Maciej Sobieraj

Lecture 1

Outline

1. **Introduction to computer programming**
2. **Advanced flow control and data aggregates**

Your first program

- First we need to define our expectations for the program. They'll be very modest.
- We want a short text to appear on the screen. Let's assume that the text will state:

  „It's me, your first program."
- What further steps does our first program need to perform? Let's try to enumerate them here:
  - to start
  - to write the text on the screen
  - to stop
- This type of structured and semi-formal description of each step of the program is called an algorithm.

Your first program

- # (hash) at the beginning of the first line means that the content of this line is the so-called **preprocessor directive**

```
#include <iostream>
using namespace std;
int main(void) {
        cout << "It's me, your first program.";
        return 0;
}
```

- it's a separate part of the compiler whose task is to pre-read the text of the program and make some modifications in it.

Your first program

- The changes the preprocessor will introduce are fully controlled by its directives.
- We are dealing with the include directive.
- When the preprocessor meets that directive, it replaces the directive with the content of the file whose name is listed in the directive

```
#include <iostream>
using namespace std;
int main(void) {
        cout << "It's me, your first program.";
        return 0;
}
```

Your first program

- Writing a program is similar to building a construction with ready-made blocks.
- In our program, we are going to use one such block and it will happen when we want to write something on the screen.
- That block is called **cout**, but the compiler knows nothing about it so far.

Your first program

- A set of preliminary information that the compiler needs is included in **header files**.
- These files contain a collection of preliminary information about ready-made blocks which can be used by a program to write text on the screen, or to read letters from the keyboard.
- So when our program is going to write something, it will use a block called **cout**.

Your first program

- In the C++ language, all elements of the standard C++ library are declared inside the **namespace** called **std**.
- A namespace is an abstract container or environment created to hold a logical grouping of unique entities (blocks).

```
#include <iostream>
using namespace std;
int main(void) {
        cout << "It's me, your first program.";
        return 0;
}
```

Your first program

- The standard of the C++ language assumes that among many different blocks that may be put into a program, one specific block must always be present, otherwise the program won't be correct.
- This block is always a function with the same name: **main**.

```
#include <iostream>
using namespace std;
int main(void) {
        cout << "It's me, your first program.";
        return 0;
}
```

Your first program

- Inside the main function body we find a reference to a block named **cout**.
- Each instruction in C++ must end with a **semicolon**.
- This particular statement says: instruct the entity named **cout** to show the following text on the screen (as indicated by the << digraph, which specifies the direction in which the text is sent).

```
#include <iostream>
using namespace std;
int main(void) {
        cout << "It's me, your first program.";
        return 0;
}
```

Your first program

- Return is used in the function, it causes the end of function execution.
- If you perform return somewhere inside a function, this function immediately interrupts its execution.

```
#include <iostream>
using namespace std;
int main(void) {
        cout << "It's me, your first program.";
        return 0;
}
```

Numbers

- The numbers handled by modern computers are of two types:
  - integers, that is, whole numbers or those which are devoid of the fractional part,
  - floating-point numbers (or simply floats) that contain (or are able to contain) the fractional part.
- The characteristic of a number which determines its kind, range and application is called **type**.

A variable is variable

- To store numbers or results of arithmetic operation we use special “containers” called **variables**.
- Each variable have:
  - a name
  - a type
  - a value

A variable is variable

- If you want to give a name to the variable, you have to follow some strict rules:
  - the name of the variable can be composed of upper-case or lower-case Latin letters, digits and the character _ (underscore),
  - the name of the variable must begin with a letter,
  - the underline character is a letter (strange but true),
  - upper- and lower-case letters are treated as different (a little differently than in the real world - Alice and ALICE are the same given names, but they are two different variable names, and consequently, two different variables).

A variable is variable

- The **type** is an **attribute** that uniquely defines which values can be stored inside the variable.
- We've already encountered the integer (int) and floating point (float) types.
- The **value** of a variable is what we have put into it. We can only enter a value that is compatible with the variable's type.
- The variable exists as a result of a declaration.
- A declaration is a syntactic structure that binds a **name** provided by the programmer with a specific **type** offered by the C++ language.

A variable is variable

- **Declaration** of variable of type int named Counter.

```
int Counter;
```

- **Assignment operator** has a simple syntax and unambiguous interpretation.

```
Counter = 1;
```

- Another example:

```
Result = 100 + 200;
```

A variable is variable

- In the C++ language the sign = does not mean **is equal to**, but **assign a value**.
- The value of variable x is **incremented** by one, which has nothing to do with comparing the variable with any value.

```
x = x + 1;
```

A variable is variable

- List of words that play a very special role in every C++ language program.

and	do	new	switch
and_eq	double	not	template
asm	dynamic_cast	not_eq	this
auto	else	operator	throw
bitand	enum	or	true
bitor	explicit	or_eq	try
bool	export	private	typedef
break	extern	protected	typeid
case	false	public	typename
catch	float	register	union
char	for	reinterpret_cast	unsigned
class	friend	return	using
compl	goto	short	virtual
const	if	signed	void
const_cast	inline	sizeof	volatile
continue	int	static	wchar_t
default	long mutable	static_cast	while
delete	namespace	struct	xor
			xor_eq

Comments on the comments

- A line comment discards everything from where the pair of slash signs (//) is found up to the end of that same line.

```
int Counter; // counts the number of sheep in the meadow
```

- In the C++ language a block comment is a text that begins with a pair of the following characters:

```
/* Counter variable counts the number of sheep in the meadow */
int Counter;
```

Floating-point numbers

- **Note:** the decimal point is essential to recognize floating-point numbers in C++. Look at these two numbers:
  - 4
  - 4.0
- You might think that they're exactly the same, but the C++ compiler sees them completely differently:
  - 4 is an int.
  - 4.0 is a float.

Floating-point numbers

- Very important difference between these two data types

```
int i;
float x;

i = 10 / 4;
x = 10.0 / 4.0;
```

- After changing from *int* to *float*, the value of the variable *f* is 100.0, because the value of type *int* (100) is automatically converted into a *float* (100.0).

```
int i;
float f;

i = 100;
f = i;
```

Operators

- An **assignment operator** is the = sign
- An asterisk * is a **multiplication operator**.

```
int i,j,k;
float x,y,z;

i = 10;
j = 12;
k = i * j;
x = 1.25;
y = 0.5;
z = x * y;
```

Operators

- A slash / is a **divisional operator**.

```
int i,j,k;
float x,y,z;

i = 10; j = 5;
k = i / j;
x = 1.0; y = 2.0;
z = x / y;
```

- Division by zero

```
float x;

x = 1.0 / 0.0;
```

```
float x,y;

x = 0.0;
y = 1.0 / x;
```

Operators

- The **addition operator** is the + (plus) sign

```
int i,j,k;
float x,y,z;

i = 100; j = 2;
k = i + j;
x = 1.0; y = 0.02;
z = x + y;
```

- The **subtraction operator** is obviously the – (minus) sign

```
int i,j,k;
float x,y,z;

i = 100; j = 200;
k = i - j;
x = 1.0; y = 1.0;
z = x - y;
```

Operators

- The **remainder operator** is quite peculiar, because it has no equivalent among traditional arithmetic operators.
- Its graphical representation in the C++ language is the % (percent) character. It's a binary operator (it performs the modulo operation)

```
int i,j,k;

i = 13;
j = 5;
k = i % j;
```

Operators

- Operators in order from the highest to the lowest priority.

+ -	unary
* / %	
+ -	binary

Operators

- Both operators (* and %) have the same priority.

2 * 3 % 5

- Subexpressions in parentheses are always calculated first

```
int i,j,k,l;
i = 100;
j = 25;
k = 13;
l = (5 * ((j % k) + i) / (2 * k)) / 2;
```

Operators

- Operator used to increment a variable by one
  - **int** SheepCounter;
  - SheepCounter = 0;
  - SheepCounter++;
- Decrease the value by one
  - DaysUntilHoliday--;

Operators

- Operation: ++Variable --Variable
  - Effect: Increment/decrement the variable by 1 and use its value already increased/reduced.
- Operation: Variable++ Variable--
  - Effect: Use the original (unchanged) variable's value and then increment/decrement the variable by 1.

Variable++	post-increment operator
++Variable	pre-increment operator
Variable--	post-decrement operator
--Variable	pre-decrement operator

Operators

- Result?

```
int i,j;

i = 4;
j = 2 * i++;
i = 2 * --j;
```

Operators

- Shortcut operators

```
i *= 2;
SheepCounter += 10;
```

```
SheepCounter = SheepCounter + 10;
i = i * 2;
```

Character type

- char, which is an abbreviation of the word “character”.

```
char Character;
```

- Computers store characters as numbers.

Character	Dec	Hex		Character	Dec	Hex
@	64	40		`	96	60
A	65	41		a	97	61
B	66	42		b	98	62
C	67	43		c	99	63
D	68	44		d	100	64

Character type

- Enclosed in single quotes (apostrophes)

```
Character = 'A';
```

- Assign a non-negative integer value that is the code of the desired character

```
Character = 65;
```

Character type

- The C++ language uses a special convention that also extends to other characters, not only to apostrophes.
- The \ character (called backslash) acts as a so-called escape character because by using the \ we escape from the normal meaning of the character that follows the slash.

```
Character = '\';
```

Character type

- **\n** - denotes a transition to a new line and is sometimes called an LF (Line Feed)

\n

- \r - denotes a return to the beginning of the line and is sometimes called a CR (Carriage Return)

\r

Character type

- **\a** (as in alarm) is a relic of the past when teletypes were often used to communicate with computers; sending this character to a teletype turns on its ringer, hence the character is officially called BEL (as bell);

\a

Character type

- There is an assumption in the C++ language that may seem surprising at first: the char type is treated as a special kind of **int** type.

```
char Char;

Char = 'A';
Char += 32;
Char -= ' ';
```

```
Char = 'A' + 32;
Char = 'A' + ' ' ;
Char = 65 + ' ';
Char = 97 - ' ';
Char = 'a' - 32;
Char = 'a' - ' ';
```

Question: is x equal to y?

- Here we have another developer who counts black and white sheep separately and can only fall asleep when there are exactly twice as many black sheep as white ones.

```
BlackSheepCounter == 2 * WhiteSheepCounter
```

Question: is x not equal to y?

- To ask this question, we use the **!=** (exclamation equal).

```
DaysUntilTheEndOfTheWorld != 0
```

Question: is x greater than y?

- You can ask this question using the **>** (greater) operator.

BlackSheep > WhiteSheep

Question: is x greater than or equal to y?

- The “greater” operator has another special, non-strict variant but it’s denoted differently than in the classical arithmetic notation: **>=** (greater equal).

```
CentigradesOutside >= 0.0
```

Question: is x less than (or equal to) y?

- The operators we're using in this case are the **<** (less) operator and its non-strict sibling **<=** (less equal).

CurrentVelocity < 110

CurrentVelocity <= 110

How do we use the answer we got?

- What can we do with the answer the computer has given us? Well, we have at least two options: first, we can memorize it (store it in a variable) and make use of it later. How do we do that?

```
int Answer, Value1, Value2;

Answer = Value1 >= Value2;
```

Conditions and conditional executions

- Mechanism to allow us to do something if a condition is met.
- The C++ language offers us a special instruction. Due to its nature and its application, it's called a **conditional instruction** (or conditional statement).

```
if(true_or_not) do_this_if_true;
```

```
if(TheWeatherIsGood) GoForAWalk();
HaveLunch();
```

Conditions and conditional executions

- Mechanism to allow us to do something if a condition is met.
- The C++ language offers us a special instruction. Due to its nature and its application, it's called a **conditional instruction** (or conditional statement).

```
if(SheepCounter >= 120) SleepAndDream();
```

Conditions and conditional executions

- When we have to execute conditionally more than one instruction, we need to use the braces **{** and **}** which create a structure known as **a compound statement** or (much simpler) **a block**. The compiler treats the block as a single instruction.

```
if(SheepCounter >= 120) {MakeABed(); TakeAShower();
SleepAndDream(); } FeedTheSheepdogs();
```

Conditions and conditional executions

- When we have to execute conditionally more than one instruction, we need to use the braces **{** and **}** which create a structure known as **a compound statement** or (much simpler) **a block**. The compiler treats the block as a single instruction.

```
if(SheepCounter >= 120){
    MakeABed();
    TakeAShower();
    SleepAndDream();
}
FeedTheSheepdogs();
```

Output

- **cout** is one of these streams and is ready to work without any special preparations – it only needs the header file name.
- Both the << operator and the *cout* stream are responsible for two important actions:
  - **converting** the internal (machine) representation of the integer value into a form acceptable for humans
  - **transferring** the converted form to the output device e.g. console

```
int herd_size = 110;
cout << herd_size;
```

Output

- You can also connect more than one **<<** operator in one ***cout*** statement and each of the printed elements may be of a different type and a different nature.

```
int herd_size = 123;
cout << "Sheep counted so far: " << herd_size;

int square_side = 12;
cout << "The square perimeter is: " << 4 * square_side;
```

Output

- If you want a value of type *int* to be presented as a fixed-point hexadecimal number, you need to use the so-called **manipulator**.
- A manipulator that is designed to switch the stream into a hexadecimal mode is called **a hex**.

```
int byte = 255;
cout << "Byte in hex: " << hex << byte;

int byte = 255;
cout << hex << byte;
cout << byte << dec << byte;
```

Output

- The *oct* manipulator switches the stream into the octal mode.

```
int byte = 255;
    cout << oct << byte;
```

Output

- The three manipulators we showed you previously are only one of the methods (probably the simplest one) of accessing the basefield property. You can achieve the same effect by using **the setbase manipulator**, which directly instructs the stream on what base value it should use during conversion.
- It requires a header file called **iomanip**

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(void)
{
	int byte = 255;
	cout << setbase(16) << byte;
	return 0;
}
```

Output

- In general, output streams (including *cout*) are able to **recognize the type of the printed value** and act accordingly i.e. they'll use a proper form of data presentation for char and float values.

```
char Char = 'X', Minus = '-';
float Float = 2.5;
cout << Char << Minus << Float;
```

Output

- *cout* is able to **recognize the actual type of its element** even when it is an effect of a conversion.
- ASCII code of X is 88

```
char Char = 'X';
 int Int = Char;
 cout<<Char<<" "<<(int)Char<<" "<<Int<<" "<<(char)Int;
```

Output

- Sometimes we may want to (and sometimes we may have to) **break the line** being sent to the screen.

```
cout << "1\n2" << endl << "3\n";
```

Output

- The output streams try to output float values in a form that is more compact, and a decision is taken for every printed float value.
- For example, the following snippet:
  - float x = 2.5, y = 0.0000000025;
  - cout << x << endl << y << endl;
- It will produce the following output on the screen:
  - 2.5
  - 2.5e-009

Output

```
float x = 2.5, y = 0.0000000025;
cout << fixed << x << " " << y << endl;
cout << scientific << x << " " << y << endl;
```

- The program will output the following text:
  - 2.500000 0.000000
  - 2.500000e+000 2.500000e-009

Input

- The simplest way is to mentally reverse the direction of the transfer and to acknowledge that for the data input:
  - we use **cin** stream instead of cout
  - we use **>>** operator instead of <<.
- By the way, the >> operator is often referred to as **an extraction operator**.
- The *cin* stream, along with the extraction operator, is responsible for:
  - transferring the human-readable form of the data from the input device e.g. a console
  - converting the data into the internal (machine) representation of the value being input.

Input

- The user enters the value from the keyboard and the program stores it in a specified variable (*MaxSheep*).

```
cin >> MaxSheep;
```

Input

```
#include <iostream>

using namespace std;

int main(void)
{
	int value,square;

	cout << "Give me a number and I will square it!\n";
	cin >> value;
	square = value * value;
	cout << "You've given " << value << endl;
	cout << "The squared value is " << square << endl;
	return 0;
}
```

Input

```
#include <iostream>
#include <cmath>

using namespace std;

int main(void) {
	float value,squareroot;

	cout << "Give me a number and I will find its square root:" << endl;
	cin >> value;
	if(value >= 0.0) {
		squareroot = sqrtf(value);
		cout << "You have given: " << value << endl;
		cout << "The square root is: " << squareroot << endl;
	}
	return 0;
}
```

Outline

1. **Introduction to computer programming**
2. **Advanced flow control and data aggregates**

The conditional statement

- The “C++” language allows us to express these alternative plans.
- So the *if-else* execution goes as follows:
  - **if** the condition is “true” (its value is not equal to zero) the *perform_if_condition_true* is executed and the conditional statement comes to an end;
  - **if** the condition is “false” (it is equal to zero) the *perform_if_condition_false* is executed and the conditional statement comes to an end

```
if (true_or_false_condition)
    perform_if_condition_true;
else
    perform_if_condition_false;
```

```
if(TheWeatherIsGood)
    GoForAWalk();
else
    GoToATheatre();
HaveLunch();
```

The conditional statement

- Just like the other, simpler instructions we've already encountered, both *if* and *else* may **contain only one statement**.
- If you want to add more than one instruction, then you have to use a block

```
if(TheWeatherIsGood) {
        GoForAWalk();
        HaveFun();
}
else {
        GoToATheatre();
        EnjoyTheMovie();
}
HaveLunch();
```

The conditional statement

- Think about when the instruction placed after *if* is another *if*.
- Remember that every *else* refers to the closest former *if* that doesn't match any other *else*.

```
if(TheWeatherIsGood)
    if(NiceRestaurantFound)
        HaveLunch();
    else
        EatASandwich();
else
    if(TicketsAvailable)
        GoToATheatre();
    else
        GoShopping();
```

The conditional statement

- When you assemble subsequent *if* statements, it's called a **cascade**.

```
if(TheWeatherIsGood)
   GoForAWalk();
else if(TicketsAvailable)
   GoToATheatre();
else if(TableAvailable)
   GoForALunch();
else
   PlayChessAtHome();
```

Not only the int is an int

- To specify our memory requirements, we can use some additional keywords called **modifiers**:
  - **long** – used to declare that we need a wider range of ints than the standard one;
  - **short** – used to declare that we need a narrower range of ints than the standard one;
  - **unsigned** – used to declare that a variable will be used only for non-negative numbers;

Not only the int is an int

- The **Counter** variable will use fewer bits than the standard int (e.g. it could be 16 bits long - in this case, the range of the variable will be suppressed to the range of [-32768 to 32767]).
- The word *int* may be omitted as all the **declarations lacking a type name are considered to specify *int* by default**

```
short int Counter;
```

Not only the int is an int

- The *Ants* variable will use more (or just as many) bits than the standard *int* (e.g. 64 bits so it can be used to store numbers from the range of [-9223372036854775808 to 9223372036854775807]

```
long int Ants;
```

Not only the int is an int

- If we decide that a variable will never be a negative value, we can use the *unsigned* modifier

```
unsigned int Positive;
```

- We can also mix some of the modifiers together

```
unsigned long int HugeNumber;
```

Not only the int is an int

- The *short* and *long* modifiers cannot be used with the *float*, but there is a type named *double*.
- The data stored in the floating-point variable has **finite precision** - in other words, only a certain number of digits are precisely stored in the variable

```
11111111111111111.11111111111111111111
11111111318516531 20.000000
```

- We can say that the variable saves (only) 8 precise digits. This is within the expected accuracy of 32-bit long *floats*. Using a *double* (which is usually 64 bit long) guarantees that the variable will save more significa digits - about 15 to 17.

Floats and their traits

- If a very small float value is added to a very large one, you could end up with a surprise.

11111111000.0 + 0.00011111111

- If we add these two floats, we'll probably get
  - 11111110656.000000

In memory of George Boole

- Variables of this type are able to store only two distinct values: **true** and **false**.

```
bool developer_is_hungry = false;
```

Two simple programs

```
/* finding the larger of two numbers */
#include <iostream>
using namespace std;

int main(void) {
   /* the two numbers */
   int number1,number2;

   /* we will save the larger number here */
   int max;

   /* read two numbers */
   cin >> number1;
   cin >> number2;

   /* we temporarily assume that the former number is the larger one */
   /* we will check it soon */
   max = number1;

   /* we check if the assumption was false */
   if(number2 > max)
      max = number2;

    /* we print the result */
   cout << "The larger number is " << max << endl;

   /* we finish the program successfully */
   return 0;

}
```

Two simple programs

```
/* finding the largest of three numbers */
#include <iostream>
using namespace std;

int main(void) {
  /* the three numbers */
  int number1,number2, number3;

  /* we will save the larger number here */
  int max;

  /* read three numbers */
  cin >> number1;
  cin >> number2;
  cin >> number3;

  /* we temporarily assume that the former number is the larger one */
  /* we will check it soon */
  max = number1;

  /* we check if the second value is the largest */
  if(number2 > max)
  max = number2;

  /* we check if the third value is the largest */
  if(number3 > max)
  max = number3;

  /* we print the result */
  cout << "The largest number is " << max << endl;

  /* we finish the program successfully */
  return 0;

}
```

Two simple programs

- Let's ignore the "C++" language for the moment and try to analyze the problem while not thinking about the programming. In other words, let's try to write the **algorithm**, and when we're happy with it, we'll try to implement it.
- We're going to use a kind of notation that is not a programming language at all, but is formalized, concise and readable. We call this **pseudo-code**.

```
1. max = -999999999;
2. read number
3. if(number == -1) print max next stop;
4. if(number > max) max = number
5. go to 2
```

The loop named “while”

- Performing a certain part of the code more than once is called a **loop**.

```
while my hands are dirty
    I am washing my hands;
```

- *while* repeats the execution as long as the condition evaluates to “true”.

```
while(conditional_expression)
    statement;
```

The loop named “while”

- if you want *while* to execute more than one statement, you must (like *if*) use a block

```
while(conditional_expression) {
        statement_1;
        statement_2;
        :
        :
        statement_n;
    }
```

```
while(1) {
    cout << "I am stuck inside a loop" << endl;
}
```

The loop named “while”

```
#include <iostream>
using namespace std;
int main(void) {
    /* temporary storage for the incoming numbers */
    int number;
    /* get the first value */
    cin >> number;
    /* we will store the currently greatest number here */
    int max = number;
    /* if the number is not equal to -1 we will continue */
    while(number != -1) {
        /* is the number greater than max? */
        if(number > max)
          /* yes – update max */
          max = number;
        /* get next number */
        cin >> number;
    }
     /* print the largest number */
    cout << "The largest number is " << max << endl;
     /* finish the program successfully */
    return 0;
}
```

The loop named “while”

```
int main(void) {
    /* we will count the numbers here */
    int Evens = 0, Odds = 0;

    /* we will store the incoming numbers here */
    int Number;

    /* read first number */
    cin >> Number;

    /* 0 terminates execution */
    while(Number != 0) {
        /* check if the number is odd */
        if(Number % 2 == 1)
            /* increase „odd” counter */
            Odds++;
        else
            /* increase „even” counter */
            Evens++;
        /* read next number */
        cin >> Number;
    }
    /* print results */
    cout << "Even numbers: " << Evens << endl;
    cout << "Odd numbers: " << Odds << endl;
    return 0;

}
```

The loop named “while”

```
if(Number % 2 == 1) ...
if(Number % 2) ...
```

The loop named “while”

```
int main(void) {
     int counter = 5;

    while(counter != 0) {
          cout << "I am an awesome program" << endl;
          counter--;
    }
    return 0;

}
```

```
int main(void) {
    int counter = 5;

    while(counter) {
          cout << "I am an awesome program" << endl;
          counter--;
    }
    return 0;

}
```

```
int main(void) {
     int counter = 5;

     while(counter--)
          cout << "I am an awesome program" << endl;
     return 0;

}
```

The “do” loop or do it at least once

- **do while** loop:
  - the condition is checked at the end of the body execution,
  - the loop's body is executed at least once, even if the condition is not met.

```
do
        statement;
while(condition);

do {
      statement_1;
      statement_2;
      :
      :
      statement_n;
} while(condition);
```

The “do” loop or do it at least once

```
#include <iostream>

using namespace std;

int main(void) {
        int number;
        int max = -100000;
        int counter = 0;
        do {
                cin >> number;
                if(number != -1)
                        counter++;
                if(number > max)
                        max = number;
        } while (number != -1);
        if(counter)
           cout << "The largest number is " << max << endl;
        else
           cout << "Are you kidding? You haven't entered any number!" << endl;
        return 0;
}
```

“for” - the last loop

- it’s more important to **count the “turns” of the loop** than to check the conditions.
- We can distinguish three independent elements there:
  - the initiation of the counter
  - the checking of the condition
  - the modification of the counter

```
int i;

i = 0;
while (i < 100) {
      /* the body goes here */

      i++;

}
```

```
for(i = 0; i < 100; i++) {
      /* the body goes here */
}
```

“for” - the last loop

```
#include <iostream>

using namespace std;

int main(void) {
   int pow = 1;

   for(int exp = 0; exp < 16; exp++) {
      cout << "2 to the power of " << exp << " is " << pow << endl;
      pow *= 2;
   }
   return 0;
}
```

break and continue – the loop's spices

- ***break*** - exits the loop immediately and unconditionally ends the loop's operation; the program begins to execute the nearest instruction after the loop's body;

```
#include <iostream>

using namespace std;

int main(void) {
      int number;
      int max = -100000;
      int counter = 0;
      for( ; ; ){
         cin >> number;
         if(number == -1)
               break;
         counter++;
         if(number > max)
               max = number;
      }
      if(counter)
           cout << "The largest number is " << max << endl;
      else
           cout << "Are you kidding? You haven't entered any number!" << endl;
       return 0;
}
```

break and continue – the loop's spices

- ***continue*** – behaves as the program suddenly reached the end of the body; the end of the loop's body is reached and the condition expression is tested immediately.

```
#include <iostream>

using namespace std;

int main(void) {
int number;
int max = -100000;
int counter = 0;

do {
      cin >> number;
      if(number == -1)
            continue;
      counter++;
      if(number > max)
            max = number;
} while (number != -1);
if(counter)
      cout << "The largest number is " << max << endl;
else
      cout << "Are you kidding? You haven't entered any number!" << endl;
return 0;
}
```