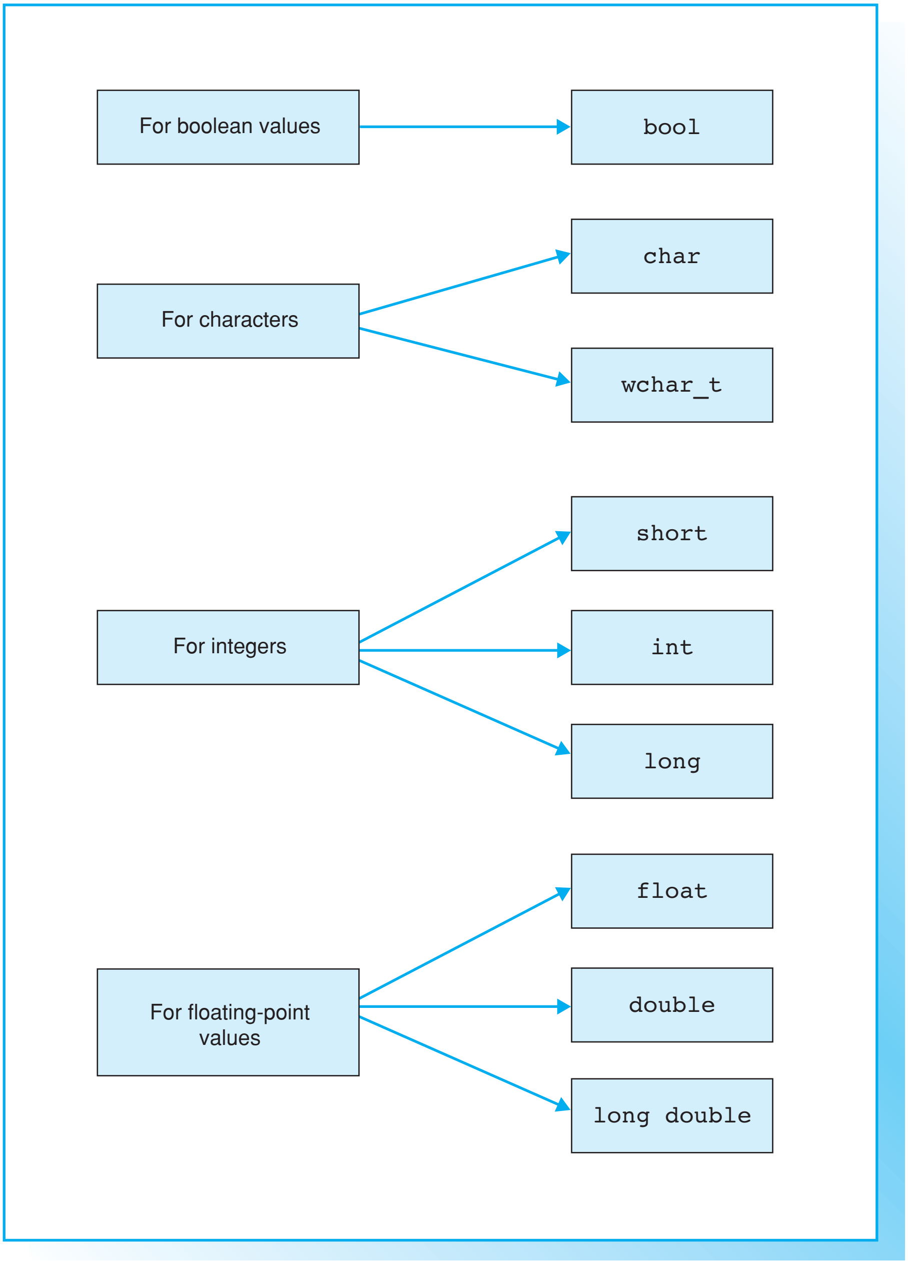
chapter 2

Fundamental Types, Constants, and Variables

This chapter introduces you to the basic types and objects used by C++ programs.

■ FUNDAMENTAL TYPES

Overview*



* without type `void`, which will be introduced later.

A program can use several data to solve a given problem, for example, characters, integers, or floating-point numbers. Since a computer uses different methods for processing and saving data, the data *type* must be known. The type defines

1. the internal representation of the data, and
2. the amount of memory to allocate.

A number such as `-1000` can be stored in either 2 or 4 bytes. When accessing the part of memory in which the number is stored, it is important to read the correct number of bytes. Moreover, the memory content, that is the bit sequence being read, must be interpreted correctly as a signed integer.

The C++ compiler recognizes the *fundamental types*, also referred to as *built-in types*, shown on the opposite page, on which all other types (vectors, pointers, classes, ...) are based.

☐ The Type `bool`

The result of a comparison or a logical association using AND or OR is a *boolean* value, which can be true or false. C++ uses the `bool` type to represent boolean values. An expression of the type `bool` can either be `true` or `false`, where the internal value for `true` will be represented as the numerical value 1 and `false` by a zero.

☐ The `char` and `wchar_t` Types

These types are used for saving character codes. A *character code* is an integer associated with each character. The letter `A` is represented by code 65, for example. The *character set* defines which code represents a certain character. When displaying characters on screen, the applicable character codes are transmitted and the "receiver," that is the screen, is responsible for correctly interpreting the codes.

The C++ language does not stipulate any particular characters set, although in general a character set that contains the *ASCII code* (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) is used. This 7-bit code contains definitions for 32 control characters (codes 0 – 31) and 96 printable characters (codes 32 – 127).

The `char` (character) type is used to store character codes in one byte (8 bits). This amount of storage is sufficient for extended character sets, for example, the ANSI character set that contains the ASCII codes and additional characters such as German umlauts.

The `wchar_t` (wide character type) type comprises at least 2 bytes (16 bits) and is thus capable of storing modern Unicode characters. *Unicode* is a 16-bit code also used in Windows NT and containing codes for approximately 35,000 characters in 24 languages.

■ FUNDAMENTAL TYPES (CONTINUED)

Integral types

Type	Size	Range of Values (decimal)
`char`	1 byte	–128 to +127 or 0 to 255
`unsigned char`	1 byte	0 to 255
`signed char`	1 byte	–128 to +127
`int`	2 byte resp. 4 byte	–32768 to +32767 resp. –2147483648 to +2147483647
`unsigned int`	2 byte resp. 4 byte	0 to 65535 resp. 0 to 4294967295
`short`	2 byte	–32768 to +32767
`unsigned short`	2 byte	0 to 65535
`long`	4 byte	–2147483648 to +2147483647
`unsigned long`	4 byte	0 to 4294967295

Sample program

```
#include <iostream>
#include <climits>       // Definition of INT_MIN, ...
using namespace std;

int main()
{
  cout << "Range of types int and unsigned int"
       << endl << endl;
  cout << "Type             Minimum          Maximum"
       << endl
       << "--------------------------------------------"
       << endl;

  cout << "int              " <<  INT_MIN << "         "
                              <<  INT_MAX << endl;

  cout << "unsigned int     " <<  "          0         "
                              << UINT_MAX << endl;
  return 0;
}
```

☐ Integral Types

The types `short`, `int`, and `long` are available for operations with integers. These types are distinguished by their ranges of values. The table on the opposite page shows the integer types, which are also referred to as *integral types*, with their typical storage requirements and ranges of values.

The `int` (integer) type is tailor-made for computers and adapts to the length of a register on the computer. For 16-bit computers, `int` is thus equivalent to `short`, whereas for 32-bit computers `int` will be equivalent to `long`.

C++ treats character codes just like normal integers. This means you can perform calculations with variables belonging to the `char` or `wchar_t` types in exactly the same way as with `int` type variables. `char` is an integral type with a size of one byte. The range of values is thus –128 to +127 or from 0 to 255, depending on whether the compiler interprets the `char` type as signed or unsigned. This can vary in C++.

The `wchar_t` type is a further integral type and is normally defined as `unsigned short`.

☐ The `signed` and `unsigned` Modifiers

The `short`, `int`, and `long` types are normally interpreted as signed with the highest bit representing the sign. However, integral types can be preceded by the keyword `unsigned`. The amount of memory required remains unaltered but the range of values changes due to the highest bit no longer being required as a sign. The keyword `unsigned` can be used as an abbreviation for `unsigned int`.

The `char` type is also normally interpreted as signed. Since this is merely a convention and not mandatory, the `signed` keyword is available. Thus three types are available: `char`, `signed char`, and `unsigned char`.

✓ NOTE

In ANSI C++ the size of integer types is not preset. However, the following order applies:

```
char <= short <= int <= long
```

Moreover, the `short` type comprises at least 2 bytes and the `long` type at least 4 bytes.

The current value ranges are available in the `climits` header file. This file defines constants such as `CHAR_MIN`, `CHAR_MAX`, `INT_MIN`, and `INT_MAX`, which represent the smallest and greatest possible values. The program on the opposite page outputs the value of these constants for the `int` and `unsigned int` types.

■ FUNDAMENTAL TYPES (CONTINUED)

Floating-point types

Type	Size	Range of Values	Lowest Positive Value	Accuracy (decimal)
`float`	4 bytes	–3.4E+38	1.2E—38	6 digits
`double`	8 bytes	–1.7E+308	2.3E—308	15 digits
`long double`	10 bytes	–1.1E+4932	3.4E—4932	19 digits

NOTE

IEEE format (IEEE = **I**nstitute of **E**lectrical and **E**lectronic **E**ngineers) is normally used to represent floating-point types. The table above makes use of this representation.

Arithmetic types

Integral types

```
bool
char, signed char, unsigned char, wchar_t
short, unsigned short
int, unsigned int
long, unsigned long
```

Floating-point types

```
float
double
long double
```

NOTE

Arithmetic operators are defined for arithmetic types, i.e. you can perform calculations with variables of this type.

☐ Floating-Point Types

Numbers with a fraction part are indicated by a decimal point in C++ and are referred to as floating-point numbers. In contrast to integers, floating-point numbers must be stored to a preset accuracy. The following three types are available for calculations involving floating-point numbers:

`float`	for simple accuracy
`double`	for double accuracy
`long double`	for high accuracy

The value range and accuracy of a type are derived from the amount of memory allocated and the internal representation of the type.

Accuracy is expressed in decimal places. This means that "six decimal places" allows a programmer to store two floating-point numbers that differ within the first six decimal places as separate numbers. In reverse, there is no guarantee that the figures 12.3456 and 12.34561 will be distinguished when working to a accuracy of six decimal places. And remember, it is not a question of the position of the decimal point, but merely of the numerical sequence.

If it is important for your program to display floating-point numbers with an accuracy supported by a particular machine, you should refer to the values defined in the `cfloat` header file.

Readers interested in additional material on this subject should refer to the Appendix, which contains a section on the representation of binary numbers on computers for both integers and floating-point numbers.

☐ The `sizeof` Operator

The amount of memory needed to store an object of a certain type can be ascertained using the `sizeof` operator:

```
sizeof(name)
```

yields the size of an object in bytes, and the parameter `name` indicates the object type or the object itself. For example, `sizeof(int)` represents a value of 2 or 4 depending on the machine. In contrast, `sizeof(float)` will always equal 4.

☐ Classification

The fundamental types in C++ are *integer types*, *floating-point types*, and the `void` type. The types used for integers and floating-point numbers are collectively referred to as *arithmetic types*, as arithmetic operators are defined for them.

The `void` type is used for expressions that do not represent a value. A function call can thus take a `void` type.

■ CONSTANTS

Examples for integral constants

Decimal	Octal	Hexadecimal	Type
16	020	0x10	int
255	0377	OXff	int
32767	077777	0x7FFF	int
32768U	0100000U	0x8000U	unsigned int
100000	0303240	0x186A0	int (32 bit-) long (16 bit- CPU)
10L	012L	0xAL	long
27UL	033UL	0x1bUL	unsigned long
2147483648	020000000000	0x80000000	unsigned long

NOTE

In each line of the above table, the same value is presented in a different way.

Sample program

```
// To display hexadecimal integer literals and
// decimal integer literals.
//
#include <iostream>
using namespace std;

int main()
{
  // cout outputs integers as decimal integers:
  cout << "Value of 0xFF = " << 0xFF << " decimal"
       << endl;                  // Output: 255 decimal
  // The manipulator hex changes output to hexadecimal
  // format (dec changes to decimal format):
  cout << "Value of 27 = " << hex << 27 <<" hexadecimal"
       << endl;               // Output: 1b hexadecimal
  return 0;
}
```

The boolean keywords `true` and `false`, a number, a character, or a character sequence (*string*) are all constants, which are also referred to as a *literals*. Constants can thus be subdivided into

- *boolean constants*
- *numerical constants*
- *character constants*
- *string constants*.

Every constant represents a value and thus a type—as does every expression in C++. The type is defined by the way the constant is written.

Boolean Constants

A boolean expression can have two values that are identified by the keywords `true` and `false`. Both constants are of the `bool` type. They can be used, for example, to set flags representing just two states.

Integral Constants

Integral numerical constants can be represented as simple decimal numbers, octals, or hexadecimals:

- a *decimal constant* (base 10) begins with a decimal number other than zero, such as 109 or 987650
- an *octal constant* (base 8) begins with a leading 0, for example 077 or 01234567
- a *hexadecimal constant* (base 16) begins with the character pair 0x or 0X, for example 0x2A0 or 0X4b1C. Hexadecimal numbers can be capitalized or non-capitalized.

Integral constants are normally of type `int`. If the value of the constant is too large for the `int` type, a type capable of representing larger values will be applied. The ranking for decimal constants is as follows:

```
int, long, unsigned long
```

You can designate the type of a constant by adding the letter `L` or `l` (for `long`), or `U` or `u` (for `unsigned`). For example,

`12L`	and	`12l`	correspond to the type `long`
`12U`	and	`12u`	correspond to the type `unsigned int`
`12UL`	and	`12ul`	correspond to the type `unsigned long`

■ CONSTANTS (CONTINUED)

Examples for floating-point constants

5.19	12.	0.75	0.00004
0.519E1	12.0	.75	0.4e-4
0.0519e2	.12E+2	7.5e-1	.4E-4
519.OE-2	12e0	75E-2	4E-5

Examples for character constants

Constant	Character	Constant Value (ASCII code decimal)
`'A'`	Capital A	65
`'a'`	Lowercase a	97
`' '`	Blank	32
`'.'`	Dot	46
`'0'`	Digit 0	48
`'\0'`	Terminating null character	0

Internal representation of a string literal

String literal: `"Hello!"`

Stored byte sequence:

`'H'`	`'e'`	`'1'`	`'1'`	`'o'`	`'!'`	`'\0'`

☐ Floating-Point Constants

Floating-point numbers are always represented as decimals, a decimal point being used to distinguish the fraction part from the integer part. However, exponential notation is also permissible.

EXAMPLES:
```
27.1      1.8E-2      // Type: double
```

Here, `1.8E-2` represents a value of `1.8*10`$^{-2}$. `E` can also be written with a small letter `e`. A decimal point or `E` (`e`) must always be used to distinguish floating-point constants from integer constants.

Floating-point constants are of type `double` by default. However, you can add `F` or `f` to designate the `float` type, or add `L` or `l` for the `long double` type.

☐ Character Constants

A character constant is a character enclosed in *single* quotes. Character constants take the type `char`.

EXAMPLE:
```
'A'     // Type: char
```

The numerical value is the character code representing the character. The constant `'A'` thus has a value of `65` in ASCII code.

☐ String Constants

You already know string constants, which were introduced for text output using the `cout` stream. A string constant consists of a sequence of characters enclosed in *double* quotes.

EXAMPLE:
```
"Today is a beautiful day!"
```

A string constant is stored internally without the quotes but terminated with a *null character*, `\0`, represented by a byte with a numerical value of `0` — that is, all the bits in this byte are set to `0`. Thus, a string occupies one byte more in memory than the number of characters it contains. An *empty string*, `""`, therefore occupies a single byte.

The terminating null character `\0` is not the same as the number zero and has a different character code than zero. Thus, the string

EXAMPLE:
```
"0"
```

comprises two bytes, the first byte containing the code for the character zero `0` (ASCII code 48) and the second byte the value `0`.

The terminating null character `\0` is an example of an escape sequence. Escape sequences are described in the following section.

■ ESCAPE SEQUENCES

Overview

Single character	Meaning	ASCII code (decimal)
`\a`	alert (BEL)	7
`\b`	backspace (BS)	8
`\t`	horizontal tab (HT)	9
`\n`	line feed (LF)	10
`\v`	vertical tab (VT)	11
`\f`	form feed (FF)	12
`\r`	carriage return (CR)	13
`\"`	" (double quote)	34
`\'`	' (single quote)	39
`\?`	? (question mark)	63
`\\`	\ (backslash)	92
`\0`	string terminating character	0
`\ooo` (up to 3 octal digits)	numerical value of a character	ooo (octal!)
`\xhh` (hexadecimal digits)	numerical value of a character	hh (hexadecimal!)

Sample program

```
#include <iostream>
using namespace std;
int main()
{
   cout << "\nThis is\t a string\n\t\t"
           " with \"many\" escape sequences!\n";
   return 0;
}
```

Program output:

```
This is         a string
                with "many" escape sequences!
```

☐ Using Control and Special Characters

Nongraphic characters can be expressed by means of *escape sequences*, for example `\t`, which represents a tab.

The effect of an escape sequence will depend on the device concerned. The sequence `\t`, for example, depends on the setting for the tab width, which defaults to eight blanks but can be any value.

An escape sequence always begins with a \ (backslash) and represents a single character. The table on the opposite page shows the standard escape sequences, their decimal values, and effects.

You can use octal and hexadecimal escape sequences to create any character code. Thus, the letter `A` (decimal 65) in ASCII code can also be expressed as `\101` (three octals) or `\x41` (two hexadecimals). Traditionally, escape sequences are used only to represent non-printable characters and special characters. The control sequences for screen and printer drivers are, for example, initiated by the ESC character (decimal 27), which can be represented as `\33` or `\x1b`.

Escape sequences are used in character and string constants.

EXAMPLES: `'\t'` `"\tHello\n\tMike!"`

The characters ', ", and **** have no special significance when preceded by a backslash, i.e. they can be represented as `\'`, `\"`, and `\\` respectively.

When using octal numbers for escape sequences in strings, be sure to use three digits, for example, `\033` and not `\33`. This helps to avoid any subsequent numbers being evaluated as part of the escape sequence. There is no maximum number of digits in a hexadecimal escape sequence. The sequence of hex numbers automatically terminates with the first character that is not a valid hex number.

The sample program on the opposite page demonstrates the use of escape sequences in strings. The fact that a string can occupy two lines is another new feature. String constants separated only by white spaces will be concatenated to form a *single* string.

To continue a string in the next line you can also use a backslash \ as the last character in a line, and then press the Enter key to begin a new line, where you can continue typing the string.

EXAMPLE:

```
"I am a very, very \
long string"
```

Please note, however, that the leading spaces in the second line will be evaluated as part of the string. It is thus generally preferable to use the first method, that is, to terminate the string with " and reopen it with ".

■ NAMES

Keywords in C++

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

Examples for names

valid:

```
a           US          us       VOID
_var        SetTextColor
B12         top_of_window
a_very_long_name123467890
```

invalid:

```
goto    586_cpu     object-oriented
US$     true        écu
```

☐ Valid Names

Within a program *names* are used to designate variables and functions. The following rules apply when creating names, which are also known as *identifiers*:

- a name contains a series of letters, numbers, or underscore characters (_). German umlauts and accented letters are invalid. C++ is case sensitive; that is, upper- and lowercase letters are different.
- the first character must be a letter or underscore
- there are no restrictions on the length of a name and all the characters in the name are significant
- C++ keywords are reserved and cannot be used as names.

The opposite page shows C++ keywords and some examples of valid and invalid names.

The C++ compiler uses internal names that begin with one or two underscores followed by a capital letter. To avoid confusion with these names, avoid use of the underscore at the beginning of a name.

Under normal circumstances the linker only evaluates a set number of characters, for example, the first 8 characters of a name. For this reason names of global objects, such as functions, should be chosen so that the first eight characters are significant.

☐ Conventions

In C++ it is standard practice to use small letters for the names of variables and functions. The names of some variables tend to be associated with a specific use.

EXAMPLES:

`c, ch`	for characters
`i, j, k, l, m, n`	for integers, in particular indices
`x, y, z`	for floating-point numbers

To improve the readability of your programs you should choose longer and more self-explanatory names, such as `start_index` or `startIndex` for the first index in a range of index values.

In the case of software projects, naming conventions will normally apply. For example, prefixes that indicate the type of the variable may be assigned when naming variables.

■ VARIABLES

Sample program

```
// Definition and use of variables
#include <iostream>
using namespace std;

int gVar1;                      // Global variables,
int gVar2 = 2;                  // explicit initialization

int main()
{
   char ch('A');  // Local variable being initialized
                  // or:  char ch = 'A';

   cout << "Value of gVar1:    " << gVar1  << endl;
   cout << "Value of gVar2:    " << gVar2  << endl;
   cout << "Character in ch:   " << ch     << endl;

   int sum, number = 3; // Local variables with
                        // and without initialization
   sum = number + 5;
   cout << "Value of sum:      " << sum  << endl;

   return 0;
}
```

HINT

Both strings and all other values of fundamental types can be output with `cout`. Integers are printed in decimal format by default.

Screen output

```
Value of gVar1:   0
Value of gVar2:   2
Character in ch:  A
Value of sum:     8
```

Data such as numbers, characters, or even complete records are stored in *variables* to enable their processing by a program. Variables are also referred to as *objects*, particularly if they belong to a class.

☐ Defining Variables

A variable must be defined before you can use it in a program. When you *define* a variable the type is specified and an appropriate amount of memory reserved. This memory space is addressed by reference to the name of the variable. A simple definition has the following syntax:

SYNTAX: `typ name1 [name2 ... ];`

This defines the names of the variables in the list `name1 [, name2 ...]` as variables of the type `type`. The parentheses `[ ... ]` in the syntax description indicate that this part is optional and can be omitted. Thus, one or more variables can be stated within a single definition.

EXAMPLES:

```
char c;
int i, counter;
double x, y, size;
```

In a program, variables can be defined either within the program's functions or outside of them. This has the following effect:

- a variable defined outside of each function is *global*, i.e. it can be used by all functions
- a variable defined within a function is *local*, i.e. it can be used only in that function.

Local variables are normally defined immediately after the first brace—for example at the beginning of a function. However, they can be defined wherever a statement is permitted. This means that variables can be defined immediately before they are used by the program.

☐ Initialization

A variable can be initialized, i.e. a value can be assigned to the variable, during its definition. Initialization is achieved by placing the following immediately after the name of the variable:

- an equals sign (=) and an initial value for the variable or
- round brackets containing the value of the variable.

EXAMPLES:

```
char c = 'a';
float x(1.875);
```

Any *global* variables not explicitly initialized default to zero. In contrast, the initial value for any local variables that you fail to initialize will have an undefined initial value.

■ THE KEYWORDS `const` AND `volatile`

Sample program

```
// Circumference and area of a circle with radius 2.5

#include <iostream>
using namespace std;

const double pi = 3.141593;

int main()
{
   double area, circuit, radius = 1.5;

   area = pi * radius * radius;
   circuit = 2 * pi * radius;

   cout << "\nTo Evaluate a Circle\n" << endl;

   cout << "Radius:         " << radius    << endl
        << "Circumference:  " << circuit   << endl
        << "Area:           " << area      << endl;

   return 0;
}
```

NOTE

By default `cout` outputs a floating-point number with a maximum of 6 decimal places without trailing zeros.

Screen output

```
To Evaluate a Circle

Radius:          1.5
Circumference:   9.42478
Area:            7.06858
```

A type can be modified using the `const` and `volatile` keywords.

☐ Constant Objects

The `const` keyword is used to create a "read only" object. As an object of this type is constant, it cannot be modified at a later stage and must be initialized during its definition.

EXAMPLE: `const double pi = 3.1415947;`

Thus the value of `pi` cannot be modified by the program. Even a statement such as the following will merely result in an error message:

```
pi = pi + 2.0;              // invalid
```

☐ Volatile Objects

The keyword `volatile`, which is rarely used, creates variables that can be modified not only by the program but also by other programs and external events. Events can be initiated by interrupts or by a hardware clock, for example.

EXAMPLE: `volatile unsigned long  clock_ticks;`

Even if the program itself does not modify the variable, the compiler must assume that the value of the variable has changed since it was last accessed. The compiler therefore creates machine code to read the value of the variable whenever it is accessed instead of repeatedly using a value that has been read at a prior stage.

It is also possible to combine the keywords `const` and `volatile` when declaring a variable.

EXAMPLE: `volatile const unsigned time_to_live;`

Based on this declaration, the variable `time_to_live` cannot be modified by the program but by external events.