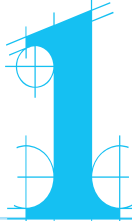




chapter



1

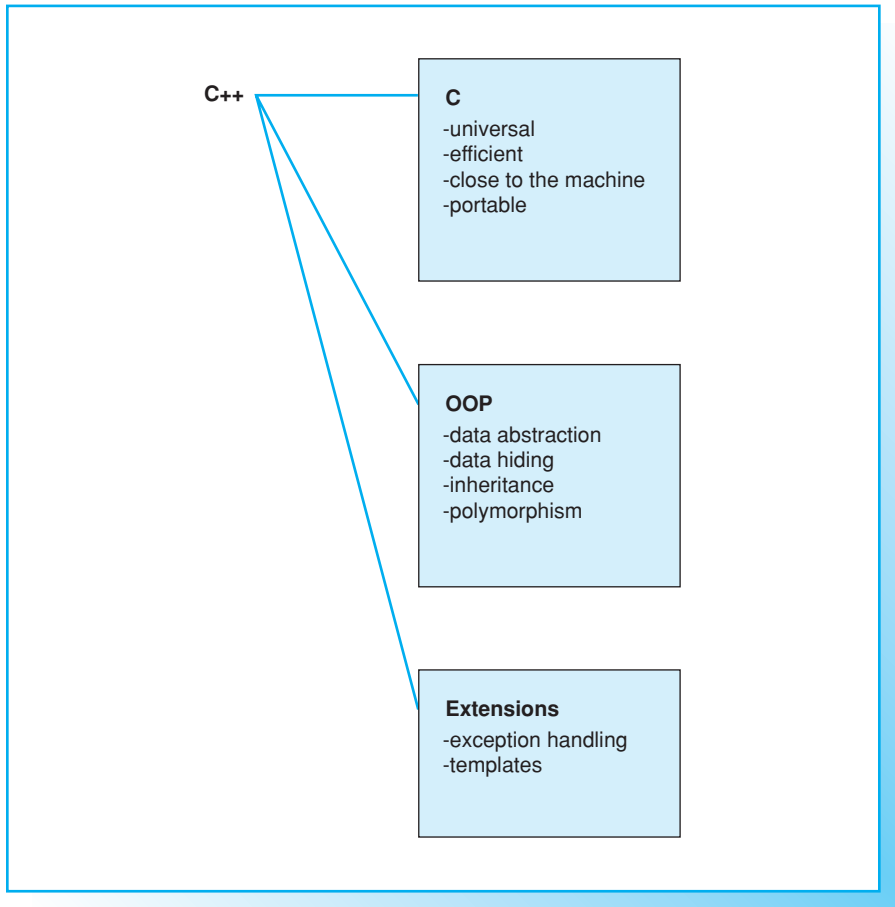
Fundamentals

This chapter describes the fundamental characteristics of the object-oriented C++ programming language. In addition, you will be introduced to the steps necessary for creating a fully functional C++ program. The examples provided will help you retrace these steps and also demonstrate the basic structure of a C++ program.

2 ■ CHAPTER 1 FUNDAMENTALS

■ DEVELOPMENT AND PROPERTIES OF C++

Characteristics



□ Historical Perspective

The C++ programming language was created by Bjarne Stroustrup and his team at Bell Laboratories (AT&T, USA) to help implement simulation projects in an object-oriented and efficient way. The earliest versions, which were originally referred to as “C with classes,” date back to 1980. As the name C++ implies, C++ was derived from the C programming language: ++ is the increment operator in C.

As early as 1989 an ANSI Committee (American National Standards Institute) was founded to standardize the C++ programming language. The aim was to have as many compiler vendors and software developers as possible agree on a unified description of the language in order to avoid the confusion caused by a variety of dialects.

In 1998 the ISO (International Organization for Standardization) approved a standard for C++ (ISO/IEC 14882).

□ Characteristics of C++

C++ is not a purely object-oriented language but a hybrid that contains the functionality of the C programming language. This means that you have all the features that are available in C:

- universally usable modular programs
- efficient, close to the machine programming
- portable programs for various platforms.

The large quantities of existing C source code can also be used in C++ programs.

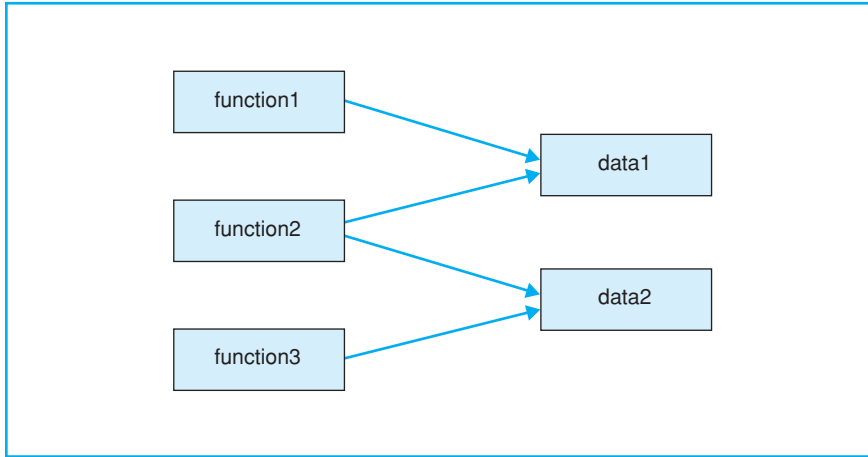
C++ supports the concepts of object-oriented programming (or OOP for short), which are:

- *data abstraction*, that is, the creation of classes to describe objects
- *data encapsulation* for controlled access to object data
- *inheritance* by creating derived classes (including multiple derived classes)
- *polymorphism* (Greek for multiform), that is, the implementation of instructions that can have varying effects during program execution.

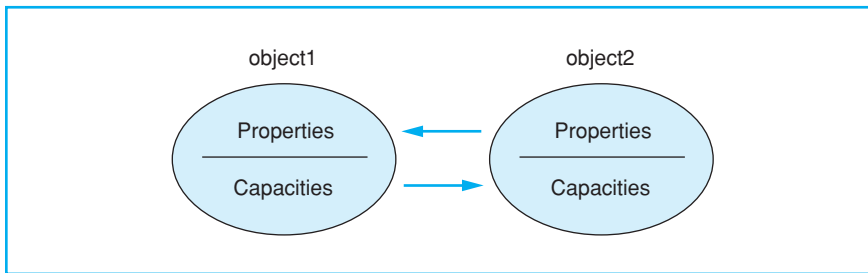
Various language elements were added to C++, such as references, templates, and exception handling. Even though these elements of the language are not strictly object-oriented programming features, they are important for efficient program implementation.

■ OBJECT-ORIENTED PROGRAMMING

Traditional concept



Object-oriented concept



□ Traditional Procedural Programming

In traditional, procedural programming, data and functions (subroutines, procedures) are kept separate from the data they process. This has a significant effect on the way a program handles data:

- the programmer must ensure that data are initialized with suitable values before use and that suitable data are passed to a function when it is called
- if the data representation is changed, e.g. if a record is extended, the corresponding functions must also be modified.

Both of these points can lead to errors and neither support low program maintenance requirements.

□ Objects

Object-oriented programming shifts the focus of attention to the *objects*, that is, to the aspects on which the problem is centered. A program designed to maintain bank accounts would work with data such as balances, credit limits, transfers, interest calculations, and so on. An object representing an account in a program will have properties and capacities that are important for account management.

OOP objects combine data (properties) and functions (capacities). A class defines a certain object type by defining both the properties and the capacities of the objects of that type. Objects communicate by sending each other “messages,” which in turn activate another object’s capacities.

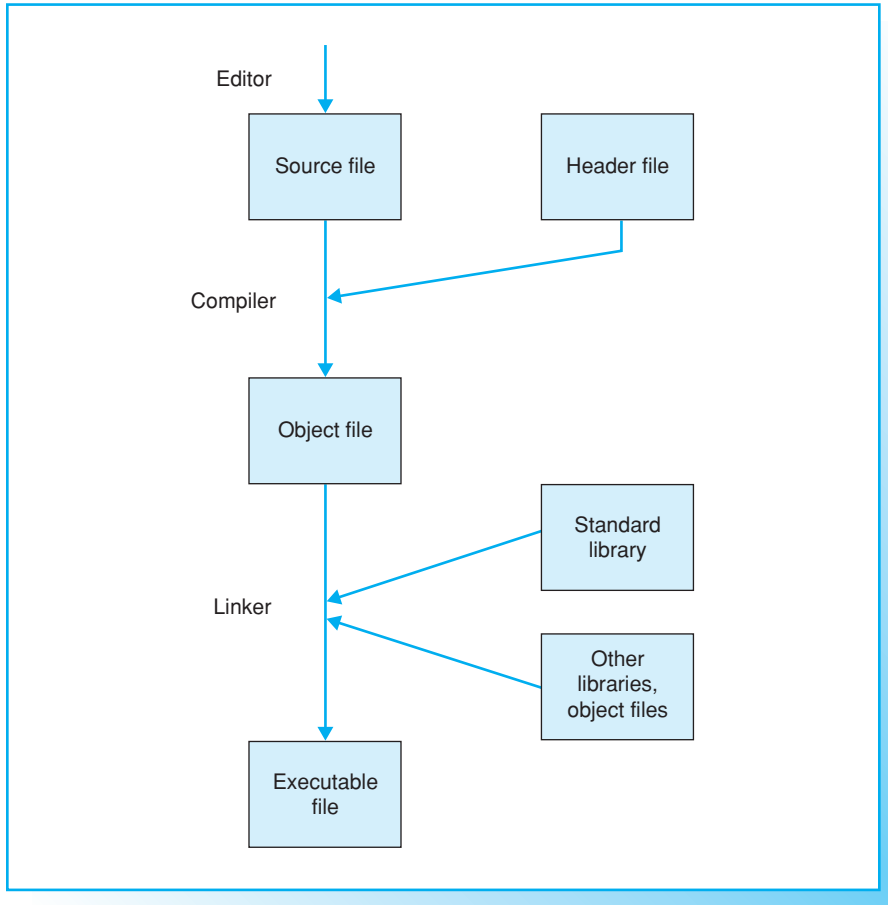
□ Advantages of OOP

Object-oriented programming offers several major advantages to software development:

- **reduced susceptibility to errors:** an object controls access to its own data. More specifically, an object can reject erroneous access attempts
- **easy re-use:** objects maintain themselves and can therefore be used as building blocks for other programs
- **low maintenance requirement:** an object type can modify its own internal data representation without requiring changes to the application.

■ DEVELOPING A C++ PROGRAM

Translating a C++ program



The following three steps are required to create and translate a C++ program:

1. First, a text editor is used to save the C++ program in a text file. In other words, the *source code* is saved to a *source file*. In larger projects the programmer will normally use *modular programming*. This means that the source code will be stored in several source files that are edited and translated separately.
2. The source file is put through a *compiler* for translation. If everything works as planned, an object file made up of *machine code* is created. The object file is also referred to as a *module*.
3. Finally, the *linker* combines the object file with other modules to form an *executable file*. These further modules contain functions from standard libraries or parts of the program that have been compiled previously.

It is important to use the correct file extension for the source file's *name*. Although the file extension depends on the compiler you use, the most commonly found file extensions are `.cpp` and `.cc`.

Prior to compilation, *header files*, which are also referred to as *include files*, can be copied to the source file. Header files are text files containing information needed by various source files, for example, type definitions or declarations of variables and functions. Header files can have the file extension `.h`, but they may not have any file extension.

The C++ *standard library* contains predefined and standardized functions that are available for any compiler.

Modern compilers normally offer an *integrated software development environment*, which combines the steps mentioned previously into a single task. A graphical user interface is available for editing, compiling, linking, and running the application. Moreover, additional tools, such as a debugger, can be launched.

 **NOTE**

If the source file contains just one *syntax error*, the compiler will report an *error*. Additional error messages may be shown if the compiler attempts to continue despite having found an error. So when you are troubleshooting a program, be sure to start with the first error shown.

In addition to error messages, the compiler will also issue *warnings*. A warning does not indicate a syntax error but merely draws your attention to a possible error in the program's logic, such as the use of a non-initialized variable.

■ A BEGINNER'S C++ PROGRAM

Sample program

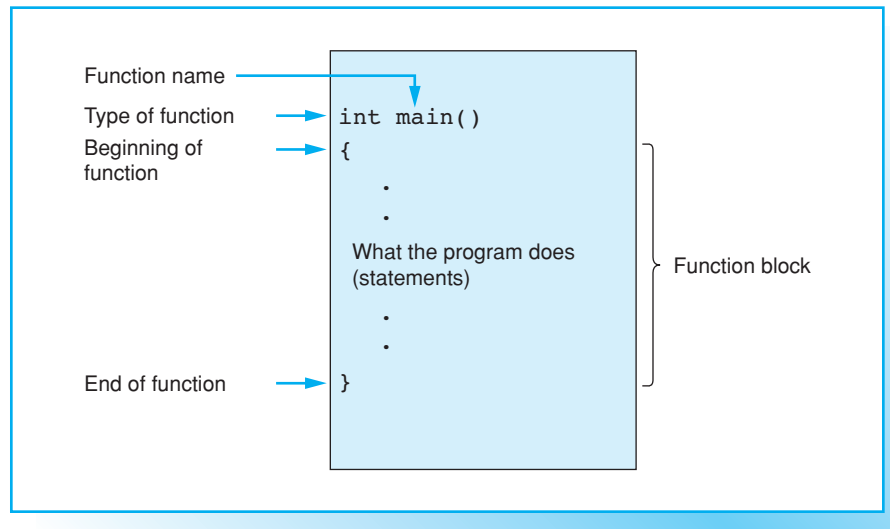
```
#include <iostream>
using namespace std;

int main()
{
    cout << "Enjoy yourself with C++!" << endl;
    return 0;
}
```

Screen output

Enjoy yourself with C++!

Structure of function main()



A C++ program is made up of objects with their accompanying *member functions* and *global functions*, which do not belong to any single particular class. Each function fulfills its own particular task and can also call other functions. You can create functions yourself or use ready-made functions from the standard library. You will always need to write the global function `main()` yourself since it has a special role to play; in fact it is the main program.

The short programming example on the opposite page demonstrates two of the most important elements of a C++ program. The program contains only the function `main()` and displays a message.

The first line begins with the number symbol, `#`, which indicates that the line is intended for the *preprocessor*. The preprocessor is just one step in the first translation phase and no object code is created at this time. You can type

```
#include <filename>
```

to have the preprocessor copy the quoted file to this position in the source code. This allows the program access to all the information contained in the header file. The header file `iostream` comprises conventions for input and output streams. The word *stream* indicates that the information involved will be treated as a flow of data.

Predefined names in C++ are to be found in the `std` (standard) namespace. The `using` directive allows direct access to the names of the `std` namespace.

Program execution begins with the first instruction in function `main()`, and this is why each C++ program must have a main function. The structure of the function is shown on the opposite page. Apart from the fact that the name cannot be changed, this function's structure is not different from that of any other C++ function.

In our example the function `main()` contains two *statements*. The first statement

```
cout << "Enjoy yourself with C++!" << endl;
```

outputs the text string `Enjoy yourself with C++!` on the screen. The name `cout` (console output) designates an object responsible for output.

The two less-than symbols, `<<`, indicate that characters are being “pushed” to the output stream. Finally `endl` (end of line) causes a line feed. The statement

```
return 0;
```

terminates the function `main()` and also the program, returning a value of 0 as an exit code to the calling program. It is standard practice to use the exit code 0 to indicate that a program has terminated correctly.

Note that statements are followed by a semicolon. By the way, the shortest statement comprises only a semicolon and does nothing.

■ STRUCTURE OF SIMPLE C++ PROGRAMS

A C++ program with several functions

```
/******  
   A program with some functions and comments  
******/  
  
#include <iostream>  
using namespace std;  
  
void line(), message();           // Prototypes  
  
int main()  
{  
    cout << "Hello! The program starts in main()."  
        << endl;  
    line();  
    message();  
    line();  
    cout << "At the end of main()." << endl;  
  
    return 0;  
}  
  
void line()                       // To draw a line.  
{  
    cout << "-----" << endl;  
}  
  
void message()                   // To display a message.  
{  
    cout << "In function message()." << endl;  
}
```

Screen output

```
Hello! The program starts in main().  
-----  
In function message().  
-----  
At the end of main().
```

The example on the opposite page shows the structure of a C++ program containing multiple functions. In C++, functions do not need to be defined in any fixed order. For example, you could define the function `message()` first, followed by the function `line()`, and finally the `main()` function.

However, it is more common to start with the `main()` function as this function controls the program flow. In other words, `main()` calls functions that have yet to be defined. This is made possible by supplying the compiler with a function *prototype* that includes all the information the compiler needs.

This example also introduces *comments*. Strings enclosed in `/* . . . */` or starting with `//` are interpreted as comments.

EXAMPLES:

```
/* I can cover
   several lines */
// I can cover just one line
```

In single-line comments the compiler ignores any characters following the `//` signs up to the end of the line. Comments that cover several lines are useful when troubleshooting, as you can use them to mask complete sections of your program. Both comment types can be used to comment out the other type.

As to the *layout* of source files, the compiler parses each source file sequentially, breaking the contents down into tokens, such as function names and operators. Tokens can be separated by any number of whitespace characters, that is, by spaces, tabs, or new line characters. The order of the source code is important but it is not important to adhere to a specific layout, such as organizing your code in rows and columns. For example

```
void message
( ) { cout <<
      "In function message()." <<
      endl; }
```

might be difficult to read, but it is a correct definition of the function `message()`.

Preprocessor directives are one exception to the layout rule since they always occupy a single line. The number sign, `#`, at the beginning of a line can be preceded only by a space or a tab character.

To improve the legibility of your C++ programs you should adopt a consistent style, using indentation and blank lines to reflect the structure of your program. In addition, make generous use of comments.

exercises

■ EXERCISES

Program listing of exercise 3

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

void pause();          // Prototype

int main()
{
    cout << endl << "Dear reader, "
         << endl << "have a ";
    pause();
    cout << "!" << endl;

    return 0;
}

void pause()
{
    cout << "BREAK";
}
```

Exercise 1

Write a C++ program that outputs the following text on screen:

```
Oh what  
a happy day!  
Oh yes,  
what a happy day!
```

Use the manipulator `endl` where appropriate.

Exercise 2

The following program contains several errors:

```
*/ Now you should not forget your glasses //  
#include <stream>  
int main  
{  
    cout << "If this text",  
    cout >> " appears on your display, ";  
    cout << " endl;"  
    cout << 'you can pat yourself on '  
        << " the back!" << endl.  
    return 0;  
}
```

Resolve the errors and run the program to test your changes.

Exercise 3

What does the C++ program on the opposite page output on screen?

solutions

■ SOLUTIONS

Exercise 1

```
// Let's go !

#include <iostream>
using namespace std;

int main()
{
    cout << " Oh what " << endl;
    cout << " a happy day! " << endl;
    cout << " Oh yes, " << endl;
    cout << " what a happy day! " << endl;

    return 0;
}
```

Exercise 2

The corrected places are underlined.

```
/* Now you should not forget your glasses */
#include <iostream>
using namespace std;
int main()
{
    cout << " If this text ";
    cout << "<< " appears on your display, ";
    cout << endl;
    cout << "<< you can pat yourself on "<<
        << " the back!" << endl;
    return 0;
}
```

Exercise 3

The screen output begins on a new line:

```
Dear reader,
have a BREAK!
```