

C++ PROGRAMMING

C S C

2 0 3

Structure of a program

The best way to learn a programming language is by writing programs. Typically, the first program beginners write is a program called "Hello World", which simply prints "Hello World" to your computer screen. Although it is very simple, it contains all the fundamental components C++ programs have:

```
1 // my first program in C++
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << "Hello World!";
7 }
```

Above shows the C++ code for this program. The numbers to the left of the panels are line numbers to make discussing programs and researching errors easier. They are not part of the program.

Let's examine this program line by line:

Line 1: // my first program in C++

Two slash signs indicate that the rest of the line is a comment which has no effect on the behavior of the program.

Line 2: #include <iostream>

Lines beginning with a hash sign (#) are directives read and interpreted by what is known as the *preprocessor*. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive `#include <iostream>`, instructs the preprocessor to include a section of standard C++ code, known as *header iostream*, that allows to perform standard input and output operations, such as writing the output of this program (Hello World) to the screen.

Line 3: A blank line.

Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: int main ()

This line initiates the declaration of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow.

The function named `main` is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the `main` function, regardless of where the function is actually located within the code.

Lines 5 and 7: (and)

The open brace ({) at line 5 indicates the beginning of main's function definition, and the closing brace (}) at line 7, indicates its end. Everything between these braces is the function's body that defines what happens when main is called.

*** All functions use braces to indicate the beginning and end of their definitions.

Line 6: std::cout << "Hello, World!";

This line is a C++ statement. A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body.

This statement has three parts: First, std::cout, which identifies the standard character output device (usually, this is the computer screen). Second, the insertion operator (<<), which indicates that what follows is inserted into std::cout. Finally, a sentence within quotes ("Hello world!"), is the content inserted into the standard output.

Notice that the statement ends with a semicolon (;). This character marks the end of the statement, just as the period ends a sentence in English. All C++ statements must end with a semicolon character. One of the most common syntax errors in C++ is forgetting to end a statement with a semicolon.

You may have noticed that not all the lines of this program perform actions when the code is executed. There is a line containing a comment (beginning with //). There is a line with a directive for the preprocessor (beginning with #). There is a line that defines a function (in this case, the main function). And, finally, a line with a statement ending with a semicolon (the insertion into cout), which was within the block delimited by the braces ({ }) of the main function.

The program has been structured in different lines and properly indented, in order to make it easier to understand for the humans reading it. But C++ does not have strict rules on indentation or on how to split instructions in different lines. For example, instead of

```
1 int main ()  
2 {  
3     std::cout << "Hello World!";  
4 }
```

We could have written:

```
int main () { std::cout << "Hello World!"; }
```

all in a single line, and this would have had exactly the same meaning as the preceding code.

In C++, the separation between statements is specified with an ending semicolon (:), with the separation into different lines not mattering at all for this purpose. Many statements can be written in a single line, or each statement can be in its own line. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it, but has no effect on the actual behavior of the program.

Now, let's add an additional statement to our first program:

```
1 // my second program in C++  
2 #include <iostream>  
3  
4 int main ()  
5 {  
6     std::cout << "Hello World! ";  
7     std::cout << "I'm a C++ program";  
8 }
```

In this case, the program performed two insertions into `std::cout` in two different statements. Once again, the separation in different lines of code simply gives greater readability to the program, since `main` could have been perfectly valid defined in this way:

```
int main () { std::cout << "Hello World! "; std::cout << " I'm a C++  
Program "; }
```

The source code could have also been divided into more code lines instead:

```
1 int main ()  
2 {  
3     std::cout <<  
4     "Hello World!";  
5     std::cout  
6     << "I'm a C++ program";  
7 }
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by `#`) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor before proper compilation begins. Preprocessor directives must be specified in their own line and, because they are not statements, do not have to end with a semicolon (`;`).

Comments

As noted above, comments do not affect the operation of the program; however, they provide an important tool to document directly within the source code what the program does and how it operates.

C++ supports two ways of commenting code:

```
1 // line comment  
2 /* block comment */
```

The first of them, known as *line comment*, discards everything from where the pair of slash signs (//) are found up to the end of that same line. The second one, known as *block comment*, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including multiple lines.

Let's add comments to our second program:

```
1 /* my second program in C++  
2   with more comments */  
3  
4 #include <iostream>  
5  
6 int main ()  
7 {  
8     std::cout << "Hello World! ";      // prints Hello World!  
9     std::cout << "I'm a C++ program"; // prints I'm a C++ program  
10 }
```

If comments are included within the source code of a program without using the comment characters combinations //, /* or */, the compiler takes them as if they were C++ expressions, most likely causing the compilation to fail with one, or several, error messages.

Using namespace std

If you have seen C++ code before, you may have seen cout being used instead of std::cout. Both name the same object: the first one uses its *unqualified name* (cout), while the second qualifies it directly within the *namespace std* (as std::cout).

cout is part of the standard library, and all the elements in the standard C++ library are declared within what is called a *namespace*: the namespace std.

In order to refer to the elements in the std namespace a program shall either qualify each and every use of elements of the library (as we have done by prefixing cout with std::), or introduce visibility of its components. The most typical way to introduce visibility of these components is by means of *using declarations*:

```
using namespace std;
```

The above declaration allows all elements in the std namespace to be accessed in an *unqualified* manner (without the std:: prefix).

With this in mind, the last example can be rewritten to make unqualified uses of cout as:

```
1 // my second program in C++  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main ()
```

```
6  
7 cout << "Hello World!";  
8 cout << "I'm a C++ program";  
9 }
```

Both ways of accessing the elements of the `std` namespace (explicit qualification and *using* declarations) are valid in C++ and produce the exact same behavior. For simplicity, and to improve readability, the examples in these tutorials will more often use this latter approach with *using* declarations, although note that *explicit qualification* is the only way to guarantee that name collisions never happen.

Namespaces are explained in more detail in a later chapter.

Variables and types

We can now define *variable* as a portion of memory to store a value.

Identifiers

A *valid identifier* is a sequence of one or more letters, digits, or underscore characters (`_`).

Spaces, punctuation marks, and symbols cannot be part of an identifier. In addition, identifiers shall always begin with a letter. They can also begin with an underline character (`_`), but such identifiers are - on most cases - considered reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case can they begin with a digit.

C++ uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a programmer cannot match these keywords. The standard reserved keywords that cannot be used for programmer created identifiers are:

```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case,  
catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast,  
continue, decltype, default, delete, do, double, dynamic_cast, else, enum,  
explicit, export, extern, false, float, for, friend, goto, if, inline, int,  
long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or,  
or_eq, private, protected, public, register, reinterpret_cast, return, short,  
signed, sizeof, static, static_assert, static_cast, struct, switch, template,  
this, thread_local, throw, true, try, typeid, typename, union,  
unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq
```

Specific compilers may also have additional specific reserved keywords.

Very important: The C++ language is a "case sensitive" language. That means that an identifier written

in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the RESULT variable is not the same as the result variable or the Result variable. These are three different identifiers identifying three different variables.

Fundamental data types

The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones. Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name. What the program needs to be aware of is the kind of data stored in the variable. It's not the same to store a simple integer as it is to store a letter or a large floating-point number; even though they are all represented using zeros and ones, they are not interpreted in the same way, and in many cases, they don't occupy the same amount of memory.

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

- **Character types:** They can represent a single character, such as 'A' or '\$'. The most basic type is `char`, which is a one-byte character. Other types are also provided for wider characters.
- **Numerical integer types:** They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values or not.
- **Floating-point types:** They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.
- **Boolean type:** The boolean type, known in C++ as `bool`, can only represent one of two states, `true` or `false`.

Here is the complete list of fundamental types in C++:

Group	Type names*	Notes on size / precision
Character types	<code>Char</code>	Exactly one byte in size. At least 8 bits.
	<code>char16_t</code>	Not smaller than <code>char</code> . At least 16 bits.
	<code>char32_t</code>	Not smaller than <code>char16_t</code> . At least 32 bits.
	<code>wchar_t</code>	Can represent the largest supported character set.
Integer types (signed)	<code>signed char</code>	Same size as <code>char</code> . At least 8 bits.
	<code>signed short int</code>	Not smaller than <code>char</code> . At least 16 bits.

	<code>signed int</code>	Not smaller than short. At least 16 bits.
	<code>signed long int</code>	Not smaller than int. At least 32 bits.
	<code>signed long long int</code>	Not smaller than long. At least 64 bits.
	<code>unsigned char</code>	
	<code>unsigned short int</code>	
Integer types (unsigned)	<code>unsigned int</code>	(same size as their signed counterparts)
	<code>unsigned long int</code>	
	<code>unsigned long long int</code>	
	<code>Float</code>	
Floating-point types	<code>Double</code>	Precision not less than float
	<code>long double</code>	Precision not less than double
Boolean type	<code>Bool</code>	
Void type	<code>Void</code>	no storage
Null pointer	<code>decltype(nullptr)</code>	

* The names of certain integer types can be abbreviated without their `signed` and `int` components - only the part not in italics is required to identify the type, the part in italics is optional. I.e., `signed short int` can be abbreviated as `signed short`, `short int`, or simply `short`; they all identify the same fundamental type.

Within each of the groups above, the difference between types is only their size (i.e., how much they occupy in memory): the first type in each group is the smallest, and the last is the largest, with each type being at least as large as the one preceding it in the same group. Other than that, the types in a group have the same properties.

Note in the panel above that other than `char` (which has a size of exactly one byte), none of the fundamental types has a standard size specified (but a minimum size, at most). Therefore, the type is not required (and in many cases is not) exactly this minimum size. This does not mean that these types are of an undetermined size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where

the program is going to run. This rather generic size specification for types gives the C++ language a lot of flexibility to be adapted to work optimally in all kinds of platforms, both present and future.

Type sizes above are expressed in bits; the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory:

Size	Unique representable values	Notes
8-bit	256	$= 2^8$
16-bit	65 536	$= 2^{16}$
32-bit	4 294 967 296	$= 2^{32}$ (~4 billion)
64-bit	18 446 744 073 709 551 616	$= 2^{64}$ (~18 billion billion)

For integer types, having more representable values means that the range of values they can represent is greater; for example, a 16-bit unsigned integer would be able to represent 65536 distinct values in the range 0 to 65535, while its signed counterpart would be able to represent, on most cases, values between -32768 and 32767. Note that the range of positive values is approximately halved in signed types compared to unsigned types, due to the fact that one of the 16 bits is used for the sign; this is a relatively modest difference in range, and seldom justifies the use of unsigned types based purely on the range of positive values they can represent.

For floating-point types, the size affects their precision, by having more or less bits for their significant and exponent.

If the size or precision of the type is not a concern, then `char`, `int`, and `double` are typically selected to represent characters, integers, and floating-point values, respectively. The other types in their respective groups are only used in very particular cases.

The properties of fundamental types in a particular system and compiler implementation can be obtained by using the `numeric_limits` classes (see standard header `<limits>`). If for some reason, types of specific sizes are needed, the library defines certain fixed-size type aliases in header `<cstdint>`.

The types described above (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: `void`, which identifies the lack of type; and the type `nullptr`, which is a special type of pointer. Both types will be discussed further in a coming chapter about pointers.

C++ supports a wide variety of types based on the fundamental types discussed above; these other types are known as *compound data types*, and are one of the main strengths of the C++ language. We will also see them in more detail in future chapters.

Declaration of variables

C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use. This informs the compiler the size to reserve in memory for the variable and how to interpret its value. The syntax to declare a new variable in C++ is straightforward: we simply write the type followed by the variable name (i.e., its identifier). For example:

```
1 int a;  
2 float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type `int` with the identifier `a`. The second one declares a variable of type `float` with the identifier `mynumber`. Once declared, the variables `a` and `mynumber` can be used within the rest of their scope in the program. If declaring more than one variable of the same type, they can all be declared in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (`a`, `b` and `c`), all of them of type `int`, and has exactly the same meaning as:

```
1 int a;  
2 int b;  
3 int c;
```

To see what variable declarations look like in action within a program, let's have a look at the entire C++ code of the example about your mental memory proposed at the beginning of this chapter:

```
1 // operating with variables  
2  
3 #include <iostream>  
4 using namespace std;  
5  
6 int main ()  
7 {  
8     // declaring variables:  
9     int a, b;  
10    int result;  
11  
12    // process:  
13    a = 5;  
14    b = 2;  
15    a = a + 1;  
16    result = a - b;  
17  
18    // print out the result:
```

```
19 cout << result;  
20  
21 // terminate the program:  
22 return 0;  
23 }
```

Don't be worried if something else than the variable declarations themselves look a bit strange to you. Most of it will be explained in more detail in coming chapters.

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time. But it is possible for a variable to have a specific value from the moment it is declared. This is called the *initialization* of the variable.

In C++, there are three ways to initialize variables. They are all equivalent and are reminiscent of the evolution of the language over the years:

The first one, known as *c-like initialization* (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized:

```
type identifier = initial_value;
```

For example, to declare a variable of type int called x and initialize it to a value of zero from the same moment it is declared, we can write:

```
int x = 0;
```

A second method, known as *constructor initialization* (introduced by the C++ language), encloses the initial value between parentheses ():

```
type identifier (initial_value);
```

For example:

```
int x (0);
```

Finally, a third method, known as *uniform initialization*, similar to the above, but using curly braces () instead of parentheses (this was introduced by the revision of the C++ standard, in 2011):

```
type identifier {initial_value};
```

For example:

```
int x {0};
```

All three ways of initializing variables are valid and equivalent in C++.

```
1 // initialization of variables
2
3 #include <iostream>
4 using namespace std;
5
6 int main ()
7 {
8     int a=5;           // initial value: 5
9     int b(3);         // initial value: 3
10    int c{2};         // initial value: 2
11    int result;       // initial value undetermined
12
13    a = a + b;
14    result = a - c;
15    cout << result;
16
17    return 0;
18 }
```

Type deduction: auto and decltype

When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use `auto` as the type specifier for the variable:

```
1 int foo = 0;
2 auto bar = foo; // the same as: int bar = foo;
```

Here, `bar` is declared as having an `auto` type; therefore, the type of `bar` is the type of the value used to initialize it: in this case it uses the type of `foo`, which is `int`.

Variables that are not initialized can also make use of type deduction with the `decltype` specifier:

```
1 int foo = 0;
2 decltype(foo) bar; // the same as: int bar;
```

Here, `bar` is declared as having the same type as `foo`.

`auto` and `decltype` are powerful features recently added to the language. But the type deduction features they introduce are meant to be used either when the type cannot be obtained by other means or when using it improves code readability. The two examples above were likely neither of these use cases. In fact they probably decreased readability, since, when reading the code, one has to search for the type of `foo` to actually know the type of `bar`.

Introduction to strings

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks.

An example of compound type is the `string` class. Variables of this type are able to store sequences of characters, such as words or sentences. A very useful feature!

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library (header `<string>`):

```
1 // my first string
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string mystring;
9     mystring = "This is a string";
10    cout << mystring;
11    return 0;
12 }
```

As you can see in the previous example, strings can be initialized with any valid string literal, just like numerical type variables can be initialized to any valid numerical literal. As with fundamental types, all initialization formats are valid with strings:

```
1 string mystring = "This is a string";
2 string mystring ("This is a string");
3 string mystring {"This is a string"};
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and change its value during execution:

```
1 // my first string
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string mystring;
9     mystring = "This is the initial string content
&
This is a different string content";
```

```
10 cout << mystring << endl;
11 mystring = "This is a different string content";
12 cout << mystring << endl;
13 return 0;
14 }
```

Note: inserting the `endl` manipulator ends the line (printing a newline character and flushing the stream).

The string class is a *compound type*. As you can see in the example above, *compound types* are used in the same way as *fundamental types*: the same syntax is used to declare variables and to initialize them.

For more details on standard C++ strings, see the string class reference.

Constants

Constants are expressions with a fixed value.

Literals

Literals are the most obvious kind of constants. They are used to express particular values within the source code of a program. We have already used some in previous chapters to give specific values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

The 5 in this piece of code was a *literal constant*.

Literal constants can be classified into: integer, floating-point, characters, strings, Boolean, pointers, and user-defined literals.

Integer Numerals

1 1776

2 707

3 -273

These are numerical constants that identify Integer values. Notice that they are not enclosed in quotes

or any other special character; they are a simple succession of digits representing a whole number in decimal base; for example, 1776 always represents the value *one thousand seven hundred seventy-six*.

Suffix Type modifier

u or U unsigned

l or L long

ll or LL long long

Unsigned may be combined with any of the other two in any order to form **unsigned long** or **unsigned long long**.

For example:

```
1 75          // int
2 75u         // unsigned int
3 75l         // long
4 75ul        // unsigned long
5 75lu        // unsigned long
```

In all the cases above, the suffix can be specified using either upper or lowercase letters.

Floating Point Numerals

They express real values, with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character:

```
1 3.14159    // 3.14159
2 6.02e23     // 6.02 x 10^23
3 1.6e-19     // 1.6 x 10^-19
4 3.0         // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated-, and the last one is the number *three* expressed as a floating-point numeric literal.

The default type for floating-point literals is **double**. Floating-point literals of type **float** or **long double** can be specified by adding one of the following suffixes:

Suffix Type

for F float

for L long double

For example:

```
1 3.14159L // long double  
2 6.02e23f // float
```

Any of the letters that can be part of a floating-point numerical constant (e, f, l) can be written using either lower or uppercase letters with no difference in meaning.

Character and string literals

Character and string literals are enclosed in quotes:

```
1 'z'  
2 'p'  
3 "Hello world"  
4 "How do you do?"
```

The first two expressions represent *single-character literals*, and the following two represent *string literals* composed of several characters. Notice that to represent a single character, we enclose it between single quotes ('), and to express a string (which generally consists of more than one character), we enclose the characters between double quotes (").

Both single-character and string literals require quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

x
'x'

Here, x alone would refer to an identifier, such as the name of a variable or a compound type, whereas 'x' (enclosed within single quotation marks) would refer to the character literal 'x' (the character that represents a lowercase x letter).

Character and string literals can also represent special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). These special characters are all of them preceded by a backslash character (\).

Here you have a list of the single character escape codes:

Escape code	Description
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\\	backslash (\)

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Internally, computers represent characters as numerical codes: most typically, they use one extension of the ASCII character encoding system (see ASCII code for more info). Characters can also be represented in literals using its numerical code by writing a backslash character (\) followed by the code expressed as an octal (base-8) or hexadecimal (base-16) number. For an octal value, the backslash is followed directly by the digits; while for hexadecimal, an x character is inserted between the backslash and the hexadecimal digits themselves (for example: \x20 or \x4A).

Several string literals can be concatenated to form a single string literal simply by separating them by one or more blank spaces, including tabs, newlines, and other valid blank characters. For example:

```
1 "this forms" "a single"      " string "
2 "of characters"
```

The above is a string literal equivalent to:

"this forms a single string of characters"

Note how spaces within the quotes are part of the literal, while those outside them are not.

Some programmers also use a trick to include long string literals in multiple lines: In C++, a backslash (\) at the end of line is considered a line-continuation character that merges both that line and the next into a single line. Therefore the following code:

```
1 x = "string expressed in \
2 two lines"
```

is equivalent to:

```
x = "string expressed in two lines"
```

All the character literals and string literals described above are made of characters of type `char`. A different character type can be specified by using one of the following prefixes:

Prefix Character type

u `char16_t`

U `char32_t`

L `wchar_t`

Note that, unlike type suffixes for integer literals, these prefixes are *case sensitive*: lowercase for `char16_t` and uppercase for `char32_t` and `wchar_t`.

Other literals

Three keyword literals exist in C++: `true`, `false` and `nullptr`:

- `true` and `false` are the two possible values for variables of type `bool`.
- `nullptr` is the *null pointer* value.

```
1 bool foo = true;
2 bool bar = false;
3 int* p = nullptr;
```

Typed constant expressions

Sometimes, it is just convenient to give a name to a constant value:

```
1 const double pi = 3.1415926;
2 const char tab = '\t';
```

We can then use these names instead of the literals they were defined to:

```
1 #include <iostream>
2 using namespace std;
3
4 const double pi = 3.14159;
5 const char newline = '\n';
6
7 int main ()
8 {
9     double r=5.0;
10    double circle;
11
12    circle = 2 * pi * r;
13    cout << circle;
14    cout << newline;
15 }
```

Preprocessor definitions (#define)

Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:

```
#define identifier replacement
```

After this directive, any occurrence of **identifier** in the code is interpreted as **replacement**, where **replacement** is any sequence of characters (until the end of the line). This replacement is performed by the preprocessor, and happens before the program is compiled, thus causing a sort of blind replacement: the validity of the types or syntax involved is not checked in any way.

For example:

```

1 #include <iostream>
2 using namespace std;
3
4 #define PI 3.14159
5 #define NEWLINE '\n'
6
7 int main ()
8 {
9     double r=5.0;
10    double circle;
11
12    circle = 2 * PI * r;
13    cout << circle;
14    cout << NEWLINE;
15
16 }

```

Note that the `#define` lines are preprocessor directives, and as such are single-line instructions that -unlike C++ statements- do not require semicolons (`;`) at the end; the directive extends automatically until the end of the line. If a semicolon is included in the line, it is part of the replacement sequence and is also included in all replaced occurrences.

Operators

Once introduced to variables and constants, we can begin to operate with them by using *operators*. What follows is a complete list of operators. At this point, it is likely not necessary to know all of them, but they are all listed here to also serve as reference.

Assignment operator (=)

The assignment operator assigns a value to a variable.

`x = 5;`

This statement assigns the integer value 5 to the variable `x`. The assignment operation always takes place from right to left, and never the other way around:

x = y;

This statement assigns to variable x the value contained in variable y. The value of x at the moment this statement is executed is lost and replaced by the value of y.

Consider also that we are only assigning the value of y to x at the moment of the assignment operation. Therefore, if y changes at a later moment, it will not affect the new value taken by x. For example, let's have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
1 // assignment operator
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int a, b;
8     a = 10;           // a:?, b:?
9     b = 4;            // a:10, b:?
10    a = b;           // a:10, b:4 a:4 b:7
11    b = 7;           // a:4, b:4
12                    // a:4, b:7
13    cout << "a:";
14    cout << a;
15    cout << " b:";
16    cout << b;
17 }
```

This program prints on screen the final values of a and b (4 and 7, respectively). Notice how a was not affected by the final modification of b, even though we declared a = b earlier.

Assignment operations are expressions that can be evaluated. That means that the assignment itself has a value, and -for fundamental types- this value is the one assigned in the operation. For example:

y = 2 + (x = 5);

In this expression, y is assigned the result of adding 2 and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

1 x = 5;
2 y = 2 + x;

With the final result of assigning 7 to y.

The following expression is also valid in C++:

`x = y = z = 5;`

It assigns 5 to the all three variables: `x`, `y` and `z`; always from right-to-left.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by C++ are:

operator description

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division correspond literally to their respective mathematical operators. The last one, *modulo operator*, represented by a percentage sign (%), gives the remainder of a division of two values. For example:

`x = 11 % 3;`

results in variable `x` containing the value 2, since dividing 11 by 3 results in 3, with a remainder of 2.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Compound assignment operators modify the current value of a variable by performing an operation on it. They are equivalent to assigning the result of an operation to the first operand:

expression	equivalent to...
<code>y += x;</code>	<code>y = y + x;</code>
<code>x -= 5;</code>	<code>x = x - 5;</code>
<code>x /= y;</code>	<code>x = x / y;</code>
<code>price *= units + 1; price = price * (units+1);</code>	

and the same for all other compound assignment operators. For example:

```
1 // compound assignment operators
2 #include <iostream>
```

```

3 using namespace std;
4 int main ()
5 {
6     int a, b=3;
7     a = b;
8     a+=2;           // equivalent to a=a+2
9     cout << a;
10 }
11 }
```

Increment and decrement (++, --)

Some expression can be shortened even more: the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to `+1` and to `-1`, respectively. Thus:

```

1 ++x;
2 x+=1;
3 x=x+1;
```

are all equivalent in its functionality; the three of them increase by one the value of `x`.

In the early C compilers, the three previous expressions may have produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally performed automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A peculiarity of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable name (`++x`) or after it (`x++`). Although in simple expressions like `x++` or `++x`, both have exactly the same meaning; in other expressions in which the result of the increment or decrement operation is evaluated, they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (`++x`) of the value, the expression evaluates to the final value of `x`, once it is already increased. On the other hand, in case that it is used as a suffix (`x++`), the value is also increased, but the expression evaluates to the value that `x` had before being increased. Notice the difference:

Example 1

```

x = 3;
y = ++x;
// x contains 4, y contains 4 // x contains 4, y contains 3
```

Example 2

```

x = 3;
y = x++;
// x contains 4, y contains 3
```

In *Example 1*, the value assigned to `y` is the value of `x` after being increased. While in *Example 2*, it is the value `x` had before being increased.

Relational and comparison operators (==, !=, >, <, >=, <=)

Two expressions can be compared using relational and equality operators. For example, to know if two values are equal or if one is greater than the other.

The result of such an operation is either true or false (i.e., a Boolean value).

The relational operators in C++ are:

operator	description
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to

Here there are some examples:

```
1 (7 == 5)      // evaluates to false
2 (5 > 4)       // evaluates to true
3 (3 != 2)       // evaluates to true
4 (6 >= 6)      // evaluates to true
5 (5 < 5)       // evaluates to false
```

Of course, it's not just numeric constants that can be compared, but just any value, including, of course, variables. Suppose that `a=2`, `b=3` and `c=6`, then:

```
1 (a == 5)      // evaluates to false, since a is not equal to 5
2 (a*b >= c)    // evaluates to true, since (2*3 >= 6) is true
3 (b+4 > a*c)  // evaluates to false, since (3+4 > 2*6) is false
4 ((b=2) == a)   // evaluates to true
```

Be careful! The assignment operator (operator `=`, with one equal sign) is not the same as the equality comparison operator (operator `==`, with two equal signs); the first one (`=`) assigns the value on the right-hand to the variable on its left, while the other (`==`) compares whether the values on both sides of the operator are equal. Therefore, in the last expression `((b=2) == a)`, we first assigned the value 2 to `b` and then we compared it to `a` (that also stores the value 2), yielding `true`.

Logical operators (`!`, `&&`, `||`)

The operator `!` is the C++ operator for the Boolean operation NOT. It has only one operand, to its right, and inverts it, producing `false` if its operand is `true`, and `true` if its operand is `false`. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
! (5 == 5)    // evaluates to false because the expression at its right (5 ==  
1 5) is true  
2 !(6 <= 4)  // evaluates to true because (6 <= 4) would be false  
3 !true      // evaluates to false  
4 !false     // evaluates to true
```

The logical operators `&&` and `||` are used when evaluating two expressions to obtain a single relational result. The operator `&&` corresponds to the Boolean logical operation AND, which yields `true` if both its operands are `true`, and `false` otherwise. The following panel shows the result of operator `&&` evaluating the expression `a & b`:

&& OPERATOR (and)

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator `||` corresponds to the Boolean logical operation OR, which yields `true` if either of its operands is `true`, thus being `false` only when both operands are `false`. Here are the possible results of `a || b`:

|| OPERATOR (or)

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

For example:

```
For example:  
1 { (5 == 5) && (3 > 6) } // evaluates to false ( true && false )  
2 { (5 == 5) || (3 > 6) } // evaluates to true ( true || false )
```

When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in the last example ($(5==5) \text{ || } (3>6)$), C++ evaluates first whether $5==5$ is true, and if so, it never checks whether $3>6$ is true or not. This is known as *short-circuit evaluation*, and works like this for these operators:

operator

short-circuit

if the left-hand side expression is `false`, the combined result is `false` (the right-hand side expression is never evaluated).

if the left-hand side expression is `true`, the combined result is `true` (the right-hand side expression is never evaluated).

This is mostly important when the right-hand expression has side effects, such as altering values:

```
if ( (i<10) && (++i<n) ) /*...*/ // note that the condition  
increments i
```

Here, the combined conditional expression would increase `i` by one, but only if the condition on the left of `&&` is `true`, because otherwise, the condition on the right-hand side (`++i<n`) is never evaluated.

Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

would first assign the value 3 to `b`, and then assign `b+2` to variable `a`. So, at the end, variable `a` would contain the value 5 while variable `b` would contain value 3.

Explicit type casting operator

Type casting operators allow to convert a value of a given type to another type. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
1 int i;  
2 float f = 3.14;  
3 i = (int) f;
```

The previous code converts the floating-point number 3.14 to an integer value (3); the remainder is lost. Here, the typecasting operator was `(int)`. Another way to do the same thing in C++ is to

use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of casting types are valid in C++.

sizeof

This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object:

```
x = sizeof (char);
```

Here, x is assigned the value 1, because char is a type with a size of one byte.

The value returned by sizeof is a compile-time constant, so it is always determined before program execution.

Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming.

? : conditional operator

16 Sequencing , comma separator Left-to-right

When an expression has two operators with the same precedence level, grouping determines which one is evaluated first: either left-to-right or right-to-left.

Enclosing all sub-statements in parentheses (even those unnecessary because of their precedence) improves code readability.

Basic Input/Output

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file. A *stream* is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs:

stream	Description
cin	standard input stream
cout	standard output stream
cerr	standard error (output) stream
clog	standard logging (output) stream

We are going to see in more detail only cout and cin (the standard output and input streams); cerr and clog are also output streams, so they essentially work like cout, with the only difference being that they identify streams for specific purposes: error messages and logging; which, in many cases, in most environment setups, they actually do the exact same thing: they print on screen, although they can also be individually redirected.

Standard output (cout)

On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is cout.

For formatted output operations, cout is used together with the *insertion operator*, which is written as << (i.e., two "less than" signs).

```
1 cout << "Output sentence"; // prints Output sentence on screen
2 cout << 120; // prints number 120 on screen
3 cout << x; // prints the value of x on screen
```

The << operator inserts the data that follows it into the stream that precedes it. In the examples above, it inserted the literal string Output sentence, the number 120, and the value of variable x into the standard output stream cout. Notice that the sentence in the first statement is enclosed in double quotes ("") because it is a string literal, while in the last one, x is not. The double quoting is what makes the difference; when the text is enclosed between them, the text is printed literally; when they are not, the text is interpreted as the identifier of a variable, and its value is printed instead. For example, these two sentences have very different results:

```
1 cout << "Hello"; // prints Hello  
2 cout << Hello; // prints the content of variable Hello
```

Multiple insertion operations (<<) may be chained in a single statement:

```
cout << "This " << " is a " << "single C++ statement";
```

This last statement would print the text This is a single C++ statement. Chaining insertions is especially useful to mix literals and variables in a single statement:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

Assuming the age variable contains the value 24 and the zipcode variable contains 90064, the output of the previous statement would be:

I am 24 years old and my zipcode is 90064

What cout does not do automatically is add line breaks at the end, unless instructed to do so. For example, take the following two statements inserting into cout:

```
cout << "This is a sentence.";  
cout << "This is another sentence.;"
```

The output would be in a single line, without any line breaks in between. Something like:

This is a sentence.This is another sentence.

To insert a line break, a new-line character shall be inserted at the exact position the line should be broken. In C++, a new-line character can be specified as \n (i.e., a backslash character followed by a lowercase n). For example:

```
1 cout << "First sentence.\n";  
2 cout << "Second sentence.\nThird sentence.;"
```

This produces the following output:

```
First sentence.  
Second sentence.  
Third sentence.
```

Alternatively, the endl manipulator can also be used to break lines. For example:

```
1 cout << "First sentence." << endl;  
2 cout << "Second sentence." << endl;
```

This would print:

```
First sentence.  
Second sentence.
```

The endl manipulator produces a newline character, exactly as the insertion of '\n' does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly *fully buffered streams*, and cout is (generally) not a *fully buffered stream*. Still, it is generally a good idea to use endl only when flushing the stream would be a feature and '\n' when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.

Standard input (cin)

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is cin.

For formatted input operations, cin is used together with the extraction operator, which is written as >> (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored. For example:

```
1 int age;  
2 cin >> age;
```

The first statement declares a variable of type int called age, and the second extracts from cin a value to be stored in it. This operation makes the program wait for input from cin; generally, this means that

the program will wait for the user to enter some sequence with the keyboard. In this case, note that the characters introduced using the keyboard are only transmitted to the program when the ENTER (or RETURN) key is pressed. Once the statement with the extraction operation on `cin` is reached, the program will wait for as long as needed until some input is introduced.

The extraction operation on `cin` uses the type of the variable after the `>>` operator to determine how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

```
// i/o example
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6     int i;
7     cout << "Please enter an
8     integer value: ";
9     cin >> i;
10    cout << "The value you entered
11 is " << i;
12    cout << " and its double is "
13 << i*2 << ".\n";
14    return 0;
}
```

Please enter an integer value: 702 [Edit &](#)
The value you entered is 702 and [Run](#)
its double is 1404.

As you can see, extracting from `cin` seems to make the task of getting input from the standard input pretty simple and straightforward. But this method also has a big drawback. What happens in the example above if the user enters something else that cannot be interpreted as an integer? Well, in this case, the extraction operation fails. And this, by default, lets the program continue without setting a value for variable `i`, producing undetermined results if the value of `i` is used later.

This is very poor program behavior. Most programs are expected to behave in an expected manner no matter what the user types, handling invalid values appropriately. Only very simple programs should rely on values extracted directly from `cin` without further checking. A little later we will see how `stringstreams` can be used to have better control over user input.

Extractions on `cin` can also be chained to request more than one datum in a single statement:

```
cin >> a >> b;
```

This is equivalent to:

```
1 cin >> a;
2 cin >> b;
```

In both cases, the user is expected to introduce two values, one for variable a, and another for variable b. Any kind of space is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.

cin and strings

The extraction operator can be used on cin to get strings of characters in the same way as with fundamental data types:

```
1 string mystring;
2 cin >> mystring;
```

However, cin extraction always considers spaces (whitespaces, tabs, new-line...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.

To get an entire line from cin, there exists a function, called getline, that takes the stream (cin) as first argument, and the string variable as second. For example:

```
// cin with strings
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main ()
6 {
7     string mystr;
8     cout << "What's your name? ";
9     getline (cin, mystr);
10    cout << "Hello " << mystr << ".\n";
11    cout << "What is your favorite team? ";
12    cout << "I like " << mystr << "
13    getline (cin, mystr);
14    cout << "I like " << mystr << "
15 too!\n";
16    return 0;
}
```

What's your name? Homer Simpson
Hello Homer Simpson.
What is your favorite team? The
Isotopes
I like The Isotopes too!

Notice how in both calls to getline, we used the same string identifier (mystr). What the program does in the second call is simply replace the previous content with the new one that is introduced.

The standard behavior that most users expect from a console program is that each time the program queries the user for input, the user introduces the field, and then presses ENTER (or RETURN). That is to say, input is generally expected to happen in terms of lines on console programs, and this can be achieved by using getline to obtain input from the user. Therefore, unless you have a strong reason not to, you should always use getline to get input in your console programs instead of extracting from cin.

stringstream

The standard header `<sstream>` defines a type called `stringstream` that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on `cin` and `cout`. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
1 string mystr ("1204");
2 int myint;
3 stringstream(mystr) >> myint;
```

This declares a string with initialized to a value of "1204", and a variable of type int. Then, the third line uses this variable to extract from a `stringstream` constructed from the string. This piece of code stores the numerical value 1204 in the variable called `myint`.

```
1 // stringstream
2 #include <iostream>
3 #include <string>
4 #include <sstream>
5 using namespace std;
6
7 int main ()
8 {
9     string mystr;
10    float price=0;
11    int quantity=0;
12    cout << "Enter price: ";
13    getline (cin,mystr);
14    stringstream(mystr) >> price;
15    cout << "Enter quantity: ";
16    getline (cin,mystr);
17    stringstream(mystr) >> quantity;
18    cout << "Total price: " << price*quantity <<
19    endl;
20    return 0;
21 }
```

Enter price: 22.25 Edit &
Enter quantity: 7 Run
Total price: 155.75

In this example, we acquire numeric values from the *standard input* indirectly: Instead of extracting numeric values directly from `cin`, we get lines from it into a string object (`mystr`), and then we extract the values from this string into the variables `price` and `quantity`. Once these are numerical values, arithmetic operations can be performed on them, such as multiplying them to obtain a total price.

With this approach of getting entire lines and extracting their contents, we separate the process of getting user input from its interpretation as data, allowing the input process to be what the user expects, and at the same time gaining more control over the transformation of its content into useful data by the program.

Statements and flow control

A simple C++ statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (`:`), and are executed in the same order in which they appear in a program.

But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.

Many of the flow control statements explained in this section require a generic (sub)statement as part of its syntax. This statement may either be a simple C++ statement, -such as a single instruction, terminated with a semicolon (`:`) - or a compound statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: `{ }:`

```
{ statement1; statement2; statement3; }
```

The entire block is considered a single statement (composed itself of multiple substatements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

Selection statements: if and else

The `if` keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:

```
if (condition) statement
```

Here, `condition` is the expression that is being evaluated. If this condition is true, `statement` is executed. If it is false, `statement` is not executed (it is simply ignored), and the program continues right after the entire selection statement.

For example, the following code fragment prints the message (`x` is 100), only if the value stored in the `x` variable is indeed 100:

```
1 if (x == 100)  
2   cout << "x is 100";
```

If `x` is not exactly 100, this statement is ignored, and nothing is printed.

If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces (`{ }`), forming a block:

```
1 if (x == 100)  
2 {  
3   cout << "x is ";  
4   cout << x;  
5 }
```

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

```
if (x == 100) { cout << "x is "; cout << x; }
```

Selection statements with `if` can also specify what happens when the condition is not fulfilled, by using the `else` keyword to introduce an alternative statement. Its syntax is:

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

where `statement1` is executed in case condition is true, and in case it is not, `statement2` is executed.

For example:

```
1 if (x == 100)
```

34

```
33  
1 cout << "x is 100";  
2 else  
3 cout << "x is not 100";
```

This prints **x** is 100, if indeed **x** has a value of 100, but if it does not, and only if it does not, it prints **x** is not 100 instead.

Several if + else structures can be concatenated with the intention of checking a range of values. For example:

```
1 if (x > 0)  
2 cout << "x is positive";  
3 else if (x < 0)  
4 cout << "x is negative";  
5 else  
6 cout << "x is 0";
```

This prints whether **x** is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: {}.

Iteration statements (loops)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords **while**, **do**, and **for**.

The while loop

The simplest kind of loop is the while-loop. Its syntax is:

```
while (expression) statement
```

The while-loop simply repeats statement while expression is true. If, after any execution of statement, expression is no longer true, the loop ends, and the program continues right after the loop. For example, let's have a look at a countdown using a while-loop:

```
1 // custom countdown using  
2 while  
3 #include <iostream>  
4 using namespace std;  
5  
6 int main ()  
7 {  
8     int n = 10;
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
liftoff!

[Edit &](#)
[Run](#)

```

9  while (n>0) {
10    cout << n << ", ";
11    --n;
12  }
13
14 cout << "liftoff!\n";
15
}

```

The first statement in `main` sets `n` to a value of 10. This is the first number in the countdown. Then the while-loop begins: if this value fulfills the condition `n>0` (that `n` is greater than zero), then the block that follows the condition is executed, and repeated for as long as the condition (`n>0`) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in `main`):

1. `n` is assigned a value
2. The `while` condition is checked (`n>0`). At this point there are two possibilities:
 - o condition is true: the statement is executed (to step 3)
 - o condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
`cout << n << ", ";`
`--n;`
 (prints the value of `n` and decreases `n` by 1)
4. End of block. Return automatically to step 2.
5. Continue the program right after the block:
`print liftoff!` and end the program.

A thing to consider with while-loops is that the loop should end at some point, and thus the statement shall alter values checked in the condition in some way, so as to force it to become false at some point. Otherwise, the loop will continue looping forever. In this case, the loop includes `--n`, that decreases the value of the variable that is being evaluated in the condition (`n`) by one - this will eventually make the condition (`n>0`) false after a certain number of loop iterations. To be more specific, after 10 iterations, `n` becomes 0, making the condition no longer true, and ending the while-loop.

Note that the complexity of this loop is trivial for a computer, and so the whole countdown is performed instantly, without any practical delay between elements of the count (if interested, see `sleep_for` for a countdown example with delays).

The do-while loop

A very similar loop is the do-while loop, whose syntax is:

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

It behaves like a while-loop, except that condition is evaluated after the execution of statement instead of before, guaranteeing at least one execution of statement, even if condition is never fulfilled. For example, the following example program echoes any text the user introduces until the user enters goodbye:

```

1 // echo machine
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string str;
9     do {
10         cout << "Enter text: ";
11         getline (cin,str);
12         cout << "You entered: " << str <<
13         '\n';
14     } while (str != "goodbye");
}

```

```

Enter text: hello
You entered: hello
Enter text: who's there? Edit &
You entered: who's Run
there?
Enter text: goodbye
You entered: goodbye

```

The do-while loop is usually preferred over a while-loop when the statement needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop ends. And thus, even if the user wants to end the loop as soon as possible by entering goodbye, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.

The for loop

The for loop is designed to iterate a number of times. Its syntax is:

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

Like the while-loop, this loop repeats statement while condition is true. But, in addition, the for loop provides specific locations to contain an initialization and an increase expression, executed before the loop begins the first time, and after each iteration, respectively. Therefore, it is especially useful to use counter variables as condition.

It works in the following way:

1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.

3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
4. increase is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

```

1 // countdown using a for
2 loop
3 #include <iostream>
4 using namespace std;
5
6 int main ()           10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
7 {                     liftoff!
8     for (int n=10; n>0; n--) {
9         cout << n << ", ";
10    }
11    cout << "liftoff!\n";
}

```

Edit &
Run

The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, `for (; n<10;)` is a loop without *initialization* or *increase* (equivalent to a while-loop); and `for (; n<10; ++n)` is a loop with *increase*, but no *initialization* (maybe because the variable was already initialized before the loop). A loop with no *condition* is equivalent to a loop with `true` as condition (i.e., an infinite loop).

Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (,): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

```

1 for ( n=0, i=100 ; n!=i ; ++n, --i )
2 {
3     // whatever here...
4 }

```

This loop will execute 50 times if neither `n` or `i` are modified within the loop:

n starts with a value of 0, and *i* with 100, the condition is $n \neq i$ (i.e., that *n* is not equal to *i*). Because *n* is increased by one, and *i* decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both *n* and *i* are equal to 50.

Range-based for loop

The for-loop has another syntax, which is used exclusively with ranges:

```
for ( declaration : range ) statement;
```

This kind of for loop iterates over all the elements in range, where declaration declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end; Most of these types have not yet been introduced in this tutorial, but we are already acquainted with at least one kind of range: strings, which are sequences of characters.

An example of range-based for loop using strings:

```
1 // range-based for loop
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string str ("Hello!");
9     for (char c : str)
10    {
11         cout << "[" << c << "]";
12     }
13     cout << '\n';
14 }
```

[H][e][l][l][o][!] [Edit & Run](#)

Note how what precedes the colon (:) in the for loop is the declaration of a char variable (the elements in a string are of type char). We then use this variable, *c*, in the statement block to represent the value of each of the elements in the range.

This loop is automatic and does not require the explicit declaration of any counter variable.

Range based loops usually also make use of type deduction for the type of the elements with auto. Typically, the range-based loop above can also be written as:

```
1 for (auto c : str)
2     cout << "[" << c << "]";
```

Here, the type of c is automatically deduced as the type of the elements in str.

Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations.

The break statement

break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, let's stop the countdown before its natural end:

```
1 // break loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     for (int n=10; n>0; n--)
8     {
9         cout << n << ", ";
10        if (n==3)
11        {
12            cout << "countdown
13 aborted!";
14            break;
15        }
16    }

```

Edit &
Run

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, let's skip number 5 in our countdown:

```
1 // continue loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()           10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
6 {
7     for (int n=10; n>0; n--) {
8         if (n==5) continue;
9         cout << n << ", ";

```

Edit & Run

40

```
cout << "liftoff!\n";
```

The goto statement

`goto` allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.

The destination point is identified by a *label*, which is then used as an argument for the `goto` statement. A *label* is made of a valid identifier followed by a colon (:).

`goto` is generally deemed a low-level feature, with no particular use cases in modern higher-level programming paradigms generally used with C++. But, just as an example, here is a version of our countdown loop using `goto`:

```
1 // goto loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int n=10;           10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff! Edit & Run
8     mylabel:
9     cout << n << ", ";
10    n--;
11    if (n>0) goto mylabel;
12    cout << "liftoff!\n";
13 }
```

Another selection statement: switch.

The syntax of the `switch` statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating `if-else` statements, but limited to constant expressions. Its most typical syntax is:

```
switch (expression)
{
    case constant1:
        group-of-statements-1;
        break;
    case constant2:
        group-of-statements-2;
```

```
        break;  
  
    default:  
        default-group-of-statements  
    }
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1; if it is, it executes group-of-statements-1 until it finds the break statement. When it finds this break statement, the program jumps to the end of the entire switch statement (the closing brace).

If expression was not equal to constant1, it is then checked against constant2. If it is equal to this, it executes group-of-statements-2 until a break is found, when it jumps to the end of the switch.

Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior, demonstrating the if-else equivalent of a switch statement:

switch example	if-else equivalent
<pre>switch (x) { case 1: cout << "x is 1"; break; case 2: cout << "x is 2"; break; default: cout << "value of x unknown"; }</pre>	<pre>if (x == 1) { cout << "x is 1"; } else if (x == 2) { cout << "x is 2"; } else { cout << "value of x unknown"; }</pre>

The switch statement has a somewhat peculiar syntax inherited from the early times of the first C compilers, because it uses labels instead of blocks. In the most typical use (shown above), this means that break statements are needed after each group of statements for a particular label. If break is not included, all statements following the case (including those under any other labels) are also executed, until the end of the switch block or a jump statement (such as break) is reached.

If the example above lacked the break statement after the first group for case one, the program would not jump automatically to the end of the switch block after printing x is 1, and would instead continue executing the statements in case two (thus printing also x is 2). It would then continue doing so until a break statement is encountered, or the end of the switch block. This makes

43

unnecessary to enclose the statements for each case in braces {}, and can also be useful to execute the same group of statements for different possible values. For example:

```
1 switch (x) {  
2     case 1:  
3     case 2:  
4     case 3:  
5         cout << "x is 1, 2 or 3";  
6     break;  
7     default:  
8         cout << "x is not 1, 2 nor 3";  
9 }
```

Notice that **switch** is limited to compare its evaluated expression against labels that are constant expressions. It is not possible to use variables as labels or ranges, because they are not valid C++ constant expressions.

To check for ranges or values that are not constant, it is better to use concatenations of **if** and **else if** statements.

Functions

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ... ) { statements }
```

Where:

- **type** is the type of the value returned by the function.
- **name** is the identifier by which the function can be called.
- **parameters** (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: `int x`), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.
- **statements** is the function's body. It is a block of statements surrounded by braces {} that specify what the function actually does.

Let's have a look at an example:

43

```

1 // function example
2 #include <iostream>
3 using namespace std;
4 int addition (int a, int b)
5 {
6     int r;
7     r=a+b;
8     return r;
9 }
10 }
11 int main ()
12 {
13     int z;
14     z = addition (5,3);
15     cout << "The result is " << z;
16 }
17 
```

The result is 8

This program is divided in two functions: `addition` and `main`. Remember that no matter the order in which they are defined, a C++ program always starts by calling `main`. In fact, `main` is the only function called automatically, and the code in any other function is only executed if its function is called from `main` (directly or indirectly).

In the example above, `main` begins by declaring the variable `z` of type `int`, and right after that, it performs the first function call: it calls `addition`. The call to a function follows a structure very similar to its declaration. In the example above, the call to `addition` can be compared to its definition just a few lines earlier:

The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, 5 and 3, to the function; these correspond to the parameters `a` and `b`, declared for function `addition`.

At the point at which the function is called from within `main`, the control is passed to function `addition`: here, execution of `main` is stopped, and will only resume once the `addition` function ends. At the moment of the function call, the value of both arguments (5 and 3) are copied to the local variables `int a` and `int b` within the function.

Then, inside `addition`, another local variable is declared (`int r`), and by means of the expression `r=a+b`, the result of `a` plus `b` is assigned to `r`; which, for this case, where `a` is 5 and `b` is 3, means that 8 is assigned to `r`.

The final statement within the function:

```
return r;
```

45

Ends function `addition`, and returns the control back to the point where the function was called; in this case: to function `main`. At this precise moment, the program resumes its course on `main` returning exactly at the same point at which it was interrupted by the call to `addition`. But additionally, because `addition` has a return type, the call is evaluated as having a value, and this value is the value specified in the return statement that ended `addition`: in this particular case, the value of the local variable `r`, which at the moment of the `return` statement had a value of 8.

Therefore, the call to `addition` is an expression with the value returned by the function, and in this case, that value, 8, is assigned to `z`. It is as if the entire function call (`addition(5, 3)`) was replaced by the value it returns (i.e., 8).

Then `main` simply prints this value by calling:

```
cout << "The result is " << z;
```

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

```
// function example
1 #include <iostream>
2 using namespace std;
3
4 int subtraction (int a, int b)
5 {
6     int r;
7     r=a-b;
8     return r;
9 }
10
11 int main ()
12 {
13     int x=5, y=3, z;
14     z = subtraction (7,2);
15     cout << "The first result is " << z << '\n';
16     cout << "The second result is " << subtraction
17 (7,2) << '\n';
18     cout << "The third result is " << subtraction
19 (x,y) << '\n';
20     z= 4 + subtraction (x,y);
21     cout << "The fourth result is " << z << '\n';
}
```

The first result
is 5
The second result
is 5
The third result [Edit &](#)
is 2 [Run](#)
The fourth result
is 6

Similar to the `addition` function in the previous example, this example defines a `subtract` function, that simply returns the difference between its two parameters. This time, `main` calls this function several times, demonstrating more possible ways in which a function can be called.

4c

Let's examine each of these calls, bearing in mind that each function call is itself an expression that is evaluated as the value it returns. Again, you can think of it as if the function call was itself replaced by the returned value:

```
1 z = subtraction (7, 2);  
2 cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
1 z = 5;  
2 cout << "The first result is " << z;
```

With the same procedure, we could interpret:

```
cout << "The second result is " << subtraction (7, 2);
```

as:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by subtraction (7, 2).

In the case of:

```
cout << "The third result is " << subtraction (x, y);
```

The arguments passed to subtraction are variables instead of literals. That is also valid, and works fine. The function is called with the values x and y have at the moment of the call: 5 and 3 respectively, returning 2 as result.

The fourth call is again similar:

```
z = 4 + subtraction (x, y);
```

The only addition being that now the function call is also an operand of an addition operation. Again, the result is the same as if the function call was replaced by its result: 6. Note, that thanks to the commutative property of additions, the above can also be written as:

```
z = subtraction (x, y) + 4;
```

With exactly the same result. Note also that the semicolon does not necessarily go after the

function call, but, as always, at the end of the whole statement. Again, the logic behind may be easily seen again by replacing the function calls by their returned value:

```
1 z = 4 + 2;      // same as z = subtraction(x,y);
2 z = 2 + 4;      // same as z = subtraction(x,y) + 4;
```

Functions with no type. The use of void

The syntax shown above for functions:

```
type name ( argument1, argument2 ... ) { statements }
```

Requires the declaration to begin with a type. This is the type of the value returned by the function. But what if the function does not need to return a value? In this case, the type to be used is **void**, which is a special type to represent the absence of value. For example, a function that simply prints a message may not need to return any value:

```
1 // void function example
2 #include <iostream>
3 using namespace std;
4
5 void printmessage ()
6 {
7     cout << "I'm a function!"; I'm a function! Edit & Run
8 }
9
10 int main ()
11 {
12     printmessage ();
13 }
```

void can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called. For example, **printmessage** could have been declared as:

```
1 void printmessage (void)
2 {
3     cout << "I'm a function!";
4 }
```

In C++, an empty parameter list can be used instead of **void** with same meaning, but the use of **void** in the argument list was popularized by the C language, where this is a requirement.

Something that in no case is optional are the parentheses that follow the function name, neither in its declaration nor when calling it. And even when the function takes no parameters, at least an

empty pair of parentheses shall always be appended to the function name. See how printmessage was called in an earlier example:

```
printmessage();
```

The parentheses are what differentiate functions from other kinds of declarations or statements.
The following would not call the function:

```
printmessage;
```

The return value of main

You may have noticed that the return type of `main` is `int`, but most examples in this and earlier chapters did not actually return any value from `main`.

Well, there is a catch: If the execution of `main` ends normally without encountering a `return` statement the compiler assumes the function ends with an implicit `return` statement:

```
return 0;
```

Note that this only applies to function `main` for historical reasons. All other functions with a return type shall end with a proper `return` statement that includes a return value, even if this is never used.

When `main` returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by `main`, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for `main` that are guaranteed to be interpreted in the same way on all platforms are:

value	Description
0	The program was successful
<u>EXIT_SUCCESS</u>	The program was successful (same as above). This value is defined in header <code><cstdlib></code> .
<u>EXIT_FAILURE</u>	The program failed. This value is defined in header <code><cstdlib></code> .

Because the implicit `return 0;` statement for `main` is a tricky exception, some authors consider it good practice to explicitly write the statement.

Arguments passed by value and by reference

In the functions seen earlier, arguments have always been passed *by value*. This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters. For example, take:

```
1 int x=5, y=3, z;  
2 z = addition ( x, y );
```

In this case, function addition is passed 5 and 3, which are copies of the values of x and y, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the function's definition, but any modification of these variables within the function has no effect on the values of the variables x and y outside it, because x and y were themselves not passed to the function on the call, but only copies of their values at that moment.

In certain cases, though, it may be useful to access an external variable from within a function. To do that, arguments can be passed *by reference*, instead of *by value*. For example, the function duplicate in this code duplicates the value of its three arguments, causing the variables used as arguments to actually be modified by the call:

```
1 // passing parameters by reference  
2 #include <iostream>  
3 using namespace std;  
4  
5 void duplicate (int& a, int& b, int& c)  
6 {  
7     a*=2;  
8     b*=2;  
9     c*=2;  
10 }  
11  
12 int main ()  
13 {  
14     int x=1, y=3, z=7;  
15     duplicate (x, y, z);  
16     cout << "x=" << x << ", y=" << y << ", z=" << z;  
17     return 0;  
18 }
```

x=2, y=6, z=14 [Edit & Run](#)

To gain access to its arguments, the function declares its parameters as *references*. In C++, references are indicated with an ampersand (&) following the parameter type, as in the parameters taken by duplicate in the example above.

When a variable is passed by reference, what is passed is no longer a copy, but the variable itself, the variable identified by the function parameter, becomes somehow associated with the argument passed to the function, and any modification on their corresponding local variables within the function are reflected in the variables passed as arguments in the call.

In fact, a, b, and c become aliases of the arguments passed on the function call (x, y, and z) and any change on a within the function is actually modifying variable x outside the function. Any change on b modifies y, and any change on c modifies z. That is why when, in the example, function duplicate modifies the values of variables a, b, and c, the values of x, y, and z are affected.

If instead of defining duplicate as:

```
void duplicate (int& a, int& b, int& c)
```

Was it to be defined without the ampersand signs as:

```
void duplicate (int a, int b, int c)
```

The variables would not be passed by reference, but by value, creating instead copies of their values. In this case, the output of the program would have been the values of x, y, and z without being modified (i.e., 1, 3, and 7).

Efficiency considerations and const references

Calling a function with parameters taken by value causes copies of the values to be made. This is a relatively inexpensive operation for fundamental types such as int, but if the parameter is of a large compound type, it may result on certain overhead. For example, consider the following function:

```
1 string concatenate (string a, string b)
2 {
3     return a+b;
4 }
```

This function takes two strings as parameters (by value), and returns the result of concatenating them. By passing the arguments by value, the function forces a and b to be copies of the arguments passed to the function when it is called. And if these are long strings, it may mean

Copying large quantities of data just for the function call.

But this copy can be avoided altogether if both parameters are made *references*:

```
1 string concatenate (string& a, string& b)
2 {
3     return a+b;
4 }
```

Arguments by reference do not require a copy. The function operates directly on (aliases of) the strings passed as arguments, and, at most, it might mean the transfer of certain pointers to the function. In this regard, the version of `concatenate` taking references is more efficient than the version taking values, since it does not need to copy expensive-to-copy strings.

On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for.

The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as `constant`:

```
1 string concatenate (const string& a, const string& b)
2 {
3     return a+b;
4 }
```

By qualifying them as `const`, the function is forbidden to modify the values of neither `a` nor `b`, but can actually access their values as references (aliases of the arguments), without having to make actual copies of the strings.

Therefore, `const` references provide functionality similar to passing arguments by value, but with an increased efficiency for parameters of large types. That is why they are extremely popular in C++ for arguments of compound types. Note though, that for most fundamental types, there is no noticeable difference in efficiency, and in some cases, `const` references may even be less efficient!

Inline functions

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

Preceding a function declaration with the `inline` specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does

not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

For example, the concatenate function above may be declared inline as:

```
1 inline string concatenate (const string& a, const string& b)
2 {
3     return a+b;
4 }
```

This informs the compiler that when concatenate is called, the program prefers the function to be expanded inline, instead of performing a regular call. `inline` is only specified in the function declaration, not when it is called.

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the `inline` specifier. Therefore, this specifier merely indicates the compiler that `inline` is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

Default values in parameters

In C++, functions can also have optional parameters, for which no arguments are required in the call, in such a way that, for example, a function with three parameters may be called with only two. For this, the function shall include a default value for its last parameter, which is used by the function when called with fewer arguments. For example:

```
1 // default values in functions
2 #include <iostream>
3 using namespace std;
4
5 int divide (int a, int b=2)
6 {
7     int r;
8     r=a/b;
9     return (r);
10 }
11
12 int main ()
13 {
14     cout << divide (12) << '\n';
15     cout << divide (20,4) << '\n';
16     return 0;
17 }
```

⁶ Edit & Run

In this example, there are two calls to function divide. In the first one:

`divide (12)`

The call only passes one argument to the function, even though the function has two parameters. In this case, the function assumes the second parameter to be 2 (notice the function definition, which declares its second parameter as `int b=2`). Therefore, the result is 6.

In the second call:

`divide (20, 4)`

The call passes two arguments to the function. Therefore, the default value for `b` (`int b=2`) is ignored, and `b` takes the value passed as argument, that is 4, yielding a result of 5.

Declaring functions

In C++, identifiers can only be used in expressions once they have been declared. For example, some variable `x` cannot be used before being declared with a statement, such as:

```
int x;
```

The same applies to functions. Functions cannot be called before they are declared. That is why, in all the previous examples of functions, the functions were always defined before the `main` function, which is the function from where the other functions were called. If `main` were defined before the other functions, this would break the rule that functions shall be declared before being used, and thus would not compile.

The prototype of a function can be declared without actually defining the function completely, giving just enough details to allow the types involved in a function call to be known. Naturally, the function shall be defined somewhere else, like later in the code. But at least, once declared like this, it can already be called.

The declaration shall include all types involved (the return type and the type of its arguments), using the same syntax as used in the definition of the function, but replacing the body of the function (the block of statements) with an ending semicolon.

The parameter list does not need to include the parameter names, but only their types. Parameter names can nevertheless be specified, but they are optional, and do not need to necessarily match

those in the function definition. For example, a function called `protofunction` with two int parameters can be declared with either of these statements:

```
1 int protofunction (int first, int second);
2 int protofunction (int, int);
```

Anyway, including a name for each parameter always improves legibility of the declaration.

```
// declaring functions prototypes
#include <iostream>
1 using namespace std;
2
3 void odd (int x);
4 void even (int x);
5
6 int main()
7 {
8     int i;
9     do {
10         cout << "Please, enter number (0 to
11 to exit): ";
12         cin >> i;
13         odd (i);
14     } while (i!=0);
15     return 0;
16 }
17
18 void odd (int x)
19 {
20     if ((x%2)!=0) cout << "It is
21 odd.\n";
22     else even (x);
23 }
24
25 void even (int x)
26 {
27     if ((x%2)==0) cout << "It is
28 even.\n";
29     else odd (x);
30 }
```

Please, enter number (0 to
exit): 9
It is odd.
Please, enter number (0 to
exit): 6
It is even.
Please, enter number (0 to
exit): 1030
It is even.
Please, enter number (0 to
exit): 0
It is even.

Edit &
Run

This example is indeed not an example of efficiency. You can probably write yourself a version of this program with half the lines of code. Anyway, this example illustrates how functions can be declared before its definition:

The following lines:

```
1 void odd (int a);
2 void even (int a);
```

Declare the prototype of the functions. They already contain all what is necessary to call them, their name, the types of their argument, and their return type (`void` in this case). With these prototype declarations in place, they can be called before they are entirely defined, allowing for example, to place the function from where they are called (`main`) before the actual definition of these functions.

But declaring functions before being defined is not only useful to reorganize the order of functions within the code. In some cases, such as in this particular case, at least one of the declarations is required, because `odd` and `even` are mutually called; there is a call to `even` in `odd` and a call to `odd` in `even`. And, therefore, there is no way to structure the code so that `odd` is defined before `even`, and `even` before `odd`.

Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for some tasks, such as sorting elements, or calculating the factorial of numbers. For example, in order to obtain the factorial of a number ($n!$) the mathematical formula would be:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

More concretely, $5!$ (factorial of 5) would be:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

And a recursive function to calculate this in C++ could be:

```
1 // factorial calculator
2 #include <iostream>
3 using namespace std;
4
5 long factorial (long a)
6 {
7     if (a > 1)
8         return (a * factorial (a-1));
9     else
10        return 1;
11 }
12
13 int main ()
14 {
15     long number = 9;
16     cout << number << "!" = " << factorial (number);
17     return 0;
18 }
```

$9! = 362880$ [Edit & Run](#)

Notice how in function `factorial` we included a call to itself, but only if the argument passed was

since it arrived to 0, it would continue multiplying by all the negative numbers (probably provoking a stack overflow at some point during runtime).

5

Overloads and templates

Overloaded functions

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type.

For example:

```
1 // overloading functions
2 #include <iostream>
3 using namespace std;
4
5 int operate (int a, int b)
6 {
7     return (a*b);
8 }
9
10 double operate (double a, double b)           10 Edit & Run
11 {
12     return (a/b);
13 }
14
15 int main ()
16 {
17     int x=5,y=2;
18     double n=5.0,m=2.0;
19     cout << operate (x,y) << '\n';
20     cout << operate (n,m) << '\n';
21     return 0;
22 }
```

2.5

In this example, there are two functions called `operate`, but one of them has two parameters of type `int`, while the other has them of type `double`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two `int` arguments, it calls to the function that has two `int` parameters, and if it is called with two `doubles`, it calls the one with two `doubles`.

In this example, both functions have quite different behaviors, the `int` version multiplies its arguments, while the `double` version divides them. This is generally not a good idea. Two functions with the same name are generally expected to have -at least- a similar behavior, but this example demonstrates that is

57

entirely possible for them not to. Two overloaded functions (i.e., two functions with the same name) have entirely different definitions; they are, for all purposes, different functions, that only happen to have the same name.

Note that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Function templates

Overloaded functions may have the same definition. For example:

```
1 // overloaded functions
2 #include <iostream>
3 using namespace std;
4
5 int sum (int a, int b)
6 {
7     return a+b;
8 }
9
10 double sum (double a, double b) 30
11                               2.5
12 {
13     return a+b;
14 }
15 int main ()
16 {
17     cout << sum (10,20) << '\n';
18     cout << sum (1.0,1.5) << '\n';
19     return 0;
20 }
```

Here, `sum` is overloaded with different parameter types, but with the exact same body.

The function `sum` could be overloaded for a lot of types, and it could make sense for all of them to have the same body. For cases such as this, C++ has the ability to define functions with generic types, known as *function templates*. Defining a function template follows the same syntax as a regular function, except that it is preceded by the `template` keyword and a series of template parameters enclosed in angle-brackets `<>`:

```
template <template-parameters> function-declaration
The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the class or typename keyword followed by an identifier.
```

This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

```
1 template <class SomeType>
2 SomeType sum (SomeType a, SomeType b)
3 {
4     return a+b;
5 }
```

It makes no difference whether the generic type is specified with keyword `class` or keyword `typename` in the template argument list (they are 100% synonyms in template declarations).

In the code above, declaring `SomeType` (a generic type within the template parameters enclosed in angle-brackets) allows `SomeType` to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the *function template*, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:

```
name <template-arguments> (function-arguments)
```

For example, the `sum` function template defined above can be called with:

```
x = sum<int>(10, 20);
```

The function `sum<int>` is just one of the possible instantiations of function template `sum`. In this case, by using `int` as template argument in the call, the compiler automatically instantiates a version of `sum` where each occurrence of `SomeType` is replaced by `int`, as if it was defined as:

```
1 int sum (int a, int b)
2 {
3     return a+b;
4 }
```

Let's see an actual example:

```
1 // function template
2 #include <iostream>
3 using namespace std;
4
5 template <class T>
6 T sum (T a, T b)
```

```
1 T result;
2 result = a + b;
3 return result;
4 }
5
6 int main () {
7     int i=5, j=6, k;
8     double f=2.0, g=0.5, h;
9     k=sum<int>(i,j);
10    h=sum<double>(f,g);
11    cout << k << '\n';
12    cout << h << '\n';
13    return 0;
14 }
```

In this case, we have used `T` as the template parameter name, instead of `SomeType`. It makes no difference, and `T` is actually a quite common template parameter name for generic types.

In the example above, we used the function template `sum` twice. The first time with arguments of type `int`, and the second one with arguments of type `double`. The compiler has instantiated and then called each time the appropriate version of the function.

Note also how `T` is also used to declare a local variable of that (generic) type within `sum`:

```
T result;
```

Therefore, `result` will be a variable of the same type as the parameters `a` and `b`, and as the type returned by the function.

In this specific case where the generic type `T` is used as a parameter for `sum`, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets.

Therefore, instead of explicitly specifying the template arguments with:

```
1 k = sum<int> (i,j);
2 h = sum<double> (f,g);
```

It is possible to instead simply write:

```
1 k = sum (i,j);
2 h = sum (f,g);
```

without the type enclosed in angle brackets. Naturally, for that, the type shall be unambiguous. If `sum` is called with arguments of different types, the compiler may not be able to deduce the type of `T`.

automatically.

Templates are a powerful and versatile feature. They can have multiple template parameters, and the function can still use regular non-templated types. For example:

```

1 // function templates
2 #include <iostream>
3 using namespace std;
4 template <class T, class U>
5 bool are_equal (T a, U b)
6 {
7     return (a==b);
8 }
10 int main ()
11 {
12     if (are_equal(10,10.0))
13         cout << "x and y are equal\n";
14     else
15         cout << "x and y are not equal\n";
16     return 0;
17 }
```

x and y are equal Edit & Run

Note that this example uses automatic template parameter deduction in the call to `are_equal`:

`are_equal(10,10.0)`

Is equivalent to:

`are_equal<int,double>(10,10.0)`

There is no ambiguity possible because numerical literals are always of a specific type: Unless otherwise specified with a suffix, integer literals always produce values of type `int`, and floating-point literals always produce values of type `double`. Therefore 10 has always type `int` and 10.0 has always type `double`.

Non-type template arguments

The template parameters can not only include types introduced by `class` or `typename`, but can also include expressions of a particular type:

```

template arguments
1 include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
    return val * N;
}

int main() {
    std::cout << fixed_multiply<int,2>(10) << '\n';
    std::cout << fixed_multiply<int,3>(10) << '\n';
}

```

20 Edit & Run
30

The second argument of the `fixed_multiply` function template is of type `int`. It just looks like a regular function parameter, and can actually be used just like one.

But there exists a major difference: the value of template parameters is determined on compile-time to generate a different instantiation of the function `fixed_multiply`, and thus the value of that argument is never passed during runtime: The two calls to `fixed_multiply` in `main` essentially call two versions of the function: one that always multiplies by two, and one that always multiplies by three. For that same reason, the second template argument needs to be a constant expression (it cannot be passed a variable).

Name visibility

Scopes

Named entities, such as variables, functions, and compound types need to be declared before being used in C++. The point in the program where this declaration happens influences its visibility:

An entity declared outside any block has *global scope*, meaning that its name is valid anywhere in the code. While an entity declared within a block, such as a function or a selective statement, has *block scope*, and is only visible within the specific block in which it is declared, but not outside it.

Variables with block scope are known as *local variables*.

For example, a variable declared in the body of a function is a *local variable* that extends until the end of the function (i.e., until the brace `}` that closes the function definition), but not outside it:

```
1 int foo;           // global variable
```

```

1 int some_function ()           // local variable
2 {
3     int bar;
4     bar = 0;
5 }
6
7 int other_function ()
8 {
9     foo = 1; // ok: foo is a global variable
10    bar = 2; // wrong: bar is not visible from this function
11 }
12
13 }
```

In each scope, a name can only represent one entity. For example, there cannot be two variables with the same name in the same scope:

```

1 int some_function ()
2 {
3     int x;
4     x = 0;
5     double x; // wrong: name already used in this scope
6     x = 0.0;
7 }
```

The visibility of an entity with *block scope* extends until the end of the block, including inner blocks. Nevertheless, an inner block, because it is a different block, can re-utilize a name existing in an outer scope to refer to a different entity; in this case, the name will refer to a different entity only within the inner block, hiding the entity it names outside. While outside it, it will still refer to the original entity. For example:

```

1 // inner block scopes
2 #include <iostream>
3 using namespace std;
4
5 int main () {
6     int x = 10;
7     int y = 20;
8     {
9         int x; // ok, inner scope.
10        x = 50; // sets value to inner x
11        y = 50; // sets value to (outer) y
12        cout << "inner block:\n";
13        cout << "x: " << x << '\n';
14        cout << "y: " << y << '\n';
15    }
16    cout << "outer block:\n";
17    cout << "x: " << x << '\n';
18    cout << "y: " << y << '\n';
19    return 0;
```

Note that `y` is not hidden in the inner block, and thus accessing `y` still accesses the outer variable.
Variables declared in declarations that introduce a block, such as function parameters and variables declared in loops and conditions (such as those declared on a `for` or an `if`) are local to the block they introduce.

Namespaces

Only one entity can exist with a particular name in a particular scope. This is seldom a problem for local names, since blocks tend to be relatively short, and names have particular purposes within them, such as naming a counter variable, an argument, etc...

But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

The syntax to declare a namespaces is:

```
namespace identifier  
{  
    named_entities  
}
```

Where `identifier` is any valid identifier and `named_entities` is the set of variables, types and functions that are included within the namespace. For example:

```
1 namespace myNamespace  
2 {  
3     int a, b;  
4 }
```

In this case, the variables `a` and `b` are normal variables declared within a namespace called `myNamespace`.

use operator ::. For example, to access the previous variables from outside my namespace they
will be qualified like:

```
myNamespace::a  
myNamespace::b
```

Namespaces are particularly useful to avoid name collisions. For example:

```
1 // namespaces  
2 #include <iostream>  
3 using namespace std;  
4  
5 namespace foo  
6 {  
7     int value() { return 5; }  
8 }  
9  
10 namespace bar  
11 {  
12     const double pi = 3.1416;  
13     double value() { return 2*pi; }  
14 }  
15  
16 int main () {  
17     cout << foo::value() << '\n';  
18     cout << bar::value() << '\n';  
19     cout << bar::pi << '\n';  
20     return 0;  
21 }
```

5
6.2832
3.1416

Edit & Run

In this case, there are two functions with the same name: value. One is defined within the namespace foo, and the other one in bar. No redefinition errors happen thanks to namespaces. Notice also how pi is accessed in an unqualified manner from within namespace bar (just as pi), while it is again accessed in main, but here it needs to be qualified as bar::pi.

Namespaces can be split: Two segments of a code can be declared in the same namespace:

```
1 namespace foo { int a; }  
2 namespace bar { int b; }  
3 namespace foo { int c; }
```

This declares three variables: a and c are in namespace foo, while b is in namespace bar. Namespaces

can even extend across different translation units (i.e., across different files of source code).

using

The keyword `using` introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example:

```
1 // using
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8     int y = 10;
9 }
10
11 namespace second
12 {
13     double x = 3.1416;
14     double y = 2.7183;
15 }
16
17 int main () {
18     using first::x;
19     using second::y;
20     cout << x << '\n';
21     cout << y << '\n';
22     cout << first::y << '\n';
23     cout << second::x << '\n';
24     return 0;
25 }
```

5
2.7183 Edit & Run
10
3.1416

Notice how in `main`, the variable `x` (without any name qualifier) refers to `first::x`, whereas `y` refers to `second::y`, just as specified by the `using` declarations. The variables `first::y` and `second::x` can still be accessed, but require fully qualified names.

The keyword `using` can also be used as a directive to introduce an entire namespace:

```
1 // using
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8 }
```

5
10 Edit & Run
3.1416
2.7183

66

```
1 int y = 10;
2
3 namespace second
4 {
5     double x = 3.1416;
6     double y = 2.7183;
7 }
8
9 int main () {
10    using namespace first;
11    cout << x << '\n';
12    cout << y << '\n';
13    cout << second::x << '\n';
14    cout << second::y << '\n';
15 }
16
17 int main () {
18    using namespace first;
19    cout << x << '\n';
20    cout << y << '\n';
21    cout << second::x << '\n';
22    cout << second::y << '\n';
23 }
24 }
```

In this case, by declaring that we were using namespace `first`, all direct uses of `x` and `y` without name qualifiers were also looked up in namespace `first`.

using and using namespace have validity only in the same block in which they are stated or in the entire source code file if they are used directly in the global scope. For example, it would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks:

```
1 // using namespace example
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8 }
9
10 namespace second
11 {
12     double x = 3.1416;      5      Edit & Run
13 }                         3.1416
14
15 int main () {
16 {
17     using namespace first;
18     cout << x << '\n';
19 }
20 {
21     using namespace second;
22     cout << x << '\n';
23 }
24 return 0;
25 }
```

66

Namespace aliasing

Existing namespaces can be aliased with new names, with the following syntax:

```
namespace new_name = current_name;
```

The std namespace

All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the `std` namespace. Most examples in these tutorials, in fact, include the following line:

```
using namespace std;
```

This introduces direct visibility of all the names of the `std` namespace into the code. This is done in these tutorials to facilitate comprehension and shorten the length of the examples, but many programmers prefer to qualify each of the elements of the standard library used in their programs. For example, instead of:

```
cout << "Hello world!";
```

It is common to instead see:

```
std::cout << "Hello world!";
```

Whether the elements in the `std` namespace are introduced with `using` declarations or are fully qualified on every use does not change the behavior or efficiency of the resulting program in any way. It is mostly a matter of style preference, although for projects mixing libraries, explicit qualification tends to be preferred.

Storage classes

The storage for variables with *global* or *namespace scope* is allocated for the entire duration of the program. This is known as *static storage*, and it contrasts with the storage for *local variables* (those

(declared within a block). These use what is known as automatic storage. The storage for local variables is only available during the block in which they are declared; after that, that same storage may be used for a local variable of some other function, or used otherwise.

But there is another substantial difference between variables with static storage and variables with automatic storage:

- Variables with static storage (such as global variables) that are not explicitly initialized are automatically initialized to zeroes.
- Variables with automatic storage (such as local variables) that are not explicitly initialized are left uninitialized, and thus have an undetermined value.

For example:

```

1 // static vs automatic storage
2 #include <iostream>
3 using namespace std;
4
5 int x;
6
7 int main ()
8 {
9     int y;
10    cout << x << '\n';
11    cout << y << '\n';
12    return 0;
13 }
```

0 Edit & Run
4285838

The actual output may vary, but only the value of x is guaranteed to be zero. y can actually contain just about any value (including zero).

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type int called foo could be represented as:

CO

where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.

Therefore, the foo array, with five elements of type int, can be declared as:

```
int foo [5];
```

NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

Initializing arrays

By default, regular arrays of local scope (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

This statement declares an array that can be represented like this:

The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

Will create an array like this:

The initializer can even have no values, just the braces:

```
int baz [5] = {};
```

This creates an array of five int values, each initialized with a value of zero:

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

```
1 int foo[] = { 10, 20, 30 };
2 int foo[] { 10, 20, 30 };
```

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

Accessing the values of an array

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

name [index]

Following the previous examples in which `foo` had 5 elements and each of those elements was of type `int`, the name which can be used to refer to each element is the following:

For example, the following statement stores the value 75 in the third element of `foo`:

```
foo[2] = 75;
```

and, for example, the following copies the value of the third element of `foo` to a variable called `x`:

```
x = foo[2];
```

Therefore, the expression `foo[2]` is itself a variable of type `int`.

Notice that the third element of `foo` is specified `foo[2]`, since the first one is `foo[0]`, the second one is `foo[1]`, and therefore, the third one is `foo[2]`. By this same reason, its last element is `foo[4]`. Therefore, if we write `foo[5]`, we would be accessing the sixth element of `foo`, and therefore actually exceeding the size of the array.

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced.

At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets [] with arrays.

```
1 int foo[5];           // declaration of a new array
2 foo[2] = 75;          // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

Some other valid operations with arrays:

```
1 foo[0] = a;  
2 foo[a] = 75;  
3 b = foo[a+2];  
4 foo[foo[a]] = foo[2] + 5;
```

For example:

```
1 // arrays example  
2 #include <iostream>  
3 using namespace std;  
4  
5 int foo [] = {16, 2, 77, 40, 12071};  
6 int n, result=0;  
7  
8 int main ()  
9 {  
10    for ( n=0 ; n<5 ; ++n )  
11    {  
12        result += foo[n];  
13    }  
14    cout << result;  
15    return 0;  
16 }
```

12206 Edit & Run

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

jimmy represents a bidimensional array of 3 per 5 elements of type int. The C++ syntax for this is:

```
int jimmy [3][5];
```

for example, the way to reference the second element vertically and fourth horizontally in an array would be:

jimmy[1][3]

(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

```
char century [100][365][24][60][60];
```

declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory!

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

```
1 int jimmy [3][5]; // is equivalent to  
2 int jimmy [15]; // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bidimensional array while the other uses a simple array:

multidimensional array

```
#define WIDTH 5  
#define HEIGHT 3  
  
int jimmy [HEIGHT][WIDTH];  
int n,m;  
  
int main ()  
{  
    for (n=0; n<HEIGHT; n++)  
        for (m=0; m<WIDTH; m++)  
    {  
        jimmy[n][m]=(n+1)*(m+1);  
    }
```

pseudo-multidimensional array

```
#define WIDTH 5  
#define HEIGHT 3  
  
int jimmy [HEIGHT * WIDTH];  
int n,m;  
  
int main ()  
{  
    for (n=0; n<HEIGHT; n++)  
        for (m=0; m<WIDTH; m++)  
    {  
        jimmy[n*WIDTH+m]=(n+1)*(m+1);  
    }
```

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

Note that the code uses defined constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

Arrays as parameters

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
void procedure (int arg[])
```

This function accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
1 // arrays as parameters
2 #include <iostream>
3 using namespace std;
4
5 void printarray (int arg[], int length) {
6     for (int i = 0; i < length; i++) {
7         cout << arg[i] << " ";
8     }
9 }
```

5 10 15 Edit & Run
2 4 6 8 10

7A

```

8     for (int n=0; n<length; ++n)
9         cout << arg[n] << ' ';
10    cout << '\n';
11 }
12 int main ()
13 {
14     int firstarray[] = {5, 10, 15};
15     int secondarray[] = {2, 4, 6, 8, 10};
16     printarray (firstarray, 3);
17     printarray (secondarray, 5);
}

```

In the code above, the first parameter (`int arg[]`) accepts any array whose elements are of type `int`, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the array passed, without going out of range.

In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

`base_type[][][depth]`

For example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets [] are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers. Although a clear understanding of pointers, explained in a coming chapter, helps a lot.

Library arrays

The arrays explained above are directly implemented as a language feature, inherited from the C language. They are a great feature, but by restricting its copy and easily decay into pointers, they

ably suffer from an excess of optimization.

overcome some of these issues with language built-in arrays, C++ provides an alternative array type standard container. It is a type template (a class template, in fact) defined in header `<array>`.

Containers are a library feature that falls out of the scope of this tutorial, and thus the class will not be explained in detail here. Suffice it to say that they operate in a similar way to built-in arrays, except that they allow being copied (an actually expensive operation that copies the entire block of memory, and thus to use with care) and decay into pointers only when explicitly told to do so (by means of its member data).

Just as an example, these are two versions of the same example using the language built-in array described in this chapter, and the container in the library:

language built-in array	container library array
#include <iostream> using namespace std; int main() { int myarray[3] = {10,20,30}; for (int i=0; i<3; ++i) ++myarray[i]; for (int elem : myarray) cout << elem << '\n'; }	#include <iostream> #include <array> using namespace std; int main() { array<int,3> myarray {10,20,30}; for (int i=0; i<myarray.size(); ++i) ++myarray[i]; for (int elem : myarray) cout << elem << '\n'; }

As you can see, both kinds of arrays use the same syntax to access its elements: `myarray[i]`. Other than that, the main differences lay on the declaration of the array, and the inclusion of an additional header for the *library array*. Notice also how it is easy to access the size of the *library array*.

Character sequences

The `string` class has been briefly introduced in an earlier chapter. It is a very powerful class to handle and manipulate strings of characters. However, because strings are, in fact, sequences of characters, we can represent them also as plain arrays of elements of a character type.

For example, the following array:

```
char foo [20];
```

an array that can store up to 20 elements of type `char`. It can be represented as:

Therefore, this array has a capacity to store sequences of up to 20 characters. But this capacity does not need to be fully exhausted: the array can also accommodate shorter sequences. For example, at some point in a program, either the sequence "Hello" or the sequence "Merry Christmas" can be stored in `foo`, since both would fit in a sequence with a capacity for 20 characters.

By convention, the end of strings represented in character sequences is signaled by a special character: the *null character*, whose literal value can be written as '`\0`' (backslash, zero).

In this case, the array of 20 elements of type `char` called `foo` can be represented storing the character sequences "Hello" and "Merry Christmas" as:

Notice how after the content of the string itself, a null character ('`\0`') has been added in order to indicate the end of the sequence. The panels in gray color represent `char` elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays, they follow the same rules as these. For example, to initialize an array of characters with some predetermined sequence of characters, we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The above declares an array of 6 elements of type `char` initialized with the characters that form the word "Hello" plus a *null character* '`\0`' at the end.

But arrays of character elements have another way to be initialized: using *string literals* directly.

In the expressions used in some examples in previous chapters, string literals have already shown up several times. These are specified by enclosing the text between double quotes (""). For example:

the result is:

This is a *string literal*, probably used in some earlier example.

Sequences of characters enclosed in double-quotes ("") are *literal constants*. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

Therefore, the array of char elements called myword can be initialized with a null-terminated sequence of characters by either one of these two statements:

```
1 char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 char myword[] = "Hello";
```

In both cases, the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello", plus a final null character ('\0'), which specifies the end of the sequence and that, in the second case, when using double quotes ("") it is appended automatically.

Please notice that here we are talking about initializing an array of characters at the moment it is being declared, and not about assigning values to them later (once they have already been declared). In fact, because string literals are regular arrays, they have the same restrictions as these, and cannot be assigned values.

Expressions (once myword has already been declared as above), such as:

```
1 myword = "Bye";
2 myword[] = "Bye";
```

would not be valid, like neither would be:

```
myword = ( 'B', 'y', 'e', '\0' );
```

This is because arrays cannot be assigned values. Note, though, that each of its elements can be assigned a value individually. For example, this would be correct:

```
1 myword[0] = 'B';
2 myword[1] = 'y';
3 myword[2] = 'e';
4 myword[3] = '\0';
```

Strings and null-terminated character sequences

Plain arrays with null-terminated sequences of characters are the typical types used in the C language to represent strings (that is why they are also known as C-strings). In C++, even though the standard library defines a specific type for strings (class `string`), still, plain arrays with null-terminated sequences of characters (C-strings) are a natural way of representing strings in the language; in fact, string literals still always produce null-terminated character sequences, and not `string` objects.

In the standard library, both representations for strings (C-strings and library strings) coexist, and most functions requiring strings are overloaded to support both.

For example, `cin` and `cout` support null-terminated sequences directly, allowing them to be directly extracted from `cin` or inserted into `cout`, just like strings. For example:

```
// strings and NTCS:  
1 #include <iostream>  
2 #include <string>  
3 using namespace std;  
4  
5 int main ()  
6 {  
7     char question1[] = "What is your name?  
8     ";  
9     string question2 = "Where do you live?  
10    ";  
11    char answer1 [80];  
12    string answer2;  
13    cout << question1;  
14    cin >> answer1;  
15    cout << question2;  
16    cin >> answer2;  
17    cout << "Hello, " << answer1;  
18    cout << " from " << answer2 << "!\n";  
19    return 0;  
}
```

What is your name? Homer
Where do you live?
Greece
Hello, Homer from
Greece!

[Edit &](#)
[Run](#)

In this example, both arrays of characters using null-terminated sequences and strings are used. They are quite interchangeable in their use together with `cin` and `cout`, but there is a notable difference in their declarations: arrays have a fixed size that needs to be specified either implicit or explicitly when declared; `question1` has a size of exactly 20 characters (including the terminating null-characters) and `answer1` has a size of 80 characters; while strings are simply strings, no size is specified. This is due to the fact that strings have a dynamic size determined during runtime, while the size of arrays is

determined on compilation, before the program runs.

In any case, null-terminated character sequences and strings are easily transformed from one another.

Null-terminated character sequences can be transformed into strings implicitly, and strings can be transformed into null-terminated character sequences by using either of string's member functions `c_str` or `data`:

```
1 char myntcs[] = "some text";
2 string mystring = myntcs; // convert c-string to string
3 cout << mystring; // printed as a library string
4 cout << mystring.c_str(); // printed as a c-string
```

(note: both `c_str` and `data` members of `string` are equivalent)

Pointers

In earlier chapters, variables have been explained as locations in the computer's memory which can be accessed by their identifier (their name). This way, the program does not need to care about the physical address of the data in memory; it simply uses the identifier whenever it needs to refer to the variable.

For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.

This way, each cell can be easily located in the memory by means of its unique address. For example, the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776.

When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored. Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime. However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

INTRODUCTION TO PROGRAMMING IN JAVA

Java Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java -

Primitive Data Types

Reference/Object Data Types

Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

byte

Byte data type is an 8-bit signed two's complement integer

Minimum value is -128 (-2^7)

Maximum value is 127 (inclusive) ($2^7 - 1$)

Default value is 0

Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.

Example: byte a = 100, byte b = -50

short

Short data type is a 16-bit signed two's complement integer

Minimum value is -32,768 (-2^{15})

Maximum value is 32,767 (inclusive) ($2^{15} - 1$)

Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer

Default value is 0.

Example: short s = 10000, short r = -20000

int

Int data type is a 32-bit signed two's complement integer.

Minimum value is - 2,147,483,648 (- 2^{31})

Maximum value is 2,147,483,647(inclusive) ($2^{31} - 1$)

Integer is generally used as the default data type for integral values unless there is a concern about memory.

The default value is 0

Example: int a = 100000, int b = -200000

long

Long data type is a 64-bit signed two's complement integer

Minimum value is -9,223,372,036,854,775,808(- 2^{63})

Maximum value is 9,223,372,036,854,775,807 (inclusive) ($2^{63} - 1$)

This type is used when a wider range than int is needed

Default value is 0L

Example: long a = 100000L, long b = -200000L

float

Float data type is a single-precision 32-bit IEEE 754 floating point

Float is mainly used to save memory in large arrays of floating point numbers

Default value is 0.0f

Float data type is never used for precise values such as currency

Example: float f1 = 234.5f

double

double data type is a double-precision 64-bit IEEE 754 floating point

This data type is generally used as the default data type for decimal values, generally the default choice

Double data type should never be used for precise values such as currency

Default value is 0.0d

Example: double d1 = 123.4

boolean

boolean data type represents one bit of information

There are only two possible values: true and false

This data type is used for simple flags that track true/false conditions

Default value is false

Example: boolean one = true

char

char data type is a single 16-bit Unicode character

Minimum value is '\u0000' (or 0)

Maximum value is '\uffff' (or 65,535 inclusive)

Char data type is used to store any character

Example: char letterA = 'A'

Reference Datatypes

Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.

Class objects and various type of array variables come under reference datatype.

Default value of any reference variable is null.

A reference variable can be used to refer any object of the declared type or any compatible type.

Example: Animal animal = new Animal("giraffe");

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example -

byte a = 68;

char a = 'A'

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example -

```
int decimal = 100;  
int octal = 0144;  
int hexa = 0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are -

Example

"Hello World"

"two\nlines"

"\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example -

```
char a = '\u0001';
```

```
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are -

Notation	Character represented
----------	-----------------------

\n	Newline (0x0a)
----	----------------

\r	Carriage return (0x0d)
----	------------------------

\f	Formfeed (0x0c)
----	-----------------

\b	Backspace (0x08)
----	------------------

\s	Space (0x20)
----	--------------

\t	tab
----	-----

\"	Double quote
----	--------------

\'	Single quote
----	--------------

\\\	backslash
-----	-----------

\ddd	Octal character (ddd)
------	-----------------------

\xxxxx	Hexadecimal UNICODE character (xxxx)
--------	--------------------------------------

Different variable types and their usage

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration -

```
data type variable [= value][, variable [= value] ...];
```

Here data type is one of Java's datatypes and variable is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java -

Example

```
int a, b, c;           // Declares three ints, a, b, and c.  
int a = 10, b = 10;   // Example of initialization  
byte B = 22;          // initializes a byte type variable B.  
double pi = 3.14159; // declares and assigns a value of PI.  
char a = 'a';          // the char variable a is initialized with value 'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java -

Local variables

Instance variables

Class/Static variables

Local Variables

Local variables are declared in methods, constructors, or blocks.

Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.

Access modifiers cannot be used for local variables.

Local variables are visible only within the declared method, constructor, or block.

Local variables are implemented at stack level internally.

There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

Here, age is a local variable. This is defined inside pupAge() method and its scope is limited to only this method.

```
public class Test {  
    public void pupAge() {  
        int age = 0;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

| This will produce the following result

Output

Puppy age is: 7

Example

Following example uses age without initializing it, so it would give an error at the time of compilation.

```
public class Test {  
    public void pupAge() {  
        int age;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

This will produce the following error while compiling it -

Output

Test.java:4:variable number might not have been initialized

age = age + 7;

^

1 error

Instance Variables

Instance variables are declared in a class, but outside a method, constructor or any block.

When a space is allocated for an object in the heap, a slot for each instance variable value is created.

Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.

Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.

Instance variables can be declared in class level before or after use.

Access modifiers can be given for instance variables.

The instance variables are visible for all methods, constructors and block in the class. Normally, it is

recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.

Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.

Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. ObjectReference.VariableName.

Example

```
import java.io.*;
public class Employee {
    // this instance variable is visible for any child class.
    public String name;
    // salary variable is visible in Employee class only.
    private double salary;
    // The name variable is assigned in the constructor.
    public Employee (String empName) {
        name = empName;
    }
    // The salary variable is assigned a value.
    public void setSalary(double empSal) {
        salary = empSal;
    }
    // This method prints the employee details.
    public void printEmp() {
        System.out.println("name : " + name );
        System.out.println("salary :" + salary);
    }
}
public static void main(String args[]) {
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
}
```

This will produce the following result -

87

8

Output

name : Ransika

salary : 1000.0

Class/Static Variables

Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.

There would only be one copy of each class variable per class, regardless of how many objects are created from it.

Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.

Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Static variables are created when the program starts and destroyed when the program stops.

Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.

Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.

Static variables can be accessed by calling with the class name Classname.VariableName.

When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

Example

```
import java.io.*;
public class Employee {
    // salary variable is a private static variable
    private static double salary;
    // DEPARTMENT is a constant
    public static final String DEPARTMENT = "Development";
    public static void main(String args[]) {
```

```
    salary = 1000;
    System.out.println(DEPARTMENT + "average salary:" + salary);
}
}
```

This will produce the following result -

Output

Development average salary:1000

Note - If the variables are accessed from an outside class, the constant should be accessed as Employee.DEPARTMENT

Java - Modifier Types

Modifiers are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following -

Java Access Modifiers

Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following example.

Example

```
public class className {
    // ...
}

private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;
```

```
public static void main(String[] arguments) {
    // body of method
}
```

Access Control Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are -

Visible to the package, the default. No modifiers are needed.

Visible to the class only (private).

Visible to the world (public).

Visible to the package and all subclasses (protected).

Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

The static modifier for creating class methods and variables.

The final modifier for finalizing the implementations of classes, methods, and variables.

The abstract modifier for creating abstract classes and methods.

The synchronized and volatile modifiers, which are used for threads.

Java - Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups -

Arithmetic Operators

Relational Operators

Bitwise Operators

Logical Operators

Assignment Operators

Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators -

Assume integer variable A holds 10 and variable B holds 20, then -

Show Examples

Operator Description Example

+ (Addition) Adds values on either side of the operator. A + B will give 30

- (Subtraction) Subtracts right-hand operand from left-hand operand. A - B will give -10

* (Multiplication) Multiplies values on either side of the operator. A * B will give 200

/ (Division) Divides left-hand operand by right-hand operand. B / A will give 2

% (Modulus) Divides left-hand operand by right-hand operand and returns remainder. B % A will give 0

++ (Increment) Increases the value of operand by 1. B++ gives 21

-- (Decrement) Decreases the value of operand by 1. B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then -

Show Examples

Operator Description Example

$=$ (equal to) Checks if the values of two operands are equal or not, if yes then condition becomes true. ($A == B$) is not true.

$!=$ (not equal to) Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. ($A != B$) is true.

$>$ (greater than) Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. ($A > B$) is not true.

$<$ (less than) Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. ($A < B$) is true.

\geq (greater than or equal to) Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. ($A \geq B$) is not true.

\leq (less than or equal to) Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. ($A \leq B$) is true.

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if $a = 60$ and $b = 13$; now in binary format they will be as follows -

$a = 0011\ 1100$

$b = 0000\ 1101$

$a \& b = 0000\ 1100$

$a | b = 0011\ 1101$

$a ^ b = 0011\ 0001$

$-a = 1100\ 0011$

The following table lists the bitwise operators -

Assume integer variable A holds 60 and variable B holds 13 then -

Show Examples

Operator Description Example

& (bitwise and) Binary AND Operator copies a bit to the result if it exists in both operands.
(A & B) will give 12 which is 0000 1100

|(bitwise or) Binary OR Operator copies a bit if it exists in either operand. (A | B) will give 61
which is 0011 1101

^ (bitwise XOR) Binary XOR Operator copies the bit if it is set in one operand but not both.(A ^ B) will give 49 which is 0011 0001

- (bitwise compliment) Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. (~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.

<< (left shift) Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand. A << 2 will give 240 which is 1111 0000

>> (right shift) Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. A >> 2 will give 15 which is 1111

>>> (zero fill right shift) Shift right zero fill operator. The left operand's value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. A >>> 2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators -

Assume Boolean variables A holds true and variable B holds false, then -

Show Examples

Operator Description Example

&& (logical and) Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. (A && B) is false

|| (logical or) Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. (A || B) is true

! (logical not) Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. !(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language -

Show Examples

Operator Description Example

= Simple assignment operator. Assigns values from right side operands to left side operand. C =
A + B will assign value of A + B into C

+= Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand. C += A is equivalent to C = C + A

-= Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand. C -= A is equivalent to C = C - A

*= Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand. C *= A is equivalent to C = C * A

$/=$ Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand. $C /= A$ is equivalent to $C = C / A$

$\%=$ Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand. $C \%= A$ is equivalent to $C = C \% A$

$<<=$ Left shift AND assignment operator. $C <<= 2$ is same as $C = C << 2$

$>>=$ Right shift AND assignment operator. $C >>= 2$ is same as $C = C >> 2$

$\&=$ Bitwise AND assignment operator. $C \&= 2$ is same as $C = C \& 2$

$\sim=$ bitwise exclusive OR and assignment operator. $C \sim= 2$ is same as $C = C \sim 2$

$|=$ bitwise inclusive OR and assignment operator. $C |= 2$ is same as $C = C | 2$

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (?:)

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as -

`variable x = (expression) ? value if true : value if false`

Following is an example -

Example

```
public class Test {
    public static void main(String args[]) {
        int a, b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " + b );
        b = (a == 10) ? 20: 30;
        System.out.println( "Value of b is : " + b );
    }
}
```

This will produce the following result -

Output

Value of b is : 30

Value of b is : 20

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as -

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example -

Example

```
public class Test {  
  
    public static void main(String args[]) {  
  
        String name = "James";  
  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println(result);  
    }  
}
```

This will produce the following result -

Output

true

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example -

Example

```
class Vehicle {}  
  
public class Car extends Vehicle {  
  
    public static void main(String args[]) {  
  
        Vehicle a = new Car();  
        boolean result = a instanceof Car;  
        System.out.println(result);  
    }  
}
```

This will produce the following result -

Output

True

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator -

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3 * 2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category Operator Associativity

Postfix $>() [] .$ (dot operator) Left to right

Unary $>++ - - ! ~$ Right to left

Multiplicative $>* /$ Left to right

Additive $>+ -$ Left to right

Shift $>>> >>> <<$ Left to right

Relational $>>= <<=$ Left to right

Equality $>== !=$ Left to right

Bitwise AND $>&$ Left to right

Bitwise XOR $>^$ Left to right

Bitwise OR $>|$ Left to right

Logical AND $>&&$ Left to right

Logical OR $>||$ Left to right

Conditional $? :$ Right to left

Assignment $>= += -= *= /= \% = >>= <<= \&= ^= |=$ Right to left

What is Next?

The next chapter will explain about loop control in Java programming. The chapter will describe various types of loops and how these loops can be used in Java program development and for what purposes they are being used.

Java - Loop Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages -

Loop Architecture

Java programming language provides the following types of loop to handle looping requirements.
Click the following links to check their detail.

Loop & Description

1 while loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

2 for loop

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

3 do...while loop

Like a while statement, except that it tests the condition at the end of the loop body.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Java supports the following control statements.

Control Statement & Description

1 break statement

Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.

2 continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Enhanced for loop in Java

As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.

Syntax

Following is the syntax of enhanced for loop -

```
for(declaration : expression) {
```

```
    // Statements
```

```
}
```

Declaration - The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.

Expression - This evaluates to the array you need to loop through. The expression can be an array

variable or method call that returns an array.

Example

```
public class Test {
```

```
    public static void main(String args[]) {
        int [] numbers = {10, 20, 30, 40, 50};

        for(int x : numbers ) {
            System.out.print( x );
            System.out.print(",");
        }
        System.out.print("\n");
        String [] names = {"James", "Larry", "Tom", "Lacy"};

        for( String name : names ) {
            System.out.print( name );
            System.out.print(",");
        }
    }
}
```

This will produce the following result -

Output

```
10, 20, 30, 40, 50,
James, Larry, Tom, Lacy,
tutorialspoint
```

Java - Decision Making

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages -

Decision Making

Java programming language provides following types of decision making statements. Click the following links to check their detail.

Statement & Description

- | if statement

An if statement consists of a boolean expression followed by one or more statements.

2 if...else statement

An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

3 nested if statement

You can use one if or else if statement inside another if or else if statement(s).

4 switch statement

A switch statement allows a variable to be tested for equality against a list of values.

The ?: Operator

We have covered conditional operator ? : in the previous chapter which can be used to replace if...else statements. It has the following general form -

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

To determine the value of the whole expression, initially exp1 is evaluated.

If the value of exp1 is true, then the value of Exp2 will be the value of the whole expression.

If the value of exp1 is false, then Exp3 is evaluated and its value becomes the value of the entire expression.

Java - Object and Classes

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts -

Polymorphism

Inheritance

Encapsulation

Abstraction

Classes

Objects

Instance

Method

Message Parsing

concepts of Classes and Objects.

Object - Objects have states and behaviors. Example: A dog has states - color, name, breed as well

as behaviors - wagging the tail, barking, eating. An object is an instance of a class.

Class - A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

Example

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
  
    void barking() {  
    }  
  
    void hungry() {  
    }  
  
    void sleeping() {  
    }  
}
```

A class can contain any of the following variable types.

Local variables - Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be

destroyed when the method has completed.

Instance variables - Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Class variables - Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor -

Example

```
public class Puppy {  
    public Puppy() {  
    }  
  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
    }  
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Note - We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

(07)

There are three steps when creating an object from a class -

Declaration - A variable declaration with a variable name with an object type.

Instantiation - The 'new' keyword is used to create the object.

Initialization - The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object -

Example

```
public class Puppy {  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
        System.out.println("Passed Name is :" + name );  
    }  
  
    public static void main(String []args) {  
        // Following statement would create an object myPuppy  
        Puppy myPuppy = new Puppy( "tommy" );  
    }  
}
```

If we compile and run the above program, then it will produce the following result -

Output

Passed Name is :tommy

Accessing Instance Variables and Methods

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path -

```
/* First create an object */  
ObjectReference = new Constructor();  
  
/* Now call a variable as follows */  
ObjectReference.variableName;
```

```
/* Now you can call a class method as follows */  
ObjectReference.MethodName();
```

Example

This example explains how to access instance variables and methods of a class.

```
public class Puppy {  
    int puppyAge;
```

```

public Puppy(String name) {
    // This constructor has one parameter, name.
    System.out.println("Name chosen is :" + name);
}

public void setAge( int age ) {
    puppyAge = age;
}

public int getAge( ) {
    System.out.println("Puppy's age is :" + puppyAge );
    return puppyAge;
}

public static void main(String []args) {
    /* Object creation */
    Puppy myPuppy = new Puppy( "tommy" );

    /* Call class method to set puppy's age */
    myPuppy.setAge( 2 );

    /* Call another class method to get puppy's age */
    myPuppy.getAge( );

    /* You can access instance variable as follows as well */
    System.out.println("Variable Value :" + myPuppy.puppyAge );
}
}

```

If we compile and run the above program, then it will produce the following result -

Output

Name chosen is :tommy

Puppy's age is :2

Variable Value :2

Source File Declaration Rules

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, import statements and package statements in a source file.

There can be only one public class per source file.

A source file can have multiple non-public classes.

The public class name should be the name of the source file as well which should be appended

by java at the end. For example: the class name is public class Employee{} then the source file should be as Employee.java.

If the class is defined inside a package, then the package statement should be the first statement in the source file.

If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.

Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.

Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import Statements

In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory java_installation/java/io -

```
import java.io.*;
```

A Simple Case Study

For our case study, we will be creating two classes. They are Employee and EmployeeTest.

First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.

The Employee class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

Example

```
import java.io.*;
```

```

public class Employee {
    String name;
    int age;
    String designation;
    double salary;

    // This is the constructor of the class Employee
    public Employee(String name) {
        this.name = name;
    }

    // Assign the age of the Employee to the variable age.
    public void empAge(int empAge) {
        age = empAge;
    }

    /* Assign the designation to the variable designation.*/
    public void empDesignation(String empDesig) {
        designation = empDesig;
    }

    /* Assign the salary to the variable salary.*/
    public void empSalary(double empSalary) {
        salary = empSalary;
    }

    /* Print the Employee details */
    public void printEmployee() {
        System.out.println("Name:" + name );
        System.out.println("Age:" + age );
        System.out.println("Designation:" + designation );
        System.out.println("Salary:" + salary);
    }
}

```

As mentioned previously in this tutorial, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.

Following is the EmployeeTest class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file.

```
import java.io.*;
public class EmployeeTest {
    public static void main(String args[]) {
        /* Create two objects using constructor */
        Employee empOne = new Employee("James Smith");
        Employee empTwo = new Employee("Mary Anne");

        // Invoking methods for each object created
        empOne.empAge(26);
        empOne.empDesignation("Senior Software Engineer");
        empOne.empSalary(1000);
        empOne.printEmployee();

        empTwo.empAge(21);
        empTwo.empDesignation("Software Engineer");
        empTwo.empSalary(500);
        empTwo.printEmployee();
    }
}
```

Now, compile both the classes and then run EmployeeTest to see the result as follows -

Output
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0