**Natural language vs programming language**

We're here to show you what the C++ language is and what we can use it for. Let’s consider for a moment what language itself is, not only the C++ language, but any language people use. We’ll try not to use scientific sounding definitions and we’ll try to speak informally. We can say that **a language is a tool for expressing and recording human thoughts**. We really like this definition. In other words, it's a mechanism known to us and to our partners, allowing us all to understand and to be understood. We use language for speaking, writing, reading, listening and thinking.

At least one language accompanies us throughout our entire lives - it's our native language (our mother tongue), which we learn almost unconsciously from the very beginning. Many of us will learn other languages too, mostly as a result of a conscious decision, forced by social conditions or business needs. The languages we use to communicate with other people are called natural languages. They were created over many centuries and still are subject to change. If we ignore languages that have been created artificially, such as Esperanto or even Quenya (the language used by elves in Tolkien's world), their development is almost independent of their speakers and evolves naturally, in a way that gives us little to no control over it.

However, there are languages whose creation and development were (and often continue to be) dictated by some specific needs, and whose development is fully subject to control by very wide groups of people like international committees or work groups. The shape of these languages is defined by international standards, and although they may be understood by many people, the exchange of thoughts between human-beings is not their priority.

Such languages are, among others, **programming languages**. You are probably familiar with this concept already. A programming language is defined by a certain set of rigid rules, much more inflexible than any natural language. For example, these rules determine which symbols (letters, digits, punctuation marks, and so on) could be used in the language. This part of the definition of language is called **lexicon**. Another set of rules determines the appropriate ways of collating the symbols - this is the **syntax** of the language. We also need to be able to recognize the meaning of every statement expressed in the given language, and this is what we call semantics. Any program we write must be error-free in these three ways: **lexically, syntactically and semantically**, otherwise, the program won’t run, or it will produce unacceptable results. You can be sure that you’ll encounter all of these errors, as to err is human and it’s these fallible humans who write the computer programs.

You might be asking yourself why we need to use a programming language at all, and that's a good question. Let’s try to answer it.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A computer, even the most technically sophisticated one, is devoid of even a trace of intelligence. You could say that it is like a well-trained dog – it responds only to a predetermined **set of known commands**. And sometimes, like a dog, it just shuts down and refuses to do what it’s told. The recognized commands are very simple. We can imagine that the computer responds to orders like “take that number, add to another and save the result”.

A complete set of well-known commands is called an **instruction list**, sometimes abbreviated to **IL**. Different types of computers may vary depending on the size of their ILs and the instructions themselves can be entirely different in different models.

The IL is in fact an alphabet that is commonly known as **machine language**. This is the simplest and the most primary language we can use to give commands to our computer. We can say it's the computer’s mother tongue.

It is possible (and this is often done in practice) for a computer program to be coded directly in machine language using elementary instructions (orders). This kind of programming is tedious, time-consuming and **highly susceptible to a programmer's mistakes**. During the early stages of computer technology, it was the only available method of programming, and it very quickly revealed its serious flaws.

Programming in machine language requires full knowledge of a computer’s hardware design and its internal structure. This also means that replacing the computer with one that differs in design from its predecessor can make the programmer’s entire knowledge unusable. Also, the old programs could be completely useless if the new computer used a different IL. Thus, a program written for a specific type of computer could be completely useless to other computers and vice versa. Secondly, programs written in machine language are very **difficult to understand for humans**, including experienced programmers. It is also the case that to develop a program in machine language takes a lot of time and is very costly and cumbersome.

All these circumstances lead to a need for some kind of bridge between the people's language (natural language) and computer language (machine language). That bridge is also a language – an intermediate common language for both humans and computers working together. Such a language is often called **a high-level programming language**.

A language like this is at least somewhat similar to a natural language: it uses symbols, words and conventions readable to humans. This language enables humans to express complex commands for computers.

You may be asking how to persuade a computer to understand programs written in this way. Try as you might, encouraging the computer is just not enough, but you can translate your program into machine language. Moreover, the translation can be done by a computer, making the whole process fast and efficient.

How great is that? You don’t need to learn a whole bunch of different machine languages – you just need to know one high-level programming language. If there is a translator designed for a specific computer, you can run your program without any problems. In other words, the programs written in high-level languages can be translated into any number of different machine languages, and thus enable them to be used on many different computers. This is called **portability**.

Natural language vs programming language

The translation we are referring to is made by a specialized computer program called a **compiler**. The process of translating from a high-level language into a machine language is called **compilation**.

Now let's get back to more interesting issues related to the process of creating a new program. We already know that the main task is to write a program in accordance with the rules of the chosen programming language. Such a program (which in fact is just text) is called the source code, or simply source, while the file which contains the source is called the **source file**.

To write the source code, you will need a text editor that allows you to manipulate text without any formatting information (for this reason Microsoft Word isn't a good choice - Notepad is better). This code is placed in a file and the name of the file should imply its content. For example, it's common for a file containing the source code in the C++ language to have its name ending with the suffix .cpp, so if you write a computer program and decide to name it proggie, it would be a good idea to put the source code into a file named proggie.cpp. Some platforms may prefer other suffixes like cc, cp, cxx, c++ or even C (note that this is a capital letter). Please consult your compiler documentation for details.

Next, your source code needs to be compiled. To do this you need to run an appropriate compiler and instruct it where you stored the source code you want it to translate into machine language. The compiler reads your code, makes some complex analysis, and then determines whether or not you made any errors during the coding process. If you have, it will tell you. This is not to make you feel bad - these analyses are very insightful and helpful. But remember that they are made by a machine, not a human, and you shouldn't expect too much from them. Okay, if your mistake was that you tried to add up two numbers using # instead of +, the compiler will kindly inform you of your mistake. However, if you typed - instead of +, the compiler can't guess that your intention was to add two numbers, rather than to subtract them. **The compiler will not think for you**. But that's okay too. If the compiler did everything, developers wouldn't have jobs.

If the compiler doesn't find any mistakes in your source, the result will be a file containing your program translated into machine language. That file is commonly called an executable file. The name of the file depends on the compiler you use and the operating system you are working with. For example, most compilers designed for Unix/Linux system create an output file named "a.out" by default. Compilers designed for use in MS Windows® can name the same name to this file as the source file, while only changing the suffix from .cpp to .exe.

Of course the whole process is actually a bit more complicated. Your source code might be huge and divided among several or even dozens of source files. It may also be that the program was not written by you alone, but by a whole team, in which case the division of sources into multiple files is simply a must. In this case, the compiling process splits into two phases: a compilation of your source in order to translate it into machine language and a joining (or gluing) of your executable code with the executable code derived from the other developers into a single and unified product. The phase of "gluing" the different executable codes is commonly known as **linking**, while the program that conducts the process is called a **linker**.

What is the most common use of C++? It is the so-called object programming language, i.e. suitable mostly for large, complex applications, especially those working in graphic environments. Knowing the C++ language is very helpful if you want to learn C# or Java. The conceptual apparatus used by the C++ language is common for all object programming languages, as C++ is one of the oldest languages of that class.

# Further readings

If you take the learning of the "C++" language seriously (and we expect nothing less of you), you will certainly not end your education with this awesome course.

Among the hundreds of books written on the C++ language, there is one that we recommend in particular. The book has been issued dozens of times all around the world and is available in over 20 different (natural) languages. The book’s title is simply "**The C++ Programming Language**". It was the first book to describe the C++ programming language, written by the language’s creator, Bjarne Stroustrup. Stroustrup began developing C++ in 1979 and currently – not surprisingly – is considered the most important authority in this field. You can buy the book [here](http://www.stroustrup.com/4th.html).

Another book worth reading is "**Thinking in C++**" by Bruce Eckel. This book is available to download for free. You can find it [here](http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html).

One of the recently published books that we particularly enjoy is one written by Alex Allain, called "Jumping into C++." It’s a great book for beginners, presenting a step-by-step guide to becoming a C++ programmer. You can order the book [here](http://www.cprogramming.com/c++book/?inl=sb).

Last but not least, for those who aim to improve their programs and learn more about best practices in C++, there is a great book by Scott Meyers, "Effective C++." You can order the book [here](http://www.amazon.com/gp/product/0321334876?ie=UTF8&tag=aristeia.com-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0321334876).

For a more comprehensive list of good C++ books, you may wish to check [this thread](http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list) at *stackoverflow.com*.

If you know any other good books worth recommending, please feel free to contribute to [this thread](https://www.facebook.com/CppInstitute/photos/a.624438340916602.1073741825.478483122178792/747492841944484/?type=1&theater) on our Facebook page.

Once you're a proficient programmer, you may want to have a source of knowledge through which you can quickly find the answers to emerging questions, or just fill in the gaps in your memory. Instead of a handbook, you’ll want a book that briefly describes the language standards - everything that’s really important and nothing more. You need what is called a report (permanently improved and updated), published by the **ISO standardization committee**. You can find the most recent version of the standard at <https://isocpp.org/std>.

But hey, it’s too soon for that. Look into that when you finish this course. No sooner.

**Your first program**

Now we’re going to show you a very simple (and, at the same time, completely useless) program written in the C++ language. We’re going to use this example to present you some basic rules governing the language. The program itself will be modified many times, as it becomes enriched by various additional elements in expanding our programming knowledge. Ready? All right then, let’s go.

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.

That’s all we want at the moment.

What further steps should our first program perform? Let's try to enumerate them here:

1. to **start**;
2. to **write** the text on the screen;
3. to **stop**

This type of structured and semi-formal description of each step of the program is called an **algorithm**. Sources of the word algorithm can be traced back to the Arabic language and originated in early medieval times, and for us, this represents the beginning of computer programming.

Now it's time to see our program. It’s on the right side of the screen, in the editor.

It looks a bit mysterious, doesn't it? Let’s check out each line of the program carefully, and uncover its meaning and purpose. The description is not particularly accurate and those of you who know a little C++ already will probably conclude that it’s too simplistic and somewhat childish. We did this on purpose – we’re not building Rome in a day. Not even in a week!

Let's move on.

**#include <iostream>**

**using namespace std;**

**int main(void) {**

**cout << "It's me, your first program.";**

**return 0;**

**}**

Pay attention to the character # (hash) at the beginning of the first line. It means that the content of this line is the so-called **preprocessor directive**. We’re going to tell you more about the preprocessor later, but for now we’ll only say that 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. The prefix "pre" suggests that these operations are performed before the full processing (compilation) takes place.

The changes the **preprocessor** will introduce are fully controlled by its directives. In our program, 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 (in our case, this is the file named iostream). The changes made by the preprocessor never modify the content of your source file in any way. Any alterations are made on a volatile copy of your program that disappears immediately after the compiler finishes its work.

So why do need the preprocessor to include the content of a completely unknown file called iostream? 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 (you can find it inside our code), but the compiler knows nothing about it so far. In particular, the compiler has no idea that cout is a valid name for that block while cuot isn't. The compiler must be warned about it – it needs to be aware of the fact.

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, which is able to do the trick (and much more). We don't want the compiler to be surprised, so we need to warn it about that. The compiler’s developers put a set of this anticipatory information in the iostream file. All we have to do is use the file. This is exactly what we expect from the include directive.

But where is the iostream file located? The answer is simple, but not as clear as you might want: that’s not our problem. The preprocessor knows where it is. Good job, preprocessor!

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

**#include <iostream>**

**using namespace std;**

**int main(void) {**

**cout << "It's me, your first program.";**

**return 0;**

**}**

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).

An entity defined in a namespace is associated only with that namespace. If you want to use many of the standard C++ entities (we’re going to tell you all about them later) you must insert the using namespace instruction at the top of each file, outside of any function.

The instruction should specify the name of the desired namespace (std in our case). This will make the standard facilities available throughout the program.

**#include <iostream>**

**using namespace std;**

**int main(void) {**

**cout << "It's me, your first program.";**

**return 0;**

**}**

We’ve already said something about the blocks. Now let's go a little deeper. One of the most common types of blocks used to build C++ programs are **functions**.

If you associate a function with mathematics, you’re on the right track. Imagine a function as a black box, where you can insert something into it (though this is not always necessary) and take something new out of it, as if from a magic hat. Things to be put into the box are called **function arguments** (or function parameters). Things to be taken out of the box are called **function results**. In addition, a function can do something else on the side.

If this sounds rather vague, don't worry, we’ll talk about functions many times and in much more detail later.

Back to our 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.

Every function in C++ begins with the following set of information:

* what is the result of the function?
* what is the name of the function?
* how many parameters does the function have and what are their names?

Take a close look at our program and try to read it properly, accepting the fact that you won’t yet fully understand everything.

* wthe result of the function is an **integer** value (we can read it from the word int, which is short for integer)
* the name of the function is main (we know why already)
* the function doesn't require any parameters (which we can read from the word **void**).

This set of information is sometimes called a **prototype**, and it’s like a label affixed to a function announcing how you can use that function in your program. The prototype says nothing about what the function is intended to do. It’s written inside the function and the interior of the function is called the **function body**. The function body begins with the first opening bracket { and ends with the corresponding closing bracket }. It might sound surprising, but the function body can be empty, which means that the function does exactly nothing.

We can even create a function that is lazy – it can be encoded like this:

void lazy(void) { }

This drone provides no result (the first void), its name is "lazy", it doesn't take any parameters (the second void) and it does absolutely nothing (the blank space between brackets).

By the way, the names of the functions are subject to some fairly rigid constraints. More on this later.

**#include <iostream>**

**using namespace std;**

**int main(void) {**

**cout << "It's me, your first program.";**

**return 0;**

**}**

Inside the main function body we need to write what our function (and thus the program) is supposed to do. If we look inside, we find a reference to a block named cout.

Firstly, note the **semicolon** at the end of the line. Each instruction (more precisely, each **statement**) in C++ must end with a semicolon – without it the program will be incorrect.

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). You might ask – how do we know that cout will do that for us? Well, we know it because it says so in the C++ language standards.

The cout entity (in fact, it's an **object**, but we don't want to bring up this word too soon – more on it later) must be fed with something that is intended to be shown on the screen. In our example, the feed is just text (**string**). For simplicity, we can assume that strings in the program in C++ are always enclosed in quotes – that way the compiler recognizes the text that is sent to the user of the program, and the text that is intended to be compiled is translated into machine language. This distinction is very important. Take a look:

int main(void);

The line above is the main function prototype.

"int main(void);"

The line above is not the main function prototype, but a string that only looks like part of a source code. The compiler is not interested in what is between the quotes, and therefore doesn’t recognize such strings as code.

How does it work? We can picture it like this: the execution of our main function is suspended (we can say that the main function falls asleep) and during that time cout, together with << (this kind of symbol is named **operator**) prints the text on the screen. When the text is printed, the main function wakes up and continues its activity.

The above form of source code is the most natural and perhaps the most easily read by humans, but you’re still free to write it in a different way. For example:

cout

<<

"It's me, your first program."

;

In the C++ language it isn’t necessary to write just one statement per line. You can place two (or more) statements on the same line, or split one statement into several lines, but bear in mind that **readability** (for humans) is a very important factor. However, compilers, unlike humans, will never complain about your style.

**#include <iostream>**

**using namespace std;**

**int main(void) {**

**cout << "It's me, your first program.";**

**return 0;**

**}**

We’re almost at the end now. There’s only one line left in our program. This is:

return 0;

This is another (beside the function invocation) statement of the C++ language. Its name is just return and that’s what it does. 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.

The zero after the word return is a result of your function main. It's important - this is how your program tells the operating system the following: *I did what I had to do, nothing prevented me from doing this, and everything is okay.*

If you were to write:

return 1;

this would mean that something had gone wrong, it did not allow your program to be performed successfully and the operating system could use that information to react appropriately.

So is that all? Yes, that’s it! Let's look again at our program and see what’s happening step by step:

* we introduced the function named main into our program - it will be executed when you start the program;
* we made use of an entity named cout inside the main function - it will print the text on the screen;
* the program finishes immediately after printing, indicating that everything you expected to achieve has been achieved.

So hopefully it wasn’t as difficult as it seemed at first glance. Now let’s try to persuade the computer to compute something for us – after all, that’s what they’re for.

**#include <iostream>**

**using namespace std;**

**int main(void) {**

**cout << "It's me, your first program.";**

**return 0;**

**}**

**Numbers and how computers see them**

Do you know how computers perform calculations on numbers? Maybe you've heard of the **binary system** and know that it’s the system computers use for storing numbers and that they can perform any operation upon them. We’re not going to go the intricacies of positional numeral systems here, but we will say that the numbers handled by modern computers are of two types:

* **integers**, that is, those which are devoid of the fractional part;
* **floating-point** numbers (or simply floats), that contain (or are able to contain) the fractional part.

This definition is quite simplistic, but good enough for our purposes. This distinction is very important and the boundary between these two types of numbers is very strict. Both of these kinds of numbers significantly differ in how they are stored in a computer memory and in the range of acceptable values. Additionally, the characteristic of a number which determines its kind, range and application is called **type**.

So now we know two types of the C++ language – an integer type (known as int) and a floating point type (known as float).

For now, let's leave the floating-point numbers aside (that’s right: more on them later) and let’s consider a question that’s maybe a bit banal at first glance: how does the C++ language recognize the integers?

Well, it’s almost the same as when you write them on a piece of paper – they’re simply a string of digits that make up a number. But there’s a catch – you can’t include any characters that are not digits inside the number.

Take for example the number eleven million one hundred and eleven thousand one hundred and eleven. If you took a pencil in your hand right now, you would probably write the number like this:

11,111,111

or (if you are a Pole or a German) like this:

11.111.111

or even like this:

11 111 111

For sure, this makes it easier to read if the number consists of many digits. However, C++doesn’t like this at all. You have to write this number as follows:

11111111

If you don’t, the compiler will shout at you. And how do you code negative numbers in C++? As usual, by adding a minus. You can write:

-11111111

Positive numbers don’t need to be preceded by the plus sign, but you can do it if you want. The following lines describe the same number:

+123

123

For now we’ll deal only with integers – we’ll introduce floating-point numbers very soon.

There are two additional conventions, unknown to the world of mathematics. The first of them allows us to use the numbers in an **octal representation**. If an integer number is preceded by the 0 digit, it will be treated as an octal value. This means that the number must contain digits taken from the 0 to 7 range only.

0123

is an octal number with a (decimal) value equal to **83**.

The second allows us to use hexadecimal numbers. Such number should be preceded by the prefix written as 0x or 0X.

0x123

is a hexadecimal number with the (decimal) value equal to 291.

**A variable is variable**

So the C++ language allows us to write numbers. It probably won't surprise you that we can do some arithmetic operations with these numbers too: add, subtract, multiply and divide. Contain your excitement – we’ll be doing that soon enough. But it’s quite normal to ask how to store the results of such operations in order to use them in other operations. There are special "containers" for that purpose and these containers are called variables. As the name variables suggests, the content of a container can be varied in (almost) any way.

What does every variable have?

* a **name**;
* a **type**;
* a **value**;

Let's start with the issues connected with a variable’s name. Variables don’t magically appear in our program. We (as developers) decide how many, and which variables, we want to have in our program. We also **give them their names**, almost becoming their godparents. If you want to give a name to the variable, you have to follow some strict rules:

* the name of the variable must 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, consequently, two different variables);

These same restrictions apply to all entity names used in C++

The standard of the C++ language does not impose restrictions on the length of variable names, but a specific compiler may have a different opinion on this matter. Don't worry; usually the limitation is so high that it’s unlikely you would actually want to use such long variable names (or functions).

Here are some correct, but not always convenient variable names:

* variable
* i
* t10
* Exchange\_Rate
* counter
* DaysToTheEndOfTheWorld
* TheNameOfAVariableWhichIsSoLongThatYouWillNotBeAbleToWriteItWithoutMistakes
* \_

The last name may be ridiculous from your perspective, but your compiler thinks it’s just great.

And now some incorrect names:

* 10t (does not begin with a letter)
* Adiós\_Señora (contains illegal characters)
* Exchange Rate (contains a space)

You can find more information about C++ naming style and conventions in the [C++ Core Guidelines](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#nl8-use-a-consistent-naming-style). (http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#nl8-use-a-consistent-naming-style)

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. Of course, you can only enter a value that is compatible with the variable’s type. Only an integer value can be assigned to an integer variable (or in other words, to a variable of type int). The compiler will not allow us to enter a floating-point number here.

Let's talk now about two important things – how the variables are created and how to enter a value inside them (or rather – how to give them a value).

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. The construction of such a declaration (or the declaration syntax) is simple: just use the name of the desired type, then the variable name (or variable names separated by commas if there are more than one). The whole statement ends with a semicolon.

Let's try to declare a variable of type *int* named *counter*. The relevant portion of the program looks like this:

int counter;

What is declared by the following fragment of a program?

int variable\_1, account\_balance, invoices;

It declares three variables of type *int* named (respectively) variable\_1, account\_balance and invoices.

Remember that you are allowed to use **as many variable declarations as you need** to achieve your goal.

And how do we give a value to the newly declared variable? You need to use the so-called **assignment operator**. Although this sounds rather mysterious, the operator has a simple syntax and unambiguous interpretation. The assignment operator looks very familiar – here it is:

=

Let's look at some examples:

counter = 1;

The above statement says: *assign a value of 1 to a variable named counter* or a bit shorter *assign 1 to counter*.

Some programmers use a different convention and read such a statement as: ***counter becomes 1***.

Another example:

result = 100 + 200;

In this case, the new value of the variable result will be the result of adding 100 to 200, but you probably already guessed that, right?

And now a slightly more difficult example:

x = x + 1;

Seeing that, a mathematician would probably protest – no value may be equal to itself plus one. This is a contradiction.

But in the "C" language the sign "=" does not mean is equal to, but **assign a value**.

So how do we read such a record in the program?

*Take the current value of the variable x, add 1 to it and store the result in the variable x*

In effect, the value of variable x was **incremented** by one, which has nothing to do with comparing the variable to any value.

**Keywords – why they are the keys?**

Take a look at the right side of the screen – you’ll see a list of words that play a very special role in every C++ language program. They are called **keywords** or (more precisely) **reserved keywords**. They are reserved because you **can’t use them as names**: neither for your variables, nor for your functions or any other named entities you want to create. The meaning of the reserved word is predefined and can’t be changed in any way.

Fortunately, because the C++ compiler is case-sensitive, you can modify any of these words by changing the case of any letter, thus creating a new word, which is not reserved any more.

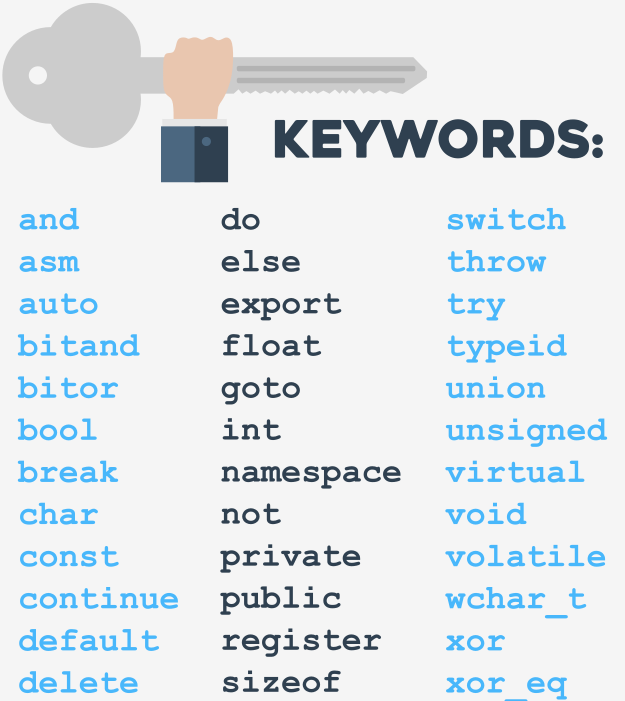
For example - **you can't do this**:

int int;

You can’t have a variable named int - it’s prohibited. But **you can do this** instead:

int Int;

The compiler will be happy, very happy.



**Comments on the comments**

Now we’re going to make some **comments**. We don't mean comments on either your achievements or our achievements, although we’re sure you have many achievements to be proud of. We’re referring to other comments, namely comments on the program and inside the program at the same time.

The developer may want to add a few words addressed not to the compiler but to humans, usually to **explain** to other readers of the code how the tricks used in the code work, or the means of variables and functions, and possibly, to store information on who the author is and when the program was written.

Good and responsible developers describe each newly created important entity; in particular, explaining the role of the parameters and the values modified or returned as results, as well as explaining what the code actually does.

How do we leave something like this in the source code? We have to do it in a way that won't force the compiler to interpret it as part of the code. The remark inserted into the program, which is omitted at the time of compiling, is called a *comment*.

If we want to be precise, we should say that **each comment is lexically equivalent to one space**. Whenever the compiler encounters a comment in your program, the comment is completely transparent to it - from its point of view this is only one space (regardless of how long the real comment is).

The C++ language supports two ways of inserting comments:

// comment - line comments

and

/\* comment \*/ - block comments.

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

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

/\*

and ends with a pair of the following characters:

\*/

The comment can span several lines or can occupy only one line or part of a line.

Here yyou can see an example in which everything from the pair of slash signs on is ignored by the compiler. The line comment can start anywhere on the line. This could be a blank line too, with no content at all.

int counter; // counts the number of sheep in the meadow

Look at the snippet below. Here you can see an example of a similar comment, but introduced into the code using the second method.

Any new developer reading the program will be aware of the true meaning of the variable. The developer will read the code faster and it’ll take less time to understand it.

/\* the counter variable counts the number of sheep in the meadow \*/

int counter;

Developers often place a note at the beginning of the source informing when they write the program stating who amended it and why. The note may appear like this:

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Counting sheep version 1.0

Author: Ronald Sleepyhead, 2012

email: rs@insomnia.org

Changes:

2012-09-13: Ginny Drowsy: counting black sheep improved

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

Note that, despite the complicated structure and the multitude of stars, the condition stating how the comment should be started and finished is fully met.

Comments may be **useful** in another respect - you can use them **to mark a piece of code that you currently don’t need** for whatever reason. We often do this during the testing of the program in order to isolate the place where an error could be hidden.

We’ve just got one more thing to say about comments. Compilers differ in assessing whether another comment may be placed within a single comment. Consider the following program:

/\* int i; /\* int j; \*/ int k; \*/

The question is: are you allowed to nest one block comment (like /\* int j; \*/) inside another block comment?

The answer is no.

You can’t use such a construction in your code.

# Floating-point numbers

A word (or 2.0 words) about floating-point numbers in real life and in the C++ language.

Previously, we became acquainted with the concept of **data type**, and learned that one of the basic types known in the C++ language is the integer type named int. Now it's the time to talk about another type, designed to represent and store the numbers that (as a mathematician would say) have a **non-empty decimal fraction**.

These are the numbers that have (or may have) a fractional part after the decimal point, and although this is a very simplistic definition, it is sufficient for our purposes. Whenever we use a term like "two and a half" or "zero point four" we think of numbers which the computer considers to be **floating numbers**.

Let's go back to the values we quoted a moment ago. "Two and a half" looks normal when you write it in a program, although if your native language uses a comma instead of a point in the number, you should **make sure that your number contains points and not commas**. The compiler will either not accept it or (in very rare circumstances) will misunderstand your intentions, as the comma itself has its own reserved meaning in the C++ language.

If you want to use a value of just "two and a half", you should write it as shown here:

2.5

Note once again – there is a point between "2" and "5" - not a comma.

As you can probably imagine, you can write the value of "zero point four" in C++ as:

.4

Don’t forget this simple rule – you can **omit zero** when it’s the only digit in front of or after the decimal point. In essence you can write the value 0.4 like on the right.

For example: the value of 4.0 could be written as 4. without changing its type or value.

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:

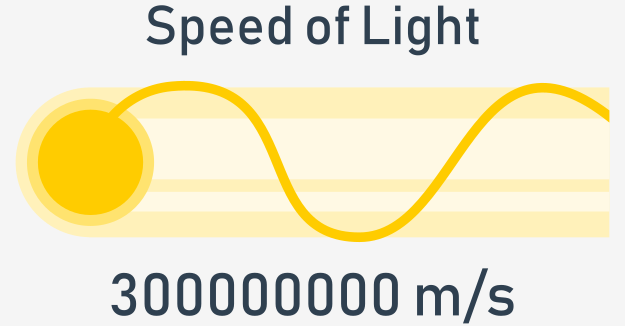
4 is an *int*.

4.0 is a *float*.

We can say that **the point makes a float**. Don't forget that.

When you want to use any numbers that are very large or very small, you can use so-called **scientific notation**. Take, for example, the speed of light expressed in meters per second. Written directly it would look like:

300000000



To avoid the tedious job of writing of so many zeros, physics textbooks use an abbreviated form, which you’ve probably seen already:

3 • 108

It reads: "*three times ten to the power of eight*"

In the C++ language, the same effect is achieved in a slightly different form – take a look:

3E8

The letter *E* (you can also use the lower case letter *e* – it comes from the word exponent) is a concise version of the phrase "*times ten to the power of*".

Note:

* the exponent (the value after the "*E*") **has to be** an integer.
* the base (the value in front of the "*E*") **may or may not be an integer.**

Let's see how we use this convention to record numbers that are very small (in the sense of their absolute value, which is close to zero). A physical constant called *Planck's constant* (and denoted as *h*) has, according to the textbooks, the value of:

6.62607 x 10-34

If you would like to use it in a program, you would write it this way:

6.62607E-34

And that’s it. Not so hard, right?

Let's go back to the floating-point values. We already know what a variable is, and we also know how to declare an integer variable, so now it’s time to declare variables of a floating-point type.

We do this using the keyword float. Knowing that we can declare two floating-point variables, named PI (we can't name it Π because - as you already know - the C++ language freaks out when variable names are written with Greek letters) and Field:

float PI, Field;

As you can see, the difference in declaring the integer and float variables is quite small from the syntax's point of view.

This difference in terms of semantics is significant, as you can see in the example below. With a little thought, we can figure out that the symbol (more precisely - the **operator**) that performs the mathematical **division** is a single character **/** (slash). Take a look at the code:

int i;

float x;

i = 10 / 4;

x = 10.0 / 4.0;

It might be a bit surprising for you to know that the value that will be entered in the variable i is 2 (yes - just 2!) while the variable x will be equal to 2.5. Look at the example carefully, because it illustrates a very important difference between these two data types.

What happens when we have to convert integer values into float values or vice versa? We can always transform from int into float, but it can lead to a **loss of accuracy**. Consider the example below:

int i;

float f;

i = 100;

f = i;

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).

The transformation affects the internal (machine) representation of those values as **computers use different methods for storing floats and ints in their memory**.

Let's consider the opposite situation now.

As you’ve probably guessed, these substitutions will result in a loss of accuracy - the value of the variable i will be 100. Twenty-five hundredths has no meaning in the ints world. Furthermore, **converting a float into an int is not always feasible**.

int i;

float f;

f = 100.25;

i = f;

Integer variables (like *floats*) have a limited capacity. They cannot contain arbitrarily large (or arbitrarily small) numbers.

For example, if a certain type of computer uses four bytes (i.e. 32 bits) to store int values, you can only use the numbers from the range of *-2147483648 to 2147483647*.

The *i* variable is unable to store such a large value, but it isn’t clear what will happen during the assignment. Certainly a **loss of accuracy** will occur, but the value assigned to the variable i is not known in advance.

In some systems it may be the maximum permissible int value, while in others an error occurs, and in others still the value assigned can be completely random.

This is what we call an **implementation dependent issue**. It's the second (and uglier) face of software portability.

int i;

float f;

f = 1E10;

i = f;

# Operators

An **operator** is a symbol of the programming language, which is able to operate on the values. For example, **an assignment operator** is the = sign. We already know that it can assign values to variables.

Now let’s look at some other operators available in the C++ language and find out which rules govern their use and how to interpret the operations they perform. Let’s begin with the operators associated with widely recognizable arithmetic operations. The order of their appearance is not accidental. We’ll talk more about this afterward.

**Multiplication**

An asterisk \* is **a multiplication operator**. If you take a look at the code here, you’ll see that the variable k will be set to the value of 120, while the z variable will be set to 0.625.

int i, j, k;

float x, y, z;

i = 10;

j = 12;

k = i \* j;

x = 1.25;

y = 0.5;

z = x \* y;

**Division**

A slash ("/") is a **divisional** operator. The value in front of the slash is a **dividend**, the value behind the slash, a **divisor**. Look at the snippet of the program below: of course, k will be set to 2, z to 0.5.

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**

As you’ve probably guessed, dividing by zero is strictly forbidden, but the penalty for violating that rule will come to you at different times.

float x;

x = 1.0 / 0.0;

If you dare to write something like that, the compiler will go nuts – you’ll get a compilation error, runtime error or some message at runtime, possibly also a few choice words about your programming capabilities. OK, the last one was a joke. Still, in some cases you won't be able to run your program. As a general rule, you shouldn't divide by zero.

In the following example the compiler won't tell you a thing, but when you try to execute that code, the result of the operation may be surprising. It’s not a number. It’s a special featured value named inf (as in **infinitive**). Generally, this kind of illegal operation is a so-called **exception**.

float x, y;

x = 0.0;

y = 1.0 / x;

When you find exceptions in your program, you should react accordingly. We’ll discuss this later.

**Addition**

The **addition** operator is the "+" (plus) sign, which most of us already know from maths class. Again, take a look at the snippet of the program – as you can surmise, k is equal to 102 and z to 1.02.

int i, j, k;

float x, y, z;

i = 100; j = 2;

k = i + j;

x = 1.0; y = 0.02;

z = x + y;

**Substraction**

The **subtraction** operator is obviously the "-" (minus) sign, although you should note that this operator also has another meaning – it can change the sign of a number. This is a good time to show you a very important distinction between **unary** and **binary** operators (in the C++ language there is also **a ternary operator** – more on that a bit later).

As usual, let’s get acquainted with a snippet of the program – you can guess that k will be equal to -100, while z will be equal to 0.0.

int i, j, k;

float x, y, z;

i = 100; j = 200;

k = i - j;

x = 1.0; y = 1.0;

z = x - y;

**Unary minus**

In "subtracting" applications, the minus operator expects two arguments: the left (a **minuend** in arithmetic terms) and the right (a **subtrahend**). For this reason, the subtraction operator is considered one of the binary operators, just like the addition, multiplication and division operators. But the minus operator may be used in a different way – take a look at the snippet.

int i, j;

i = -100;

j = -i;

As you’ve probably guessed, the variable j will be assigned the value of 100. We used the minus operator as a **unary** operator, as it expects only one argument – the right one.

**Unary plus**

The same dual nature is expressed by the "+" operator, which can be also used as a unary operator – its role is to **preserve** the sign. Take a look at the snippet.

int i, j;

i = 100;

j = +i;

Although such a construction is **syntactically correct**, using it doesn’t make much sense and it would be hard to find a good rationale for doing so.

**Remainder** Оператор залишку

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, which may look a bit confusing. It’s a binary operator (it performs the **modulo operation**) and **both arguments cannot be floats** (don't forget that!). Look at the example:

int i, j, k;

i = 13;

j = 5;

k = i % j;

The k variable is set to the value of 3 (because *2 \* 5 + 3 = 13*).

Oh, and one more thing you need to remember: **you cannot compute the remainder with the right argument equal to zero**. Can you guess why?

You probably remember what we said earlier about dividing by zero. And because division by 0 invokes undefined behaviour, the modulo operation, which relies on division, is undefined, too.

Well, that’s what the C++ Standard says. We have to accept that.

**Priorities**

So far, we’ve treated each operator as if it had no connection with the others. Of course, in real programming, nothing is ever as simple as that. Also, we very often find more than one operator in an **expression** and then things can get very complicated very quickly. Consider the following expression:

2 + 3 \* 5

If your memory hasn't failed you, you should remember from school that multiplications precede additions. You probably remember that you have to multiply 3 by 5, keep the 15 in your memory, add it to 2, thus getting the result of 17.

The phenomenon that causes some operators to act before others is known as the hierarchy of priorities. The C++ language precisely defines the priorities of all operators and assumes that **operators of larger (higher) priority perform their operations before the operators with lower priority**.

So if we know that \* has a higher priority than +, we can figure out how the final result will be computed.

**Bindings**

The **binding** of the operator determines the order of computations performed by some operators with equal priority, put side by side in one expression. Most operators in the C++ language have the **left-sided binding**, which means that the calculation of this sample expression is conducted from left to right, i.e. 3 will be added to 2, and 5 will be added to the result.

2 + 3 + 5

Now at this point you might be snorting and saying that all children know perfectly well that addition is commutative and it doesn’t matter in which order the additions are performed. And yes, you're right, but not quite. Additions performed by the computer are not always commutative. Really. We’ll show you this in more detail later. But for now, be patient and trust us.

**Priorities**

Since you're new to C++ language operators, presenting a complete list of operators' priorities may not be a good idea. Instead, we’ll show you its truncated form, and we’ll expand on it consistently during the introduction of new operators.

This table now looks as follows:

|  |  |
| --- | --- |
| + - | unary |
| \* / % |  |
| + - | binary |

Note: we’ve gone through the operators in order **from the highest to the lowest priority**.

We want to check if you understand the concept of binding. Try to work through the following expression:

2 \* 3 % 5

Both operators (\* and %) have the same priority, so the result could be guessed only when you know the binding direction.

Do you know the result? Click *Check* below to see if you were right:

**Parentheses**

Of course, we’re always allowed to use **parentheses**, which can change the natural order of calculation. Just like with arithmetic rules, **subexpressions in parentheses are always calculated first**. You can use as many parentheses as you need and we often use them to improve the readability of an expression, even if they don't change the order of operations.

An example expression involving multiple parentheses is given below. Try to compute the value given to the l variable.

int i, j, k, l;

i = 100;

j = 25;

k = 13;

l = (5 \* ((j % k) + i) / (2 \* k)) / 2;

Click *Check* below to see if you were right:

10

**Yes, it's 10.**

1.4.1.12

# Operators continued

Here are some operators in the C++ language which you won’t find in maths textbooks. Some of them are frequently used **to increment a variable by one**. This is often done when we’re counting something (e.g. sheep). Let's consider the following snippet:

int sheep\_counter;

sheep\_counter = 0;

Every time a sheep runs through our thoughts we want the variable to be incremented, like this:

sheep\_counter = sheep\_counter + 1;

Similar operations appear very frequently in typical programs, so the creators of the C++ language introduced a set of special operators for these actions. One of them is the ++ (plus plus) operator. You can achieve the same effect in a shorter way:

sheep\_counter++;

It looks much more elegant now, doesn't it?

Similarly, you can also decrease the value of a chosen variable. For example, if we can hardly wait for our holiday, our mind does the following operation every morning:

days\_until\_holiday = days\_until\_holiday - 1;

We can write it in a more compact way:

days\_until\_holiday--;

Sorry, but now we have to introduce a few new words.

The "++" is called the **increment operator**.

The "--" is called the **decrement operator**.

We’ve shown you the ++ and -- operators after a variable (a specialist in the syntax of programming languages would say that they’re used as postfix operators). However, both operators can be placed in front of a variable as well (as prefix operators), like this:

++sheep\_counter;

--days\_until\_holiday;

The effect will be exactly the same: sheep\_counter will be increased by 1, days\_until\_holiday decremented by 1. There’s a fairly significant difference, however, which is described by the precise names of these operators.

\*\*\*\*\*\*\*\*\*\*\*\*\*

Here they are.

That may seem a little weird to you, but it’ll only take a short time to understand. Let's discuss the effects of these operators.

**Operation:**

++variable

--variable

**Effect:**

Increment/decrement the variable by 1 and return its value **already** increased/reduced.

**Operation:**

variable++

variable--

**Effect:**

Return the original (unchanged) variable's value and then increment/decrement the variable by 1.

This behavior justifies the presence of the prefix *pre-* (before) and *post-* (after) in the operators’ names: *pre-* because the variable is modified **first** and then its value is **used**; *post-* because the variable's value is **used** and then **modified**.

**Pre-and post-operators and their priorities**

Take a look at two simple examples.

int i, j;

i = 1;

j = i++;

First, the variable i is set to 1. In the second statement, we’ll see the following steps:

* the value of i will be taken (as we use the *post-incrementation*);
* the variable i will be increased by 1.

In effect, j will receive the value of 1 and i the value of 2.

Things go a bit differently here.

int i, j;

i = 1;

j = ++i;

The variable i is assigned with the value of 1; next, the i variable is incremented and is equal to 2; next, the increased value is assigned to the j variable.

In effect, both i and j will be equal to 2.

Look carefully at this program. Let’s trace its execution step by step.

int i, j;

i = 4;

j = 2 \* i++;

i = 2 \* --j;

1. The i variable is assigned the value of 4;
2. We take the original value of i (4), multiply it by 2, assign the result (8) to j and eventually (post-)increment the i variable (it equals 5 now);
3. We (pre-)decrement the value of j (it equals 7 now); this reduced value is taken and multiplied by 2 and the result (14) is assigned to the variable i.

What else do you need to know about the new operators? Firstly, their priority is quite high – higher than the "\*", "/" and "%" operators. Secondly, their binding depends on whether you use the prefix or postfix version. The prefix version operators have a right-to-left binding, while the postfix operators bind from left to right.

Our priority table now reads as follows:

|  |  |
| --- | --- |
| ++ -- + - | unary |
| \* / % |  |
| + - | binary |
| = |  |

**Shortcut operators**

Now it's time for the next set of operators, ones that make a developer's life easier. The one’s we’ve already described deal with addition and subtraction of one.

i = i \* 2;

However, we often need something other than addition or subtraction, or we want to use a different value; we can use this operator when we want to calculate a series of successive values of powers of 2.

Here’s another example. We use this expression when our herd is extremely numerous:

sheep\_counter = sheep\_counter + 10;

In the C++ language there is a short way to write these operations. You can write them as follows:

i \*= 2;

sheep\_counter += 10;

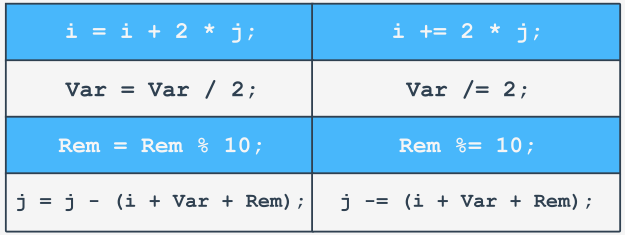
Let's try to present a general description for such operations. If op is a **two-argument operator** (this is a very important condition!) and the operator is used in the following context:

variable = variable op expression;

then this expression can be **simplified** as follows:

variable op = expression;

Take a look at the examples here:



Make sure you understand them all. And relax, because we still have a lot of work ahead.

# Character type

So far we have treated the C++ language (and the computer itself) as a tool for performing calculations on numbers. This is consistent with a common belief that a computer is just a calculator, albeit a very smart one. You know it’s not true, as the computer can be easily used for word processing, too.

We can define a *word* as a string of characters (letters, numbers, punctuation marks, etc.). We dealt with such strings in the first lesson when we used cout to write some text on the computer screen.

Now, however, we’ll ignore the string consisting of multiple characters and we’ll focus our attention on single characters. We’ll come back to the problem of processing strings when we start working on arrays, because in the C++ language **all strings are treated as arrays**.

To store and manipulate characters, the C++ language provides a special type of data. This type is called char, which is an abbreviation of the word *character*.

Let's try to declare a variable for storing a single character.

char character;

Looks familiar, doesn't it? Now let's talk a bit about how computers store characters.

1.5.1.2 Other kind of data

# ASCII code

**Computers store characters as numbers**. Every character used by a computer corresponds to a unique number, and vice versa. This system of assignments includes more characters than you would probably expect. Many of them are invisible to humans but essential for computers. Some of these characters are called **white spaces**, while others are named **control characters**, because their purpose is to **control** the input/output devices. An example of a white space that is completely invisible to the naked eye is a special code, or a pair of codes (different operating systems may treat this issue differently), which are used to mark the ends of lines inside text files. People don’t see this sign (or these signs), but they can see their effect where the lines are broken.

We can create virtually any number of assignments, but a world in which each computer type uses different character encoding would be extremely inconvenient. This has created a need to introduce a universal and widely accepted standard implemented by (almost) all computers and operating systems all over the world. **ASCII** (which is a short for *American Standard Code for Information Interchange*) is the most widely used system in the world, and it’s safe to assume that nearly all modern devices (like computers, printers, mobile phones, tablets, etc.) use this code. The code provides space for 256 different characters, but we’re only interested in the first 128. If you want to see how the code is constructed, go to the table on the right.

Look at it carefully – there are some interesting facts about it that you might notice. We'll show you one. Do you see what the code of the most common character is – the space? Yes – it’s 32. Now look at what the code of the lower-case letter “a” is. It’s 97, right? And now let's find the upper-case “A”. Its code is 65. What’s the difference between the code of “a” and “A”? It’s 32. Yes, that's the code of a space. We’ll use that interesting feature of the ASCII code soon.

Also, note that the letters are arranged in **the same order** as in the **Latin alphabet**.

By the way, ASCII code is being superseded (or rather extended) by a new international standard named UNICODE.

Fortunately, the ASCII set is a [UNICODE](https://en.wikipedia.org/wiki/Unicode) subset. UNICODE is able to represent virtually all characters used throughout the world. We’ll spend a little more time on this later.

| Character | Dec | Hex |  | Character | Dec | Hex |  | Character | Dec | Hex |  | Character | Dec | Hex |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (NUL) | 0 | 0 |  | (space) | 32 | 20 |  | @ | 64 | 40 |  | ` | 96 | 60 |  |
| (SOH) | 1 | 1 |  | ! | 33 | 21 |  | A | 65 | 41 |  | a | 97 | 61 |  |
| (STX) | 2 | 2 |  | " | 34 | 22 |  | B | 66 | 42 |  | b | 98 | 62 |  |
| (ETX) | 3 | 3 |  | # | 35 | 23 |  | C | 67 | 43 |  | c | 99 | 63 |  |
| (EOT) | 4 | 4 |  | ($) | 36 | 24 |  | D | 68 | 44 |  | d | 100 | 64 |  |
| (ENQ) | 5 | 5 |  | % | 37 | 25 |  | E | 69 | 45 |  | e | 101 | 65 |  |
| (ACK) | 6 | 6 |  | & | 38 | 26 |  | F | 70 | 46 |  | f | 102 | 66 |  |
| (BEL) | 7 | 7 |  | ' | 39 | 27 |  | G | 71 | 47 |  | g | 103 | 67 |  |
| (BS) | 8 | 8 |  | ( | 40 | 28 |  | H | 72 | 48 |  | h | 104 | 68 |  |
| (HT) | 9 | 9 |  | ) | 41 | 29 |  | I | 73 | 49 |  | i | 105 | 69 |  |
| (LF) | 10 | 0A |  | \* | 42 | 2A |  | J | 74 | 4A |  | j | 106 | 6A |  |
| (VT) | 11 | 0B |  | + | 43 | 2B |  | K | 75 | 4B |  | k | 107 | 6B |  |
| (FF) | 12 | 0C |  | , | 44 | 2C |  | L | 76 | 4C |  | l | 108 | 6C |  |
| (CR) | 13 | 0D |  | - | 45 | 2D |  | M | 77 | 4D |  | m | 109 | 6D |  |
| (SO) | 14 | 0E |  | . | 46 | 2E |  | N | 78 | 4E |  | n | 110 | 6E |  |
| (SI) | 15 | 0F |  | / | 47 | 2F |  | O | 79 | 4F |  | o | 111 | 6F |  |
| (DLE) | 16 | 10 |  | 0 | 48 | 30 |  | P | 80 | 50 |  | p | 112 | 70 |  |
| (DC1) | 17 | 11 |  | 1 | 49 | 31 |  | Q | 81 | 51 |  | q | 113 | 71 |  |
| (DC2) | 18 | 12 |  | 2 | 50 | 32 |  | R | 82 | 52 |  | r | 114 | 72 |  |
| (DC3) | 19 | 13 |  | 3 | 51 | 33 |  | S | 83 | 53 |  | s | 115 | 73 |  |
| (DC4) | 20 | 14 |  | 4 | 52 | 34 |  | T | 84 | 54 |  | t | 116 | 74 |  |
| (NAK) | 21 | 15 |  | 5 | 53 | 35 |  | U | 85 | 55 |  | u | 117 | 75 |  |
| (SYN) | 22 | 16 |  | 6 | 54 | 36 |  | V | 86 | 56 |  | v | 118 | 76 |  |
| (ETB) | 23 | 17 |  | 7 | 55 | 37 |  | W | 87 | 57 |  | w | 119 | 77 |  |
| (CAN) | 24 | 18 |  | 8 | 56 | 38 |  | X | 88 | 58 |  | x | 120 | 78 |  |
| (EM) | 25 | 19 |  | 9 | 57 | 39 |  | Y | 89 | 59 |  | y | 121 | 79 |  |
| (SUB) | 26 | 1A |  | : | 58 | 3A |  | Z | 90 | 5A |  | z | 122 | 7A |  |
| (ESC) | 27 | 1B |  | ; | 59 | 3B |  | [ | 91 | 5B |  | { | 123 | 7B |  |
| (FS) | 28 | 1C |  | < | 60 | 3C |  | \ | 92 | 5C |  | | | 124 | 7C |  |
| (GS) | 29 | 1D |  | = | 61 | 3D |  | ] | 93 | 5D |  | } | 125 | 7D |  |
| (RS) | 30 | 1E |  | > | 62 | 3E |  | ^ | 94 | 5E |  | ~ | 126 | 7E |  |
| (US) | 31 | 1F |  | ? | 63 | 3F |  | \_ | 95 | 5F |  |  | 127 | 7F |  |

1.5.1.3 Other types

**Character type values**

How do we use the values of the char type in the C++ language? We can do it in two ways, which are not entirely equivalent.

The first way allows us to specify the character itself, but enclosed in single quotes (apostrophes). Let’s assume that we want the variable we declared a few slides earlier to be assigned the value of the upper-case letter “A”.

We do this as follows:

character = 'A';

You’re not allowed to omit apostrophes under any circumstances.

Now let’s assign an asterisk to our variable. We do this as follows:

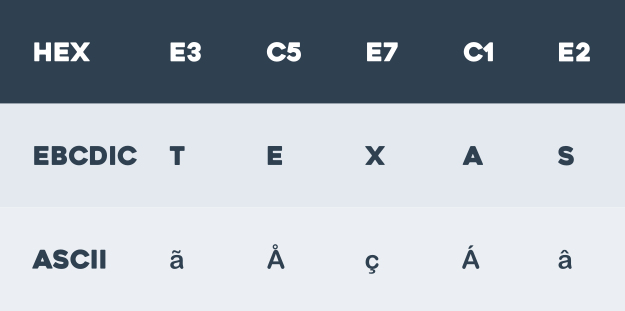
character = '\*';

The second method consists of assigning a **non-negative integer value** that is the code of the desired character. This means that the assignment below will put an “A” into the character variable.

character = 65;

The second solution, however, is less recommended and if you can avoid it, you should. Why?

The second reason is more exotic, but still true. There’s a significant number of computers in the world which use codes **other than ASCII**. For example, many of the IBM mainframes use a code commonly called [EBCDIC](https://en.wikipedia.org/wiki/EBCDIC) (*Extended Binary Coded Decimal Interchange Code*) which is very different from ASCII and is based on radically different concepts.



**ASCII vs EBCDIC**

Now imagine that you’ve written a wonderful program and decided to compile and run it on a computer utilizing the EBCDIC code. If you wrote something like this, the compiler running on that computer would notice the question mark and use the appropriate EBCDIC code for that character.

character = '?';

But if you wrote it like this:

character = 63;

**Literal**

Now’s probably a good time to bring a new term into the mix: a **literal**. The literal is a symbol which **uniquely identifies its value**. Some prefer to use a different definition: the **literal means itself**. Choose the definition that you consider to be clearer and look at the following simple examples:

* character: this is not a literal; it’s probably a variable name; when you look at it, you cannot guess what value is currently assigned to that variable;
* 'A': this is a literal; when you look at it you can immediately guess its value; you even know that it’s a literal of the char type;
* 100: this is a literal, too (of the int type);
* 100.0: this is another literal, this time of a floating point type;
* i + 100: this is a combination of a variable and a literal joined together with the + operator; this structure is called an **expression**.

1.5.1.5

If you’re an inquisitive person, you probably want to ask a question: if a literal of type char is given as the character enclosed in apostrophes, how do we code the apostrophe itself?

The C++ language uses a special convention that also extends to other characters, not only to apostrophes. Let's start with an apostrophe anyway. An apostrophe looks like this:

character = '\'';

The \ character (called *backslash*) acts as an **escape character**, because by using the \ we can escape from the normal meaning of the character that follows the slash. In this example, we **escape** from the usual role of the apostrophe (i.e. delimiting the literals of type char).

You can also use the escape character to **escape from the escape character**. Yes, it does sound weird, but the example below should make it clear. This is how we put a backslash into a variable of type char.

character = '\\';

**Escape characters**

The C++ language allows us to escape in other circumstances too. Let's start with those that denote literals representing white spaces.

\n – denotes a **transition to a new line** and is sometimes called an **LF** (**Line Feed**), as printers react to this character by pulling out the paper by one line of text.

\r – denotes the **return to the beginning of the line** and is sometimes called a **CR** (**Carriage Return** – “carriage” was the synonym of a “print head” in the typewriter era); printers respond to this character as if they are told to re-start printing from the left margin of the already printed line.

\a – (as in **alarm**) - is a relic of the past when teletypes were often used to communicate with computers (do you know what a [teletype](https://en.wikipedia.org/wiki/Teleprinter) is, are you old enough to remember them?); sending this character to a teletype turns on its ringer; hence, the character is officially called **BEL** (as in **bell**); interestingly, if you try to send the character to the screen, you’ll hear a sound – it won't be a real ringing but rather a short beep. The power of tradition works even in the IT world.

\0 (*note: the character after the backslash is a zero, not the letter O*): called **nul** (from the Latin word **nullus** – none) is a character that **does not represent any character**; despite first impressions, it could be very useful, as we’ll show you in the lessons to come.

Now we’ll try to escape in a slightly different direction. The first example explains the variant when a backslash is followed by two or three **octal digits** (the digits from the range of 0 to 7). A number coded in this manner will be treated as an **ASCII value**. It may look like this:

character = '\47';

047 octal is 39 decimal. Look at the ASCII code table and you'll find that this is the ASCII code of an apostrophe, so this is equivalent to the notation

'\''

(but only for computers implementing the ASCII code).

The second escape refers to the situation when a \ is followed by the letter X (lower case or upper case – it doesn't matter). In this case there must be either one or two **hexadecimal digits**, which will be treated as ASCII code. Here's an example:

character = '\x27';

As you’ve probably guessed, 27 hexadecimal is 39 decimal.

1.5.1.7

**char values are int values**

There’s an assumption in the C++ language that may seem surprising at first glance: the char type is treated as a special kind of int type. This means that:

* You can always **assign a char value** to an int variable;
* You can always assign an int value to a char variable, but if the value exceeds 255 (the top-most character code in ASCII), you must expect a loss of value;
* The value of the char type can be subject to the **same operators** as the data of type int.

We can check this using a simple example. We said earlier that in ASCII, the "distance" between upper and lower case letters is 32, and that 32 is the code of the space character. Look at this snippet:

char character;

character = 'A';

character += 32;

character -= ' ';

This sequence of subsequent addition and subtraction will bring the character value to its original value ("A"). You should be able to explain why, right?

All of these assignments are correct. Try to interpret their meanings – this should be a good exercise for you.

character = 'A' + 32;

character = 'A' + ' ' ;

character = 65 + ' ';

character = 97 - ' ';

character = 'a' - 32;

character = 'a' - ' ';

97, 97, 97, 65, 65, 65

**Well done! Here are the answers: 97, 97, 97, 65, 65, 65.**

1.6.1.1 Flow control: how t

# One who asks does not err

A programmer writes a program and the program **asks questions**. A computer executes the program and provides the answers. The program must be able to react according to the answers it receives. Fortunately, computers know only two kinds of answer: ***yes, this is true or no, this is false.*** You will never get a response like “I don’t know” or “Probably yes, but I don’t know for sure”.

To ask questions, the C++ language uses a set of very special operators. Let’s go through them one by one, illustrating their effects using some simple examples.

1.6.1.2 Flow control: ho

**Question: is x equal to y?**

Question: are two values equal?

To ask this question you use the == (***equal equal***) operator.

Don't forget this important distinction:

* = is an **assignment operator**
* == is the **question “*are these values equal?”***

It’s **a binary operator with a left-side binding**. It needs two arguments and **checks if they’re equal**. Now let’s ask a few questions. Try to guess the answers.

**Is x equal to y?**

1. 2 == 2

This question is simple. Of course 2 is equal to 2. The computer will answer true.

2. 1 == 2

This one’s simple, too. The answer will be false.

3. i == 0

Here we’re not able to find the answer if we do not know what value is currently stored in variable i. **If the variable has been changed many times during the execution of your program, the answer to this question can be given only by the computer and only at run time.**

**Question: is x equal to y?**

There’s another developer who counts white and black sheep separately and falls asleep only when there are exactly twice as many black sheep as white ones. The question will be as follows:

black\_sheep\_counter == 2 \* white\_sheep\_counter

Due to the low priority of the == operator, this question shall be treated as equivalent to this one:

black\_sheep\_counter == (2 \* white\_sheep\_counter)

**Question: is x not equal to y?**

To ask this question, we use the != (**exclamation equal**). It’s a very close relative of the == operator. It’s also a binary operator and has the same low priority. Imagine that we want to ask whether the number of days left to the end of the world is currently not equal to zero:

days\_until\_the\_end\_of\_the\_world != 0

The answer true gives us the chance to go to the theater or to visit our friends.

**Question: is x greater than y?**

You can ask this question by using the > (**greater than**) operator. If you want to know if there are more black sheep than white ones, you can write it as follows. The answer true confirms the statement; the answer false denies it.

black\_sheep > white\_sheep

**Question: is x greater than or equal y?**

The “**greater than**” operator has another special, non-strict variant, but it’s denoted differently in classical arithmetic notation: >= (**greater than or equal**). There are two subsequent signs, not one. Both of these operators (strict and non-strict), as well as the other two that we discuss in the next section, are binary operators with left-side binding, and their priority is greater than the ones indicated by == and !=.

If we want to find out whether or not we have to wear a warm hat, we ask the following question:

centigrade\_outside >= 0.0

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

As you’ve probably already guessed, the operators we use in this case are: the < (**less than**) operator and its non-strict sibling <= (**less than or equal**). Look at this simple example: we’re going to check if there’s a risk that we’ll be fined by the highway police (the first question is strict, the second isn't).

current\_velocity < 110

current\_velocity <= 110

**How to use the answer we got?**

What can we do with the answer we get from the computer? There are at least two possibilities: first, we can **memorize it** (store it in a variable) and make use of it later. How do we do that? Well, we would use an arbitrary variable of type int, like this:

int answer, value1, value2;

answer = value1 >= value2;

If the answer is true because value1 is greater than or equal to value2, the computer will assign 1 to answer (1 is arguably different from zero). If value1 is less than value2, the variable answer will be assigned 0.

**The priority table – an update.**

The second possibility is more convenient and far more common: we can use the answer to make a decision about the future of our program. We use a special instruction for this purpose and we’ll tell you what that is very soon.

Now we need to update our priority table. It now looks as follows:

|  |  |
| --- | --- |
| + - | unary |
| \* / % |  |
| + - | binary |
| < <= > >= |  |
| == != |  |
| = += -= \*= /= %= |  |

**Conditions and conditional executions**

You already know how to ask, but you still don’t know how to make reasonable use of the answers. We must have a mechanism which allows us to do something if a condition is met and not to do it if it isn’t. It's just like in life: we do certain things or we don’t when a specific condition is met or not, e.g. we go for a walk if the weather is good, or stay home if it’s wet and cold.

To make these decisions, the C++ language has a special instruction. Due to its nature and its application, it’s called a **conditional instruction** (or **conditional statement**).

There are several variants of the conditional instruction. We’ll start with the simplest, and slowly move on to the more difficult ones. The first form of a conditional statement, which you can see below, is written very informally but figuratively:

if(true\_or\_not) do\_this\_if\_true;

This conditional statement consists of the following, strictly necessary, elements in this and this order only:

* **if** keyword;
* left (opening) parenthesis;
* an **expression** (a **question** or an **answer**) whose value will be interpreted solely in terms of “true” (when its value is non-zero) and “false” (when it is equal to zero);
* right (closing) parenthesis;
* an **instruction** (only **one**, but we’ll learn how to deal with that limitation).

How does this statement *work*?

* if the true\_or\_not expression enclosed inside the parentheses represents the truth (i.e. its *value is not equal to zero*), the statement behind this condition (do\_this\_if\_true) will be executed;
* if the true\_or\_not expression represents a **falsehood** (its *value is equal to zero*), the statement behind this condition is **omitted** and the next executed instruction will be the one that lies after the conditional statement.

In real life we often express a will:

*if the weather is good we will go for a walk next, we will have lunch*

As you can see, having lunch is not a conditional activity and doesn’t depend on the weather (what luck). Knowing what conditions influence our behavior and assuming that we have the parameterless functions go\_for\_a\_walk() and have\_lunch() we can write the following snippet:

if(the\_weather\_is\_good) go\_for\_a\_walk();

have\_lunch();

1.6.1.7

As we already know, our friend the developer falls asleep when he counts 120 sheep. His sleep is implemented as a special function named sleep\_and\_dream(). This function does not require any parameters.

We can read it as: “***if*** *sheep\_counter is greater* ***than*** *or equal to 120, then fall asleep and dream!*”

We’ve said that there may be only one statement after the if statement. When we have to **conditionally** execute more than one instruction, we need to use braces { and } which create a structure known as a **compound statement** or (much simpler) a **block**. The block is treated as a single instruction by the compiler.

This is how we can circumvent the if statement limitation.

Let’s treat our programmer a little nicer:

if(sheep\_counter >= 120){make\_a\_bed(); take\_a\_shower(); sleep\_and\_dream(); }

feed\_the\_sheepdogs();

Now it’s time for some stylistic remarks. Writing blocks one after the other is, of course, syntactically correct, but very **inelegant**. It may cause the text of our program to run outside the right margin of the editor. There are a few styles of coding the blocks. We won't try to argue that some of them are better than others, but we will be using the [K & R style](https://en.wikipedia.org/wiki/Indent_style#K.26R_style). The letters K and R are the initials of the creators of the “C” language, Mr. Kernighan and Mr. Ritchie (the C language is the still living ancestor of C++). They presented this style in their classic book and we will follow it.

The same snippet, written in accordance with the *K & R style*, will look as follows:

if (sheep\_counter >= 120)

{

make\_a\_bed();

take\_a\_shower();

sleep\_and\_dream();

}

feed\_the\_sheepdogs();

Note that a conditionally executed block is **indented** – it improves the readability of the program and manifests its conditional nature.

In the next section, we’re going to discuss another variant of the conditional statement, which also allows you to perform an action only when the condition is not met.

Now feed your sheep dogs, please. They’ve been waiting ages for your attention.

**Input and output**

Now we’re going to set aside conditional statements for a while and spend some time on two important and extremely useful features we use to provide connectivity between the computer and the **outside world**.

Sending data in the direction from human (user) to the computer program is called **input**. The stream of data transferred in the opposite direction, i.e. from the computer to the human, is called **output**.

We've already learned about one useful entity that serves to output data – can you remember its name? Yup, it’s the cout stream, and we used it along with the << operator in the very first program we wrote in the C++ language, right at the beginning of this course.

The << operator itself is sometimes referred to as an insertion operator as it inserts a string of characters into the character device (e.g. a console).

The actual cout capabilities are much more impressive: it’s capable of writing the data of virtually any type on a computer screen.

So what do we do if we want to output the value of type int or float, or char, not only a simple string?

**Output**

To do this and other more complex tasks, we need to use any of the output streams associated with the screen (more formally: with the **console**) and send a value of a variable there.

cout is one of these streams and is ready to work without any special preparations – it only needs the header file name.

If you want to print the value of an integer variable to the screen, the only thing you have to do is send it to the cout stream through the << operator, which indicates the desired direction of data transfer.

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

Streams are very powerful and convenient tools for both input and output. They can easily output several values of different types and mix them with the text. They can also easily input many values at once.

Let's look at streams in a few applications. The first one is trivial - we use cout to print a value of an int variable. We do it like this:

int herd\_size = 110;

cout << herd\_size;

We can expect that a string consisting of characters ‘1’, ‘1’ and ‘0’ will appear on the screen immediately.

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.

Take a look at the example on the right. We’re using **a string literal** (as the former element) and **an integer variable** (as the latter element) in one cout operation.

In this example:

int herd\_size = 123;

cout << "Sheep counted so far: " << herd\_size;

This snippet of code results in the string Sheep counted so far: 123 printed on the screen.

An expression is a legal cout element too. The example below demonstrates one such case:

int square\_side = 12;

cout << "The square perimeter is: " << 4 \* square\_side;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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 is a special kind of entity that tells the stream that the data form has to be changed immediately. All elements outputted after the manipulator activation will be presented in the desired form.

A manipulator that is designed to switch the stream into a hexadecimal mode is called a **hex**. The snippet on the right will output a string consisting of characters 'F' and 'F'.

Technically, a manipulator is a function that changes one of the output stream’s properties, called **basefield**. The property is used to determine what number should be used as a base during the conversion of any int value into human readable text.

int byte = 255;

cout << "Byte in hex: " << hex << byte;

There are two important facts you need to understand here:

1. any manipulator starts its work from the point it was placed at and continues its work even after the end of the cout statement; it finishes working only when another manipulator cancels its action;
2. the name of the manipulator may be in conflict with any other name declared by the programmer; e.g. you can have your own variable named hex which could hide the manipulator’s name; such conflicts are resolved by a specialized mechanism called namespace; more on this later.

The example below demonstrates how manipulators begin and finish their work:

int byte = 255;

cout << hex < byte;

cout < byte < dec < byte;

Note: the dec manipulator switches the stream into a decimal form. We don’t have it explicitly in most cases, since **the decimal is the default** working mode for output streams.

The snippet will output the three specimens of the same value:

1. FF as a hexadecimal representation of 255 (as an effect of the hex manipulator)
2. FF again (the previous hex activation is still working here)
3. 255 (as a result of the dec manipulator activation)

The oct manipulator switches the stream into the octal mode.

The snippet below will output ‘377’ to the screen. Can you guess why?

int byte = 255;

cout << oct << byte;

Yes, 255 in decimal is 377 in octal. Well done.

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.

The only acceptable values for the setbase parameter are 8, 10 and 16. Hopefully you got the purpose of the three manipulators. If not, go back and look at them a little longer. It’ll come to you eventually... we hope.

The program in the editor demonstrates the usage of the setbase manipulator.

The vacancy variable contains 0 if all the rooms are occupied; otherwise it displays the number of available rooms.

Note: it requires a header file called iomanip (the three previous manipulators don’t).

**#include <iostream>**

**#include <iomanip>**

**using namespace std;**

**int main(void)**

**{**

**int byte = 255;**

**cout << setbase(16) << byte;**

**return 0;**

**}**

1.7.1.6 Connecting wit

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.

The snippet in the editor will cause the stream to print the following text on the screen:

X-2.5

**char Char = 'X', Minus = '-';**

**float Float = 2.5;**

**cout << Char << Minus << Float;**

**char Char = 'X', Minus = '-';**

**float Float = 2.5;**

**cout << Char << Minus << Float;**

1.7.1.7 Connecting with th

cout is able to recognize the actual type of its element even when it is an effect of a conversion.

We’ll discuss the conversions later, but for now we only want to mention that a phrase written as:

(newtype)expr

changes the type of the expr expression into the newtype type.

What it means is that we can see the ASCII code of any character stored within a char variable and vice versa, or see a character whose ASCII code is placed inside an int variable.

The snippet in the editor outputs the following text to the screen:

X 88 88 X

**char Char = 'X';**

**char Char = 'X';**

**int Int = Char;**

**cout << Char << " " << (int)Char << " " << Int << " " << (char)Int;**

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

When we present many different results one by one in the same line of text, it doesn’t look nice and you won’t want to look at it. One line is okay, but a thousand lines written like that will make you go blind.

We can break the line in two ways. First, we can use one of the control characters called “newline” and coded as \n (note: we use two characters to write it down but the compiler sees it as one character - don’t let it fool you).

We can achieve exactly the same effect by using a manipulator called endl (as “end line”).

The snippet below:

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

illustrates both methods and causes the console to display the following three lines of text:

1

2

3

**Input**

Of course, equally important as data output is **data input**. Actually, it’s difficult to imagine any non-trivial program that doesn’t require any data from the user, although you can do the following:

* encode all the data needed inside the source code (which is sometimes called **hard coding**)
* when you need to repeat the execution of the program with other data, you just modify the program, compile it and run it again.

**This isn’t a particularly convenient solution**. It’s far better to get the information from the user, transfer it to the program, and then use it for calculations. So how does a C++ language program get data from a human and store it in variables?

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**

Of course, equally important as data output is **data input**. Actually, it’s difficult to imagine any non-trivial program that doesn’t require any data from the user, although you can do the following:

* encode all the data needed inside the source code (which is sometimes called **hard coding**)
* when you need to repeat the execution of the program with other data, you just modify the program, compile it and run it again.

**This isn’t a particularly convenient solution**. It’s far better to get the information from the user, transfer it to the program, and then use it for calculations. So how does a C++ language program get data from a human and store it in variables?

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.

1.7.1.10 Connecting with the

Imagine that we want to ask the user about the maximum number of sheep we want to count before the programmer falls asleep.

The user enters the value from the keyboard and the program stores it in a specified variable (max\_sheep). That statement looks like this:

cin >> max\_sheep;

You probably see the similarity to emitting data using cout: we have a stream, we have an operator and we have a variable.

At this point the similarities end and the differences begin. First, the argument for cout may not be a variable. It can also be an expression.

Take a look:

cout << 2 \* i;

Here we want the doubled value of i to be printed – and that’s feasible. Using an input stream, we need to **explicitly specify the variable** that can store the data entered by the user.

Now we’ll show you a simple but complete program that does the following:

* prompts the user to enter a single integer value,
* squares it,
* prints the result with an appropriate comment.

Analysing this program shouldn’t be a problem for you... we hope!

**#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;**

}

So you prefer square roots to squares? No problem, but we need to remember two things: first, there’s **no such thing as a square root operator**; and second, that **square roots of negative numbers do not exist**.

We can solve the first issue by finding a function that knows how to compute the root. This type of function does exist and takes the argument of the float type.

The result is also a float (of course - the square of an integer is still an integer, but the root of any number is not always an integer, like the square root of 2).

The function we’re going to use is called sqrtf (square root float) and needs exactly one argument. Oh, one more thing - to use this function you need to include a header file named cmath.

We need to deal with negative numbers as well. If you’re careless and enter a negative number, the program will just ignore you and your input completely. It may not be polite, but at least it won’t attempt to bend the rules of mathematics. Whether we see the result or not will be decided by the conditional statement.

Now it’s time to focus on the use of floating point data and the sqrtf function.

Complete program is in the editor.

**#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;**

**}**

**Congratulations! You have completed Module 1.**

Well done! You've reached the end of Module 1 and completed a major milestone in your C++ programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 1:

* the difference between machine and high-level languages;
* the machine code and compilation;
* variables, integers, characters;
* comments;
* the basics of flow control;
* dealing with streams and basic I/O operations;
* writing simple programs.

You are now ready to take the module quiz and attempt the final challenge: Module 1 Test, which will help you gauge what you've learned so far.

**C++ Essentials: Module 2**

**In this module, you will learn about:**

* how to control the flow of the program;
* more data types;
* conditional instructions: if, else, switch;
* loops and controlling the loop execution;
* logic, bitwise and arithmetic operators;
* vectors, multidimensional arrays;
* declaring and initializing structures.

**The conditional statement – more conditional than before**

We concluded our last discussion on conditional statements with a promise that we would introduce a more complex and flexible form soon. We started our tale with a simple phrase which read: *If the weather is good, we will go for a walk*.

Note – there is not a word about what will happen if it rains cats and dogs. We only know that we certainly won’t go outdoors, but what we would do instead isn’t mentioned. We may want to plan something in case of bad weather, too.

We can say, for example: *If the weather is good we will go for a walk, otherwise we will go to a theater*. This sentence makes our plans more resistant to the whims of fate – we know what we’ll do if the conditions are met and we know what we we’ll do if not everything goes our way. In other words, we have a plan “B”.

Luckily, the “C” language allows us to express alternative plans. We do this with a second, slightly more complex form of the conditional statement – here it is:

if (true\_or\_false\_condition) perform\_if\_condition\_true; else perform\_if\_condition\_false;

Thus, we have a new word: else – this is a keyword (*reserved word*). A statement which begins with else tells us what to do if the **condition specified for the if is not met**.

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” (its value is equal to zero) the perform\_if\_condition\_false is executed and the conditional statement comes to an end;

if (the\_weather\_is\_good)

go\_for\_a\_walk();

else

go\_to\_the\_theater();

have\_lunch();

By using this form of the conditional statement we can describe our plans as follows:

As well as in other simplified forms of this instruction, like the ones we’ve already encountered, both if and else may contain **only one statement**.

If you’re going to write more than one instruction, you have to use a block, as in the example below:

if (the\_weather\_is\_good) {

go\_for\_a\_walk();

have\_fun();

} else {

go\_to\_the\_theater();

enjoy\_the\_movie();

}

have\_lunch()

Now it’s time to discuss two special cases of the conditional statement. First, think about when the instruction placed after if is another if.

Listen to what we have planned for this Sunday: *If the weather is fine, we’ll go for a walk. If we find a nice restaurant, we’ll have lunch there. Otherwise, we’ll eat a sandwich. If the weather is poor, we'll go to the theater. If there are no tickets, we’ll go shopping in the nearest mall.*

Complicated, right? Let’s write the same thing in C++ language. Consider the code below carefully:

if (the\_weather\_is\_good)

if (nice\_restaurant\_is\_found)

have\_lunch();

else

eat\_a\_sandwich();

else

if (tickets\_are\_available)

go\_to\_the\_theater();

else

go\_shopping();

Let’s examine two important points:

* such a use of the if statement is known as **nesting**; remember that every *else refers to the closest former if* which did not match any other else; we need to know this to determine how the ifs and elses pair up;
* consider how the **indentation** improves readability and emphasizes the nesting of inner conditional statements.

This second special case is somewhat similar to nesting, but with an important difference. Again, we’re going to change our plans and express them as follows: “If the weather is fine, we'll go for walk, otherwise if we get tickets, we’ll go to the theatre, otherwise if there are free tables at the restaurant, we’ll go for lunch. If everything fails, we’ll return home and play chess”.

Can you see how many alternatives we’ve listed here? Let’s write the same scenario in C++ language:

if (the\_weather\_is\_good)

go\_for\_a\_walk();

else if (tickets\_are\_available)

go\_to\_the\_theater();

else if (table\_is\_available)

go\_for\_lunch();

else

play\_chess\_at\_home();

This way of assembling subsequent if statements is called a **cascade**. Notice how the **indentation** improves the readability of the code.

Now, let our minds work out all that we’ve told them about conditional statements, while we pay attention to our old friends: types int, char and float. We’re going to meet their siblings.

**Not only the int is an int**

It would seem that the developer’s life would be organized well enough if they had type int to operate with integers, type char to manipulate characters and type float for floating-point calculations.

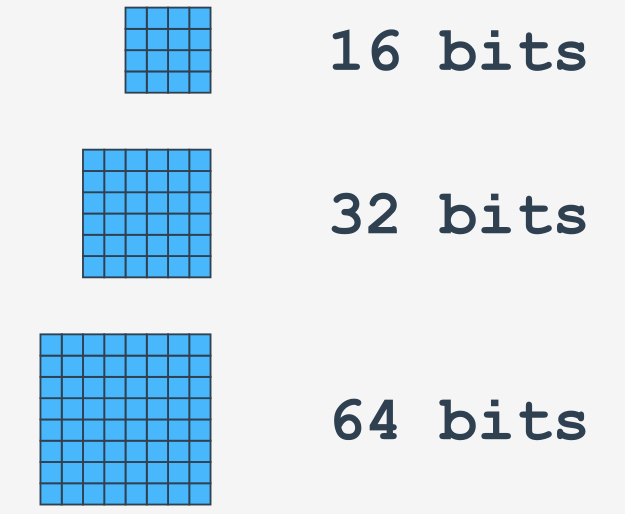
However, this practice has shown that such a narrow repertoire of types may raise some problems.

Most of the computers currently in use store ints using **32 bits** (4 bytes); this means that we can operate the ints within the range of [-2147483648 .. 2147483647]. It may happen that:

* we don’t need such big values; if we count sheep, it’s unlikely that we’ll need to count two billion of them, so why waste the majority of these 32 bits if we don’t need them;
* we need much larger values; for example, we intend to calculate the exact number of humans living on Earth; in this case we need more than 32 bits to represent that number;
* this brings us to another observation – after all, the number of inhabitants on Earth will never be a negative number; it seems like a real waste that up to half of the permissible range will never be used.

For these reasons, the C++ language provides some methods for defining precisely how we intend to store large/small numbers. This allows the compiler to allocate memory, either smaller than usual (e.g. 16 bits instead of 32) or larger than usual (e.g. 64 bits instead of 32). We can also declare that we guarantee that the value stored in the variable will be **non-negative**.

In this case the width of the variable’s **range does not change**, but is **shifted** toward the positive numbers. This means that instead of the range of -2,147,483,648 .. 2,147,483,647 we get the range of 0 .. 4294967295.



To specify our memory requirements, we can use some additional keywords called *modifiers*:

* *long* – is used to declare that we need a wider range of ints than the standard one;
* *short* – is used to determine 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; this might surprise you, but we can use this modifier together with the type char; we’ll explain it soon.

Let’s look at some examples.

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]):

short int counter;

The word int may be **omitted** as all the declarations are considered to be specifying int by default, like this:

short Counter;

The ants variable will occupy more bits than the standard int (e.g. 64 bits, so it can be used to store numbers from the range of [-9223372036854775808 .. 9223372036854775807] – can you read such huge numbers?

long int ants;

Note – we can again **omit** the word int:

long ants;

If we come to the conclusion that a variable will **never be a negative value**, we can use the **unsigned** modifier:

unsigned int positive;

Of course, we can omit the int as usual:

unsigned positive;

We can also mix some of the modifiers together – take a look:

unsigned long int big\_number;

We can remove the word int and the declaration will preserve its meaning:

unsigned long big\_number;

A more modest example is here:

unsigned short int lambs;

Its equivalent form is:

unsigned short lambs;

The *long* and *short* modifiers **must not be used** in conjunction with the type char (why?) and (for obvious reasons) must not be used simultaneously (одночасно) in a single declaration. But there’s nothing preventing us from using the *unsigned* modifier with a variable of type char. What do we get from this declaration?

Don’t forget that we’re not allowed to omit the word char. Most of the compilers currently in use assume that the chars are stored using 8 bits (1 byte). That may be enough to store a small value such as the number of months or even the day of the month.

If we treat the char variable as a signed integer number, its range would be [-128 .. 127]. If we don’t need any signed value (as in the example below), its range shifts to [0 .. 255]. This may be sufficient for many applications and may also result in significant savings in memory usage.

unsigned char little\_counter;

But we need to add an important remark. So far we’ve used integer literals, assuming that all of them are of type int. This is generally the case, but there are some cases when the compiler recognizes literals of type long. This will happen if:

* a literal value goes **beyond the acceptable** range of type int;
* **letter L or l is appended** to the literal, such as 0L or 1981l – both of these literals are of type long.

**Another float type**

The *short* modifier cannot be used alongside the float, but we may use the *long* modifier here. It’s assumed that type long float is a synonym for another type named double. The variables of type double may differ from the variables of type float, not only in **range**, but also in **accuracy**.

What does this mean? The data stored in a floating-point variable has **finite precision** – in other words, only a certain number of digits are **precisely stored** in the variable.

For example, we expect that the value:

1111111111111111111.111111111111111111111

will be stored by a specific type of computer as:

1111111131851653120.000000

We 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 bits long) guarantees that the variable will save a more significant number of digits – about **15-17**. This is where the name double comes from – its accuracy is **doubled** compared to float.

**Floats and their traits**

We told you some time ago that computer addition is **not always commutative**. Do you know why? Imagine that you have to add a large number of floating-point values – some of them are very large, some very small (close to zero). If a very small float value is added to another that’s very large, the result can be quite surprising.

Let’s go back to the previous example – we’ll assume that our computer only saves 8 precise digits of any float. If we add these two floats, we’ll probably get:

11111110656.000000

as the result. The lower value simply vanished without a trace.

We can’t avoid these effects when we add/subtract the numbers of type float (and of double as well, because they’re also affected by this issue). The phenomenon described here is what we call a **numerical anomaly**.

**In memory of George Boole**

**George Boole** (1815 –1864) was an English mathematician, philosopher and logician and we’re talking about him for a very important reason. One of his most important achievements was **algebraic logic**, referred to by Boole himself as “the laws of thought”. Algebraic logic does not operate on numbers but only on **two truth values**, and doesn’t use standard arithmetic operations like addition and multiplication, but **conjunction, disjunction** and **negation**.

Taking into account the fact that virtually all modern computers are built using Boole’s theorems, we can say without exaggeration that Boole was actually one of the founders of IT.

There is a type in the C++ language whose name commemorates George Boole – **the type bool**.

It’s a very intriguing type. Variables of this type are able to store only two distinct values: true and false. Note: all these new words (bool, true and false) are keywords. Don’t forget that.

Take a look at the example below:

bool developer\_is\_hungry = false;

We’ve declared a variable there. Neither its name nor its value requires additional comments. There are many contexts where this variable may be useful. One of the most spectacular is the following:

if(developer\_is\_hungry) {

have\_lunch();

developer\_is\_hungry = !developer\_is\_hungry;

}

The exclamation mark we’ve used in the assignment is a negation operator. It’s a **unary prefix operator** that changes the logical value of its arguments: because of the operator, true becomes false and vice versa. As you see, having lunch changes the logical state of one of the most important factors of a programmer’s well-being.

To be honest, the bool type is only a very special variant of the int type. It’s very short (variables of this type occupy only 8 bits, which is still too much, because one bit would be enough). It behaves like an int inside expressions (**true is equivalent to 1 while false is equivalent to 0**).

We’ll return to this type and its values soon, and to George Boole’s algebra and logical operators, too.

**Two simple programs**

Now we’re going to show you some simple but complete programs. We won’t explain them in detail, because we think the comments inside the code are sufficient guides.

All these programs solve the same problem – they find the largest of several numbers and print it out.

Let’s start with the simplest case – how to **identify the larger of two numbers**.

**/\* 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;**

**}**

Now let's try to find **the largest of three numbers**. We find the larger of the first two and compare it with the third one. Here we go.

**/\* 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;**

**}**

**Some simple programs**

By this point, you should be able to write a program that finds the largest of four, five, six or even ten numbers. You already know the scheme, so the extension of the program doesn’t need to be particularly complex.

But what happens if we ask you to write a program that finds the largest of a hundred of numbers? Can you imagine the code?

You’d need hundreds of declarations of type int variables. If you think you can cope with that, then try to imagine searching for the greatest of a million numbers;

Imagine the code that contains 99 conditional statements and a hundred cin statements.

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 (it could be neither compiled nor executed), but is formalized, concise and readable. We call this **pseudo-code**.

There is an example of pseudo-code below. Take a look at it. What’s going on?

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

First, we can simplify our program if, at the very beginning of the code, we assign the variable max with a value which will be smaller than any of the numbers entered. We’ll use *-999999999* for this purpose.

Second, we assume that our algorithm doesn’t know in advance how many numbers will be delivered to the program. We expect that the user will enter as many numbers as she/he wants – the algorithm will work equally well with one hundred or one thousand numbers. How do we do that? Well, we make a deal with the user: when the value -1 is entered, it will be a sign that there is no more data and the program should end its work. Otherwise, if the entered value is not equal to - 1, the program will read another number and so on.

The trick is based on the assumption that any part of the code **can be performed more than once** – in fact, as many times as you need.

Performing a certain part of the code more than once is called a loop. You probably already know what a loop is. See, steps 2 through 5 make a loop. Can we use a similar structure in the program written in the C++ language? Yes, we can. And we’re going to tell all you about it soon.

**The “while” loop**

We want to ask you a strange question: how long do you usually take to wash your hands? Don’t think about it, just answer. Well, when your hands are very dirty, you wash them for a very long time. Otherwise it takes less time. Do you agree with this statement:

while my hands are dirty

I am washing my hands;

Note that this also implies that if our hands are clean, we won’t wash them at all.

So now you've learnt one of the **loops** available in the C++ language. In general, the loop manifests itself as follows:

while(conditional\_expression)

statement;

If you think it looks similar to the if instruction, you’re quite right. Indeed, there’s only one syntactic difference: we replaced the word “if” with the word “while”.

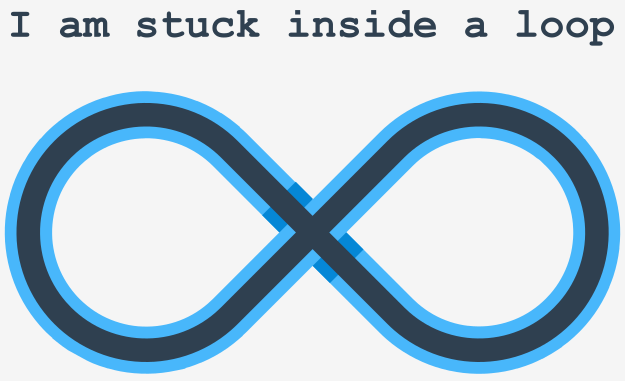
The semantic difference is more important: when the condition is met, if performs its statements only once; while repeats the execution as long as the condition evaluates to “true”.

Let’s make a few observations:

* if you want while to execute **more than one** statement, you must (like with the if statement) use a block – take a look at the code in the editor;
* an instruction or instructions executed inside the loop are called the **loop's body**;
* if the condition is “false” (equal to zero) as early as when it’s tested for the first time, the **body is not executed** even once (note the analogy of not having to wash your hands if they’re not dirty);
* the body should be able to change the condition value, because if the condition is true at the beginning, the body might **run continuously to infinity** (notice that washing changes the state of impurity).

**while(conditional\_expression) {  
statement\_1;  
statement\_2;  
:  
:  
statement\_n;  
}**

Here is an example of a loop that is not able to finish its execution. This loop will infinitely print I am stuck inside a loop on the screen.



**while(1) {**

**cout << "I am stuck inside a loop" << endl;**

**}**

Let's go back to the algorithm we talked about recently. We’re going to show you how to use the newly learnt loop. By the way, we want to introduce you to one more novelty. So far we’ve declared variables in one place and assigned values to them in another.

We can combine these two steps by **declaring the variable and assigning the value at the same time**. This is done by adding the = sign followed by an expression whose value is assigned to the variable at the time of its creation.

For example, if you need a variable that needs the value of zero then you can do this (code in the editor).

int variable = 0; As usual on such occasions, a new word arrives into our vocabulary: the part of the declaration placed on the right side of the = sign is called an **initiator**.

The initiator you saw before was a literal, but you can also use more complex expressions, like the ones in the editor.

We’ll be using initiators often. They’re extremely convenient and quite useful.

float PI = 3.1415;

double PI2 = 2.0 \* PI;

Analyze this program carefully. Locate the loop’s body and find out how the **body is exited**.

See how the above code implements the algorithm we made earlier.

**#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;**

**}**

This program counts odd and even numbers coming from the keyboard. Have a look at it.

Certain snippets can be simplified without changing the program’s behavior. Take a look at the next slide.

**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;**

**}**

Try to recall how the “C++” language interprets the truth of a condition and note that these two forms are equivalent.

while(number !=0) {...}

while(number) {...}

The condition that checks if a number is odd can be coded in like this:

if(number % 2 ==1)...

if(number % 2)...

We guess that nothing surprises you, right? But there are two things that we can write more compactly. First, the condition of the while loop.

int main(void) {

int counter = 5;

while(counter != 0) {

cout << "I am an awesome program" << endl;

counter--;

}

return 0;

}

Another change requires us to have some knowledge of how the post-decrement works. We’ll use it to compact our program once again.

int main(void) {

int counter = 5;

while(counter) {

cout << "I am an awesome program" << endl;

counter--;

}

return 0;

}

We’re convinced that this is the simplest form of this program, but you can challenge us if you dare.

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

We already know that the while loop has two important features:

* it checks the condition **before** entering the body,
* the body will not be entered if the condition is false.

These two properties can often cause unnecessary complications. For this reason, there’s another loop in the C++ language which **acts like a mirror image of the while loop**.

We say this because in that 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.

This loop is called the do loop. Its simplified syntax is listed in the editor.

If you want to execute a body containing more than one statement, you need to use a block.

**do  
statement;  
while(condition);  
do {  
statement\_1;  
statement\_2;  
:  
:  
statement\_n;  
} while(condition);**

Let’s return to the program that searches for the largest number. Firstly, we will use the “do” loop instead of “while” for teaching purposes. Secondly, we remove the vulnerability involved in the excessive trust in the user’s good will. Our new program won’t be misled by entering the value of -1 as the first number. Look at the editor. Here's our code.

Take a look. We used the counter variable to count the numbers entered so we can instruct the user that we cannot search for the greatest number if no number is given.

As we have to **read at least one number**, it makes sense to use the do loop. We use this approach in the program.

**#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;**

**}**

10

11

-1

The largest number is 11

**“for” - the last loop**

The last available kind of loop in C++ language comes from the fact that sometimes it’s more important to **count the “turns” of the loop** than to check the conditions.

Imagine that a loop's body needs to be executed exactly one hundred times. If you want to use the while loop for that purpose, it may look something like this:

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

We can distinguish three independent elements there:

* the initialization of the counter
* the checking of the condition
* the modification of the counter

It’s possible to create something like a generalized scheme for these kinds of loops, here it is:

initialization; while (checking) { /\* the body goes here \*/ modifying; }

This way of coding the loop is very common, so there’s a special, brief way of writing it in “C++” language.

Below we’ve gathered all three decisive parts together. The loop is clear and easy to understand. Its name is for.

for(initialization; checking; modifying) { /\* the body goes here \*/ }

The for loop can take the form shown in the editor.

The variable used for counting the loop's turns is often called a **control variable**.

Notice, that the control variable doesn’t have to be declared before it’s used within the for loop. It can be declared inside the loop, but in this case it’ll be available during and only during the loop execution.

Look at an example in the editor.

**for(i = 0; i < 100; i++) {**

**/\* the body goes here \*/**

**}**

The for loop has an interesting singularity. If we omit any of its three components, it is presumed that there is a 1 there instead.

One of the consequences of this is that a loop written in this way is an infinite loop (do you know why?).

Well, the conditional expression is not there, so it is automatically assumed to be true. And because the condition never becomes false, the loop becomes infinite.

**for( ; ; ) {**

**/\* the body goes here \*/**

**}**

Let’s look at a short program whose task is to write some of the first powers of 2.

The exp variable is used as a control variable for the loop and indicates the current value of the exponent. The exponentiation itself is replaced by multiplying by 2. Since 20 is equal to 1, then 2 ∙ 1 is equal to 21, 2 ∙ 21 is equal to 22 and so on.

Answer this question: what is the greatest exponent for which our program still prints the result?

**#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;**

**}**

2 to the power of 12 is 4096

2 to the power of 13 is 8192

2 to the power of 14 is 16384

2 to the power of 15 is 32768

# break and continue – the loop's spices

So far, we’ve treated the body of the loop as an **indivisible and inseparable** sequence of instructions that are performed completely at every turn of the loop. However, as a developer, you could be faced with the following choices:

* it appears that it is unnecessary to continue the loop as a whole; we should stop executing the loop's body and go further;
* it appears that we need to start the condition testing without completing the execution of the current turn.

The C++ language provides us with two special instructions to implement both these tasks. Let's say for the sake of accuracy that their existence in the language is not necessary - an experienced programmer can code any algorithm without these instructions.

The famous Dutch computer scientist [Edsger Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra) proved it in 1965. These additions, which don't improve the language's expressive power but only simplify the developer's work, are sometimes called **syntactic candies**.

These two instructions are:

* 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;
* 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.

Both these words are keywords.

Now let’s look at two simple examples. We’ll return to our program that recognizes the largest of the numbers entered. We’ll convert it twice, using both instructions. Analyze the code and judge whether and how you would use any of them.

You can see the break variant in the editor.

Note that the only way to exit the body is to perform the break, as the loop itself is infinite (for (;;)).

**#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;  
}**

100

10

8

-1

The largest number is 100

**break and continue – the loop's spices**

And now the continue variant.

**#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;**

**}**

52

88

-1

The largest number is 88

**Computers and their logic**

Have you noticed that the conditions we’ve used so far are very simple, quite primitive, in fact? The conditions we use in real life are much more complex. Let's look at this following sentence:

*If we have some free time, and the weather is good, we will go for a walk.*

We have used the conjunction “and”, which means that going for a walk depends on the **simultaneous fulfillment of the two conditions**. (одночасне виконання двох умов)

In the language of logic, connecting conditions like this is called a conjunction. And now another example:

*If you are in the mall or I am in the mall, one of us will buy a gift for Mom.*

The appearance of the word “or” means that the purchase depends on at least one of these conditions. In logic terms, this is called a **disjunction**.

So clearly, the C++ language needs to have operators to build conjunctions and disjunctions. Without them, the expressive power of the language would be substantially weakened. They are called **logical operators**.

**Pride && Prejudice**

The logical conjunction operator in the C++ language is a digraph && (*ampersand ampersand*).

This is a binary operator with a priority lower than the comparison operators. It allows us to code complex conditions without the use of parentheses like this one:

counter > 0 && value == 100

The result provided by the && operator can be determined on the basis of the **truth table**. If we consider the conjunction of:

left && right

the set of possible values of arguments and corresponding values of the conjunction looks as follows:

|  |  |  |
| --- | --- | --- |
| left | right | left && right |
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

# To be || not to be

The disjunction operator is the digraph || (*bar bar*). It’s a binary operator with a lower priority than && (just like “+” compared to “\*”). Its truth table looks as follows:

|  |  |  |
| --- | --- | --- |
| left | right | left || right |
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

In addition, there’s another operator that can be used to construct conditions. It’s a unary operator performing a **logical negation**. Its operation is simple: it turns true into false and false into true. This operator is written as a single character ! (*exclamation mark*) and its priority is very high: the same as the increment and decrement operators.

Its truth table is truly simple:

|  |  |
| --- | --- |
| arg | !arg |
| false | true |
| true | false |

# Some logical expressions

Note that the following conditions are **equivalent, respectively**:

variable > 0 !(variable <= 0)

variable != 0 !(variable == 0)

**You may remember** [**De Morgan's laws**](https://en.wikipedia.org/wiki/De_Morgan's_laws) **from school. They say that:**

*The negation of a conjunction is the disjunction of the negations. The negation of a disjunction is the conjunction of the negations.*

Let's try to write the same thing using the “C” language:

!(p && q) == !p || !q

!(p || q) == !p && !q

Note how the parentheses have been used to code the expressions.

We should add that none of the previous two-argument operators can be used in the abbreviated form known as op=. This exception is worth remembering.

**How to deal with single bits**

Logical operators take their arguments as a whole, **regardless of how many bits they contain**. The operators are aware only of the value: 0 (when all the bits are reset) means “false”; not 0 (when at least one bit is set) means “true”. The result of their operations is one of the values: 0 or 1. This means that the following snippet:

bool i, j;

j = !!i;

will assign a value of 1 to the j variable if i is not zero; otherwise, it will be 0 (why?).

However, there are four operators that allow you to manipulate single bits of data. We call them **bitwise operators**. They cover all the operations we mentioned before in the logical context and one additional operator. This is the *xor (exclusive or)* and is denoted as ^ (caret). Here are all of them:

|  |  |
| --- | --- |
| & (ampersand) | bitwise conjunction |
| | (bar) | bitwise disjunction |
| ~ (tilde) | bitwise negation |
| ^ (caret) | bitwise exclusive or |

Let's make it easier:

* & requires exactly two “1s” to provide “1” as the result
* | requires at least one “1” to provide “1” as the result
* ^ requires only one “1” to provide “1” as the result
* ~ (is one argument and) requires “0” to provide “1” as the result

But take note: arguments of these operators **must be integers** (int as well as long, short or char); we must not use floats here.

The difference in the operation of the logical and bit operators is important: the logical operators do not penetrate into the bit level of its argument. They’re only interested in the final integer value.

Bitwise operators are stricter: **they deal with every bit separately**. If we assume that the int variable occupies 32 bits, you can imagine the bitwise operation as a 32-fold evaluation of the logical operator for each pair of bits of the arguments. Obviously, this analogy is somewhat imperfect, as in the real world all these 32 operations are performed at the same time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| left | right | left&right | left|right | left^right |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

|  |  |
| --- | --- |
| arg | ~arg |
| 0 | 1 |
| 1 | 0 |

Let’s have a look at an example of the difference in operation between logical and bit operations. Let’s assume that the following declaration has been performed:

int i = 15, j = 22;

If we assume that the ints are stored with 32 bits, the bitwise image of the two variables will be as follows:

i: 00000000000000000000000000001111 j: 00000000000000000000000000010110

The declaration is given:

int log = i && j;

We’re dealing with a logical conjunction. Let’s trace the course of the calculations. Both variables i and j are not zeros so will be deemed to represent “true”.

If we look at the truth table for the && operator, we can see that the result will be “true” and that it’s an integer equal to 1. This means that the bitwise image of the log variable is as follows:

log:00000000000000000000000000000001

Now the bitwise operation – here it is:

int bit = i & j;

The & operator will operate with each pair of corresponding bits separately, producing the values of the relevant bits of the result. Therefore the result is this:

bit:00000000000000000000000000000110

These bits correspond to the integer value of 6.

Let's try the **negation operators** now. First the logical one:

int logneg = !i;

The logneg variable will be set to 0, so its image will consist of zeros only.

The result:

logneg:00000000000000000000000000000000

The bitwise negation goes here:

int bitneg = ~i;

The result:

bitneg:11111111111111111111111111110000

It may surprise you to learn that the bitneg variable value is -16. Strange? No, not at all!

If it surprises you, try to spend some time looking into the secrets of the binary numeral system and the rules governing so-called **two's complement numbers**. It makes for good bedtime reading.

We can use each of the previous two-argument operators in their abbreviated forms. These are the examples of equivalent notations:

|  |  |
| --- | --- |
| x = x & y; | x &= y; |
| x = x | y; | x |= y; |
| x = x ˜ y; | x ˜= y; |

We'll now show what you can use bitwise operators for.

Imagine that you have to write an important piece of an operating system. You’ve been told that you’re to use a variable declared in the following way:

int flag\_register;

The variable stores the information about various aspects of system operation. Each bit of the variable stores one yes/no value.

You’ve also been told that only one of these bits is yours – bit number three (remember that bits are numbered from 0 and bit number 0 is the lowest one, while the highest is number 31).

The remaining bits are not allowed to change because they’re intended to store other data.

Here's your bit marked with the letter “x”:

0000000000000000000000000000x000

You may face the following tasks:

**#1:**

Check the state of your bit – you want to find out the value of your bit; comparing the whole variable to zero will not do anything, because the remaining bits can have completely unpredictable values, but we can use the following conjunction property:

x & 1 = x x & 0 = 0

If we apply the & operation to the flag\_register variable along with the following bit image:

00000000000000000000000000001000

(note the "1" at your bit's position) we obtain one of the following bit strings as a result

00000000000000000000000000001000

if your bit was set to “1”

00000000000000000000000000000000

if your bit was reset to “0”.

A sequence of zeros and ones whose task is to grab the value or to change the selected bits is called **a bitmask**. Let’s try to build a bitmask to **detect the state** of your bit. It should point to the **third** bit. That bit has the weight of 23 = 8. A suitable mask could be created by the following declaration:

int the\_mask = 8;

We can also make a sequence of instructions depending on the state of your bit – here it is:

if(flag\_register & the\_mask) {

/\* my bit is set \*/

} else {

/\* my bit is reset \*/

}

**#2:**

**Reset your bit** – you assign a zero to the bit while all other bits remain unchanged; we’ll use the same property of the conjunction as before, but we’ll use a slightly different mask – just like this:

1111111111111111111111111111110111

Note that the mask was created as a result of the negation of all bits of the\_mask variable.

Resetting the bit is simple and looks like these (choose the one you like most):

flag\_register = flag\_register & ~the\_mask; flag\_register &= ~the\_mask;

**#3:**

**Set your bit** – you assign a “one” to your bit while all the remaining bits must remain unchanged; we’ll use the following disjunction's property:

x | 1 = 1

x | 0 = x

We’re ready to set your bit with one of the following instructions (look at the editor once again).

flag\_register = flag\_register | the\_mask; flag\_register |= the\_mask;

**#4:**

**Negate your bit** – you replace a “one” with a “zero” and a “zero” with a “one”. We’ll use an interesting property of the xor operator:

x ^ 1 = !x

x ^ 0 = x

Now let's negate your bit with the instructions:

flag\_register = flag\_register ^ the\_mask; flag\_register ^= the\_mask;

The C++ language offers us yet another operation relating to single bits: **shifting**. It applies only to integer values and you can’t use it with floats as arguments. You use this operation unconsciously all the time. How do you multiply any number by 10? Take a look:

12345 ∙ 10 = 123450

As you can see, multiplying by ten is in fact a shift of all the digits to the left and filling the resulting gap with a “0”. Division by 10? Let's look:

12340 ÷ 10 = 1234

Dividing by 10 is nothing more than shifting the digits to the right.

The same kind of operation is performed by the computer, but with one difference: as 2 is the base for binary numbers (not 10), shifting a value one bit to the left corresponds to multiplying it by 2; respectively, shifting one bit to the right is like dividing by 2 (notice that the right-most bit is lost).

Bit shifting can be:

* logical, if all the bits of the variable are shifted; shifting takes place when you apply it to the unsigned integers;
* arithmetic, if the shift omits the **sign bit** – in two's complement notation, the role of the **sign bit** is played by the **highest bit** of a variable; if it’s equal to "1", the value is treated as a negative; this means than the arithmetic shift cannot change the sign of the shifted value.

The shift operators in the C++ language are a pair of digraphs, << and >>, clearly suggesting in which direction the shift will act. The left argument of these operators is the integer value whose bits are shifted. The right argument determines the size of the shift. This shows that this operation is certainly not commutative.

The priority of these operators is very high. You'll see them in the updated table of priorities which we’ll show you at the end of this section.

value << bits  
value >> bits

Let’s assume the following declarations exist:

int signed = -8, var\_s;

unsigned unsigned = 6, var\_u;

Take a look at the shifts in the editor.

Both operators can be used in the shortcut form as below:

signed >>= 1; /\* division by 2 \*/

unsigned <<= 1; /\* multiplication by 2 \*/

**/\* equivalent to division by 2 –> var\_s == -4 \*/  
var\_s = signed >> 1;  
/\* equivalent to multiplication by 4 –> var\_s == -32 \*/  
var\_s = signed << 2;  
/\* equivalent to division by 4 –> var\_u == 1 \*/  
var\_u = unsigned >> 2;  
/\* equivalent to multiplication by 2 –> var\_u == 12 \*/  
var\_u = unsigned << 1;**

And here’s the updated priority table, containing all the operators introduced in this section.

|  |  |
| --- | --- |
| ! ~ (type) ++ -- + - | unary |
| \* / % |  |
| + - | binary |
| << >> |  |
| < <= > >= |  |
| == != |  |
| & |  |
| | |  |
| && |  |
| || |  |
| = += -= \*= /= %= &= ^= | >>= <<= |  |

# Case and switch vs. if

As we already know, an *if-cascade* is a name for a construction of code where many if instructions are **placed consecutively one after another**, as in the example in the editor.

Of course, there are no obstacles to using and maintaining a code like this, but there are a few disadvantages that may be discouraging. The longer the cascade, the harder it is to read and understand what it’s indented for.

Amending the cascade is also hard: it's hard to add a new branch into it and it's hard to remove any previously created branch.

The C++ language offers us a way to make these selections easier. By the way, this is only more syntax candy. You can manage without it, but don't hesitate to use it when your ifs start growing extensively.

**if(i == 1)   
puts("Only one?");  
else if(i == 2)  
puts("I want more");  
else if(i == 3)  
puts("Not bad");  
else  
puts("OK");**

Let's take a look at the snippet in the editor. It's an example of how to replace an if cascade with a specialized instruction. Note that the words switch and case are keywords.

The new instruction is called switch and it is, in fact, a switch. So how does it work?

First, the value of the expression enclosed inside the parenthesis after the switch keyword is **evaluated** (this is sometimes called a *switching expression*).

Then the block is searched in order to **find a literal** preceded by the case keyword which is equal to the value of the expression.

When this case is found, the instructions behind the colon are **executed**. If there’s a break among them, the execution of the switch statement is terminated, otherwise, all instructions are executed until the end of the block is reached or another break is met.

What happens if the switching expression has a value that does not occur in any of the cases? The answer is simple: nothing will happen – none of the branches of the switch statement are executed.

**switch(i) {**

**case 1: cout << "Only one?" << endl; break;**

**case 2: cout << "I want more" << endl; break;**

**case 3: cout << "Not bad" << endl; break;**

**case 4: cout << "OK"<< endl;**

**}**

Let's modify the requirements. We’ll assume now that our program is satisfied (it says “OK”) if the i variable is equal to 4 or to 3. Does this mean that we have to create two branches for both possibilities?

Fortunately not. We’re allowed to place more than one case in front of any branch, like the one in the editor.

**switch(i) {**

**case 1: cout << "Only one?" << endl; break;**

**case 2: cout << "I want more" << endl; break;**

**case 3:**

**case 4: cout << "OK" << endl;**

**}**

We can also assume that our program does not have an opinion when i values are different from the ones specified so far and we want the program to express it in a clear form. Have we made a million new cases covering the entire int type's range?

No. We can use a generalized case that covers all these events which haven’t been enumerated in the previous cases. Take a look at the code in the editor.

Note that default is a keyword too.

Don't forget to use the break. Leaving out this keyword is one of **the most common mistakes** developers make (not only at the beginning of their careers).

Simple, right? And how elegant.

But now a few more important remarks to note:

* the value after the case **must not be an expression** containing variables or any other entities whose values aren't known at compilation time;
* the case branches are **scanned in the same order** in which they are specified in the program; this means that the most common selections should be placed first (in fact, this could make your program a little faster in some cases).

Now we say goodbye to our switch statement; it's time to take up an important challenge – we’re going to discuss **arrays**.

**switch(i) {  
case 1: cout << "Only one?" << endl; break;  
case 2: cout << "I want more" << endl; break;  
case 3:   
case 4: cout << "OK" << endl; break;  
default: cout << "Don't care" << endl;  
}**

# Arrays – why?

There may come a time when we have to read, store, process, and finally, print dozens, maybe hundreds, perhaps even thousands of numbers. What then? Must we declare a separate variable for each value? Will we have to spend long hours writing statements like the one in the editor.

If you think this is a simple task, then we suggest that you take a piece of paper and write a program that reads five numbers of type int and prints them in order from the smallest to the largest (NB this is called *sorting*). We don’t think your piece of paper will be big enough for the task.

So far, we’ve declared variables that can store exactly one given value at a time. These variables are called *scalars*, analogous to mathematical terms. All variables we’ve used so far have actually been scalars.

Think of how convenient it would be if we could declare a variable that can store more than one value. For example, 100 or 1000 or even 10,000 variables. It would still be one and the same variable, but very wide and extensive. Does that sound appealing? Perhaps, but how would it handle all these different values? How would it choose just the one we need?

Should we just number them? And we'll say: *give me the value number 2; assign the value number 15; increase the value number 10000.*

Maybe... what do you think?

We'll show you how to declare these **multi-value variables**. We’ll do this with the example that we suggested before. We’ll try to write a program that sorts a sequence of integers, but we won't be particularly ambitious right now – we’ll assume that there will be 5 numbers.

int var1, var2, var3, var4, var5, var6, var7, var8, var9;

We read this record as follows: we create a variable called numbers>; it’s intended to store five values (note the number enclosed inside brackets) of type int (which we know from the keyword int at the beginning of the declaration). Let’s say the same thing using the appropriate terminology: numbers **is an array consisting of 5 values of type int**. Since such an array is called a *vector* in mathematical terms, we'll also say that this statement declares an int vector of a size equal to 5.

All the elements of an array **have the same type**. There are no exceptions to this rule. There are other programming languages which allow the use of arrays with elements of various types, but “C” is not one of them. It’s not, as you might think, a troublesome limitation and it can be effectively avoided, if necessary. However, this is a very complex subject and you have to wait some time for the solution to this puzzle.

It’s time for a bit of intrigue. The “C++” language has a convention which says that the elements in an array are numbered **starting from 0**. This means that the item stored at the beginning of the array will have the number 0. Since there are 5 elements in our array the last one will have the number 4. **Don't forget this**. However, you’ll soon get used to it and it’ll become second nature.

Before we go any further, we have to note the following: our vector is a **collection of elements**, but each **element is a scalar**.

int number[5];

How do we assign a value to the chosen element of the array?

Let's assign the value of 111 to the **first** element of the array. We do it this way:

numbers[0] = 111;

We need a value stored in the **third** element of the array and we want to assign it to the variable i. This is how we can do it:

i = numbers[2];

And now we want the value of the **fifth** element to be copied to the **second** element – can you guess how to do it?

numbers[1] = numbers[4];

The value inside the brackets, which selects one element of the vector, is called an **index**, while the operation of selecting an element from the array is known as *indexing*.

Note: all the indices we’ve used so far are literals. Their values are fixed at run time, but any expression could be the index too. This opens up lots of opportunities.

We want to calculate the **sum of all values** stored in the numbers array. We declare a variable where the sum will be stored and initially assign a value of 0 to it – its name is sum.

Then we add to it all the elements of the array using the for loop, which is a great tool for processing arrays. Take a look at the snippet in the editor.

Let’s talk about this example for a moment. The i variable will take the values 0, 1, 2, 3, and 4 subsequently and will index the numbers array by selecting subsequent elements: the first, second, third, fourth and fifth. Each of these elements will be added by the += operator to the sum variable, giving the final result at the end of the loop.

**int numbers[5], sum = 0;  
for(int i = 0; i < 5; i++)  
sum += numbers[i];**

The next task is to assign the same value (e.g. 2012) to all the elements of the array.

**int numbers[5];**

**for(int i = 0; i < 5; i++)**

**numbers[i] = 2012;**

Now let‘s try to rearrange the elements of the array i.e. **reverse the order of the elements**: the first and the fifth as well as the second and fourth elements will be **swapped**. The third one will remain untouched.

Question: how can we swap the values of two variables? Let's look at the snippet in the editor - if we do something like this, we would lose the value we stored previously in variable2.

Changing the order of the assignments won‘t help us either. Unfortunately, we need a third variable that serves as an auxiliary storage.

**int variable1 = 1, variable2 = 2;  
variable2 = variable1;  
variable1 = variable2;**

Look at the editor - this is how we do it.

**int variable1 = 1, variable2 = 2, auxiliary;  
auxiliary = variable1;  
variable1 = variable2;   
variable2 = auxiliary;**

We don’t know how you feel, but we definitely do not like this. It’s acceptable with an array of 5 elements, but with 99 elements it certainly wouldn't work.

**/\* swap elements #1 and #5 \*/  
auxiliary = numbers[0];  
numbers[0] = numbers[4];  
numbers[4] = auxiliary;  
/\* swap elements #2 and #4 \*/  
auxiliary = numbers[1];  
numbers[1] = numbers[3];  
numbers[3] = auxiliary;**

Let’s use the services of a for loop. Look carefully at how we manipulate the indices values.

During the first turn of the loop, the i variable will be equal to 0, so the instructions in the body will actually perform the following operations:

auxiliary = numbers[0];

numbers[0] = numbers[4];

numbers[4] = auxiliary;

At the second turn, i will be equal to 1, so:

auxiliary = numbers[1];

numbers[1] = numbers[3];

numbers[3] = auxiliary;

As you can see, the loop does the same job by shortening the source code and making it more readable.

**for(int i = 0; i < 2; i++) {  
auxiliary = numbers[i];  
numbers[i] = numbers[4 – i];  
numbers[4 – i] = auxiliary;  
}**

**Array initialization**

Yes, you can initiate arrays, i.e. assign initial values to them at the time of declaration. We do this slightly differently than the initiation of scalars because we need to **specify more than one value**.

The syntax of an array (to be precise, a vector) initiator is clear and legible. Imagine that we want to create an array where the value of any element is equal to its index. A suitable initiator would look like this:

int vector[5] = {0,1,2,3,4};

As you can see, the vector initiator is simply a list of values enclosed inside **curly brackets**.

If you provide fewer values than the size of an array, like this, nothing bad will happen. The compiler determines that those elements you didn‘t specify any value to should be set to 0.

int vector[5] = {0,1,2};

If you provide more elements than can be stored in an array, like this:

int vector[5] = {0,1,2,3,4,5,6};

it‘ll be an **error**. The compiler will be most dissatisfied.

One more example. Look at this:

int vector[] = {0,1,2,3,4,5,6};

We didn't specify the size of the array but **provided an initiator**.

This is legal and it’ll force the compiler to assume that the size of the array is the same as the length of the initiator.

The vector array will be declared in the following way:

int vector[7] = { 0,1,2,3,4,5,6};

**Not only ints**

So far, we’ve discussed vectors whose elements are of type int only. Don't worry. You can also use **arrays of any other type**.

For example - this is an array in which you can store ten floating-point values.

float float\_arr[10];

And you can store twenty characters here:

char surname[20];

...as well as the logical (Boolean) values.

bool votes[100];

Virtually every piece of data can be aggregated into an array. Even an array. We‘ll show you that soon.

**Not only vectors**

We’ve assumed thus far that arrays consist of scalars, but in fact, **arrays can contain elements of a much more complex structure**. Let’s consider the case when an **array's elements are just arrays**.

Surprising? Not at all! We often find arrays like this in our lives. Probably the best example of this is simply a chessboard.

What we’re going to say now will probably outrage experienced chess players, so we apologize in advance for all our simplifications and inaccuracies. If you’re a chess player... it’s nothing personal.

A chessboard is composed of **rows** and **columns**. There are 8 rows and 8 columns. Each column is marked with the letters **A** through **H**. Each row is marked with a number from **1** to **8**. The location of each square is identified by letter-digit pairs. Thus, we know that the bottom right corner of the board (the one with the white rook) is A1, while the opposite corner is H8.

Let’s assume that we can use the selected int values to represent any chess piece. We can also assume that every row on the chessboard is a... **vector**!

Let's try to declare it – here it is:

int row[8];

Unfortunately, we have 8 of these rows. Does this mean that we have to declare 8 arrays like this?

int row1[8], row2[8], row3[8], row4[8], row5[8], row6[8], row7[8], row8[8];

Do you feel some sort of *déjà vu*? We’ve already gone through a similar dilemma, when we were trying to figure out the reason for using vectors.

A *chessboard* is in fact an 8-element array of elements as single rows. Let's summarize our observations:

* **elements of rows** are fields, 8 of them per row;
* **elements of the chessboard are rows**, 8 of them per chessboard.

We’re now ready to create an array for the chessboard – here’s the declaration:

int chessboard[8][8];

The chessboard variable is a **two dimensional array**. It’s also called, by analogy to algebraic terms, a **matrix**.

The appearance of two pairs of brackets tells the compiler that the declared array is not a vector – it's an **array whose elements are vectors**.

Access to the selected field of our board requires two indices – the first selects the row, the second, the field number inside the row, which is *de facto* a column number.

Look at our chessboard. Every field contains a pair of indices which should be given in order to access the field's content.

Glancing at the figure here → we can set some chess pieces on our board. First, let's put all the rooks on the board:

chessboard[0][0] = ROOK;

chessboard[0][7] = ROOK;

chessboard[7][0] = ROOK;

chessboard[7][7] = ROOK;

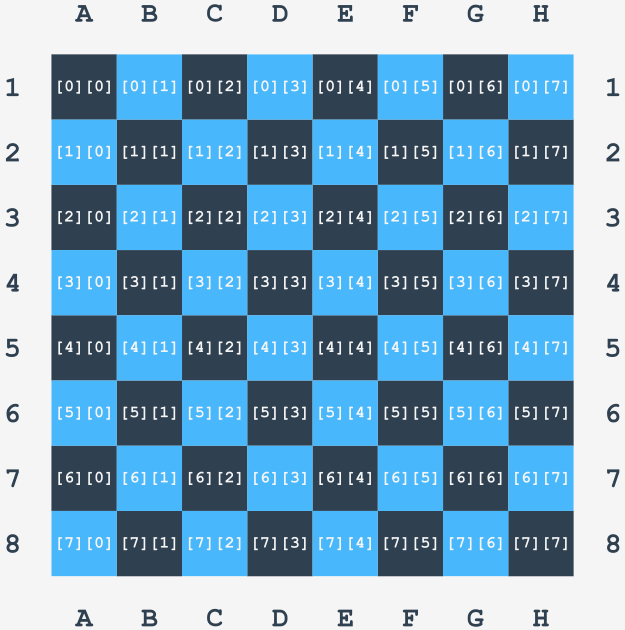
If we wanted to place a knight on C4, we would do this as follows:

chessboard[3][2] = KNIGHT;

And now a pawn to E5:

chessboard[4][4] = PAWN;

As clear as check mate, right?



**Arrays of arrays**

Now let's go deeper into the multi-dimensional nature of arrays. To find any element of a two-dimensional array, we have to use two “***coordinates***”: a vertical (row number) one and a **horizontal** (column number) one. Imagine that we develop a piece of software for an automatic weather station. The device records the air temperature on an hourly basis and does it throughout the month. This gives us a total of 24 \* 31 = 744 values. Let's try to design an array capable of storing all these results.

First, we have to decide which data type would be adequate for this application. We think that a float would be the best, since our thermometer can measure the temperature with an accuracy of 0.1 degree Centigrade. Then we decide that the rows will record the readings every hour on the hour (so the row will have 24 elements) and each of the rows will be assigned to one day of the month (so we need 31 rows). Here’s the appropriate declaration:

float temp[31][24];

Now we’ll try to determine the monthly average noon temperature. We’ll add all 31 readings recorded at noon and divide the sum by 31. We assume that the midnight temperature is stored first. In the editor you can find the relevant code along with the necessary declarations.

**float temp[31][24];  
float sum = 0.0, average;  
for (int day = 0; day < 31; day++)  
sum += temp[day][11];  
average = sum / 31;  
cout << "Average temperature at noon: " << average << endl;**

Now let's find the highest temperature for the whole month – see the code in the editor.

**float temp[31][24];  
float max = -100.0;  
for (int day = 0; day < 31; day++)  
for (int hour = 0; hour < 24; hour++)  
if (temp[day][hour] > max)  
max = temp[day][hour];  
cout << "The highest temperature was " << max << endl;**

We want to count the days when the temperature at noon was at least 20oC.

**float temp[31][24];  
int hotdays = 0;  
for (int day = 0; day < 31; day++)  
if (temp[day][11] >= 20.0)  
hotdays++;  
cout << hotdays << " days were hot.";**

We’re going to fill the entire array with zeros in order to prepare it for use in the coming month.

**float temp[31][24];  
int d, h;  
for (d = 0; d < 31; d++)  
for (h = 0; h < 24; h++)  
temp[d][h] = 0.0;**

The C++ language **doesn’t limit the size of the array's dimensions**. Here we show an example of a 3-dimensional array.

Now imagine a hotel. It's a huge hotel consisting of three buildings, 15 floors each. There are 20 rooms on each floor. We need an array that can collect and process information on the number of guests registered in each room.

Step one – the type of the array's element. We think an int would fit, although it can be unassigned as there’s no such thing as a negative number of guests.

Step two – calm analysis of the situation. Summarize the available information: **3 towers, 15 floors, 20 rooms**.

Now we can write the declaration (look at the editor).

The first index (0 through 2) selects one of the buildings; the second (0 through 14) selects the floor, the third (0 through 19) selects the room number.

Now we can book a room for two newlyweds: in the second building, on the tenth floor, room fourteen:

guests[1][9][13] = 2;

and release the second room on the fifth floor located in the first building:

guests[0][4][1] = 0;

Before we say goodbye and finish this part of our course, let's check if there are any vacancies on the fifteenth floor of the third building:

int room;

int vacancy = 0;

for (room = 0; room < 20; room++)

if (guests[2][14][room] == 0)

vacancy++;

The vacancy variable contains 0 if all the rooms are occupied; otherwise it displays the number of available rooms.

int guests[3][15][20];

**Structures – why do we need them?**

Before we start going on about structures, we need to tell you about a completely new type, named **string**. For now, we’re not going to tell you any more than what we use it for and how we declare variables of this type. We promise we’ll give you more details in a separate section devoted exclusively to strings, and we’ll show you how to manipulate them and how to enjoy using them.

To be honest, a string is little more than a type, and all we want to tell you at this point is that variables of type string are able to store strings of characters, like surnames, family names, street names and all other names you can think of, including R2D2.

Variables of type string may be assigned with the same operators as any other variable we’ve previously encountered. For example, if we want to store the name of our favourite dish in a string variable, we do it in the following way:

string dish\_to\_order = "pizza";

Imagine that we, the developers, have the following job to do: we’re obliged to design a data structure that can store information about students attending our course. It needs to store the name of each student, the time spent on studying the chapters and the number of the last completed chapter. We know that the total number of all students will not exceed 100,000. We also know that most names are rarely longer than 25 characters and this leads us to write the following declaration:

char student\_name[100000][26];

The array allows us to store 100,000 names of up to 25 characters. Wait a minute – you might say – if it’s 25 characters, why did we write “26” in the declaration? We just reserved one more character for the terminating (closing) character, that’s all.

Let's try to manipulate that array. For example, suppose that the first registered student was Mr. Bond (James Bond). Let’s store the information in our array:

strcpy(student\_name[0],"Bond");

The time spent on the site will be stored as a float. This is not particularly convenient, but we can certainly handle it. The number of hours will be represented as a decimal fraction. This leads us directly to the following statement.

float student\_time[100000];

We know that Mr. Bond spent three hours and thirty minutes studying our course. We’ll denote it in the following way:

student\_time[0] = 3.5;

Note: 3h30m = three and a half hours, thus *3.5*.

The main issue here is that the data concerning the same object (a student) is **dispersed between three variables**, although it should logically exist as a consolidated unit. Handling multiple arrays is cumbersome and error-prone, and when life forces us to collect additional information (e.g. e-mail address) we’re going to need to declare another array and make a lot of other changes throughout the program. We don’t like it, and you can be sure you won't like it either.

We already know what an array is. The array is an aggregate of elements. The elements are numbered and are of the same type. Can we use **an aggregate whose elements could be of different types**? Could they be identified by names, not by numbers? And is it a good idea?

Yes, it's a great idea! This magical aggregate is called a **structure**.

A structure contains **any number of elements of any type**. Each of these elements is called a **field**. Each field is identified by its name, not by its number. Obviously, the field names must be **unique** and cannot be doubled within a single structure. We’ll show you how to declare a structure suitable for our needs and we’ll explain its meaning.

You can see the declaration of the structure in the editor.

* the declaration of the structure always starts with the keyword struct;
* there is a so-called struct tag after the keyword (STUDENT in this case); it's the name of the structure itself; there is a widely accepted custom of composing structure tags with capital letters simply to distinguish them from ordinary variables
* here comes the opening curly bracket – a sign that the field declaration begins at this point;
* our structure has three fields: the first is a string and is called name; the second is a float and is called time; the third is an int and it’s called recent\_chapter
* the declaration ends with the closing curly bracket followed by a semicolon.

**struct STUDENT {  
string name;  
float time;  
int recent\_chapter;  
};**

We want to emphasize that the previous declaration doesn't create a variable, but only describes the structure we’re going to use in our program. If we want to declare a variable as a structure, we can do it in one of two possible ways:

struct STUDENT stdnt;

STUDENT stdnt2;

This declaration sets up two variables (**structured variables**) named stdnt and stdnt2 respectively. The variables are of type struct STUDENT or just STUDENT (notice, that the structure declaration creates a new type name). We know that this variable consists of three named fields, but we don’t yet know how to access them.

As the C++ language offers a specialized indexing operator [] for arrays, it also gives us a so-called **selection operator** designed for structures and is denoted as a single character . (dot).

The priority of the selection operator is very high, equal to the priority of the [] operator.

This is a binary operator. Its left argument **must identify the structure** while the right argument must be the **name of the field** known in this structure.

The result of this operator is the selected field of structure, and therefore the expression containing this operator is sometimes called a **selector**.

This means that the selector here:

stdnt.time

results in the selection of a field named time. The type of this expression is the type of the selected field and this expression is an l-value.

Consequently, you can use both of these selectors:

stdnt.time = 1.5;

and

float t;

t = stdnt.time;

Virtually any data could be used as a structure's field: scalars (including pointers), arrays and also almost all of the structures. We say “almost” because a **structure cannot be a field of itself**.

Structures can be aggregated inside an array, so if we want to declare an array consisting of STUDENT structures, we do it in this way:

struct STUDENT stdnt[100000];

Access to the selected fields requires two subsequent operations:

* in the first step, the [] operator indexes the array in order to access the structure we need;
* in the second step, the selection operator selects the desired field.

This means that if we want to select the time field of the fourth stdnts' element, we'll write it like this:

stdnts[3].time

We’ve collected all these assignments which have been performed for the three separate arrays. Analyze them carefully:

stndts[0].name = "Bond";

stndts[0].time = 3.5;

stdnts[0].recent\_chapter = 4;

# Declaring the structures

For the purposes of further considerations we’ll use a simple structure designed to store the date. It’s equipped with three fields, each of type int, named year, month and day, which clearly denote their role and purpose.

The first possible way of declaring the structure is in the editor.

Of course, we can write this declaration much more compactly:

**struct DATE {**

**int year, month, day;**

**};**

Both variants are equivalent.

That declaration doesn't create any new variables, but **only announces to the compiler** our intention to use this structure tag to declare new variables. The new variable would be declared, for example, in this way:

**DATE DateOfBirth;**

We can use it to store Harry Potter's date of birth:

**DateOfBirth.year = 1980;**

**DateOfBirth.month = 7;**

**DateOfBirth.day = 31;**

We can also use the structure tag to declare an array of structures:

**DATE visits[100];**

Accessing a single structure stored in the array is easy. If we want to modify the data of the first visit, we do this:

**Visits[0].year = 2012;**

**Visits[0].month = 1;**

**Visits[0].day = 1;**

It’s also possible to kill two birds with one stone by defining the structure tag and declaring any number of variables simultaneously in the same statement, like this:

**struct DATE {**

**int year, month, day;**

**} DateOfBirth, Visits[100];**

**DATE current\_date;**

We can also omit the tag and declare the variables only:

**struct {**

**int year, month, day;**

**}**

**the\_date\_of\_the\_end\_of\_the\_world;**

In this case, however, determining the type of the variable the\_date\_of\_the\_end\_of\_the\_world (e.g. if we want to use it with the sizeof operator) becomes troublesome. Without a tag it has to be denoted as:

sizeof(struct {int year, month, day;})

We find it too complex and unreadable, compared to sizeof(struct DATE).

**struct DATE {  
int year;  
int month;  
int day;  
};**

# Structures – how do we use them?

**A structure could be a field inside another structure**. Imagine that we have to extend our STUDENT structure and add a field to save the date when a particular student last accessed the course. We can do it the way shown in the editor.

It means that we’ll have to use two subsequent selection operations to **go deeper** into the structure i.e. first we select a structure within the structure and then we select the desired field of the inner structure.

This is how it works:

**HarryPotter.last\_visit.year = 2012;**

**HarryPotter.last\_visit.month = 12;**

**HarryPotter.last\_visit.day = 21;**

So now you should be able to answer the following question: when did Harry visit us recently?

**struct STUDENT {  
string name;  
float time;  
int recent\_chapter;  
struct DATE last\_visit;  
}  
HarryPotter;**

# Structures – a few important rules

A structure's field names may overlap with the tag names, and this is generally not considered a problem, although it may cause some difficulty in reading and understanding the program.

The snippet shown in the editor is completely correct.

**struct STRUCT {  
int STRUCT;  
}  
Structure;  
Structure.STRUCT = 0; /\* STRUCT is a field name here \*/**

It may be the case that the particular compiler you’re working with doesn't like it when a structure’s tag name overlaps with the variable's name (you know how those compilers are); – therefore, it's better to avoid tricks like the ones shown in the editor.

Alternatively, get yourself a new compiler – one that doesn’t complain so much. OK, just kidding.

**struct STR {  
int field;  
}  
Structure;  
int STR;  
Structure.field = 0;  
STR = 1;**

Two structures **can contain fields with the same names** – the snippet in the editor is correct.

**struct {  
int f1;  
}  
str1;  
struct {  
char f1;  
}  
str2;  
str1.f1 = 32;  
str2.f1 = str1.f1;**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Structures can be initialized as early as **at the time of declaration**. The structure's initiator is enclosed in curly brackets and contains **a list of values assigned to the subsequent fields**, starting from the first.

The values listed in the initiator **must conform to the types** of the fields. If the initiator contains fewer elements than the number of the structure's fields, it’s presumed that the list is automatically extended with zeros.

If the particular field is an array or a structure, it should have its own initiator, which is also subject to the rule of zero extension. If an “internal” initiator is complete, we can omit the surrounding curly parentheses.

Take a look at the example:

struct DATE date = { 2012, 12, 21 };

This initiator is equivalent to the following sequence of assignments:

date.year = 2012;

date.month = 12;

date.day = 21;

The initiator of this form is functionally equivalent to the following assignments:

he.name = "Bond";

he.time = 3.5;

he.recent\_chapter = 4;

he.last\_visit.year = 2012;

he.last\_visit.month = 12;

he.last\_visit.day = 21;

Due to the completeness of the inner initializer, we can write the following, simplified form:

STUDENT he = { "Bond", 3.5, 4, 2012, 12, 21};

This simplification (omitting the internal curly brackets) cannot be applied in the following case:

STUDENT she = { "Mata Hari", 12., 12, { 2012 } };

The internal initiator, referring to the last\_visit field, doesn't cover all the fields. This means that it’ll be equivalent to the following sequence of assignments:

she.name= "Mata Hari";

she.time = 12.;

she.recent\_chapter = 12;

she.last\_visit.year = 2012;

she.last\_visit.month = 0;

she.last\_visit.day = 0;

What happens when we apply an “empty” initializer?

STUDENT nobody = {};

Here's the answer:

nobody.name = "";

nobody.time = 0.0;

nobody.recent\_chapter = 0;

nobody.last\_visit.year = 0

nobody.last\_visit.month = 0;

nobody.last\_visit.day = 0;

**Congratulations! You have completed Module 2.**

Well done! You've reached the end of Module 2 and completed a major milestone in your C++ programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 2:

* how to control the flow of the program;
* more data types;
* conditional instructions: if, else, switch;
* loops and controlling the loop execution;
* logic, bitwise and arithmetic operators;
* vectors, multidimensional arrays;
* declaring and initializing structures.

You are now ready to take the module quiz and attempt the final challenge: Module 2 Test, which will help you gauge what you've learned so far.

**Pointers – the absolute basics**

**Pointers** are also **values**, but different from those we ‘ve been using so far. The types we’ve been using are closely linked to computer data processing, but fully reflect our ideas and intuition. All of us use integers in everyday life to count. We use *floats* when we pay for something and *ints* when we count something. Pointers have no simple and obvious analogy to our daily lives and their values are unreadable to humans and completely useless. Computers, however, can make great use of pointers, giving developers powerful options when designing algorithms and data structures.

Now, be prepared for the fact that not everything will be immediately obvious to you. This is normal – it’ll take some time before you can understand the pointers themselves and their specific traits.



Modern computer memories are large and fast. You probably already know that memory size is expressed in units called bytes. You probably also know that when you declare any variable, the variable occupies a small piece of the computer memory. So far, it’s been important for us to know what value is stored in the variable. From now on, we are also going to be interested in where this value is stored.

This trait of the data (to say it more formally, this **attribute**) is often called the **address**. We all live at certain addresses just like every variable “lives” at its address, too.

Try to see this important difference:

* the **value** of the variable is what the variable stores;
* the **address** of the variable is information about where this variable is placed (where it *lives*).

**Pointers are used to store information about the location (address) of any other data**. We can say that pointers are like **signposts**. They don’t say anything about the place itself, but they show us clearly how to reach it.

int i;

**The first pointer**

Let's start off with a simple declaration and an equally simple operation. We’re going to encounter a few new operators, too. Already at the time of declaration, we can see the difference between regular data types and pointer types. Take a look:

int \*p;

We use the asterisk (“\*”) here in a completely new way. It has nothing to do with multiplication. You may be feeling a bit confused by this, but don’t worry.

This dual nature of asterisks will become clear soon enough. The C++ language syntax ensures that you'll always be able to figure out what meaning the asterisk is being used for in any particular context. This declaration sets up a variable named p. It isn't an int - the asterisk means that p is a pointer and will be used to store information about the location of the data of type int. The pointers are always used to point to the specific data referred to in the declaration. The C++ language may use so-called **amorphous pointers**, too, which can be used to point to any data of any type, but we’re going to discuss them at the very end of our story.

What is the type of variable p? There are a few answers, identical in meaning, but considerably different. First, we can say that p is a variable of type “*pointer to int*”. Secondly, it would be as follows: *p is a variable of type int \** where “\*” reads simply as “*asterisk*”.

**How do we assign a value?**

**How do we assign a value to the pointer variable?**

Can we assign a value to the pointer? Of course, in the same way that you can assign any value to any other variable: by using the = operator. A more important question is which values are we allowed to assign to the pointers?

Using a literal is **not an option**. The compiler won’t allow you to write something like this (look at the line of code below) because, firstly, the C++ language syntax doesn’t allow it and secondly, if you know what’s stored in the computer's memory at the address 148324, then you’re a clairvoyant and computer programming is just a waste of your time.

p = 148324;

There’s one distinctive exception. You can assign **zero** to the pointer variable. Doing this doesn’t prompt any questions from the compiler.

p = 0;

A pointer which is assigned a value of zero is called a **null pointer** (as in Latin, *nullus* – none). It doesn’t point to anything. It's like a signpost with all the writing removed. It still exists, but with no purpose to the traveler. Despite appearances, this pointer can be very useful and sometimes even necessary.

Due to the fact that assigning a value of zero to a pointer variable sometimes causes misunderstandings and mistakes (some may confuse the pointer with a variable of type int), there’s an unwritten agreement that developers will avoid assigning null pointers. Instead, they should follow this next convention:

p = NULL;

The **NULL** symbol is actually equal to **zero**. It looks like a variable but you cannot modify its value. It’s called a **macro**. We’ll tell you about it in the future.

The same convention also says that *NULL* should be assigned only to pointers. This is a matter of care and elegance with regard to the programming style. We have one caveat: if you want to use the NULL symbol, you have to **include the header file** named cstring, or any other header file which includes cstring itself (one of them is iostream).

How do we assign any meaningful value other than *NULL* to a pointer? We may assign to the pointer a value which **points to any already existing variable**.

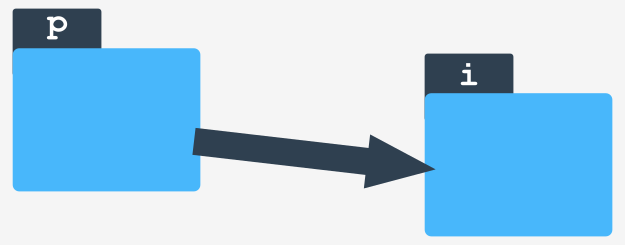
To do that, we need an & operator, called the **reference operator**. This is a unary prefix operator.

The operator simply finds the address of its argument. See the statement:

p = &i;

After completing the assignment, the p variable will point to the place where the i variable is stored in the memory.

This can be shown in the following way:



If you assign NULL to the pointer, it’ll look like this. From now on, the p pointer points to neither the i variable nor to any other variable.

We’ll use the following symbols to express this. As you can see, the p pointer is **grounded**.

What can we do with a pointer that is not null and points to a value? We can *dereference* it.

Dereferencing is an operation by which the pointer variable (as we’ll see later, it’s not only a variable, but also an expression that yields a pointer) becomes **synonymous** with the value it points to.

Look at how we can declare a variable of type int (ivar) and a variable of type int \* (ptr). We can do it within a single statement:

int ivar, \*ptr;

Now let’s assign the value of 2 to the ivar variable – everything’s clear so far, right?

ivar = 2;

Now we make the ptr pointer point to the ivar variable.

ptr = &ivar;

How do we get a value pointed to by the pointer? We have to use a well-known operator (the asterisk: “\*”) but in a completely new way – as a **dereferencer**.

If you place an asterisk in front of a pointer, you get a value which is stored at the location pointed to by the pointer.

\*ptr

The following invocation will display 2 on the screen, as the **dereferenced ptr value** is sent to cout (note: ptr points to ivar and ivar is equal to 2).

cout << \*ptr;

If you write a statement like the one here:

\*ptr = 4

you **won't change the pointer value**. You’ll instead change the value pointed to by the pointer. This is an important difference.

Don't forget that if you declare a pointer in the following way:

ANY\_TYPE \*pointer;

it means that:

* the pointer variable is of type ANY\_TYPE\*
* the \* pointer expression is of type ANY\_TYPE

Notice the place the asterisk occupies in both cases.

Also, don't forget that **dereferencing NULL pointers** is strictly forbidden and leads to serious problems very quickly.

**sizeof operator**

The sizeof operator is distinguished by its appearance. The operators we’ve encountered so far are coded as single characters or digraphs. This new operator looks like a variable.

Don't be misled: this is a **unary prefix operator** and with the highest possible priority. There’s another difference too: a typical operator requires a value as its argument and usually changes the value in certain ways.

The new operator expects that its argument is a **literal**, or a **variable**, or an **expression** enclosed in parentheses, or the **type name** (this is the only “C” operator which allows its argument to be a type).

The operator provides information on **how many bytes of memory its argument occupies** (or may occupy). The name explains the purpose – here it is:

sizeof

Note: there’s no space between “*size*” and “*of*”.

The sizeof is not only an operator – it’s also a **keyword**.

Variable i will be assigned the value of 1, because char values always occupy one byte. Note that we can achieve the same effect by writing:

i = sizeof(char);

You may not use parentheses when the argument is a literal or a value, but you must use them when the argument is a type.

int i; char c;

i = sizeof c;

Variable i will be set to the value of 10, because this is the number of bytes occupied by the **entire** *tab* array.

char tab[10];

i = sizeof tab;

Variable i will be set to the value of 1 – can you explain why?

char tab[10];

i = sizeof tab[1];

The following example is not so obvious.

Values of the int type occupy 32 bits, i.e. 4 bytes in most modern compilers/computers, **but we cannot guarantee** that this is true in all cases.

int i;

i = sizeof i;

We think it would be a good exercise for you to compile and run the following program on your computer. By doing this, you’ll learn **how your computer and compiler use the memory**.

The program is perhaps a bit simplistic, but its role is to identify interesting features of your environment, and it’s good enough for this.

## Code

#include <iostream>  
  
using namespace std;  
  
int main(void) {  
cout << "This computing environment uses:" << endl;  
cout << sizeof(char) << " bytes for chars" << endl;  
cout << sizeof(short int) << " bytes for shorts" << endl;  
cout << sizeof(int) << " bytes for ints" << endl;  
cout << sizeof(long int) << " bytes for longs" << endl;  
cout << sizeof(float) << " bytes for floats" << endl;  
cout << sizeof(double) << " bytes for doubles" << endl;  
cout << sizeof(bool) << " byte for bools" << endl;  
cout << sizeof(int \* ) << " bytes for pointers" << endl;  
return 0;  
}

**Pointers vs. arrays**

What do **pointers and arrays** have in common?

Well, a lot. Let's start with an important definition: if you see the **name of an array** without the indices, then it’s always a **synonym of the pointer pointing to the first element of the array**.

How does it work?

We’ve declared a pointer to int and the three-element array of type int.

int \*ptr, arr[3];

The two assignments that follow the declaration set ptr to the same value. In other words, the following comparison is always true:

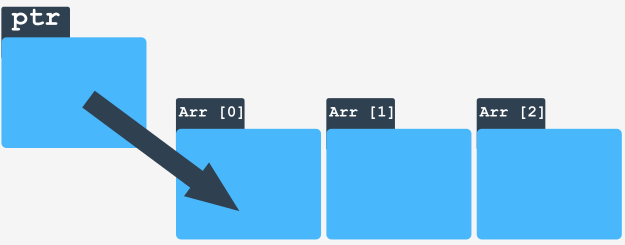
arr == &arr[0]

This figure illustrates the effect of the assignment.

int \* ptr, arr[3];

ptr = &arr[0];

ptr = arr;



# The arithmetic of pointers

TThe pointers' arithmetic is significantly different from the integers' arithmetic as it is relatively reduced and allows the following operations only:

* **adding an integer** value to a pointer, giving a pointer (*ptr + int → ptr*);
* **subtracting an integer** value from a pointer, giving a pointer (*ptr – int → ptr*);
* **subtracting a pointer from a pointer**, giving an integer (*ptr – ptr → int*);
* **comparing the two pointers** for equality or inequality (this gives a value of type int of either true or false) (*ptr == ptr → int; ptr != ptr → int*).

Any other operations are either prohibited or meaningless and only the ones mentioned above may be used.

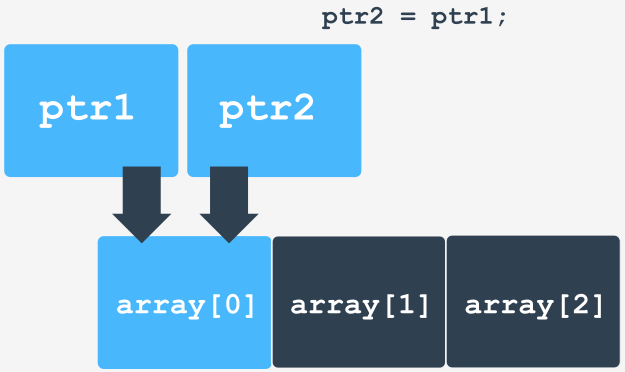
Let’s discuss briefly what’s happening to a pointer subjected to these operations. We’ll do this by assuming the declarations and assignments shown in the editor.

At this point, ptr1 points to the first element of array.

## Code

int \*ptr1, \*ptr2, array[3], i;  
ptr1 = array;

After the following assignment, ptr2 **points to the first element of array**, too – the figure shows the current state of the variables.



We can check if the two pointers are equal – yes, they are, as they point to the same element of the array.

if(ptr2 == ptr1) {

:

:

}

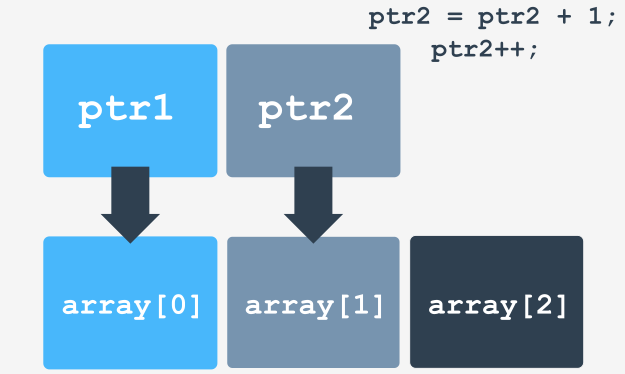
Let's check how the addition works. These statements do the same operation: they add 1 to ptr2.

We can interpret this operation as follows:

* it’s taken into account what type is pointed to by the pointer - in our example it’s int;
* it has determined **how many bytes of memory the type occupies** (the sizeof operator is used automatically for that purpose) – in our case it’s sizeof (int);
* the value we want to add to the pointer is multiplied by the given size;
* the address which is stored in the pointer is **increased** by the resulting product.

In effect, the pointer moves itself to the next int value in the memory.

The effect of this incrementation is shown in the figure.



What would happen if we added 2 instead of 1?

In this case the ptr2 would be increased by (2 \* sizeof (int)) and thus ptr2 would move through **two** int values and would point to the third element of the array (namely, array[2]).

The comparison

ptr1 == ptr2

is obviously false, while this one

ptr1 != ptr2

is true, as the addresses the pointers point to differ.

Now let's subtract the pointers in the following way:

We said earlier that subtraction gives a result of type int. How is it calculated?

* taken into account: the type to which the pointers point (int); this means that both pointers need to point to the same type; the compiler will check it
* the addresses stored in the pointers are subtracted
* the result of the subtraction is divided by the size of the type pointed to by the pointers

The final result tells us how many variables of a given type (i.e. int) fit between the addresses stored in the pointers. In our case, it’s obviously 1, and this value will be assigned to the i variable.

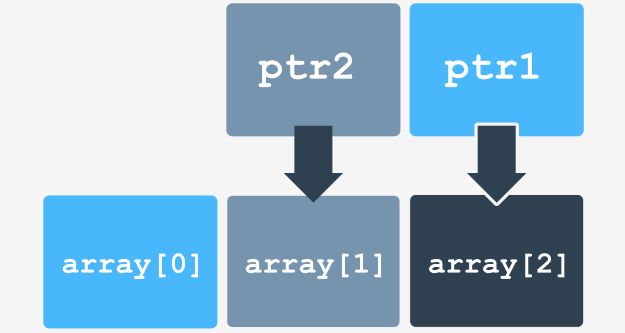
i = ptr2 - ptr1;

The result will be greater than 0 if ptr2 points to the memory located after ptr1; otherwise, it’ll be less than 0.

Try to guess the result of the following operation:

ptr1 = ptr1 + 2;

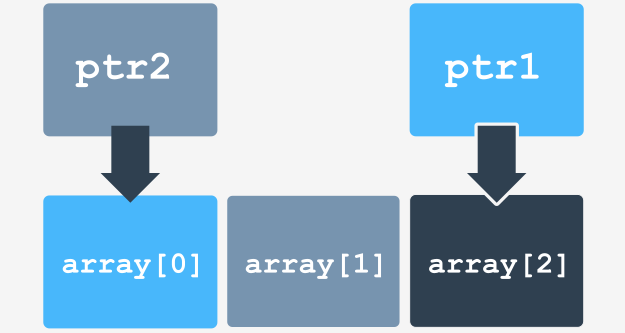
You can find the answer below:



Let’s assume that the following operation has been performed. Can you guess the effect?

ptr2 = ptr2 - 1;

Here’s the answer:



Try to determine the result of the following subtraction.

ptr1 - ptr2

Is it 2?

Yes, it is.

Congratulations!

**What is a function?**

The word "**function**" is probably not so alien to you. Certainly, you’ll have heard it many times before, even in circumstances that had nothing to do with programming.

We can bet that you’ve met this word in a math class. Do you remember? Sine is just one of the many examples.

A function is a kind of box (not always black) that can do something useful for us, e.g. to evaluate a value or perform some actions. The former means that a function has (or may have) **a result**. The latter means that a function has (or may have) **an effect**.

In other words, a function is just a separate part of a code that can be used at (almost) any time to evaluate something, to do something, or both. **We identify a function by its name**.

The name of a function is subject to the same restrictions as a variable's name. Additionally, we can’t have a variable and a function of the same name.



Every function is able to modify its own behaviour using **parameters** (do you remember? The sine function has one parameter, too). Parameters may affect what is calculated or what is performed inside the function. Moreover, a function may modify its parameter's values if necessary.

When you want a particular function to perform its actions and/or evaluations, you need to **invoke** it.

The **function invocation** is a special instruction in the C++ language. The invocation has to specify the name of the function being invoked (always), the parameters that the function should use (if needed) and how to use the result evaluated by the function (if any).

There’s one catch: you shouldn't invoke a function that is unknown to the compiler. The compiler needs to be aware of the function's nature (name, parameters, result) and there are some ways to inform the compiler about all the functions you may want to use.

In general, we can divide functions into two groups:

* functions written by someone else (not you) which are made available by the environment, sometimes called **predefined** or **library functions**
* functions written by you

You can use both of them equally easily.

**Why do we need functions?**

Functions facilitate the creation of programs in multiple ways. We can say that it’s nearly impossible to write a large, complex program without using functions. Even if it succeeds, it won’t be a piece of good software for many reasons.

Library functions are the first functions a novice developer uses, but it very soon becomes clear that they’re never sufficient for solving complicated problems. This is the moment when the developer decides to write his/her own functions.

Don't forget: it’s always better to ensure that the function you need is really worth being implemented from scratch. **It’s always a good idea to check if the function you want hasn’t already been written by someone else**, so as to avoid reinventing the open door.

Functions enable developers to divide a problem (and also a code) into smaller parts. A smaller code is easier to write, to test, to maintain and to understand.

Having a set of well written and well tested functions lets the developer construct the code in a way that’s similar to house building: using ready-made blocks. This method of dividing the problem is known as **decomposition**.

In general, there are two possible approaches you can use during decomposition: the **top-down** approach (when you try to define the most general functions first and then you divide them into simpler and more specialized ones) and the **bottom-up** approach (when you start your work by creating a set of narrowly defined and highly specialized functions, and then assembling them into more complex structures).

Experienced developers know that creating functions is always a good idea even if the new function is invoked only once during the program's execution. If the code you’ve written is too long to fit on one screen, consider dividing it into functions.

It's easier to control a herd of small and well-defined functions than one large portion of code that you can't see at a glance.

**Introduction to functions**

Each function is characterized by the following traits:

* name
* parameters
* type of result

The part of the code that specifies all these elements is known as the **function declaration**. The compiler must know the function declaration to enable it to properly interpret the invocations of the function. The function declaration is sometimes called the **function prototype**.

A function declaration says nothing about what the function does exactly. That information is provided by the function body, which is a separate part of the code, enclosed in brackets.

A function declaration enriched with a function body forms the so-called **function definition**.

Imagine that we need a function that can evaluate the second power of any float number.

We realize that:

* square would be quite a good name for the function; of course, we can call it in many different ways, e.g. sqr, SecondPowerOf or even JohnDoe (although it’s probably not a good idea, is it?)
* functions need one parameter: the value which will be raised to the power of 2; x will be a good name for the parameter, although not very original;
* the result type is float (like the parameter)

We can write the function declaration now. You can see it here:

float square(float x);

Note that the first float specifies the type of the result, while the second is the type of the parameter.

# First function

If we want to take advantage of this function, we need to **deliver its definition**. You can see it in the editor.

Note: transforming a declaration into a definition requires us to add a body, but the body also replaces the semicolon that ends the declaration (see previous slide). The function body doesn't end with a semicolon.

The body contains:

* a declaration of the result variable; the variable will be used **inside the function and only inside the function**; it’s neither visible nor accessible in any other part of your code;
* the result variable is assigned the value of the x parameter multiplied by itself; this is the simplest and fastest method of raising a number to the power of 2;
* the last instruction of the square function is return; this instruction is responsible for two important actions:
  1. it indicates **which value is returned** (provided) as the function result
  2. it **terminates the execution** of the function

## Code

float square(float x) {  
float result;  
  
result = x \* x;  
return result;  
}

We’re now going to make use of our first function and make it a part of a complete, runnable program. Here it is in the editor.

The parameters defined within the function are called **formal parameters**. The values actually transferred to the function (thus existing outside the function) are called **actual parameters**.

The function invocation is just the name of the function being invoked along with the values transferred (passed) into the function as actual parameters.

As you can see, we’ve declared a variable named arg and assigned it the value of 2.0. Next, we’ve invoked the square function, delivering the arg variable as its argument (the actual parameter).

The result of the function is then sent to the screen and displayed as part of the message saying “The second power of 2 is 4”.

## Code

#include <iostream>  
  
using namespace std;  
float square(float x) {  
float result;  
result = x \* x;  
return result;  
}  
int main(void) {  
float arg = 2.0;  
cout << "The second power of " << arg << " is " << square(arg) << endl;  
return 0;  
}

[Prev](https://edube.org/learn/c-essentials-part-1-1/functions-why-9) [Next](https://edube.org/learn/c-essentials-part-1-1/functions-why-11)Is it possible to place the square function **after** the main function and not **before**? Yes, it is, but don't forget that the compiler must be aware of all the traits of the invoked function.

Therefore, you have to put the function declaration before the first function invocation.

Take a look at the modified code in the editor.

Note: you can omit the formal parameter's name in the function prototype.

You’re now ready to learn about some of the more advanced properties of functions.

## Code

#include <iostream>  
  
using namespace std;  
float square(float);  
int main(void) {  
float arg = 2.0;  
cout << "The second power of " << arg << " is " << square(arg) << endl;  
return 0;  
}  
  
float square(float x) {  
float result;  
result = x \* x;  
return result;  
}

# Declaring functions

A function declaration is intended to inform a compiler about the function's name, its return type and type of the parameters (if any). A general form of function declaration (or function prototype – we can use these terms interchangeably) is here: return\_type function\_name(parameter\_list);

* return\_type describes the **type of result returned (delivered) by the function** (e.g. we expect that the sine function will return a value of type float as int data is completely unusable in this context); you can use any of the C++ types as a return\_type, including a very special type named void; a function of type void returns no result at all; we can say that such a function may have an effect but definitely has no result; if you omit the return\_type, the compiler assumes that the function returns a value of type int
* function\_name is an identifier which **names the function and distinguishes it from all other functions** (note: you can have more than one function of the same name, in contrast to variables; this is called **overloading** and we’re going to tell you about it soon)
* parameters\_list (you have to enclose this one in parentheses!) is a comma-separated list of “type name” pairs, where each of the names may be omitted; the list may be empty but the parentheses must still be there; you can emphasize the fact that your function has no parameters by putting the word void inside the parentheses, like this:

void fun(void);

This declaration describes a function named fun which doesn't return a result and has no parameters; here’s an example of two equivalent function declarations:

int func(int number);

int func(int);

In this context, the name of the parameter (number) means nothing to the compiler and it completely ignores it; it's not an argument against specifying the parameter's name in the declarations as it could be very useful for people trying to understand your code

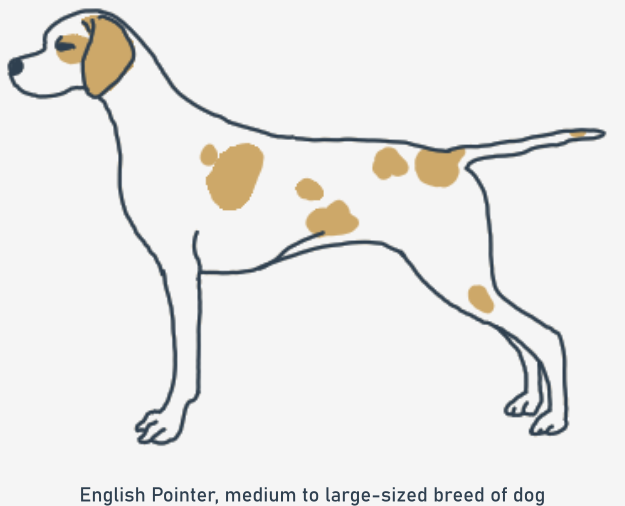
* the declaration ends with a semicolon which must not be omitted.

Although each of the elements of the parameter list resembles the syntax of a variable declaration, you're not allowed to declare more than one parameter name at the same time. You can do this:

int x, y;

but you have to use the following verbose form if you declare two (or more) function parameters:

void fun(int x, int y);



**Defining functions**

A function definition specifies the code to be executed on each function invocation. It differs from the declaration in the fact that a **semicolon is replaced by a body containing a sequence of declarations and/or instructions.**

A function definition specifies the code to be executed on each function invocation. It differs from the declaration in the fact that a **semicolon is replaced by a body containing a sequence of declarations and/or instructions.**

**If the return\_type is void, the body may not contain the return instruction**, but if it’s used anyway, it has to have the following form:

return;

Note: there’s no value after the return (obviously, a void function isn’t able to return any value).

It’s assumed that **the return statement is implicitly executed within the void function's body** just before the closing bracket. You don't have to write it there explicitly.

If the return\_type is not void, the body must contain at least one return statement specifying the value of the function's result. You can’t leave a function's body without specifying the result's value.

Any value you intend the function body to return must have a type compatible with the type specified as the function type.

You can use **as many return statements as you need** to effectively implement your algorithm.

# Example functions

Let's assume that we need a highly specialized function to greet a user who’s dared to run our program.

We choose “Greet” as the name for that function, and decide that the new function should have no result and no parameters.

You can see the definition of the greet function in the editor.

## Code

void greet(void) {  
cout << "Ave user!" << endl;  
}

# Example functions

Some users (those with very big egos) need to be greeted more than once. To avoid writing separate functions for different categories of users, we’ll write a universal function that can be instructed on **how many times the greetings should be shouted out**. We’ll use a parameter of type int for this purpose.

The function is ready. You can see it in the editor.

As you can see, we’ve made use of a previously defined function. In this way, if we changed the text of the greeting (e.g. to “I'm ready to serve you, my Master”), it would have an effect on both functions.

Now pay attention to what the function invocation looks like in the code. We’ve had to use a pair of parentheses, even though we haven’t specified a parameter.

Also note how we named the function or, rather, how we glued all the words together. This technique of combining words to receive a variable's or a function's name is called **CamelCase** (really!), as the consecutive capital letters look like humps compared to the rest of the name.

We use *CamelCase* in many of our examples, although it’s a question of aesthetics and not a language requirement.

## Code

void greet\_many\_times(int how\_many\_times) {  
while (how\_many\_times > 0) {  
greet();  
how\_many\_times--;  
}  
}

# Example functions

Now we’re going to show you a complete program demonstrating the functioning of our functions. The program asks a user for the size of his/her ego and responds with an adequate greeting. The size of the ego is measured in meters (apologies to all users of imperial units).

Look at how we send the actual parameter value to the invoked function. Try to imagine how the actual parameter replaces a formal parameter within a function.

The user gets one greet per kilometre and one extra greet (for those with no ego).

## Code

#include <iostream>  
  
using namespace std;  
  
void greet(void) {  
cout << "Ave user!" << endl;  
}  
  
void greet\_many\_times(int how\_many\_times) {  
while (how\_many\_times > 0) {  
greet();  
how\_many\_times--;  
}  
}  
  
int main(void) {  
int size\_of\_ego;  
  
cout << "How big is your ego? [km]" << endl;  
cin >> size\_of\_ego;  
greet\_many\_times(1 + size\_of\_ego);  
return 0;  
}

# Example functions

Let’s look at a more serious function now. Its task is to convert a temperature value expressed in Fahrenheit to Celsius.

As you might know, the formula says that:

[°C] = ([°F] − 32) × 5 ⁄ 9

We’ve embedded the formula inside the fahrenheit\_to\_celsius function.

We are going to test our function by forcing it to evaluate results for some characteristic values. We’ll do this by using the test\_the\_function function, which is designed to produce clear and legible output allowing testers (us) to check the correctness of the function.

The complete code is in the editor.

We expect the program to produce the following output:

Fahrenheit 32 corresponds to 0 Centigrade Fahrenheit 212 corresponds to 100 Centigrade Fahrenheit 451 corresponds to 232.778 Centigrade

## Code

#include <iostream>  
  
using namespace std;  
  
float fahrenheit\_to\_celsius(float temp) {  
return ((temp - 32.0) \* 5.0) / 9.0;  
}  
  
void test\_the\_function(float temp) {  
cout << "Fahrenheit " << temp << " corresponds to " <<  
fahrenheit\_to\_celsius(temp) << " Centigrade" << endl;  
}  
  
int main(void) {  
test\_the\_function(32.0);  
test\_the\_function(212.0);  
test\_the\_function(451.0);  
return 0;  
}



**The invocation syntax - supplement**

As you already know, a function may:

* **return a value** when it has a type name in front of its name or it doesn’t have the type name there (in this case the function is considered as returning an int value); such a function has a result and may have an effect, too
* **return nothing** when the void keyword is in front of its name; such a function doesn't have a result and we can expect that it has an effect

We've mentioned that these two kinds of functions differ in the way they use the return statement, but there is also another difference regarding invocations.

Let's assume that we have two functions, schematically shown below:

void void\_function(int par){...; return;}

int non\_void\_function(int par){...; return par \* par;}

Note that:

* the only acceptable form of the void\_function invocation looks like this:

void\_function(2);

* the non\_void\_function can be invoked in the following two ways:

value = non\_void\_function(2);

non\_void\_function(2);

It means that any non-void function's result may be **honoured** by the invoker (the former) and assigned to a variable, or used in any other way, or may be **ignored** by the invoker (the latter) and just forgotten immediately after the return from the function.

# ide effects

Any function needs to have the ability to communicate with its environment. It must be able to receive data (numbers, texts, etc.), process it and share the results. We already know two kinds of communication like this:

* **transferring data to a function using actual parameters** whose values are assigned to formal parameters
* **transferring data from a function using the function's result**; note that only one value may be transferred by such means because the syntax of the return statement allows you to specify only one value

We've said before that the variable defined inside the function's body can be neither used nor accessed from outside the function. Furthermore, there is a special kind of variable called a **global variable**.

Global variables are declared outside any function and thus are accessible for all the functions declared in the same source file.

Note that the variable declaration has to precede the function definition in order to be recognizable by the function.

**Global variables allow functions to get and to provide data of any kind**. If a function modifies any global variable that isn’t using any other mechanism of transferring data, we say that this function has a **side effect**.

Side effects, although useful sometimes, are not recommended and are considered a sign of bad programming style because they make the code difficult to understand.

Take a look at the code in the editor.

The globvar is a global variable. Its declaration is not contained in any function. The func function increments the globvar upon every invocation. We can say that globvar is used to count func's executions.

Note that the main function makes use of this variable too, although it doesn't modify the variable's value. For this reason, we assume that the main function doesn't cause side effects.

We’ll avoid using side effects in future examples. Treat them only as a possibility, not a routine.

## Code

#include <iostream>  
  
using namespace std;  
  
int globvar = 0;  
  
void func(void) {  
cout << "Thank you for invoking me :)" << endl;  
globvar++;  
}  
  
int main(void) {  
for (int i = 0; i < 5; i++)  
func();  
cout << endl << "The function was happy " << globvar <<  
" times" << endl;  
return 0;  
}

# Passing parameters by value

So far, we’ve been assuming that the actual parameter's values are sent to the function and we haven’t said a word about the way back. Our parameters travel to the function's body and we don't expect them to return from there.

Let's do a simple experiment showing if a function is able to change the value of its parameter.

Now take a look at the code in the editor.

As you can see, the can\_i\_change\_my\_parameter function increments its parameter's value. It also reports it to the user. The question is: is the modified value visible outside the function? In other words: **does the formal parameter's change reflect the actual parameter's value?**

Let's compile the code and run it. It should produce the following output:

var = 1 ---------- I've got: 1 I'm about to give back: 2 ---------- var = 1

**output**

Analyse the example carefully and do the experiment yourself.

As you can see, **the formal parameter's value doesn't replace the actual parameter's value upon return from the function**. We can say that the actual parameter has a one-way ticket: it transports a value to the function and doesn't take it up to the invoker.

Don't forget that. This way of communication is based on transferring a value from the invoker to the function. And that’s why this method is called **passing parameters by value**.

## Code

#include <iostream>  
  
using namespace std;  
  
void can\_i\_change\_my\_parameter(int param) {  
cout << "----------" << endl;  
cout << "I have got: " << param << endl;  
param++;  
cout << "I'm about to give back: " << param << endl;  
cout << "----------" << endl;  
}  
  
int main(void) {  
int  
var = 1;  
  
cout << "var = " <<  
var << endl;  
can\_i\_change\_my\_parameter(var);  
cout << "var = " <<  
var << endl;  
return 0;  
}

# Passing parameters by reference

The C++ language doesn’t just offer one method for passing parameters. The second method is called **passing by reference** and allows functions to affect an actual parameter's values.

If you’re going to pass any parameters by reference, you need to announce it while declaring the function. See the example in the editor.

It looks very similar to the previous example, doesn't it? Look carefully and find one specific difference. This is a very important difference, one which radically changes the function's behaviour.

Yes, you're right. The difference is caused by the & sign being placed in front of the parameter's name. Let’s clarify:

* type name – the name parameter is passed **by value**
* type &name – the name parameter is passed **by reference**

Notice that this is the third role that the & character plays in the C++ language. Do you remember the other two?

When a parameter is passed by reference it means that a **formal parameter is just a synonym of an actual parameter**. Every modification made into a formal parameter immediately affects an associated actual parameter. We can informally say that parameters passed by reference have return tickets and bring their modified values back to the invoker.

As you may suspect, the code will produce the following output:

var = 1 ---------- I have got: 1 I'm about to give back: 2 ---------- var = 2

## Code

#include <iostream>  
  
using namespace std;  
  
void can\_i\_change\_my\_parameter(int & param) {  
cout << "----------" << endl;  
cout << "I have got: " << param << endl;  
param++;  
cout << "I'm about to give back: " << param << endl;  
cout << "----------" << endl;  
}  
  
int main(void) {  
int  
var = 1;  
  
cout << "var = " <<  
var << endl;  
can\_i\_change\_my\_parameter(var);  
cout << "var = " <<  
var << endl;  
return 0;  
}

We do **the selection of the passing method for each parameter individually**. You can mix parameters of both kinds if you find it useful. Use the “**passing by value**” if you don't need to share the function's results using the parameter's values, and use “**passing by reference**” in all other cases.

See the example in the editor.

The mixed\_styles function assigns its second parameter with an incremented value of its first parameter. This means that the former parameter is passed by value, while the latter is passed by reference.

The program produces the following output:

var1 = 1, var2 = 2

## Code

#include <iostream>  
  
using namespace std;  
  
void mixed\_styles(int bval, int & bref) {  
bref = bval + 1;  
}  
  
int main(void) {  
int var1 = 1, var2;  
  
mixed\_styles(var1, var2);  
cout << "var1 = " << var1 << ", var2 = " << var2 << endl;  
return 0;  
}

Тhe “passing by reference” method has one important and obvious **limitation. If a parameter is declared as passed by reference (so it is preceded by the & sign) its corresponding actual parameter must be a variable**.

An actual parameter referring to a “passed by value” formal parameter may be an expression in general, so we can use not only a variable but also a literal, or even a function invocation's result.

We say this limitation is “obvious” because the function can’t place a value in something other than a variable. It cannot assign a new value to a literal, or force an expression to change its result. See the snippet in the editor.

All the following invocations are permitted:

* ByVal(i);
* ByVal(i + 2);
* ByVal(intfun(0));

If you want to modify the invocations to take advantage of the ByRef function, you can only use the first one. **All the others will cause a compilation error**.

## Code

#include <iostream>  
  
using namespace std;  
  
void mixed\_styles(int bval, int & bref) {  
bref = bval + 1;  
}  
  
int main(void) {  
int var1 = 1, var2;  
  
mixed\_styles(var1, var2);  
cout << "var1 = " << var1 << ", var2 = " << var2 << endl;  
return 0;  
}

# Passing parameters by value

Is it possible to utilize “passing by value” and be able to propagate the value outside the function despite the one-way nature of this method?

The answer is “yes”. We’re going to show you how to do that, but we want to emphasize that this is not the way we recommend doing it. This method is inherited from the C programming language, the C++ ancestor, and was the only available way of performing two-way parameter based communication. The C language offers nothing similar to the “passing by reference” mechanism, and therefore using other (we might even say slightly risky) methods is fully justified.

The idea is based on transferring a pointer to a value, not the value itself. If you declare a function with a prototype like this one:

void by\_ptr(int \*ptr);

you enable the function to deal with the addresses pointing to int values, and therefore you **give the function the chance to modify the values pointed to by the parameter**.

Now take a look at the code in the editor.

The by\_ptr function takes one parameter, which is a **pointer**, and **accesses the value pointed to by the pointer** using the \* (dereference) operator.

Note: if p is a pointer to a value, the \*p represents the value itself.

In effect, the by\_ptr function modifies the variable without even knowing about its existence.

Some would say that this is the third method of passing parameters. We don't agree – it's still an old “passing by value” method. Only the values are different.

The example code produces the following output:

variable = 2

## Code

#include <iostream>  
  
using namespace std;  
  
void by\_ptr(int \* ptr) {  
\* ptr = \* ptr + 1;  
}  
  
int main(void) {  
int variable = 1;  
int \* pointer = & variable;  
  
by\_ptr(pointer);  
cout << "variable = " << variable << endl;  
return 0;  
}

# Parameters – continue

We’re now going to rewrite our Greet function to make it more flexible. We want it to:

* be able to emit **any greeting**, not only the one predefined in the source code,
* be able to emit the greeting **more than once**, on the invoker's demand.

This means that our new\_greet (this is how we name the new function) has to have two parameters intended to:

* store the greeting,
* store the number of greeting repetitions.

The complete function along with the short main function is here in the editor.

## Code

#include <iostream>  
  
using namespace std;  
void new\_greet(string greet, int repeats) {  
for (int i = 0; i < repeats; i++)  
cout << greet << endl;  
}  
int main(void) {  
new\_greet("Hi!", 5);  
return 0;  
}

# Default parameters – a simple example

The new function makes our life easier, but we want more from it (isn’t that always the case?). We’ve learnt that most of our users require only one greeting at the same time, probably due to the average size of their egos. This may mean that we could use the following form of greeting more often than others:

new\_greet("Good morning", 1);

We may just want to omit the “1”, hoping that the function will be smart enough to guess what we're going to do.

Is it reasonable to expect that any function will be able to behave in such a convenient way?

The answer is yes, but... But we have to inform it first that some of the invocations will not specify all the expected parameters, and indicate which values should be used instead of the absent ones. This mechanism is called “default parameters” and we’ll present its rules by modifying our previous function slightly.

We can modify the second formal parameter's declaration by using a phrase:

= value

to signal that we want the compiler to assume the default value for the parameter when we omit it during the invocation. If we declare the function in the following way:

new\_greet(string greet, int repeats = 1)

the compiler will treat the one-parameter invocations like this one:

new\_greet("Hello");

as if they were (explicitly) written as follows:

new\_greet("Hello", 1);

Now look carefully at the example in the editor.

Note the different forms of invocation. The program will produce the following output:

Hello Hello Good morning Hi

As you can see, the explicitly specified parameter value invalidates the default value.

## Code

#include <iostream>  
  
using namespace std;  
void new\_greet(string greet, int repeats = 1) {  
for (int i = 0; i < repeats; i++)  
cout << greet << endl;  
}  
  
int main(void) {  
new\_greet("Hello", 2);  
new\_greet("Good morning");  
new\_greet("Hi", 1);  
return 0;  
}

# Default parameters – a more complex example

You might want to ask now if it’s possible to have more than one default parameter in one function, i.e. if we may choose the default value not only for the repeats parameter but also for the greet.

Yes, it’s possible. Here’s an example of how to do it in the editor.

This mechanism is useful, but to take full advantage of it you mustn’t forget about the following limitations:

* the order of parameters is very important (in contrast to regular, non-default parameters which may be in virtually any order); intuitively, we can say that non-default parameters must be coded before the default ones; the compiler won't be able to identify the parameters otherwise
* if more than one parameter is declared with a default value and at least one actual parameter is specified in the invocation, the actual parameters are assigned to their formal counterparts in the same order in which they are listed in the function declaration; this means that you are not allowed to use the default value for the first parameter and specify an explicit value for the second

## Code

#include <iostream>  
  
using namespace std;  
void new\_greet(string greet = "Good morning", int repeats = 1) {  
for (int i = 0; i < repeats; i++)  
cout << greet << endl;  
}  
int main(void) {  
new\_greet("Hello", 2);  
new\_greet("Hi");  
new\_greet();  
return 0;  
}

# Anatomy of a function invocation

So far, we’ve been using the term “**function invocation**” without going into detail. We’ve simply assumed that the invocations do their job in an automatic or magical way (or as developers used to say, “automagically”). It's now time to fix this and to explain how this mechanism works.

There is a function that takes its argument, multiplies it by 2 and returns the result to the invoker. The function is invoked three times within the main function. Clear enough? Okay, let’s keep going.

We’re going to look at the code from the compiler' s perspective. It reads the function code, translates it into the machine code and stores the translated code in a separate place in the memory, but the code cannot be used “as is” with no extra steps. Each function's code has to be supplemented with two important elements: a **prologue** and an **epilogue**.

A **prologue is the part of the code implicitly executed before the function**. The prologue is responsible for transferring parameters from the invoker's code and for storing them in a special transient area called a “stack”.

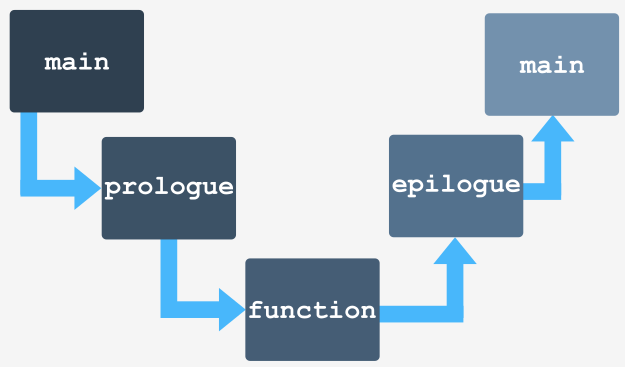
The **epilogue is implicitly executed just after the function's code** and is responsible for transferring the result of the function and for clearing the stack of the values placed there by the prologue.

## Code

#include <iostream>  
  
using namespace std;  
int  
function (int parameter) {  
return parameter \* 2;  
}  
  
int main(void) {  
int  
var = 1;  
var = function (var);  
var = function (var);  
var = function (var);  
cout << var << endl;  
return 0;  
}

**Prologues and epilogues**

Take a look at the diagram below:



This diagram illustrates the flow of control during one function invocation from the previous example. All the other invocations will be carried out in the same way.

This approach has some obvious **advantages**. The **function code, the prologue and the epilogue occupy the same memory space** regardless of how many times the function is invoked. It means that invoking it in this way saves memory and makes your program more compact.

But...

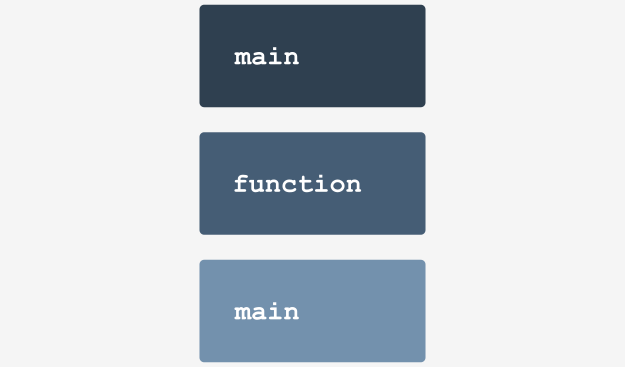
One of the most interesting **paradoxes** of computer programming says that **when a code is compact, it cannot be fast** at the same time; and vice versa, when the code is fast, it cannot be compact. Of course, it's more of a joke rather than a scientific law, but in this case the rule works very well.

Try to imagine that our program is going to invoke the function many times (e.g. hundreds or thousands of times). It may mean that you’ll have to pay a high price (in the sense of time) for all those transfers of control and prologue/epilogue executions. The price is higher when the function is short (i.e. shorter than the the prologue and epilogue).

This example shows that sometimes, in well-defined cases, it would be better to avoid the prologue-function-epilogue chain and to **insert the function's code directly into to the invoker's code**.

**Function inlining**

Take a look at the modified diagram:



This diagram illustrates another approach to the problem of function invocation. When the function is short and fast, and when **we expect the function to be invoked very often**, it’s better (and more time effective) to write the function code at each invocation.

Of course, we’ll have to pay for that. Don't be surprised by the fact that the total code size will be significantly larger than previously.

The tactic of compiling function invocations is called **function inlining**. A function compiled like this is called an **inline function**.

# Inline functions

If you want a certain function to be compiled and invoked as an inline function, you have to mark it in a special way.

You need to precede the function declaration with the inline keyword. Look at the example in the editor.

Luckily for us, the syntax of this construction has some flexibility:

* **it doesn't matter whether the inline keyword is placed** before or after the name of the type; both of the following lines are syntactically correct:

inline int function(int parameter)

int inline function(int parameter)

* if you need to use both the declaration and the definition for the same function, it doesn't matter where you put the inline keyword; it’s correct to use it in the declaration and omit it in the definition; it’s also equally valid to use it in the definition and omit it in the declaration; of course, using the keyword in both places is also valid.

## Code

#include <iostream>  
  
using namespace std;  
inline int  
function (int parameter) {  
return parameter \* 2;  
}  
  
int main(void) {  
int  
var = 1;  
var = function (var);  
var = function (var);  
var = function (var);  
cout << var << endl;  
return 0;  
}

# Different tools for different tasks

Different tasks require different tools, although these tools may have exactly the same name. For example, the tool named “knife” looks very different when it’s used by a surgeon, a cook or a serial killer. We said before that we’re not allowed to have a variable and a function of the same name. It's time to ask if we can have more than one function of the same name.

Yes, we can. It’s natural that we may want to have **different tools of the same name used for different purposes**. For example, we need a function to find the larger of two float numbers. It doesn’t look difficult, right?

Well, we've written this function for you. Take a look at the snippet in the editor.

You might argue that we could have written the function in a more compact way. We agree with you. It’s way too verbose. Feel free to rewrite it in a smarter fashion.

## Code

float max(float a, float b) {  
if (a > b)  
return a;  
else  
return b;  
}

# Max – extended version

Imagine that one day our needs increased and we suddenly wanted to have a very similar function that was able to find the largest of three values. What could we do?

Of course, we can make use of the previous function and, assuming that we want to find the largest of the a, b and c variables, do something like this:

x = max(max(a, b), c);

What, you don’t like it? We don't like it, either. It’s neither nice nor effective. It would be better to forget our old function and write a snazzy new one that was much more tailored to what we want.

It would also be a good idea to name it like the old one: max. This name perfectly illustrates the function's role and purpose.

Take a look at the snippet in the editor.

## Code

float max(float a, float b, float c) {  
int m = a;  
if (b > m)  
m = b;  
if (c > m)  
m = c;  
return m;  
}

# How it works?

The mechanism that allows us to have more than one function of a certain name is called **overloading** (since the one and the same name is *overloaded* with different meanings).

There is one, important limitation: all the **overloaded functions must be clearly distinguishable to the compiler**. The compiler cannot hesitate over which of the overloaded variants need to be used in a particular part of the code.

Our examples leave no doubt. The choice is simple: if the invocation contains three actual parameters, the second variant is chosen. If there are two parameters, the compiler uses the first variant. Any other variant of the invocation is considered an error.

What circumstances does the compiler take into consideration when choosing the one target function (one of the few available)?

* **the number of parameters**: for example, if there are three overloaded functions with (respectively) two, three and four parameters, and the invocation specifies two parameters, only the first of the functions may be used as a target (this function is called '**the best candidate**')
* **the parameters' types**: if there’s more than one function with the same number of parameters, the target function is selected on the basis of the **parameters' type conformity**

Note: **the return type is not taken into consideration** when the compiler is looking for the best candidate for a certain invocation. This should be obvious to you if you remember that the return value of any typed function may be ignored.

In this sense, two functions which differ only in the return type are indistinguishable to the compiler. This means that the following snippet is wrong:

int fnc(int a) {

return a;

}

void fnc(int a) {}

## Code

float max(float a, float b, float c) {  
int m = a;  
if (b > m)  
m = b;  
if (c > m)  
m = c;  
return m;  
}

# How to find the best candidate?

The number of different data types in the C++ language is significant, so the mechanism responsible for finding the best candidate has to use some simplifications so as not to force developers to create a separate function for each different data type. Take a look at the following example in the editor.

Try to answer this question: which of these two overloaded functions is the best candidate for the invocation?

Yes, you're absolutely right (we hope) – it’s the first one, with the int x parameter declaration.

## Code

void play\_with\_number(int x) {  
...  
}  
void play\_with\_number(float x) {  
...  
}  
:  
play\_with\_number(1);  
:

We've changed our example a bit. Can you see the difference?

Which of the functions is the best candidate now? You want to say that it’s the second one, don’t you? Sorry, you're wrong.

**There is no good candidate for the invocation** – why?

**The literal 1.0 is not of type float**. It's of type double (really!). The C++ language compiler tries to promote the types if there is no exact fit (as in our example: obviously a float is not a double).

Unfortunately (to be honest, fortunately) the direction of this type of promotion goes from less precise to more precise, not vice versa. It means, that any float can be promoted to double, but no double can be promoted (or rather degraded) to float.

In this situation, the compiler will inform you that it is unable to find the best candidate and the compilations will fail.

You have two choices:

* you can write the third instance of the play\_with\_number with one parameter of type double
* you can convince the compiler that your literal is of type float; you can do that just by adding the suffix f to the number, like this:

play\_with\_number(1.0f);

The compiler will be satisfied.

## Code

void play\_with\_number(int x) {  
...  
}  
void play\_with\_number(float x) {  
...  
}  
:  
play\_with\_number(1.0);  
:

**A new operator: a three-argument one**

Now's a good time to show you another C++ language operator: ?:. This operator is very original because it requires three arguments. This is how we use it:

expression1 ? expression2 : expression3

This operator works as follows:

* **calculates** the value of the expression1
* if the calculated value is **non-zero**, the operator returns the value of expression2, completely neglecting expression3
* if the value calculated in step 1 is **zero**, the operator returns the value of expression3, omitting expression2.

This means that the result of the following expression:

i = i > 0 ? 1 : 0;

will be calculated in the following way:

* variable i will be assigned with a 1 if its previous value was greater than zero, and 0 otherwise. Note that we can achieve the same effect using a conditional statement:

if (i > 0)

i = 1;

else

i = 0

This form is somewhat more extensive, although we can’t deny that it’s more readable at the same time.

**A new operator: an example**

We can use the new ternary operator (it’s worth mentioning that this is the only ternary operator in the “C++” language) to implement our two-argument max function in a more compact way. Here it is:

float max(float a, float b) {

return a > b ? a : b;

}

**A new operator: an example number 2**

The three-argument variant can make use of the ternary operator too, although we have to admit that the code below is not the most readable:

float max(float a, float b, float c) {

return a > b ? (a > c ? a : c) : (b > c ? b : c);

}

Please use the ?: operator with caution. It doesn’t offer much expressivity, but does destroy a lot of clarity. But it’s up to you what you decide to do.

**Sorting an array**

Think of this chapter as a digression, but a very useful digression. We’re going to tell you about **sorting**. Sorting is very serious and many sorting algorithms have been invented so far which differ a lot in speed as well as in complexity.

We’re going to show you a very simple algorithm, easy to understand, but also not too efficient. It’s used very rarely, and certainly not for large and extensive arrays.

We can say that the array can be sorted in two ways:

* **increasing** (or more precisely – **non-decreasing**) if, in every pair of adjacent elements, the former element is not greater than the latter;
* **decreasing** (or more precisely – **non-increasing**) if, in every pair of adjacent elements, the former element is not less than the latter.

In the following sections, we’re going to sort the array in increasing order, so the numbers will be ordered from the smallest to the largest.

Here’s our array:

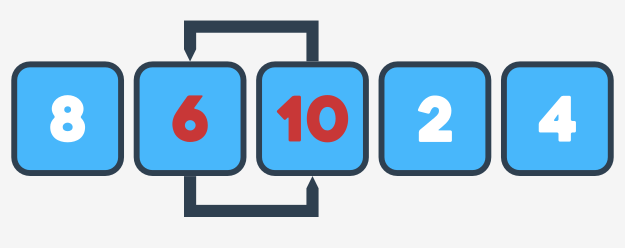


We’ll try to use the following approach: we’ll take the first and second elements and compare them; if we determine that they’re in the **wrong order** (the first is greater than the second), we’ll **swap** them around; if they’re in the right order, we’ll do nothing. A glance at our table confirms the second condition – the elements #1 and #2 are in the proper order, as *8 < 10*.

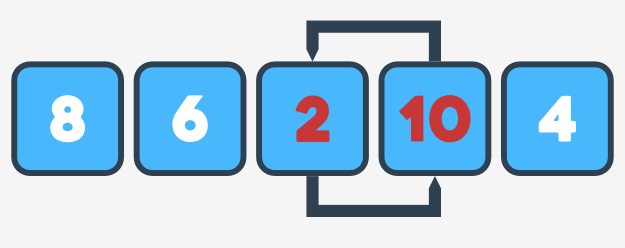
Now look at the second and third elements. They’re in the wrong positions. We have to swap them around. Let's do it.

We can go further and look at the third and fourth elements. Again, this is not what it’s supposed to be like.

We have to swap them around:



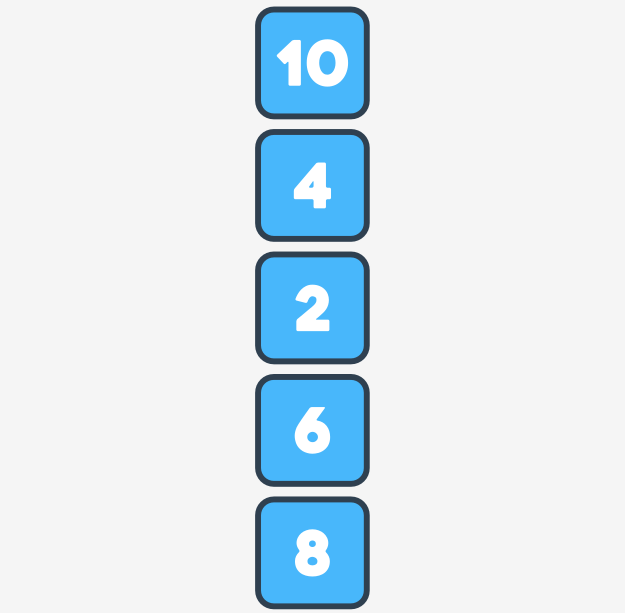
Now we check the fourth and fifth elements. Well, yes, they too are in the wrong positions. Another swap.



The first pass through the array is complete. We’re still far from finishing our job, but something curious has happened in the meantime. The largest element (10) has gone to the end of the array. Note that this is where we want it. All the remaining elements form a picturesque mess, but this one is already home.



Now, for a moment, try to imagine this array in a slightly different way – namely, this

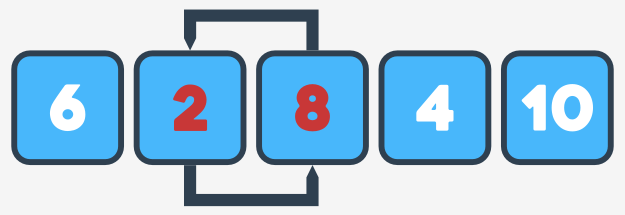


Look – 10 is at the top. We could say that it floated up from the bottom to the surface, just like the bubbles in a glass of champagne. The sorting method derives its name from this same observation – it's called a **bubble sort**.

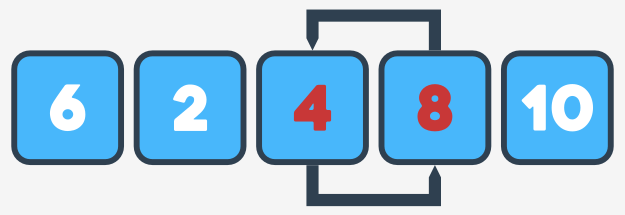
We had a quick break with a glass of champagne, but it’s time to get back to sorting. We do it with pleasure, starting with the second pass through the array. We look at the first and second elements – uh oh, a swap is necessary!



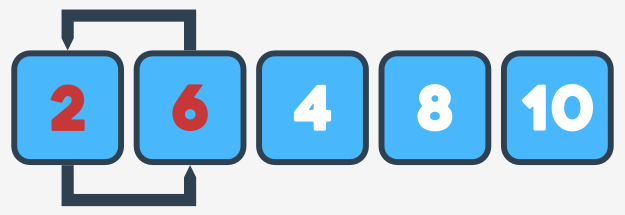
Now the second and third elements: yep, 8 is a bubble and goes up to the surface:



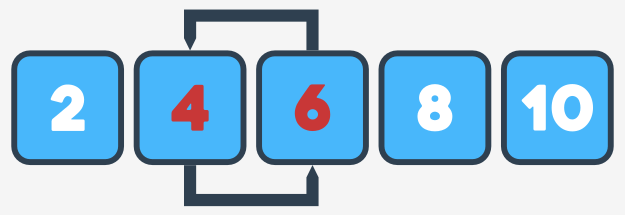
Time for the third and fourth elements: we have to swap them too:



The second pass is finished and 8 is already in place. We start the next pass immediately. Watch the first and second elements carefully - yes, it's time for a swap:



Now 6 wants to find its place. We’ll help it and swap the second and third elements.



Hey! Look! The array is already sorted! We have nothing more to do! This is exactly what we wanted!

As you can see, this algorithm is simple: we compare the adjacent elements and by swapping some of them we achieve our goal.

We’ll try to code in the C++ language all the actions performed during a single pass through the array, and then we’ll think about how many passes we actually need. We haven't analysed this so far, and we will discuss this more a little later.



How many passes do we need to sort the entire array?

We answer this by doing the following: we introduce another variable; its task is to observe if any swap was done during the pass or not; if there was no swap, then the array is already sorted and nothing more has to be done.

We declare a variable named swapped and we assign a value of false to it to indicate that there were no swaps. Otherwise, it will be *assigned true*.

## Code

int numbers[5]; // array to be sorted   
int aux; // auxiliary variable for swaps  
  
// we need 5 – 1 comparisons – why?   
for (int i = 0; i < 4; i++) {  
// compare adjacent elements  
if (numbers[i] > numbers[i + 1]) {  
/\* if we went here it means that we have to swap the elements \*/  
aux = numbers[i];  
numbers[i] = numbers[i + 1];  
numbers[i + 1] = aux;  
}  
}

You shouldn’t have any problem reading or understanding the program. On the next slide you can see a complete program, enriched by a conversation with the user and allowing the user to enter and print elements of the array.

## Code

int numbers[5];  
int aux;  
bool swapped;  
  
do { // we will decide if we need to continue this loop   
swapped = false; // no swap occured yet  
  
for (int i = 0; i < 4; i++)  
if (numbers[i] > numbers[i + 1]) {  
swapped = true;  
aux = numbers[i];  
numbers[i] = numbers[i + 1];  
numbers[i + 1] = aux;  
}  
} while (swapped);

he bubble sort – final version.

## Code

#include <iostream>  
  
using namespace std;  
  
int main(void) {  
int numbers[5];  
int aux;  
bool swapped;  
// ask the user to enter 5 values   
for (int i = 0; i < 5; i++) {  
cout << endl << "Enter value #" << i + 1 << ": ";  
cin >> numbers[i];  
}  
// sort them   
do {  
swapped = false;  
for (int i = 0; i < 4; i++) {  
if (numbers[i] > numbers[i + 1]) {  
swapped = true;  
aux = numbers[i];  
numbers[i] = numbers[i + 1];  
numbers[i + 1] = aux;  
}  
}  
} while (swapped);  
// print results   
cout << endl << "Sorted array: " << endl;  
for (int i = 0; i < 5; i++)  
cout << numbers[i] << " ";  
cout << endl;  
return 0;  
}

**void – the very exceptional type**

We've seen the unusual type called void in our examples a few times. We’ve used it to indicate that a function either doesn't return a result, or doesn't expect any parameters.

According to the following function prototype:

void nothing\_at\_all(void);

the function should be invoked without parameters and return no result. This is how we should invoke it:

nothing\_at\_all();

Despite the fact that the void type doesn't represent any useful value, it’s possible to declare pointers to this type, as in the following example:

void \*ptr;

You may ask how to use **a pointer that points at nothing**, and then ask why have such a pointer at all. This kind of pointer, which is of the type void \*, is called an amorphous pointer to emphasize the fact that it can point to any value of any type. This means that the pointer of type void \* cannot be subject to the dereference operator, so you must not write anything like this:

\*ptr;

If ptr was of type void \*, \*ptr would be of type void and the assignment of a value of type int would be prohibited by the compiler.

However, pointers of type void \* are very useful when you need to have a pointer, but don’t know what it may be used for in the future.

As soon as it becomes clear, the pointer can easily be converted into another pointer of the desired type (of course, a pointer type) which is always possible, and doesn’t cause any loss of accuracy.

**Memory on demand**

In the examples we’ve seen so far, memory management has taken place outside of our consciousness. The parts of memory we’ve used to store values were hidden behind the names of scalars and arrays. They appeared as soon as they had been declared and disappeared when our program ended its operation. All work associated with memory allocation was organized by the compiler and we didn't care how it worked. And this is exactly how it should be – high level languages and their compilers are designed to exonerate the developers' minds of such responsibilities.

It frequently happens that the developer wants to have full control over how much memory is used and when exactly it’s used. This is especially important when you don’t know in advance what the size of the data to be processed is. To manage the allocating and freeing up of memory, the “C++” language gives us two specialized keywords. Here are both of them for you:

* new
* delete

**The *new* keyword**

The new keyword is used to **request the creation of a new memory block**. When the allocated memory is no longer needed and/or utilized, it would be a good idea to return it to the operating system. We do this with the **delete** keyword.

float \*array = new float[20];

int count = new int;

* it needs precise specifications regarding the entity being created; it must be expressed as a type description and if the created entity is an array, the size of the array must be given too (as in the first example);
* the **new returns a pointer** of type conforming to the newly created entity;
* the newly allocated memory area **is not filled (initiated)** in any way, so you should expect it to just contain **garbage**.

**The *delete* keyword**

When we no longer need the memory, we can release it (free it) using the delete keyword in the following way:

delete [] array;

delete count;

What happens here is the following:

* we use the delete [] form if we want to free up the memory allocated for an array, otherwise we use delete,
* you can only release the entire allocated block, not a part of it,
* after performing the delete function, all the pointers that point to the data inside the freed area become illegal; attempting to use them may result in an abnormal program termination.

Now we’re going to show you a complete, albeit not very useful, program that demonstrates the use of both keywords.

* We declare a variable called arr which will point to the data of type float (the pointer's type is float \*); no value is initially assigned to this variable;
* We use **the new keyword** to allocate a block of memory sufficient to store a float array consisting of 5 elements;
* We make use of the newly allocated array (to be precise, a vector) and next we release it using the delete keyword

Pay attention to the fact that the pointer returned by new is treated as if it’s an array. Surprising?

The handling of the **dynamic arrays** (created during the run of the program) is no different than using regular arrays declared in the usual way.

We owe it to the [] operator. Regardless of the nature of the array, we can access its elements in the same way.

## Code

#include <iostream>  
  
using namespace std;  
  
int main(void) {  
float \* arr;  
  
arr = new float[5];  
for (int i = 0; i < 5; i++)  
arr[i] = i \* i;  
for (int i = 0; i < 5; i++)  
cout << arr[i] << endl;  
delete[] arr;  
return 0;  
}

Being able to allocate the amount of memory that’s really needed lets us write programs that can **adapt** themselves to the size of the data currently being processed. Let's go back to the bubble sort algorithm that we showed you earlier. This program assumed that there were exactly 5 numbers to sort. This is obviously a serious inconvenience.

It may happen one day that we want to sort 10,000 numbers or possibly even hundreds of thousands. You can, of course, declare an array of the maximum predictable size, but it would be unreasonable. A much better way is to ask the user how many numbers will be sorted and then allocate an array of the appropriate size.

Let's try to start with a simpler example. In the following program, we allocate an array containing 5 elements of type int, set their values, sum them up and, finally, release the previously allocated memory.

## Code

int \* tabptr, sum = 0;  
  
tabptr = new int[5];  
for (int i = 0; i < 5; i++)  
tabptr[i] = i;  
sum = 0;  
for (int i = 0; i < 5; i++)  
sum += tabptr[i];  
delete[] tabptr;

The improved bubble sort program is in the editor.

We encourage you to compile and run the program yourself.

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
int \* numbers, how\_many\_numbers;  
int aux;  
bool swapped;  
  
cout << "How many numbers are you going to sort? ";  
cin >> how\_many\_numbers;  
if (how\_many\_numbers <= 0 || how\_many\_numbers > 1000000) {  
cout << "Are you kidding?" << endl;  
return 1;  
}  
numbers = new int[how\_many\_numbers];  
for (int i = 0; i < how\_many\_numbers; i++) {  
cout << "\nEnter the number #" << i + 1 << ": ";  
cin >> numbers[i];  
}  
do {  
swapped = false;  
for (int i = 0; i < how\_many\_numbers - 1; i++)  
if (numbers[i] > numbers[i + 1]) {  
swapped = true;  
aux = numbers[i];  
numbers[i] = numbers[i + 1];  
numbers[i + 1] = aux;  
}  
} while (swapped);  
cout << endl << "The sorted array:" << endl;  
for (int i = 0; i < how\_many\_numbers; i++)  
cout << numbers[i] << " ";  
cout << endl;  
delete[] numbers;  
return 0;  
}

**Congratulations! You have completed Module 3.**

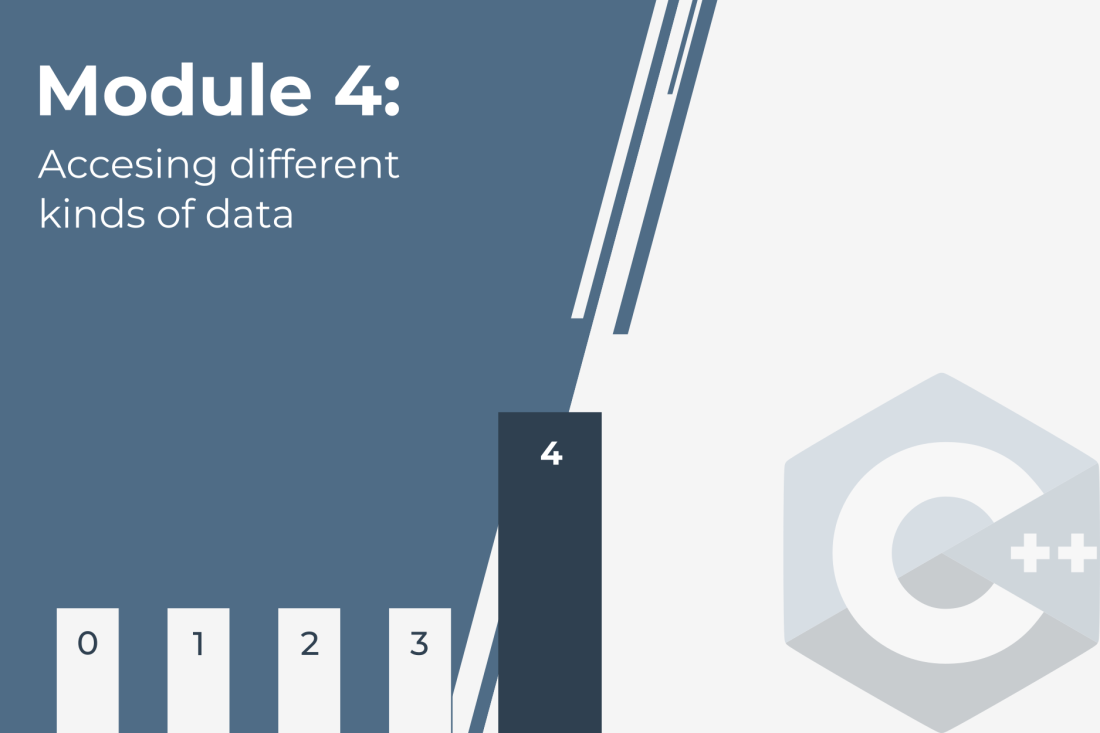
Well done! You've reached the end of Module 3 and completed a major milestone in your C++ programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 3:

* designing, declaring, and invoking functions;
* pointers;
* different methods of passing parameters and their purpose;
* default parameters;
* inline functions;
* overloaded functions;
* sorting;
* memory on demand.

**C++ Essentials: Module 4**

**In this module, you will learn about:**

* arrays of pointers;
* conversions;
* strings: declarations, initializations, assignments;
* strings as an example of objects: (methods and properties)
* using and declaring namespaces;
* dealing with exceptions.



**Arrays of pointers**

There’s nothing preventing the array elements from being pointers. Imagine we need a dynamic array. The array is two-dimensional and we can’t predict how many rows it’ll have, or how many columns.

We only know that the number of columns is stored in the cols variable and the number of rows in the rows variable.

We’re not even going to attempt to allocate the array in the following way:

new int[rows][cols];

It looks quite reasonable at first glance, but it’s actually wrong, because the compiler doesn't know anything about the size of the rows during compilation and it won’t be able to generate the correct indexing code, and the only result we can expect from this piece of code is the compilation error, and a small dose of personal disappointment.

So what should we do if we want our array to be fully configurable and easily accessible with a clear and simple notation like this:

ptrarr[r][c]

Is this possible?

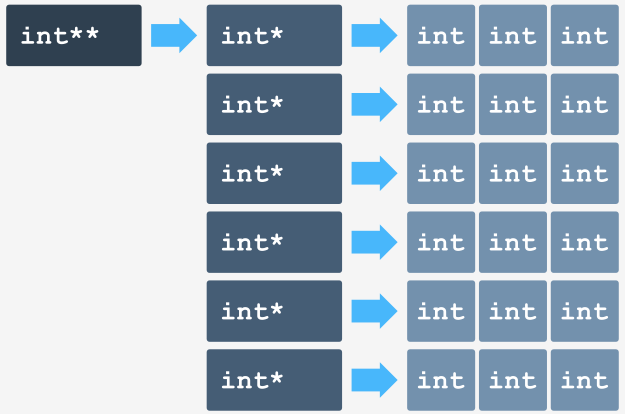


Yes, it is possible.

This is how we do it:

* we store the pointer to the beginning of every row separately so that we can reach each row like any other vector. How do we store these pointers? In the array, of course! We’ll call it **the array of rows**; every row will have as many elements as columns of the desired array;
* every element in the array of rows will be a **pointer to a separate row**;
* we need one more pointer to point to the array of rows – we call it ptrarr.

Let's look at the figure illustrating the essence of our idea.



What’s the type of the variable ptrarr?

First, let's say what type it isn't: certainly it's not a pointer to an int (int \*). This is because the result of dereferencing the ptrarr is not an int but a pointer to int.

This means that the type of ptrarr is "**a pointer to a pointer to int**" which we denote as "int \*\*"

We can write a complete declaration now (look at the editor).

## Code

int \*\*ptrarr;

Once we’ve declared the pointer, we can allocate the array of rows. This is done as follows - the code is in the editor.

Let us explain the meaning of this clause.

Firstly, the pointer returned by new is of type int \*\* (because it’s a pointer to pointer to int) and is assigned to ptrarr.

Secondly, the elements of the array of rows will be pointers to the rows, so their type is int \*.

## Code

ptrarr = new int \* [rows];

Finally, we need to allocate memory for every row and store the resulting pointer inside the right element of the array of rows.

By far the easiest way to do this is to use a loop.

Look at the code in the editor.

## Code

for (int r = 0; r < rows; r++)  
ptrarr[r] = new int[columns];

It’s surprisingly simple to use this kind of array.

For example, if we want to assign 0 to the element lying in row r, column c, we'll do it this way:

ptrarr[r][c] = 0;

How does it work?

* the ptrarr[r] expression is interpreted as \*(ptrarr + r), which means the **dereferencing of the element** pointing to the selected row;
* the pointer is **dereferenced once more** so the entire indexing expression looks as follows:

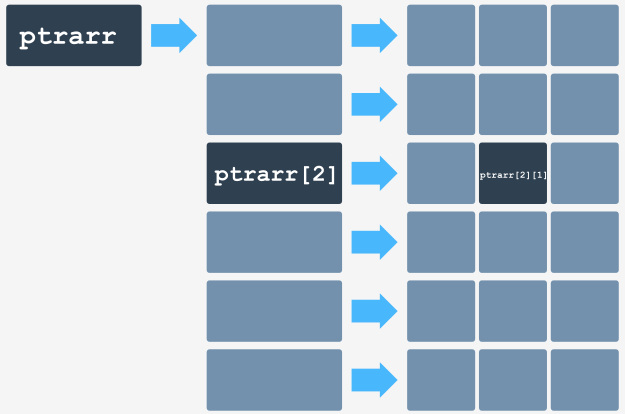
\*(\*(ptrarr + r) + c)

and this is simply the desired value of type int.

The process of dereferencing this element:

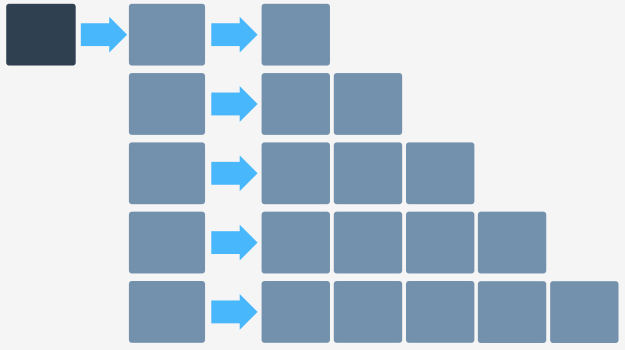
ptrarr[2][1]

is shown in the figure:



# Triangular matrices

The advantage of such arrays is that, unlike ordinary arrays, every row may be of a **different length**. This is useful for the algorithms that don’t need the entire array to run but only a slice of it. It refers specifically to ***triangular matrices***. Such an array can be allocated in this way:



Take a look at the code in the editor - this is how it’s built.

Pay attention to how the “**triangularity**” is obtained (the size of the allocated memory block depends on the row number) and how the value assigned to the elements reflects their location in the array.

Keep in mind that the order of freeing up memory is in reverse to the order of allocation.

## Code

#include <iostream>  
  
using namespace std;  
  
int main(void) {  
int rows = 5, cols = 5;  
int \*\* arr;  
// allocate and initialize the array  
arr = new int \* [rows];  
for (int r = 0; r < rows; r++) {  
arr[r] = new int[r + 1];  
for (int c = 0; c <= r; c++)  
arr[r][c] = (r + 1) \* 10 + c + 1;  
}  
// print the array  
for (int r = 0; r < rows; r++) {  
for (int c = 0; c <= r; c++)  
cout << arr[r][c] << " ";  
cout << endl;  
}  
// free the array  
for (int r = 0; r < rows; r++)  
delete[] arr[r];  
delete[] arr;  
return 0;  
}

# Implicit conversions

As you already know, **implicit conversions may occur without your input**. The compiler decides where they should be performed. Of course, we need to be aware of this, and we need to able to find all the circumstances that can cause the compiler to convert our data.

We can say that implicit conversions affect data of fundamental (basic) types and are most often associated with a change of **the internal representation**.

This kind of conversion happens when you assign, for example, int to float or float to long or double to int. There are lots of other possible combinations.

Let’s list some of the contexts where implicit conversions play an important (although almost invisible) role:

* a value is used as **part of a complex expression** built of many values of different types (see example 1 in the editor where the Short variable is converted into the int type to be compatible with the Int variable)
* a value plays **the role of a logical condition** within instructions like if, while, do, etc (see example 2 where the Double variable has to be converted into int to be usable for controlling the path of execution)
* a value is **subject to assignment** and is used to:
  1. change the value of a variable (example 3: int is converted to float)
  2. set the value of a formal parameter (example 4: the actual parameter of type float is converted into the formal parameter of type int)
  3. specify the return value of a function (example 5: the value of type int is returned by the function having the return type specified as float)

Of course, we haven't listed all the possibilities. See if you can identify any others.

## Code

int Int = 1;  
short Short = 2;  
long Long = 3;  
float Float = 4.0;  
double Double = 5.0;  
  
int f(int x) {  
return x;  
}  
  
// example no. 1  
Int = Int + Short;  
// example no. 2  
if (Double)  
Double--;  
// example no. 3  
Float = 1;  
// example no. 4  
f(Float);  
// example no. 5  
float g(void) {  
return -1;  
}

**What is a conversion?**

A conversion is the act of changing the nature of the data without (if possible) changing the value. One of the simplest examples of conversion is a change of data type. Take a look at the snippet below:

long data = 1;

The literal of value 1 and of type int is converted to data of the same value but of type long. This means that the internal representation of the subjective data may be changed (in this particular case, the number of bits used to store the value could be increased) but the value itself remains untouched.

You can avoid conversion by coding the literal in a clearer way, e.g. like this:

long data = 1L;

The suffix 'L' (or 'l' – they’re interchangeable) says that the literal is explicitly defined as long.

All possible conversions are divided into two classes:

* **automatic conversions** which are performed somewhat behind our backs: we don't require them but the compiler knows where they need to be applied and applies them without asking our permission; for this reason they’re also called **implicit conversions**
* **explicit conversions** which are performed at the developer's command and expressed using special language; the compiler obeys the programmer's will and converts data according to it; this kind of conversion is often called **typecasting**, due to its syntax construction

# Explicit conversions

In general, the C++ language give us two ways to specify explicit conversions:

* the so-called **C-style casting** (named after the C++ predecessor, the C programming language, which used, and still uses, a syntax of that kind); in this case, we specify the conversion in the following form:

(new\_type\_name) expression\_of\_old\_type

* the so-called **functional notation**, which is a native C++ syntax convention, unavailable in C (named because of its similarity to the well-known function invocation); in this case the name of a new (target) type is treated like a function and the conversion takes the following form:

new\_type\_name(expression\_of\_old\_type)

Both forms are the same when applied to the values of standard (basic) types, but functional notation can also be used for objects and when doing that, it can have more than one value.

By the way, these two options are not all of the options available. We’ll return to this issue later when we talk about classes.

Take a look at the example code in the editor.

It illustrates both kinds of explicit conversions.

Which one do you like more?

Oh, we forgot to mention that the program outputs '4' to the screen. But you knew that without our help, didn't you?

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
float f = 3.21;  
double d = 1.23;  
int k = int(f) + (int) d;  
cout << k << endl;  
return 0;  
}

# Conversions – gains and losses

Every time a conversion happens, the compiler does its best to preserve the original value, but that’s not always possible. We’re going to show you some examples which demonstrate possible gains and losses arising from conversions.

Let's analyse some of the possibilities. We start with integral types.

A good scenario is when **the length of the memory representation remains the same or increases**; we can be confident then that the original value will be preserved. We can expand it with zero bits to fill the target memory space and the sign bit may be moved to its new position, but the value itself will not change.

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
short s = 32767;  
int i = s;  
if (i == s)  
cout << "equal" << endl;  
else  
cout << "not equal" << endl;  
return 0;  
}

We’re now going to test the worse scenario. Take a look at the example in the editor.

We want to transfer the maximum value allowed for the int variables to the short variable.

A good scenario is when **the length of the memory representation remains the same or increases**; we can be confident then that the original value will be preserved. We can expand it with zero bits to fill the target memory space and the sign bit may be moved to its new position, but the value itself will not change.

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
int i = 2147483647;  
short s = i;  
  
if (i == s)  
cout << "equal" << endl;  
else  
cout << "not equal" << endl;  
return 0;  
}

Let’s do two tests for the floating point values. The test program is almost the same. You can see it in the editor.

We’ll convert the float value into a double value. As you know, doubles have not only a wider range than floats, but also better accuracy (precision).

We expect the conversion to be successful and both, original and target variables, will store the same value.

And we're right. The program will output 'equal'.

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
float f = 1234.5678;  
double d = f;  
  
if (d == f)  
cout << "equal" << endl;  
else  
cout << "not equal" << endl;  
return 0;  
}

s before, we've reversed the direction of the conversion, but here the range isn’t a problem. The value being converted is small enough to be stored in any float variable. The issue here is precision.

Floats can’t store as many significant digits as specified in the literal assigned to the d variable.

We won't lose any value. Does this mean that we'll suffer from a loss of accuracy?

Unfortunately, yes. It outputs “not equal” to the screen.

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
double d = 123456.789012;  
float f = d;  
  
if (d == f)  
cout << "equal" << endl;  
else  
cout << "not equal" << endl;  
return 0;  
}

The example in the editor seems to be obvious. Conversions from floating point types to integral types always cause a loss of accuracy. There’s no escaping this. We always lose the fractional part of a float number.

But be prepared also for the fact that, when the float is extremely large (or extremely small) you’ll also experience a loss of value. This applies to the values beyond the scope of the target integral type.

Try to compile and run the example program yourself. The results will surprise you, we promise you that.

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
float f = 123.456;  
float g = 1e100;  
int i = f;  
int j = g;  
  
cout << i << endl;  
cout << j << endl;  
return 0;  
}

To avoid both types of losses, the compiler uses a strategy called **promotion**. A promotion involves the conversion of data taking part in an evaluation to the safest type.

For example, when one int and one short are used in the same expression, we can expect that the value of the narrower range (short) will be promoted to the type with the wider range (int) and there won’t be any loss of value.

Similarly, when a float meets a double in the same expression, the float will be promoted to a double.

Formally, all the promotions are conducted according to the following set of rules (the rules are applied in the order below until all the data used in a particular expression has the same type – this condition is very important!):

* data of type char or short int will be converted to type int (**this is called an integer promotion**);
* data of type float undergoes a conversion to type double (**floating point promotion**);
* if there’s any value of type double in the expression, the other data will be converted to a double;
* if there’s any value of type long int in the expression, the other data will be converted to long int;

If the context in which the expression is calculated requires another type than that resulting from the conversions, the last conversion will be to the type requested by the context.

Knowing how the implicit conversions (sometimes called **automatic**) work, we can analyse the process of calculating the value of this expression along with the complete program (look at the editor).

We can predict that the following implicit conversions will take place:

* promotions go first, resulting in the following conversions:

int(Short) + int(Char)

* the sum of Short and Char as well as the Float variable will be converted to double, that is:

double((int(Short) + int(Char)) + double(Float))

* the final sum will be calculated as a double, but because of the context (arising from the type of the left argument of the = operator) which is int, another conversion into an int type takes place; hence, the final form of the expression looks like this:

int(double((int(Short) + int(Char)) + double(Float)))

Now try to do some experiments yourself, by changing the set of variables involved in the expression and type of the target variable. Continue the experiments until you're sure you understand the operation of conversions.

## Code

#include <iostream>  
  
using namespace std;  
int main(void) {  
int Int = 2;  
char Char = 3;  
short Short = 4;  
float Float = 5.6;  
  
Int = Short + Char + Float;  
cout << Int << endl;  
return 0;  
}

# What is a string?

A **string** (in the C++ language sense) is not a cord or a ribbon, but a **set of characters**. You might ask what we can use it for if we have a data type like char. That's a good question.

The char variables are useful (and generally all we need) when we want to process single characters, but they’re extremely difficult when we have to deal with data containing all kinds of names (e.g. surnames, city names, street names, etc) or just text (e.g. agreements, statements or simply books or emails). Processing that kind of data as a set of chars resembles spelling – it’s tedious and slow. It’s much easier to treat all characters as a whole, stored, assigned and processed at the same time.

At our current level of knowledge, we can assume that string is a type like int or float. It's not quite true, but we can neglect all the boring details and start from this point. In due time, we’ll show you why the string is somewhat different to an ordinary data type. It’ll also be a good opportunity to show you some interesting traits of **object-oriented programming**.

Let's start with a simple declaration. Take a look at the line below:

#include <string>

string pet\_name;

We’ve declared a variable named pet\_name. It’s designed to store the names of animals.

We’ve declared a variable named pet\_name. It’s designed to store the names of animals.

Note one very important fact: **the string type is not a built-in type** (like int which can be used at any time with no special preparation). If we want to use strings in our code, **we have to put the #include directive** at the top of our program and **request a header file named string** to be included during the compilation.

This lack of directive will cause a compilation error. Remember – you’ve been warned.

There’s another important fact that applies to strings. The word “*string*” is not a keyword unlike other type names like int or float.

**Initializing a string**

**A string can be initialized** in a way that is identical to the one used for other regular types i.e. by using an assignment operator followed by a **string literal** (a set of characters surrounded by double quotes). But don't forget: **apostrophes are for character literals only. Strings always use quotes.**

string pet\_name = "Lassie";

In effect, the newly created variable named pet\_name will be assigned with a string consisting of the characters “Lassie”. This isn’t the only information stored inside the pet\_name variable, but we don't want to get ahead of the facts.

Notice that sometimes we have to use the word string in two different meanings: as **a type name** (as in a variable of type string) and as **an entity consisting of a finite number of characters**.

There’s also another way to initialize string variables, more suited to the style of object programming. Take a look at the snippet:

string pet\_name("Lassie");

**We don't use the assignment operator here**. We use a syntax that clearly resembles the function invocation, with a pair of parentheses and an argument within. You may ask if there’s a real function named pet\_name being invoked. The answer is “yes and no”. Yes because there’s a specialized function responsible for creating strings and the function is being invoked every time you want to create a new string. No because you're not allowed to directly invoke the function like any other regular function.

Both forms of initialization are the same, and their results are exactly the same.

You can also use a value stored in any string variable to initialize the newly declared variable like in the example:

string is\_home = pet\_name; string has\_returned(pet\_name);

Both forms (assigning and functional) are permissable.

# String operators: +

Just like any other data type, **the string type has its own operators.** One of the most important and most frequently used is the + operator. Don't be misled – it doesn’t add strings in the same way as traditional arithmetic addition. Instead, **it concatenates strings** (i.e. it glues them together, giving a new string which has arisen out of the values of its arguments).

It may seem obvious, but it’s worth mentioning here: **the concatenating + is not commutative** (a + b aren’t always equal to b + a) in contrast to the adding +.

Now take a look at the program in the editor.

Try to predict its output.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string good = "Jekyll", bad = "Hyde";  
cout << good + " & " + bad << endl;  
cout << bad + " & " + good << endl;  
return 0;  
}

The + (concatenation) operator has one important limitation. **It cannot concatenate literals**. It can concatenate any variable with a literal, a literal with a variable, and obviously a variable with another variable, but concatenating literals is something the operator will never do for us.

The program in the editor contains an **error**.

Try to spot it. Can you see it?

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string s;  
s = "A" + "B";  
s = s + "C";  
s = "B" + s;  
cout << s << endl;  
return 0;  
}

# String operators: +=

We hope you aren’t surprised with the news that the + (concatenation) operator, just like its arithmetic cousin, may be used as a short-cut operator.

The example in the editor explains the whole thing, doesn't it?

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string the\_question = "To be ";  
the\_question += "or not to be";  
cout << the\_question << endl;  
return 0;  
}

# Inputting strings

As you’ve probably noticed, outputting strings is easy and generally doesn’t cause any problems. You can place a string variable among other expressions intended to be displayed on the screen by using the cout stream, and its content will be sent character by character.

Inputting strings is a bit more complicated, due to the fact that the cin stream treats spaces (to be precise, not only regular spaces but also all so-called **white characters**) as delimiters, demarcating limits between data. It means that you may run into trouble if you want to input and store a string containing white characters.

Try to compile and run the program in the editor.

Enter some words from the keyboard and separate them with spaces. You may write something like this:

To be or not to be

**input**

and press *Enter*. Don't be surprised if you only see the first word printed on the screen (e.g. To). The cin stream is convinced that you have entered the space in order to mark the end of the string. It's not an error. It's by design.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string line\_of\_types;  
cin >> line\_of\_types;  
cout << line\_of\_types << endl;  
return 0;  
}

If you want **to input a whole line of text** and treat the white characters just like any other character, you have to use the getline function. This function gets/reads all the characters entered as-is and does not favour any character except from the char representing the key, which marks the end of the line. As a result, all the characters entered before pressing the key will be input as one string.

Now take a look at the modified program in the editor.

Compile and run it. Make sure that Prince Hamlet's famous question is not cut off any more.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string line\_of\_types;  
getline(cin, line\_of\_types);  
cout << line\_of\_types << endl;  
return 0;  
}

# Comparing strings

Strings (just like any other data) may be **compared**. The simplest case occurs when you want to check if two variables of type string **contain identical strings**.

Something similar happens when you enter your password on logging into an account. The password you entered is compared to the password stored in the system and if both strings are equal, access is granted. (The real process is obviously more complex, but our description is good enough to show you the idea of comparing strings).

If you want to check if two strings (i.e. variables of type string) contain the same string (i.e. a chain of characters) you can use the old good == operator. It can answer your question and satisfy your curiosity.

Take a look at the program in the editor.

The program illustrates the process of entering the password and checking if it’s valid.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string secret = "abracadabra";  
string password;  
cout << "Enter password:" << endl;  
getline(cin, password);  
if (secret == password)  
cout << "Access granted" << endl;  
else  
cout << "Sorry";  
return 0;  
}

Of course, you can **compare two strings** in more flexible ways too. All the operators designed to compare data are at your disposal: > < >= <= !=. You can check if one of the strings is greater/lesser than the other, but remember that these comparisons are carried out in alphabetical order where 'a' is greater than 'A' (sic) and obviously 'z' is greater than 'a', but less obviously, 'a' is greater than '1'.

You can use this kind of string comparison e.g. to determine alphabetical order. The code in the editor gets two lines of text and prints them in alphabetical order.

You can use it to clear away any doubts about how the computer understands the alphabet.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string str1, str2;  
cout << "Enter 2 lines of text:" << endl;  
getline(cin, str1);  
getline(cin, str2);  
cout << "You've entered:'" << endl;  
if (str1 == str2)  
cout << "\"" << str1 << "\" == \"" << str2 << "\"" << endl;  
else if (str1 > str2)  
cout << "\"" << str1 << "\" > \"" << str2 << "\"" << endl;  
else  
cout << "\"" << str2 << "\" > \"" << str1 << "\"" << endl;  
return 0;  
}

# Comparing strings – the object-oriented approach

Strings offer another more complex, but also more powerful, method of comparison. We can make use of it if we have some background knowledge of how objects manipulate their data. This’ll be a good introduction to practising the object-oriented programming style.

In the classical approach, we usually have **some data and a set of functions** operating on the data. We’re used to the following way of initiating data processing:

function(data)

In the object-oriented approach, we use slightly different terminology. Both the data and the functions are embedded in the so-called object. The data is the object's **property** or **member variable**, and the functions are the object's **methods** or **member functions**. We’ll return to these terms soon and explain their meaning in more detail.

If we want a particular method (member function) to process data embedded within an object, we activate the member function for the object. It looks like this:

object.member\_function()

Let’s start with a very simple example. We’re going to rewrite the password checking program to show you one of the numerous member functions that exist in every string object (we’re so advanced now that we no longer use the term "*variable*" in this context).

The member function is called “compare” and the name speaks for itself: it’s designed to compare a string with another string. The member function returns 0 (zero) if the strings are identical.

Take a look at the program in the editor. We’ve replaced the occurrence of the == operator with the equivalent compare member function activation. Now look at it carefully. Note the expression:

secret.compare(password)

It says: activate the compare member function for the secret object in order to compare the string stored within secret with the string stored within password. We can write this expression in the following way:

password.compare(secret)

without changing the program's behaviour.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string secret = "abracadabra";  
string password;  
cout << "Enter password:" << endl;  
getline(cin, password);  
if (secret.compare(password) == 0)  
cout << "Access granted" << endl;  
else  
cout << "Sorry";  
return 0;  
}

Of course, the possibilities of the compare member function don’t stop at checking the identity of the strings. The function can also diagnose all of the possible relations between two strings. Here’s how it works:

* str1.compare(str2) == 0 when str1 == str2
* str1.compare(str2) > 0 when str1 > str2
* str1.compare(str2) < 0 when str1 < str2

In the object-oriented approach, we use slightly different terminology. Both the data and the functions are embedded in the so-called object. The data is the object's **property** or **member variable**, and the functions are the object's **methods** or **member functions**. We’ll return to these terms soon and explain their meaning in more detail.

Note, that the direction of the inequality is the same for both functional and objective notations.

Again, let’s rewrite one of our previous programs just to demonstrate the look of the code. The new version is in the editor.

Can you figure out how it works?

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string str1, str2;  
cout << "Enter 2 lines of text:" << endl;  
getline(cin, str1);  
getline(cin, str2);  
cout << "You've entered:'" << endl;  
if (str1.compare(str2) == 0)  
cout << "\"" << str1 << "\" == \"" << str2 << "\"" << endl;  
else if (str1.compare(str2) > 0)  
cout << "\"" << str1 << "\" > \"" << str2 << "\"" << endl;  
else  
cout << "\"" << str2 << "\" < \"" << str1 << "\"" << endl;  
return 0;  
}

# Substrings

All string operations we’ve shown you thus far have **referred to whole strings** i.e. both strings were taken integrally, from the first to the last character. The strings allow themselves to be processed in a more precise way when only selected parts of them are taken into consideration. A part of a string is called **a substring**.

If we want to create a new string consisting of characters taken from another (or even the same) string's substring, we can use a member function called substr, and its simplified, informal prototype looks like this:

newstr = oldstr.substr(substring\_start\_position, length\_of\_substring)

The substring of any string is defined by two “**coordinates**”:

a place where the substring begins (specified by the substring\_start\_position parameter) and its length (specified by the length\_of\_substring parameter).

Note that characters within a string are numbered, and the first character is that number 0. Therefore, if a string contains n characters, the last one is the number n – 1.

Both parameters have default values. This enables us to use the function in a more flexible way. So:

* s.substr(1,2) describes a substring of the s string, starting at its second character and ending at its third character (inclusively)
* s.substr(1) describes a substring starting at the second character of the s string and containing all of the remaining characters of s, including the last one; the omitted length\_of\_substring parameter defaults to covering all the remaining characters in the s string
* s.substr() is just a copy of the whole s string (the substring\_start\_position parameters defaults to 0)

Take a look at the example on the right. Try to predict its output.

Yes, it's BEFABCDEF.

And now pay attention, because this is very important: **getting a substring requires you to be accurate**. You mustn't define a substring that doesn't wholly fit inside the original string (e.g. starts beside the original string's end). Doing this will cause your instruction to be rejected, and is what we call **an exception**.

We’ll discuss exceptions in the next section – stay tuned.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string str1, str2;  
str1 = "ABCDEF";  
str2 = str1.substr(1, 1) + str1.substr(4) + str1.substr();  
cout << str2 << endl;  
return 0;  
}

# The length of a string

Every string has a length. Even an empty string (containing no characters at all) has a length of zero. It's obvious that at some point we may want to know how long a particular string is.

This information is provided by two twin member functions. Their names are different, but their behaviours are identical. We can say that these functions are **synonyms**.

Their informal prototypes look like these:

int string\_size = S.size();

int string\_length = S.length();

Of course, both of these int variables are equal.

**Both functions return a value equal to the number of all the characters currently stored within a string**. Note that the substr function returns a string. This means that the resulting string retains all its original traits and therefore has its own member functions like substr or size.

The program on the right, although complicated, is correct. Try to predict its output.

What did you say? 2? That's a good answer.

We encourage you to experiment. Compile and run the program with the line:

int pos = 1;

changed to:

int pos = 2;

Can you explain what happened to your program during execution and why?

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string str = "12345";  
int pos = 1;  
cout << str.substr(pos).substr(pos).substr(pos).size() << endl;  
return 0;  
}

# More detailed string comparison

Because we already know how to define and use substrings, we can take a little step back and show you some other variants of the compare functions. They allow us not only to compare whole strings, but also their substrings. Their prototypes are clear to understand and easy to use (well, we hope you agree). Here are two of them (as usual, we give them to you informally):

S.compare(substr\_start, substr\_length, other\_string)

S.compare(substr\_start, substr\_length, other\_string, other\_substr\_start, other\_substr\_length)

We’ll start with the first, obviously simpler case. This variant of the compare member function compares the entire other string to the substring of the source string. This means that the following snippet will output 0 (the strings are equal):

string S = "ABC";

cout << S.compare(1, 1, "B");

The second variant allows us to use only a part of the other\_string. Similarly, the following snippet will also output 0:

string S = "ABC";

cout << S.compare(1, 1, "ABC", 1, 1);

Now analyze the code on the right. Your task is to guess the number output by the program. Good luck!

PS. Do you think it's -1? You're right.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string S = "ABC";  
cout << S.compare(1, 1, "BC") + S.compare(2, 1, S, 2, 2) << endl;  
return 0;  
}

# Finding strings inside strings

Sometimes we have not only to extract a substring from a particular string, but also (which is much more complex) **to find a substring** within another string, taking into account the possibility that it may fail (not all haystacks contain needles).

Strings can do this for us. They can search for a substring or for a single character. For this purpose, we need to use one of the variants of the find member function. Two of them are particularly useful:

int where\_it\_begins = S.find(another\_string, start\_here);

int where\_it\_is = S.find(any\_character, start\_here);

In both variants, the start\_here parameter defaults to 0, so when you omit it, the string will be searched from the **beginning**.

The result returned by the functions points to **the first location within the string where the searched string begins or where the searched char is located** (depending on the variant). If the search fails, both functions return a special value denoted as string::npos (we’ll discuss this syntax convention soon). You can use it to check if your haystack contains the desired find.

Of course, it's now time for an example. Check it out in the editor. In our opinion, its operation is quite predictable. What do you think?

Try to change the code to test some of the possible results.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string greeting = "My name is Bond, James Bond.";  
string we\_need\_him = "James";  
if (greeting.find(we\_need\_him) != string::npos)  
cout << "OMG! He's here!" << endl;  
else  
cout << "It's not him." << endl;  
int comma = greeting.find(',');  
if (comma != string::npos)  
cout << "Interesting. He used a comma." << endl;  
return 0;  
}

# How big is the string actually?

We've said previously that we can retrieve information regarding the string's length by using the size of length member functions, but we didn't mention that the value obtained from the function is only partially true. It tells us **how many characters are currently stored within the string**, but says nothing about the memory occupied by the buffers allocated to that string. Of course, the buffer size has to be greater than the string itself if we want the whole machine to operate efficiently.

Every time we extend the string, e.g. concatenating another string to it, the new content is placed in the buffers. If the buffers are large enough, then extending the string doesn't require the buffers to extend. Of course, when the buffers are full and the string is being extended again, the buffers are reallocated to fit the new content. This process is fully **transparent** and in most cases, you don't even need to know about it.

Obviously, if you want, you can control the process when you know the nature of the particular string and its purpose. The strings offer some means of checking how the buffers are working and to recognize the condition of the string.

For example, you can ask any string for the size of the currently allocated buffers. The answer comes from the member function called capacity. You can use it in the following way:

int currently\_used = S.capacity();

Of course, the result of the function is always greater or equal to the string's size/length (as you already know, these words are synonymous in this context).

Every string can grow, but there’s a limit to its extension and in this case, the limit is not the sky, but a value defined for all the strings in the implementation. You can find it out by using another function called max\_size (it does exactly what it says). You can expect that the value will be really big – bigger than you probably need in typical programs. Here’s an example of how to use it:

Note: this value is common for all the strings you use in your program. Obviously, the previous value may be different for every instance of the string.

The program on the right demonstrates the coexistence of all these three values. Compile and run it on your own, and also change the variable's values and test its behaviour.

Don’t try to increase the upper limit of the for loop by a significant amount. It may crash your system. What, you don’t believe us? Well then, try it!

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
void print\_info(string & s) {  
cout << "length = " << s.length() << endl;  
cout << "capacity = " << s.capacity() << endl;  
cout << "max size = " << s.max\_size() << endl;  
cout << "---------" << endl;  
}  
  
int main(void) {  
string the\_string = "content";  
print\_info(the\_string);  
for (int i = 0; i < 10; i++)  
the\_string += the\_string;  
print\_info(the\_string);  
return 0;  
}

# How can we control the size of the string?

We can control the size of the memory a string uses with the reserve member function. It can work in both directions with the same ease, i.e. it can shrink the buffers as well as expand them. The function requires one parameter of type int to specify the desired size of the allocated buffers.

One important thing you need to know here: the content of the string isn't changed in any way – we can say that the content is **immune** to the effect of the reserve function.

The function, however, may not be as strict as you when it comes to determining the eventual size of the allocated memory. It may round the value of the parameter to fit it to the current memory requirements and/or target platform conditions.

The program in the editor shows what can you expect from this function. Note that the content remains untouched in each case.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
void print\_info(string & s) {  
cout << "content =\"" << s << "\" ";  
cout << "capacity = " << s.capacity() << endl;  
cout << "---------" << endl;  
}  
  
int main(void) {  
string the\_string = "content";  
print\_info(the\_string);  
the\_string.reserve(100);  
print\_info(the\_string);  
the\_string.reserve(0);  
print\_info(the\_string);  
return 0;  
}

# How to control the content of the string?

We can also control the content length of the string in many ways. The functions we’re going to show you here may also change the memory allocations – this is a possible side effect.

First of all, we can empty the string, **completely removing all the characters currently stored inside it**. This is equivalent to assigning an empty string to the string, but it could be a little bit faster.

Emptying the string is done by the member function called clear. It requires no parameters.

The program in the editor shows what can you expect from this function. Note that the content remains untouched in each case.

Changing the size of the string is carried out by the member function called resize. It's basic variant requires one parameter of type int, specifying the desired new size of the string. If the parameter is smaller than the current string size, the string will be **truncated**.

If the parameter is larger than the string size, the string will be **expanded**. You can also use an overloaded form of the function allowing you to specify a character to be used to fill the newly allocated space (the ***null*** character '\0' is used for that purpose by default).

You can also check if a particular string is **empty** (i.e. it contains no characters at all – it has nothing to do with the actual size of the allocated buffers). You can achieve the same effect by comparing the string to an empty string, but the member function will be more effective at performing this task.

The function's name is empty, it requires no parameters and it returns a value of type bool, determining the truth of the following sentence: this string is empty now.

The program on the right is a simple example showing how you can use these three functions.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
void print\_info(string & s) {  
cout << "content =\"" << s << "\" ";  
cout << "capacity = " << s.capacity() << endl;  
cout << "is empty? " << (s.empty() ? "yes" : "no") << endl;  
cout << "---------" << endl;  
}  
  
int main(void) {  
string the\_string = "content";  
print\_info(the\_string);  
the\_string.resize(50, '?');  
print\_info(the\_string);  
the\_string.resize(4);  
print\_info(the\_string);  
the\_string.clear();  
print\_info(the\_string);  
  
return 0;  
}

We’re already quite accustomed to strings as an indivisible whole, but, of course, it doesn't mean that we aren't able to get access to every char separately. Such access is a must in many algorithms, e.g. it isn't possible to implement even the simplest cipher code without being able to analyse and modify every single char of any string.

Strings offer some convenient member functions to do this. We've said previously that strings aren't arrays. Yes, it's still true, but they are able to present their content **as if it were an actual array**. We can assume that it's a kind of very useful masquerade – a string can wear a mask and show itself as an ordinary (well, almost ordinary) array. It allows us to **read and write every character separately**.

The program on the right makes use of this valuable string property. It converts its every char to upper case. We’re sorry: the algorithm is too simple to work everywhere and always. Don't use it for any serious applications. That would definitely be a bad idea. A good idea is to explain how it really works and where its weakness comes from. We urge you to try to do it yourself.

So far, we haven’t said a word about how it’s possible for a string to do something like this. Well, it's due to a mechanism called “operator overloading”. We’ll tell you more about it when we come to constructing our own objects. Again, we’re sorry: right now, you have to confine yourself to the role of user of that mechanism. We’ll change this soon, we promise.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string the\_string = "content";  
for (int i = 0; i < the\_string.length(); i++)  
the\_string[i] = the\_string[i] - 'a' + 'A';  
cout << the\_string << endl;  
return 0;  
}

# Appending a (sub)string

We’re going to change topic for a while now, but we plan to come back to this in the near future, when we show you some advanced techniques related to the advanced object capabilities of strings. And now we’re going to finish up by showing you some useful member functions that can make a programmer's life much easier.

At the end of this section we’ll tell you precisely which aspects of the strings we've omitted and why. But until then, try to guess.

One of the functions we’re going to introduce here is append. This name speaks for itself – it’s designed to **append one string to another**. If you’re an attentive student, you've probably noticed that the same task can be performed by the += operator and you may be wondering why we need another tool for the same task.

You're partially right. The append function is much more flexible and helpful than += and can give you more options. Now we’re going to try to convince you of it.

First of all, you can use the append function to append a string – here’s an example:

string str1 = "content"; str2 = "appendix"; str1.append(str2);

// str1 contains "contentappendix" now

Next, the append function is able to append not only a string, but also a substring of the string, like this:

string str1 = "content"; str1.append(3, 'x');

// str1 contains "contentxxx" now

And finally, the append can append a char (possibly repeated n times), like this:

string str1 = "content"; str1.append(3, 'x');

// str1 contains "contentxxx" now

All these variants not only affect the content of the string, but also return the modified string as a result. You may (but you don't have to) make use of this effect, e.g. building a chain of subsequent append invocations.

You’re now ready to analyse the example in the editor. You’re also ready to guess what the output produced by the program will be. It's easy, isn't it?

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string the\_string = "content";  
string new\_string;  
new\_string.append(the\_string);  
new\_string.append(the\_string, 0, 3);  
new\_string.append(2, '!');  
cout << new\_string << endl;  
return 0;  
}

# Appending a character

If you want to append just one character to a string, you can do it by using the append function, but there’s a more efficient way, by using the push\_back member function. The example in the editor is more meaningful than any description we can give you, so here it is.

Take a look at it.

When you compile and run the code, you’ll be able to admire the beauty of the classic Latin alphabet. Can you believe that it’s over 2000 years old? It still looks very young.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string the\_string;  
for (char c = 'A'; c <= 'Z'; c++)  
the\_string.push\_back(c);  
cout << the\_string << endl;  
return 0;  
}

# Іnserting a (sub)string or a character

Inserting a string into a string is like distending its contents from within. A new set of contents is just “pushed” into the old one. For example: the following snippet of code will print “to be or not to be” to the cout stream:

string quote = "to be "; quote.append(quote); quote.insert(6, "or not ");

cout << quote << endl;

As you’ve probably guessed, the first parameter specifies **where** the insertion should be, while the second says **what** should be inserted there. This is only one among various possibilities offered by the function. Another member function can insert a substring of a specified string in a way very similar to the append function.

There’s also a function that can insert a single character, optionally duplicated more than once. Both these functions are put to work in the following example. Look at the code in the editor.

The example will output:

Why so serious?

to the screen.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string quote = "Whyserious?", anyword = "monsoon";  
quote.insert(3, 2, ' ').insert(4, anyword, 3, 2);  
cout << quote << endl;  
return 0;  
}

# Assigning a (sub)string or a character

The assign member function does a job which is very similar to the insert’s job, but **does not retain** the previous string content, and instead just **replaces it with a new one**. You might ask why on earth we’d want to use the assign function when the exact same task can be performed by the = operator.

Well it’s not that simple. The assign is as universal as the insert or append functions and so is more convenient in some specific applications.

The example in the editor assigns a string with a series of stars.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string sky;  
sky.assign(80, '\*');  
cout << sky << endl;  
return 0;  
}

# Replacing a (sub)string

The replace member function is more subtle. It can **replace a part of the string with another string or another string’s substring**.

The function needs to know which part of the string it’s going to replace and you have to specify this by delivering two numbers: the first describing **the starting position** and the second saying **how many characters** will be replaced.

The first overloaded function needs a string (as a third parameter) to replace the old content (it may be either longer or shorter or equal in size in comparison to the original). The second function enables you to specify the substring to be used as a substitution.

The second variant of the function is here in the editor.

As usual (almost), the function not only modifies the string, but also returns the modified string as a result that we can use to build a series of replacements occurring in one statement.

The code will output:

I'll think about that tomorrow

to the screen.

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string to\_do = "I'll think about that in one hour";  
string schedule = "today yesterday tomorrow";  
  
to\_do.replace(22, 12, schedule, 16, 8);  
cout << to\_do << endl;  
return 0;  
}

# Erasing a (sub)string

We can also **remove a part of a string**, making the string shorter than before. We can do this by using a member function called erase (a name which leaves no room for doubt) and the function requires two numbers in order to perform its duty: the place where the substring to be removed starts (this value defaults to zero) and the length of the substring (this value defaults to the original string’s length).

This means that an invocation like this:

the\_string.erase();

erases all the characters from the string and leaves it empty.

The convention that causes the function to make a modification and to return the modified string as a result is still valid, so the example in the editor produces the following output on the screen:

I've got a feeling we're in Kansas

Note that the sequence of erase invocations cannot be reversed i.e. a statement like this:

where\_are\_we.erase(25, 4).erase(38, 8);

will generate completely different results. Can you explain why?

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string where\_are\_we = "I've got a feeling we're not in Kansas anymore";  
  
where\_are\_we.erase(38, 8).erase(25, 4);  
cout << where\_are\_we << endl;  
return 0;  
}

# Exchanging the contents of two strings

When we’re dealing with sorting, we have to be able to exchange the content of two selected cells of an array. We have to have an auxiliary variable for that purpose, since the operation resembles replacing the liquid content of two glasses – you can’t do this successfully without using a third glass. So, if we want to replace a string (or a drink) stored within glass\_1 and glass\_2 variables, we need a third variable (glass) to do it. This is how it goes:

glass = glass\_1;

glass\_1 = glass\_2;

glass\_2 = glass;

Unfortunately, this is very ineffective when it relates to strings. It would be much simpler (and also much faster) not to transfer the entire contents of both strings, but just to replace the pointers identifying them (it’s like a replacing the glasses’ owners rather than exchanging the contents of the glasses between them). This action is performed by a member function called swap. This function is many times faster than ordinary (physically performed) swapping.

Take a look at the example in the editor. We’re sure you won’t have any problems identifying the strings outputted to the console.

We’ll finish our trip to the world of strings at this point, although we’re aware that there are many more issues to discuss and to explain. We’ll return to them when we’re more familiar with object terminology and technology.

Hasta la vista, strings! See you later!

## Code

#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
int main(void) {  
string drink = "A martini";  
string needs = "Shaken, not stirred";  
  
cout << drink << ". " << needs << "." << endl;  
drink.swap(needs);  
cout << drink << ". " << needs << "." << endl;  
return 0;  
}

**Name spaces are all around us**

The term “**name space**” may sound very mysterious to you, but we want to assure you that you know this phenomenon very well, and in fact you encounter many different name spaces each and every day. Moreover, you are a part of many name spaces even though you don’t realize it. Yes, you’ve encountered the “namespace” keyword in almost all the programs in this course, but we want to give you a wider background to this concept before we go into more technical and detailed... uh, details.

**The name space is a space in which a particular name has an unambiguous and clear meaning**. For example, your closest family is a name space in which your given name identifies you and (most probably) only you. When you leave your home name space, your given name ceases to be clear. In large communities, there is the possibility that you may meet other people with the same name and it’s likely that if somebody shouted out your name in public, a few people might turn their heads in the direction of the voice.

Is there a method to make your given name more explicit? Yes, of course. Your name has to be **qualified** with another name in the hope that this new verbal cluster will be unambiguous in the new, broader space.

You’ve probably already noticed that you already have something like this. It’s your family name, isn’t it?

Of course, **it’s quite possible that this kind of qualification isn’t enough**. There may be another person with the same given name and the family names as yours, even when you’re completely unrelated. This means that another level of qualification needs to be used and as you probably know, different cultures have created their own ways of handling such issues, e.g. using one of your parents’ names, or your birth place, or your birth date, and so on.

Is there a method to make your given name more explicit? Yes, of course. Your name has to be **qualified** with another name in the hope that this new verbal cluster will be unambiguous in the new, broader space.

You’ve probably already noticed that you already have something like this. It’s your family name, isn’t it?

Of course, **it’s quite possible that this kind of qualification isn’t enough**. There may be another person with the same given name and the family names as yours, even when you’re completely unrelated. This means that another level of qualification needs to be used and as you probably know, different cultures have created their own ways of handling such issues, e.g. using one of your parents’ names, or your birth place, or your birth date, and so on.

All these tricks lead you to the same solution: to create or to choose a name space in where you can be yourself and not be confused with anyone else.

By now you should be able to identify dozens of different **name spaces** affecting your life. Some of them are quite real while others are more abstract, but all of them exist to organize our lives and to reduce the risk of misunderstanding, or (more formally) to avoid the possibility of a **name conflict**.

Good examples of real name spaces are **car licence plates** (they’re very real for sure). Depending on local law regulations, it may happen that each country, state, province, region, community etc. may create their own namespace to distinguish their plates and the plates’ owners. There’s also another level to this space: international vehicle registration codes which define a worldwide name space.

Internet domains are more abstract, but no less useful (and definitely not cheaper). They allow us all to distinguish different internet sites of the same name by their purpose, language, residence or any other aspect that may be coded according to common Internet conventions. A server called “chat” may be a good site for French cat enthusiasts, while another server with the same name may group lots of people hungry for conversations in English.

Try to list any three name spaces which accompany you in your life. It should be easy enough.

Ready? Shall we continue?

# Introducing the namespace

We don’t want you to write erroneous programs, but this time we encourage you to compile the code in the editor. Can you explain what’s wrong with it?

Yes, it lacks the line:

using namespace std;

Is this line crucial? Yes, it is. It is **fatal**. The compiler will be completely disoriented if we try to force it to use names without defining the name space to which they belong (these two: cout and endl).

You can expect many complaints from your irate compiler.

Is there a way to qualify both of these names in order to bind them to their home name space? Yes, there is. But we need to use a special syntax for this purpose. It looks like this (very informally written):

home\_name\_space :: entity\_name

Hint: the home namespace for both names is called “std” (as in *standard*). Can you rewrite the program to make it correct and not use the mysterious "*using namespace*" phrase?

## Code

#include <iostream>  
  
int main(void) {  
cout << "Play it, Sam" << endl;  
return 0;  
}

We’ve modified the code a little. Can you see the change? We’ve qualified all the ambiguous names with a prefix consisting of the home name space (std) and a special operator written as “::”.

The operator’s official name is “**scope resolution operator**”. It has some more applications, not only qualifying names with their home name spaces; hence its name refers to more general concepts.

As you may suspect, a qualifying act like this is an alternative way of using the “*using namespace*” statement.

## Code

#include <iostream>  
  
int main(void) {  
std::cout << "Play As time goes by" << std::endl;  
return 0;  
}

# Defining a name space

Do you know what a troll is? Well, of course you do. A troll is a very unpleasant creature known to everyone. Trolls come to our world from many different places, for example from Hogwarts’ dungeons or from the dark depths of Mordor, or even from many internet forums where they perform their destructive activities. They’ve even given us the verb “to troll”.

Remember, don’t feed the trolls!

Imagine we want to use trolls (two, to be precise) in our software. We’re going to take one from Hogwarts and one from Mordor. Can we cooperate with them simultaneously without the risk of mistaking one for the other?

We have to create two different name spaces and to bind the trolls to their origins. Defining a name space looks like this:

namespace the\_name\_of\_the\_space { }

Any entity declared inside a namespace (between its opening and closing brackets) will be bound to this namespace and therefore, logically separated from any other entity with the same name.

Take a look at the example in the editor. We’ve defined two namespaces and placed a troll inside each of them. If we want to make use of them, we have to qualify their names with the home namespace names.

Of course, real developers don’t use trolls in their software. You can imagine a more realistic case for yourself if you don’t like them (trolls, not developers).

Let’s assume that two independent programmers have implemented different parts of a very complex system. We cannot guarantee that all the names they’ve used are pairwise different. Such a requirement would be impractical and difficult to enforce (developers would have to agree on every new name introduced to their codes). It’s easier to assume that all their entities (e.g. variables and functions) are bound to (or rather enclosed inside) different name spaces named after the developer’s given names or their nicknames.

## Code

#include <iostream>  
  
using namespace std;  
  
namespace hogwarts {  
int troll = 1;  
}  
  
namespace Mordor {  
int troll = 2;  
}  
  
int main(void) {  
cout << hogwarts::troll << " " << Mordor::troll << endl;  
return 0;  
}

# Using a name space

If any of the available name spaces is more usable or preferable (for any reason – it’s up to the developer), it may be **used** in a way that suggests that the compiler tries to qualify every unqualified name with this/these name space names (there may be more than one name space of this type).

The act of **using a selected namespace** is carried out by the using namespace statement. If the statement is placed outside any block (a part of a code enclosed within { and } brackets), it affects the code after the statement until the end of the source file.

The example in the editor shows a case when two “*using namespace*” statements have been specified in the same code. The former allows us to conveniently use the cout and endl identifiers. The latter declares that any unqualified troll belongs to the hogwarts name space.

Note that the using namespace statements must not lead to a situation where an identifier could be considered to have originated from more than one name space. This means that you’re not allowed to use the following two statements in the same scope (block) of code:

using namespace hogwarts;

using namespace mordor;

It’ll make all the references to the unqualified trolls ambiguous. This isn’t what we want. The trolls don’t want it, either.

## Code

#include <iostream>  
  
using namespace std;  
  
namespace hogwarts {  
int troll = 1;  
}  
  
namespace Mordor {  
int troll = 2;  
}  
  
using namespace hogwarts;  
  
int main(void) {  
cout << troll << " " << Mordor::troll << endl;  
return 0;  
}

If the using namespace statement is **placed inside a block**, its scope ends in the same place where the block ends. You can use this effect to **selectively use** (and disuse) any of the available namespaces.

In the example in the editor we’ve used all of the three available namespaces and have done it in the following way:

* The std name space is used **globally** (in the whole source file)
* The hogwarts and mordor name spaces are used **selectively** (in the selected parts of the code)

## Code

#include <iostream>  
  
using namespace std;  
namespace hogwarts {  
int troll = 1;  
}  
namespace mordor {  
int troll = 2;  
}  
int main(void) {  
{  
using namespace hogwarts;  
cout << troll << " ";  
} {  
using namespace mordor;  
cout << troll << endl;  
}  
return 0;  
}

# Expanding a name space

The current example (look at the editor) shows how to extend any previously defined namespace. As you can see, we’ve constructed a code which contains “double” definitions of both our namespaces. In fact, they aren’t doubled – they’re **extended**.

Let’s assume that all the name space definitions that use the same identifier are “*glued*” together and build one, larger name space together. Note that each extension may be placed inside a separate source file. This means that any of the name spaces may be dispersed among many source files created by different developers.

Note that the first appearance of a name space is called “**an original name space**”. Any name space with the same name (identifier) that appears after the original name space is called “**an extension name space**”.

There may be more than one extension of a name space.

## Code

#include <iostream>  
  
using namespace std;  
namespace hogwarts {  
int troll = 1;  
}  
namespace mordor {  
int troll = 2;  
}  
namespace hogwarts {  
float wizard = -0.5;  
}  
namespace mordor {  
float wizard = 0.5;  
}  
int main(void) {  
cout << hogwarts::troll << " " << hogwarts::wizard << endl;  
cout << mordor::troll << " " << mordor::wizard << endl;  
return 0;  
}

# Using an entity

The using statement we’ve been working with so far has made use of the whole name space, i.e. it’s implicitly qualified all the entities bound to the space. There’s another form of statement that allows us to **selectively decide which entities should be used** and which should remain hidden inside the space. But there’s one important condition: none of the statements used may cause ambiguity in the process of identifying all the entities used in your code.

The example in the editor shows a code which employs the using statement to select two different creatures from two different worlds. In effect, their use is easier without compromising the uniqueness and clarity.

## Code

#include <iostream>  
  
using namespace std;  
namespace hogwarts {  
int troll = 1;  
}  
namespace mordor {  
int troll = 2;  
}  
namespace hogwarts {  
float wizard = -0.5;  
}  
namespace mordor {  
float wizard = 0.5;  
}  
int main(void) {  
cout << hogwarts::troll << " " << hogwarts::wizard << endl;  
cout << mordor::troll << " " << mordor::wizard << endl;  
return 0;  
}

# An unnamed name space

There’s an interesting alternation in defining the name space. We may define a name space without a name (**an anonymous namespace**).

This kind of namespace is **implicitly and automatically used** in a source file where its definition is visible. It’s another way of introducing entities (e.g. variables) accessible through the whole source file.

The example in the editor shows two name spaces: one anonymous and one created with a name. All the entities defined inside the anonymous name space can be accessed without any other preparations.

## Code

#include <iostream>  
  
using namespace std;  
namespace {  
int troll = 1;  
float wizard = -0.5;  
}  
namespace mordor {  
int troll = 2;  
float wizard = 0.5;  
}  
int main(void) {  
cout << troll << " " << wizard << endl;  
cout << mordor::troll << " " << mordor::wizard << endl;  
return 0;  
}

# Renaming a name space

It may happen that one of the name spaces you have or want to use has an inconvenient name (for any reason). You can **rename** it locally i.e. it’ll be recognized under its new name only in the source file where the act of renaming has occurred.

You can use it with a special form of the namespace statement – here it is:

namespace new\_name = old\_name;

The new name of the name space may be used together with the old one. This means that the renaming is actually **aliasing**, since both of the names are still valid.

Look at the example in the editor.

It shows how a long and twisted name could be aliased to make the developer’s life a bit easier.

Let’s leave the C++ name space alone for now. The real world is waiting.

## Code

#include <iostream>  
  
using namespace std;  
namespace what\_a\_wonderful\_place\_for\_a\_young\_sorcerer {  
int troll = 1;  
float wizard = -0.5;  
}  
namespace mordor {  
int troll = 2;  
float wizard = 0.5;  
}  
namespace hogwarts = what\_a\_wonderful\_place\_for\_a\_young\_sorcerer;  
int main(void) {  
cout << hogwarts::troll << " " <<  
what\_a\_wonderful\_place\_for\_a\_young\_sorcerer::wizard << endl;  
cout << mordor::troll << " " << mordor::wizard << endl;  
return 0;  
}

**Congratulations! You have completed Module 4.**

Well done! You've reached the end of Module 4 and completed a major milestone in your C++ programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 4:

* arrays of pointers;
* conversions;
* strings: declarations, initializations, assignments;
* strings as an example of objects: (methods and properties)
* using and declaring namespaces;
* dealing with exceptions.