Script for the vhb lecture

Programming in C++
Part 1

Prof. Dr. Herbert Fischer
Deggendorf Institute of Technology

Table of Contents

2 Basic syntax in C++

Chapter 2 contains the basic syntax of C++.
You will learn how to use variables and functions, perform calculations, read values entered from the keyboard and print them to the screen.

2.1 Expressions and Statements

C and C++ distinguish between *expressions* and *statements*. The »official« definition of a statement is the following: »any expression followed by a semicolon is a statement«. The semicolon concludes an expression and turns it into a single-line source text block. Expressions are code sequences of operators and operands, computing a value from the operands, while a statement represents a concluded expression. Let's have a look at the following statement:

- c = a + b;

In this example, the part on the right side of the equals sign, (a + b). is an *expression.*

However, the entire line is a *statement*. For the moment, it is sufficient to know that a *statement* is followed by a semicolon and is also a *full expression*.

2.2 Data types

In C/C++, the *data type* determines how information can be stored in the reserved memory by the compiler. The following table lists the data types, how much memory it takes to store a value in the memory, and the maximum and minimum value that can be stored in the respective type.

Type	Bit width	Range of values	Typical applications
unsigned char	8	0 <= x <= 255	Small numbers and complete PC character set
char	8	- 128 <= x <= 127	Very small numbers and ASCII character set, e. g. char character = 'A';
short int	16	- 32 768 <= x <= 32 767	Counter, small numbers, loop processing
unsigned int	32	0 <= x <= 4 294 967 295	Large numbers and loops
int	32	-2 147 483 648 <= x <= 2 147 483 647	Counter, small numbers, loop control
unsigned long	32	0 <= x <= 4 294 967 295	Distance in astronomical units
long	32	-2 147 483 648 <= x <= 2 147 483 647	Very large numbers, statistical populations
float	32	1,18 10^-38 <= x <= 3,40 10^38	Scientific, with an accuracy of 7 digits
double	64	2,23 10^-308 <= x <=1.79 10^308	Scientific, with an accuracy of 15 digits
long double	80	3,37 10^-4932 <= x <= 1.18 10^4932	Financial statements, with an accuracy of 18 digits
bool	8	„true“ (1) or „false“ (0)	Boolean operations

Several of these data types can be modified using the modifiers *signed* or *unsigned.* A *signed* data type contains both negative and positive numbers, whereas an *unsigned* data type contains positive numbers only.
You might wonder why there are two similar data types with different names in C/C++? This is a relic from the early days. In a 16-bit programming environment, the *int* data type occupies a memory space of 2 bytes and the *long* data type 4 bytes. In a 32-bit programming environment, however, both data types occupy a memory space of 4 bytes each and also use the same range of values. For 32-bit programs, *int* and *long* are identical.

Only the *double, long double* and *float* data types can hold floating-point values (numbers with decimal places). The other data types use integer values only. Although it is permissible to assign a value with fractions to an integer data type, the decimal part will be skipped after the point and only the integer part will be used.

Example:

- `int x = 3.75;`

Value 3 is assigned to variable x because *int* can only hold integer values. (The topic of "variables" is dealt with in the following chapter.)

Please note: In C++ programs, floating-point numbers accept decimal points only, not decimal commas.

The **bool** data type can only hold the values "true" or "false" and 1 or 0. It is mainly used to query truth values (Boolean values) in if-statements (see **Chapter 3.1.1**).

2.3 Variables

Variables are reserved memory locations to store values.

2.3.1 Variables declaration

The name of a variable can begin with an underscore. However, it is usually not a good idea to start a variable name with an underscore, since many compilers also flag specific variable and function names with one or a double underscore at the first character. The maximum length of a variable name varies from compiler to compiler. You should not have any problems if you use 31 characters or less.

C/C++ distinguishes between upper and lower case for names of variables. The following examples show two different variables:

```
int gross;
int GROSS;
```

In C++, you must declare the *type of a variable* (= data type of the variable) before you can define the variable. The *variable type* determines what can be stored in the variable (e.g. text, floating-point numbers, etc.).
First the data type has to be specified, then the name of the variable.

Some examples of valid or invalid names of variables are listed in the following:

- `int aVeryLongVariableName;` // a long variable name
- `int my_variable;` // a variable with an underscore
- `int _x;` // OK, but not recommended
- `int Label2;` // a variable name with a number
- `int return;` // invalid since return is a keyword
- `int zähler;` // invalid since umlauts (vowels with two dots on them) and special characters are not allowed
- `int 123number;` // invalid since a variable must not begin with a number

Specifying the type of variable enables the compiler to perform a type check and to ensure that everything is correct when the program is executed. Using an incorrect data type induces a compiler error or error message. By analyzing and correcting these errors, problems can be resolved before they have a negative impact.

Note:
Variables are reserved memory locations that are kept free to store values.

2.3.2 Initializing variables

The *initialization* assigns a value to the variable.

A variable can be initialized at the same time as the declaration. Thus, the following examples are both permissible:

```
int Accountbalance_Bank1;
Accountbalance _Bank1=100;
```

or:

```
int Accountbalance _Bank2=150;
```

Random numbers are assigned to variables that are *declared* but not *initialized*. Because the storage location which the variable points at has not been initialized, it is impossible to identify what values the memory contains.

Example:

```
int x;
int y;
x = y + 10;      // ATTENTION! y is assigned a random value
```

In the above example, the variable x can hold any value since y was not initialized before its use.

You can also use a single assignment to initialize several variables:

Example:

```
int x = 2;
int y = 2;
```

or:

```
int x=2, y=2;
```

2.3.3 Type conversions

If two variables of different data types are merged by an operator, the result is assigned the more exact data type of the two variables. Two data types are merged, but C/C++ is able to perform an automatic *type conversion*.

Example:

```
float x=1.0;
int i=2;
x=x/i;
```

Therefore, the result of the calculation is 0.5.

2.3.4 Arithmetic overflow

The result of a multiplication of two numbers of the *long* data type is to be assigned to a variable of the *short* data type.

Example:

```
short result;
long a1 = 200;
long a2 = 200;
result = a1 * a2;
```

The result of the multiplication is -25.536. This is because the numbers are *wrapped*: 200 * 200 = 40 000, but the *short* data type has a smaller range of values than *long* and can therefore only hold numbers from - 32 768 to + 32 767.
If the result exceeds 32 767, the result is said to wrap around the maximum, i. e. the excess is added to - 32 768.

40 000 - 32 767 = 7 233
7 233 - 1 = -7 232 // 1 is consumed from 32 767 to - 32 768
- 32 768 + 7 232 = - 25 536

Example: data type short

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

int main()
{
        short x = 32767;
        cout << "x = " << x << endl;        // program output: x=32767
        x = x + 1;
        cout << "x = " << x << endl;        // program output: x= -32768
        system("pause");

}
```

2.3.5 Cast operator

The data type resulting from arithmetic operations can also be specified with the *cast operator*. The data type it is supposed to be converted to is written in brackets.

Syntax: (data type)(variable)

Example: cast

Without cast operator	**With cast operator**

#include <cstdlib>
#include <iostream>
using namespace std;
 	
int main()
{
int a=5, b=2;
float c;
c=a/b;
 	
cout << "c = " << c << endl;
system("pause");
}
	#include <cstdlib>
	#include <iostream>
	using namespace std;
	
	int main()
	{
	int a=5, b=2;
	float c;
	c=(float)(a)/(float)(b);
	
	cout << "c = " << c << endl;
	system("pause");
	}

Result: 2	Result: 2.5

2.3.6 Constants

If you put the keyword *const* in front of the data type, a variable can only be read, i. e. the value of the variable can only be set during its initialization and therefore cannot be altered later in the further course of the source code.

Example: const double pi = 3.14159;

Constants have to be initialized during their declaration! Once you have declared a variable as const, the compiler ensures that the variable will not be altered.

2.4 Operators

Operators are used to manipulate data. Operators are used to perform calculations, check expressions for equality, make assignments, manipulate variables, and perform many other tasks.

Overview of the most important operators in C++.
For part 1 of this course you only need the operators formatted in blue and bold.

Operator	Description	Example
➢ **Arithmetic operators**		
+	**Addition**	**x = y + z;**
-	**Subtraction**	**x = y – z;**
•	**Multiplication**	**x = y * z;**
/	**Division**	**x = y / z;**
➢ **Assignment operators**		
=	**Assignment**	**x = 20;**
+=	**Assignment and addition**	**x += 20; (is equal to x = x + 20;)**
-=	**Assignment and subtraction**	**x -= 20; (is equal to x = x - 20;)**
***=**	**Assignment and multiplication**	**x *= 20; (is equal to x = x * 20;)**
/=	**Assignment and division**	**x /= 20; (is equal to x = x (20;)**
&=	Assignment, bitwise AND	x &= 0x03;
|=	Assignment, bitwise OR	x |= 0x03;
➢ **Logical operators**		
&&	**Logical AND**	**if (x && 0xFF) {...}**
||	**Logical OR**	**if (x || 0xFF) {...}**
➢ **Relational operators**		
==	**Equal**	**if (x == 20) {...}**
!=	**Not equal**	**if (x != 20) {...}**
<	**Less than**	**if (x < 20) {...}**
>	**Greater than**	**if (x > 20) {...}**
<=	**Less than or equal to**	**if (x <= 20) {...}**
>=	**Greater than or equal to**	**if (x >= 20) {...}**
➢ **Unary operators**		
*	Dereferencing operator	int x = *y;
&	Referencing operator	int* x = &y;
~	Bitwise NOT	x &= ~0x03;
!	**Logical NOT**	**if (!valid) {...}**
++	**Increment operator**	**x++; (is equal to x = x + 1;)**
--	**Decrement operator**	**x--; (is equal to x = x - 1;)**
➢ **Member access operators**		
::	**Scope resolution operator**	**MyClass::Function();**
->	Indirect member access operator	MyClass->Function();
.	**Direct member access operator**	**MyClass.Function**

Example: assignment operators

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

int main()
{
        int x= 2;                   // The variable x is assigned the value 2
        x+=10;                      // Is equal to x = x +10;
        cout << x<< endl;           // Prints value 12 to the screen

        system("pause");
}
```

Increment and decrement operators

The *pre-increment operator* (++x) tells the compiler to increment the value of the variable (add 1 to the variable) and only then reuse it, whereas the *post-incremental* operator (x++) uses the variable first and then increases the value by 1. The decrement operator (--) works analogously to the increment operator (reduces the value of the variable by 1).

Exercise:
Determine the values of the following output statements and check your solution via the result of the console output:

```
int x = 10;
cout << "x = " << x++ << endl;
cout << "x = " << x << endl;
cout << "x = " << ++x << endl;
cout << "x = " << x << endl;
```

2.5 Functions

A *function* is a group of statements that are separated from the main program. *Functions* are called (executed) when they are needed in a program to perform specific actions. For example, there could be a function that takes two values, uses them to perform a complex mathematical calculation and then returns the result.

Functions are an integral part of any programming language, and C/C++ are no exceptions. The simplest type of function does not require any *parameters* and returns *void* (i.e. no value will be returned). Other functions require one or more *parameters* and they can return a value. Functions are subject to the same name conventions as variables. A *parameter* is a value that is passed to a function and is used to manipulate its execution or to indicate the extent of its operation.

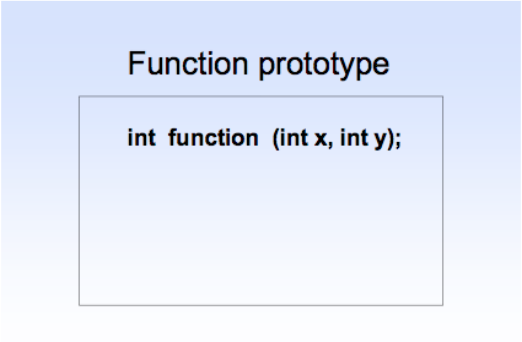
Note:
A function can return no value (void) or a specific value.
A function can have one or more input parameters.

2.5.1 Declaration of a function statement (prototype):

Functions have to be declared before they can be used. A *function declaration* (also known as a *prototype*) tells the compiler how many parameters can be passed to the function. In addition, the compiler is also informed about the data types of the individual parameters and the return value of the function.

- ret_type function_name(argtype_1 arg_1, argtype_2 arg_2, ..., argtype_n arg_n);

The declaration of a function identifies a function that is to be included into the code. The declaration specifies the data type of the return value (ret_type) and the name of the function (function_name). It also determines the order (arg_1, arg_2,..., arg_n) and the data types (argtype_1, ar-gtype_2,..., argtype_n) of the data arguments that will be passed to the function.



At least the data type of the return value and the data types of the input parameters must be specified in the prototype:

Examples:
```
int function(int,float,float);      // 1 return value, 3 input parameters
void function();                    // no input parameters, no return value
```

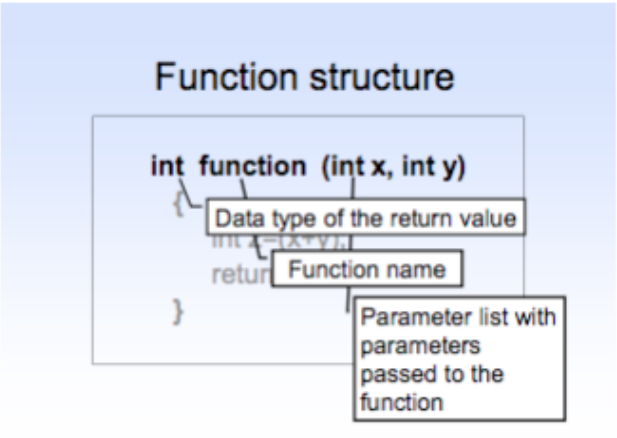
The function prototype can be omitted if the function is only called after its definition (see below).

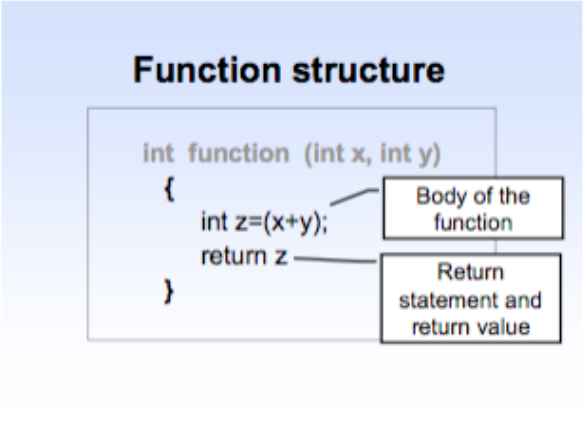
2.5.2 Definition of a function statement

```
ret_type function_name(argtype_1 arg_1, argtype_2 arg_2,..., argtype_n arg_n)
{
    Statement;
    return ret_type;      // Type of the variable or value must be the same as ret_type
}
```

The definition of a function statement identifies the code block (statements) the function consists of and it specifies the data type of the return value ret_type. The function can by identified by its name (function_name). The parameters passed to the function (arg_1, arg_2,..., arg_n) and their data types (argtype_1, argtype_2,..., argtype_n) are also part of the definition.

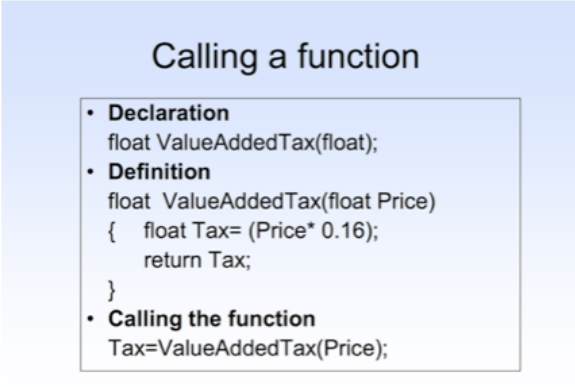
If the function prototype contains a return value, the definition must also contain a *return* statement that returns the value.
If the value *void* is returned, the return statement can be omitted or return 0; can be used.
If you try to return a value from a function with the return type *void*, you will be notified of a compiler error. If no return statement has been specified, the function will automatically return when the end of the function block is reached (the closing curly bracket).





2.5.3 Calling a function

To call a function, you need to pass the function name along with a list of the required parameters (within brackets, separated by commas).



Declaration, definition and call of a function are independent components and therefore reusable. In this way, the prototype (declaration) can also be stored in a header file and made accessible to several programs.
In the main function, a function can be called several times with different input parameters and it will then return the corresponding result.
Please note, however, that the number and the data type of the input parameters must match the functions definition.

Example: functions for multiplication

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

int multiply(int, int);                    // Prototypes of the multiply and showResult functions
void showResult(int);

int main()
{
        int result;                        // Declaration of a variable

         result = multiply(3, 4);          // Calling multiply; function with constants 3 and 4 as input parameters
                                           // The result of the calculation is assigned to the result variable
         showResult(result);               // Calling showResult function with result variable as input parameter

        cout << endl;
        getchar();
}

int multiply(int a, int b)                 // Definitions of the multiply and showResult functions
{
    return a * b;                          // The result of the multiplication of variables a and b is the return value
}

void showResult(int result)
{
    cout << "Result of the multiplication: " << result << endl;      // No return value, only screen output
}
```

Explanation:
Each program starts with the *main* function.
This program passes the values 3 and 4 to the multiply() function and then calls it to multiply the two numbers. Afterwards, the showResult() function is called to display the result. The two *prototypes* of the multiply() and the showResult() functions are located directly above the main program main(). The prototype only specifies the type of the return value, the function name and the data type of the function parameters. This is the minimum requirement for a *function declaration*. If required, names of variables can also be included in the function prototype in order to better document how the function works. The multiply() function could also have been declared as follows:

```
int multiply(int firstNumber, int secondNumber);
```

In this case, it is quite obvious what the multiply() function is being used for, but it can do no harm to document your code through comments and the code itself.

The definition of the multiply() function is located outside the main function. The *function definition* contains the actual body of the function. In the example, the body of the function is minimal since the function only multiplies two function parameters and then returns the result.
There are different ways to call the multiply() function. In the above example, the function was called using constants. You can also pass variables, or even the results of other function calls.

Example: variables as parameters

```
#include <cstdlib>
#include <iostream>

using namespace std;

int multiply(int, int);                          // Prototypes of the multiply and showResult functions
void showResult(int);
int main()                                       // Program starts with the main function
{
        int x, y, result;                        // Declaration of 3 variables

        cout << " Please enter the first value: ";          // Output on the screen
        cin >> x;                                           // User enters the value of x
        cout << " Please enter the second value: ";         // Output on the screen
        cin >> y;                                           // User enters the value of y

        result = multiply(x, y);                 // Calling multiply function; result is assigned to the result variable

        showResult(result);                      // Return value of the showResult function
                                                 // with result variable as input parameter

        cout << endl << endl << " Continue with keystroke  ..."; // Output on the screen
        system("pause");                                    // Program end
        return 0;
}

int multiply(int a, int b)                                  // Definition of multiply function
{
    return a * b;                                           // Return value is the result of the multiplication
                                                            // of variables a and b
}

void showResult(int result)                                 // Definition of showResult function
{
     cout << " Result of the multiplication: " << result;   // No return value (data type void), only screen output
}
```

Example: Return value (of *multiply* function) serves as parameter for another function (*showResult*)

```
#include <cstdlib>
#include <iostream>

using namespace std;

int multiply(int, int);
void showResult(int);

int main()
{
        int x=3, y=4;

        showResult(multiply(x,y));               // Return value of multiply function serves as
                                                 // input parameter of showResult function
        cout << endl;
        system("pause");
        return 0;
}

int multiply(int a, int b)
{
    return a * b;
}

void showResult(int result)
{
     cout << " Result of the multiplication: " << result << endl;
}
```

Functions can call other functions and even themselves. This is called *recursion*. The functions described in this section are exclusively standalone functions (*standalone* because they are not part of a class). The use of standalone functions in C++ does not differ in any way from C, although the spectrum of functions in C++ has been extended.

Rules for functions:

- A function can take no or any number of parameters, but it can only return one or no value.
- If a function defines a return value of the void type, it cannot return a value.
- If the function prototype specifies that the function returns a value, the function body should contain a return statement that returns a value. Otherwise, you will receive a warning from the compiler.
- Variables can be passed to the functions as a value, by pointers or by references (described in Part 2).

2.5.4 Scope of variables

The term *scope* refers to the validity and visibility of variables within the different parts of the source code. Most variables are *local variables*. Local variables are declared inside a function or block and are only valid and visible (can only be used by statements) within that function or block of code.

Example: test program for the scope of variables

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

int a,b;                                    // Declaration of global variables a and b

void change()                               // Definition of change() function
{
    int a;                                  // Declaration of a local variable a
    a=0;
    {                                       // Second block
        int a=20;                           // Declaration of a local variable a
        cout << "The value of a within the second block is: " << a << endl;
    }
    cout << "The value of a within the change function is: " << a << endl;
}

int main()                                  // Calling main function = program start
{
    a=b=10;                                 // The global variables a and b are assigned the value 10

    cout << "Before calling change function, the value of a is: " << a << " and b: " << b << endl;
    change();                               // Calling change() function
    cout << "After calling change function, the value of a is: "<< a << " and b: " << b << endl;;
    system("pause");
    return 0;
}
```

Screen output:

Before calling change function, the value of a is: 10 and b: 10
The value of a within the second block is 20
The value of a within the change function is 0
After calling change function, the value of a is: 10 and b: 10

Explanation:
Variable a is declared twice within the *change()* function (line7 and line11).
Usually the compiler returns an error if a variable has accidentally been declared more than once (*multiple declaration for 'a'*) and aborts the compilation. For this program, however, compilation and execution are successful. You might wonder why.
It is because each of the variables a has a different scope and is therefore a ***local variable***. The definitions of local variables always have to be the first statements of the block in which they are included. Variables that are defined inside a block are local variables and can only be used by statements that are inside that block (in the above example: the second statement block). If there are several variables with the same name, the name is used to refer to the most local (from the innermost block) of the variables.

Please note:
All local variables defined inside a block are destroyed as soon as the block ends.

Finally, the declaration of variable a in line 3 will be explained. Because this variable was declared outside of all the functions, it is also called a ***global variable*** and has a global scope. This means that the global variable is available for use throughout your entire program after its declaration: within the main function, within the second block, and within the change() function.

Important:
Define variables as local as possible and as global as necessary. This is one of the basic rules of modular programming.

2.5.5 Inline function

Calling a function takes a certain amount of time. The return address for function calls is stored on a stack memory, also known as “stack”. The parameters are also stored on the stack. The program jumps to the memory location of the called function, executes the function code, stores the return value of the function, and then jumps back to the address of the statement that was saved just before executing the called function. The overall administration effort becomes apparent as soon as the number of calls increases, e.g. in loops or when time-critical applications are involved. To avoid such an effort, a function can be declared as inline. This means that there is no actual function call. The validation of the syntax remains and the parameters are replaced accordingly.

It should be noted that the inline declaration is only suitable for functions with a short execution time compared to the effort required for the call. Furthermore, inline is only a recommendation to the compiler to perform the replacement, but the compiler does not have to adhere to it. If the compiler finds that a replacement does not have any runtime advantages, it is free to either compile the inline function or not.

Example: inline function

```
#include <cstdlib>
#include <iostream>

using namespace std;

inline int maximum (int a, int b) {
      return a > b ? a : b;
}

int main()
{
   cout << maximum(4,7) << endl;
   cout << maximum(3,9) << endl;

   system("pause");
   return 0;
}
```

In the example, the maximum function is not compiled as a function, but the program code of the function is inserted directly at the location where the function is called. This saves a jump to the function, and it saves copying the parameters and the jump back to the address of the statement. The compiled program code looks as if it was directly at the location of the call (in the main function).

Example:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
   cout << (4 > 7 ? 4 : 7) << endl;
   cout << (3 > 9 ? 3 : 9) << endl;

   system("pause");
   return 0;
}
```

2.6 Input and Output

2.6.1 I/O Streams

In addition to the input and output systems of C, C++ has new streams based on classes. The term stream reflects the situation that character strings are treated as a data stream during input and output.
A major advantage of the new streams is type checking:
With the classic C functions, such as *printf(),* the compiler cannot always check the correct call of the functions. An incorrect format element or the wrong type of an argument inevitably leads to a runtime error.

Example:

```
double x = 1.5;
printf("/d", x);          // Error!
                          // Incorrect format element.
```

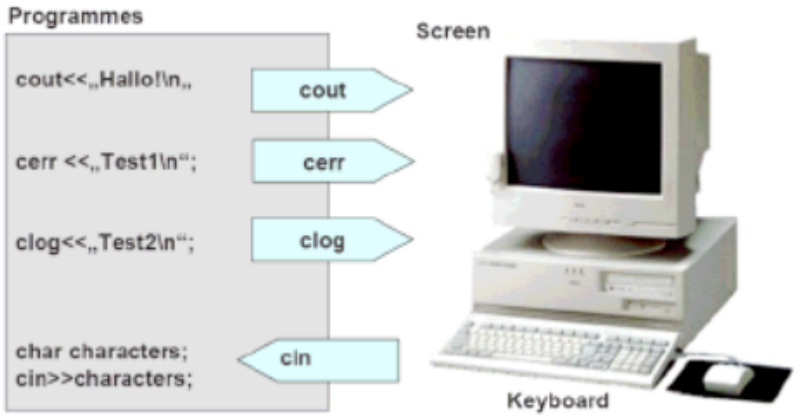
This kind of error is impossible when using the new streams: Based on the argument type, the compiler decides which input or output routine is to be called.

Screen output using printf()	Screen output using cout

float number = 12.345f;
printf(„The number is: %f\n“, number);
	float number = 12.345f; // At the end if is used,
	// because by default the memory
	// space for variables with the data
	// type *double* is used, which is
	// larger than float. So we force
	// the memory space to use float.
	
	cout << “The number is:“; // a string
	cout << number; // a floating-point value
	cout << ’\n’ // a new line
	
	// Usually you only write one statement:
	cout << “The number is:“ << number << ’\n’;

2.6.2 Basic Input / Output

The classes for the new streams are declared in the *iostream* header file, and there are the following objects:



cout: Standard output stream

cerr: Standard error stream

clog: Standard log stream

cin: Standard input stream

Unlike cerr, the clog stream is buffered and is normally used for log outputs.
Old and new stream functions should not be mixed in one program, i.e. if, for example, cout is used for input streams, printf() should not be used elsewhere.
The following text shows how to use the new streams. All global identifiers defined in the C++ standard library are assigned to the std; namespace.

Following the directive

- using namespace std;

these identifiers can be called directly.

Instead of calling the identifiers with *std::cout*, you can call them with *cout* by using the namespace.

2.6.3 Output with cout

The *cout << "Hello world! \n";* statement causes the "*Hello world! \n*" string to be “connected to” the standard output device (which is usually the display screen). With its direction, the stream insertion operator << indicates that the string is pushed to the left into the output stream.

Just like strings, you can also output values with a different data type.

Example:
```
int counter = 10;
cout << counter;        // Output: 10
```

The cout object is "intelligent" enough to identify the data type: Using cout and the << stream insertion operator, all values with an elementary data type can be output, except for the string data type. In addition, the scope of the << stream insertion operator can be extended to other data types at any time, including self-defined data types. To do so, the << operator has to be overloaded. This technique is dealt with in the second part of the course. The *cout << value;* expression also represents the cout object itself. Therefore, the << operator can be used several times in a row.

Example: cout << "This string and a character: " << 'A';

In this example, the character constant (single character) 'A' is used. Please note the following difference between C and C++:

C: A character constant is of data type int.
C++: A character constant is of data type char.

The character A is in fact printed on the screen, and not the character code of 'A'.

To be able to output umlauts in C++, you have to address them via the escape sequences and then the hexadecimal/octal value.

Character	Hex	Oct
Ä	8E	216
ä	84	204
Ö	99	231
ö	94	224
Ü	9A	232
ü	81	201
ß	E1	341

The output of the umlaut 'ö' using the hexadecimal notation would look like this:

```
cout << "Heute ist ein sch\x94ner Tag!" << endl; // \x initiates a hexadecimal escape sequence
```
(“Heute ist ein schöner Tag!” translated into English is “Today is a beautiful day.”)

However, this version has a catch: If the next character can also be displayed in hexadecimal or octal form, this character will also be interpreted. For example, in the word 'Oberfl\x84che' (Oberfläche=surface), '\x84' would not be replaced with the character 'ä', but 'x84c' with the number '2124'. However, there is a little trick for this problem:

```
cout << "Oberfl\x84 \bche!" << endl;
```

After the hexadecimal value, a blank space is added which is immediately deleted with the escape sequence '\b' , which stands for the backspace key.

2.6.4 Input with cin

Reading from the standard input stream is done with cin and the >> stream extraction operator. The cin object is “to be attached” to the standard input device (which is usually the keyboard).

Example:
```
float var;
cin >> var;      // Store floating-point number in var.
```

With its direction, the >> stream extraction operator indicates that the data “is pushed” into the variable var. Using the data type of the variable, cin also determines what is accepted as input value and how the extraction takes place. In this way, values with an elementary data type and strings can be read in.

Contrary to the scanf() input function (corresponding command in C), no careless mistakes can be made by choosing the wrong format elements.
The above example is equal to the following scanf statement:

```
float var;
scanf("%f", &var);  // Store floating-point number in var.
```

If using cin and the operator >>, the following rules apply:
- Leading whitespace characters are skipped.
- The execution of the input terminates if a character cannot be processed.

Example:
```
float var;
cin >> var; // Store floating-point number in var.
```

If you enter 1.20 €, the value 1.20 will be stored in the var variable, and any other characters will remain in the input buffer. If you entered € 1.20, no value would be assigned to var and the input would remain in the buffer.

2.6.5 Output with cerr

The cerr standard error output offers the same options as cout. In particular, cerr usually prints its output on the screen.

The distinction between cout and cerr is important when the output is redirected (e.g. to a file). Therefore, error messages should be output via cerr. This is the only way to ensure that they are printed on the screen even if the standard output has been redirected.

Example:
```
float var;
if( ! (cin >> var) )        // If no value was read in
cerr << "Error: No number was read in!\n";
```

The program can determine whether a value has been read in. That is to say if the input is made during a branch or race condition, the expression "true" is returned if the input was successful, otherwise "false".