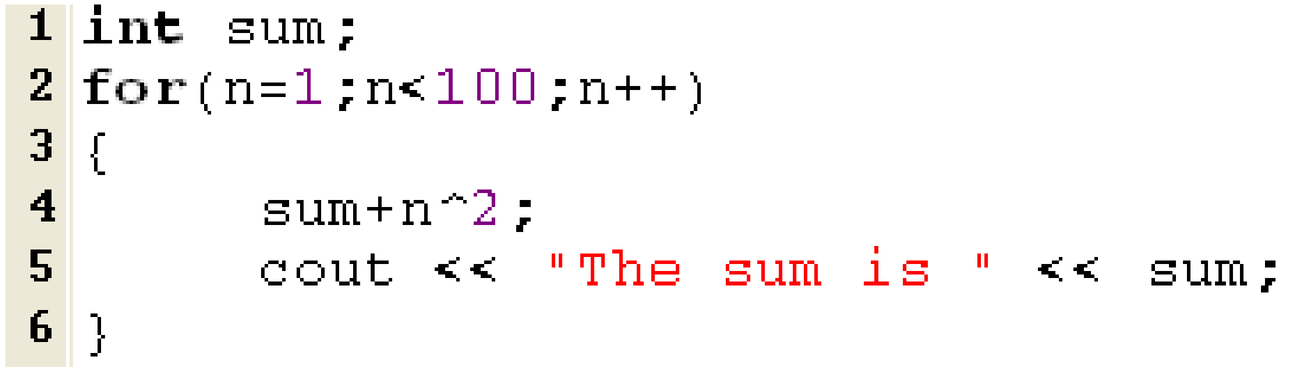
Let’s run the program step by step

```
n=3                // suppose the user enters 3
sum=0              // init sum
i=1                // loop initialization
(i < n) → true     // check loop condition
sum += i           // loop body: sum=0+1=1
i++                // i=2, loop increment
(i < n) → true     // check loop condition
sum += i           // loop body: sum=1+2=3
i++                // i=3, loop increment
(i < n) → true     // check loop condition
sum += i           // loop body: sum=3+3=6
i++                // i=4, loop increment
(i < n) → false    // check loop condition
print sum
```

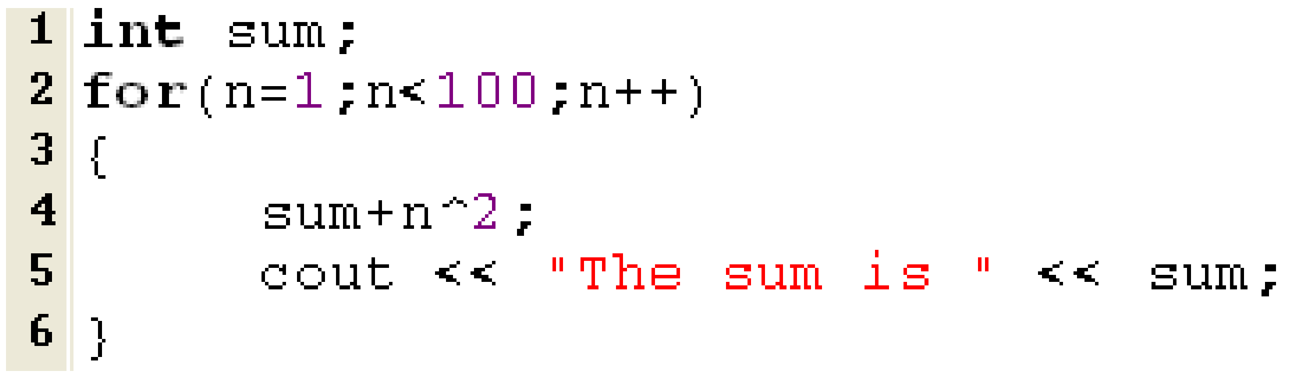
# Computing a sum… with bugs!

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;
}
```



```
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
```

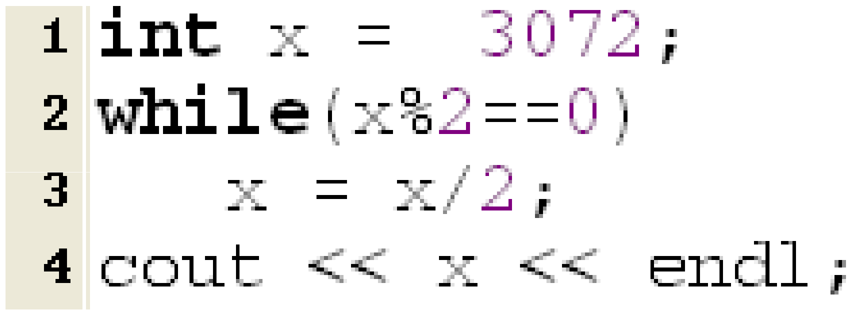
**Explanations:**

- 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

- 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

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

## Problem

- 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.

## Do-While-Loops

## `while`-loop

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

**Explanations:**

- The ***statement*** is executed as long as the ***condition*** is true. Note it is **<u>executed at least once</u>**.
- Each execution of ***statement*** is called an iteration
- Make sure that the ***condition*** becomes false after finitely many iterations!

## Problem

- Write a program that generates 1000,000 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

- 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

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

## Interrupting Loops with `continue` and `break`

Fabio Cannizzo - NUS

## `continue`

```
continue;
```

**Explanations:**

- ❑ `continue` interrupts the current iteration of a `while`- or `for`-loop
- ❑ The program immediately proceeds to the next iteration
- ❑ Use `continue` to discard some iterations (not most!) of a loop

## `continue`: Example 1

Print the numbers in the range 1,2,…,100 to the screen which are not divisible by 13.

## Solution

```
for (int i=1; i <= 100; ++i)
{
    if (i % 13 == 0)
         continue;
    cout << i << endl;
}
```

## Problem 2 (use of continue)

a) Print the pairs (a,b) with a=1,…,10, b=1,…,10, to the screen for which a≠b.

b) Write a program that prints the numbers from 1 to 100 on the screen except the numbers which are multiples of 7 or 11.

## continue

- In general it is not efficient to conditions inside loop
- If possible, reorganize the loop

```
for (int i=1; i <= 100; ++i)
{
    if (i % 13 == 0)
         continue;
    cout << i << endl;
}
```

```
int begin = 1;
do {
   int end = begin+11;
   end = end <= 100 ? end : 100;
   for (int i=begin; i <= end; ++i)
         cout << i << endl;
   begin = begin + 13;
} while (begin < 100);
```

## `break`

```
break;
```

- `break` terminates a complete for- or while-loop immediately
- If a loop has fulfilled its purpose, but is still running, then `break` it.

## `break`: example

Test if 1001 has any nontrivial divisor. When a divisor is found, print it to the screen, and stop the search at once.

## Solution

```
for (int d=2; d <= 1001; d+=2) {
    if (1001 % d == 0) {
         cout << "divisor found: " << d << endl;
         break;
    }
}
```

## Problem (use of break)

- Write a program that reads 5 double values from the keyboard.
- The program should interrupt the input immediately if the absolute value of the product becomes larger than 1e10. In this case, an error message should be printed to the screen.
- If 5 values are entered successfully,  the product of the numbers entered should be printed on the screen.
- Hint: Absolute value of a double x : std::abs(x) (needs **#include<cmath>**)

# The Switch Instruction

- It is equivalent to a chain of if, where a variable is compared to literals of integral type (including **enum**)
- Often the compiler resolve this more efficiently than an if

```
if (x == 1)
        z = 3;
else if (x == 5)
        z = 9;
else if (x == 7 || x == 8)
        z = 9;
else
        z = 0;
```

```
switch(x)
{
   case 1:
         z = 3;
         break;
   case 5:
         z = 9;
         break;
   case 7:
   case 8:
         z = 9;
         break;
   default:
         z = 0;
         break;
};
```

## Problem

- Prompt the user to enter a number in 0-6 and print to the screen the corresponding day of the week using the *switch* instruction

## Good Practice

- Use indentation correctly
- Prefer structured loop over unstructured loop when possible

# Additional Practice Problems

## Problem

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 the day is a Saturday, w=1 a Sunday etc. etc.

# Problem

Ask a person the age and assign to the variable x:

- if the age is less than 20, say “I thought you were older”
- if the age is more than 40, say: “I thought you were younger”
- Otherwise say “My guess was about right”
- Hint: Use a sequence of if-else statements

## Problem

Ask the user to enter a positive integer *n*:

- print the sum of the first numbers smaller or equal than n which are multiples of 3
- Hint: instead of putting an **if** statement inside the loop, which would be inefficient, simply initialize the loop counter to 3 and change the loop increment to `i+=3` instead of `i++`

## Problem

Ask the user to enter a positive integer *n* and print the first n Fibonacci numbers (https://en.wikipedia.org/wiki/Fibonacci_number)