chapter 3

Using Functions and Classes

This chapter describes how to

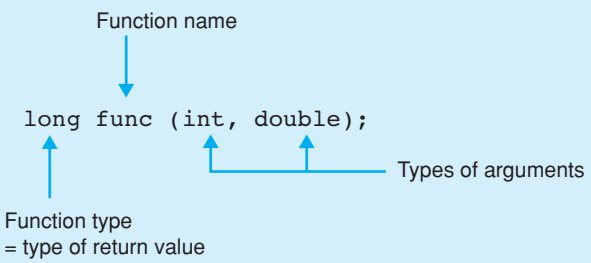
- declare and call standard functions and
- use standard classes.

This includes using standard header files. In addition, we will be working with string variables, i.e. objects belonging to the standard class `string` for the first time.

Functions and classes that you define on your own will not be introduced until later in the book.

■ DECLARING FUNCTIONS

Example of a function prototype



The prototype above yields the following information to the compiler:

- `func` is the function name
- the function is called with two arguments: the first argument is of type `int`, the second of type `double`
- the return value of the function is of type `long`.

Mathematical standard functions

```
double sin (double);           // Sine
double cos (double);           // Cosine
double tan (double);           // Tangent
double atan (double);          // Arc tangent
double cosh (double);          // Hyperbolic Cosine
double sqrt (double);          // Square Root
double pow (double, double);   // Power
double exp (double);           // Exponential Function
double log (double);           // Natural Logarithm
double log10 (double);         // Base-ten Logarithm
```

☐ Declarations

Each name (*identifier*) occurring in a program must be known to the compiler or it will cause an error message. That means any names apart from keywords must be *declared*, i.e. introduced to the compiler, before they are used.

Each time a variable or a function is defined it is also declared. But conversely, not every declaration needs to be a definition. If you need to use a function that has already been introduced in a library, you must declare the function but you do not need to redefine it.

☐ Declaring Functions

A function has a name and a type, much like a variable. The function's type is defined by its *return value*, that is, the value the function passes back to the program. In addition, the type of arguments required by a function is important. When a function is declared, the compiler must therefore be provided with information on

- the name and type of the function and
- the type of each argument.

This is also referred to as the function *prototype*.

Examples:
```
int toupper(int);

double pow(double, double);
```

This informs the compiler that the function `toupper()` is of type `int`, i.e. its return value is of type `int`, and it expects an argument of type `int`. The second function `pow()` is of type `double` and two arguments of type `double` must be passed to the function when it is called. The types of the arguments may be followed by names, however, the names are viewed as a comment only.

Examples:
```
int toupper(int c);

double pow(double base, double exponent);
```

From the compiler's point of view, these prototypes are equivalent to the prototypes in the previous example. Both junctions are standard junctions.

Standard function prototypes do not need to be declared, nor should they be, as they have already been declared in standard header files. If the header file is included in the program's source code by means of the `#include` directive, the function can be used immediately.

Example:
```
#include <cmath>
```

Following this directive, the mathematical standard functions, such as `sin()`, `cos()`, and `pow()`, are available. Additional details on header files can be found later in this chapter.

■ FUNCTION CALLS

Sample program

```
//  Calculating powers with
//  the standard function pow()

#include <iostream>     // Declaration of cout
#include <cmath>        // Prototype of pow(), thus:
                        // double pow( double, double);
using namespace std;

int main()
{
  double x = 2.5, y;

  // By means of a prototype, the compiler generates
  // the correct call or an error message!

  // Computes x raised to the power 3:
  y = pow("x", 3.0);    // Error! String is not a number
  y = pow(x + 3.0);     // Error! Just one argument
  y = pow(x, 3.0);      // ok!
  y = pow(x, 3);        // ok! The compiler converts the
                        // int value 3 to double.

  cout << "2.5 raised to 3 yields:       "
       << y << endl;

  // Calculating with pow() is possible:
  cout << "2 + (5 raised to the power 2.5) yields: "
       <<  2.0 + pow(5.0, x) << endl;

  return 0;
}
```

Screen output

```
2.5 raised to the power 3 yields:         15.625
2 + (5 raised to the power 2.5) yields:   57.9017
```

☐ Function Calls

A *function call* is an expression of the same type as the function and whose value corresponds to the return value. The return value is commonly passed to a suitable variable.

Example:
```
y = pow( x, 3.0);
```

In this example the function `pow()` is first called using the arguments `x` and `3.0`, and the result, the power x^3, is assigned to `y`.

As the function call represents a value, other operations are also possible. Thus, the function `pow()` can be used to perform calculations for `double` values.

Example:
```
cout  <<  2.0 + pow( 5.0, x);
```

This expression first adds the number `2.0` to the return value of `pow(5.0,x)`, then outputs the result using `cout`.

Any expression can be passed to a function as an *argument*, such as a constant or an arithmetical expression. However, it is important that the types of the arguments correspond to those expected by the function.

The compiler refers to the prototype to check that the function has been called correctly. If the argument type does not match exactly to the type defined in the prototype, the compiler performs type conversion, if possible.

Example:
```
y = pow( x, 3);              // also ok!
```

The value 3 of type `int` is passed to the function as a second argument. But since the function expects a `double` value, the compiler will perform type conversion from `int` to `double`.

If a function is called with the wrong number of arguments, or if type conversion proves impossible, the compiler generates an error message. This allows you to recognize and correct errors caused by calling functions at the development stage instead of causing runtime errors.

Example:
```
float x = pow(3.0 + 4.7);      // Error!
```

The compiler recognizes that the number of arguments is incorrect. In addition, the compiler will issue a warning, since a `double`, i.e. the return value of `pow()`, is assigned to a `float` type variable.

■ TYPE `void` FOR FUNCTIONS

Sample program

```
// Outputs three random numbers

#include <iostream>  // Declaration of cin and cout
#include <cstdlib>   // Prototypes of srand(), rand():
                     // void srand( unsigned int seed );
                     // int rand( void );
using namespace std;
int main()
{
   unsigned int seed;
   int z1, z2, z3;

   cout << "   --- Random Numbers  --- \n" << endl;
   cout << "To initialize the random number generator, "
        << "\n please enter an integer value: ";
   cin  >> seed;       // Input an integer

   srand( seed);       // and use it as argument for a
                       // new sequence of random numbers.

   z1 = rand();        // Compute three random numbers.
   z2 = rand();
   z3 = rand();

   cout << "\nThree random numbers: "
        << z1 << "    " << z2 << "    " << z3 << endl;

   return 0;
}
```

NOTE

The statement `cin >> seed;` reads an integer from the keyboard, because `seed` is of the `unsigned int` type.

Sample screen output

```
--- Random Numbers  ---

To initialize the random number generator,
please enter an integer value: 7777

Three random numbers: 25435   6908   14579
```

☐ Functions without Return Value

You can also write functions that perform a certain action but do not return a value to the function that called them. The type `void` is available for functions of this type, which are also referred to as procedures in other programming languages.

Example: `void srand( unsigned int seed );`

The standard function `srand()` initializes an algorithm that generates random numbers. Since the function does not return a value, it is of type `void`. An `unsigned` value is passed to the function as an argument to seed the random number generator. The value is used to create a series of random numbers.

☐ Functions without Arguments

If a function does not expect an argument, the function prototype must be declared as `void` or the braces following the function name must be left empty.

Example: `int rand( void );` `// or` `int rand();`

The standard function `rand()` is called without any arguments and returns a random number between 0 and 32767. A series of random numbers can be generated by repeating the function call.

☐ Usage of `srand()` and `rand()`

The function prototypes for `srand()` and `rand()` can be found in both the `cstdlib` and `stdlib.h` header files.

Calling the function `rand()` without previously having called `srand()` creates the same sequence of numbers as if the following statement would have been proceeded:

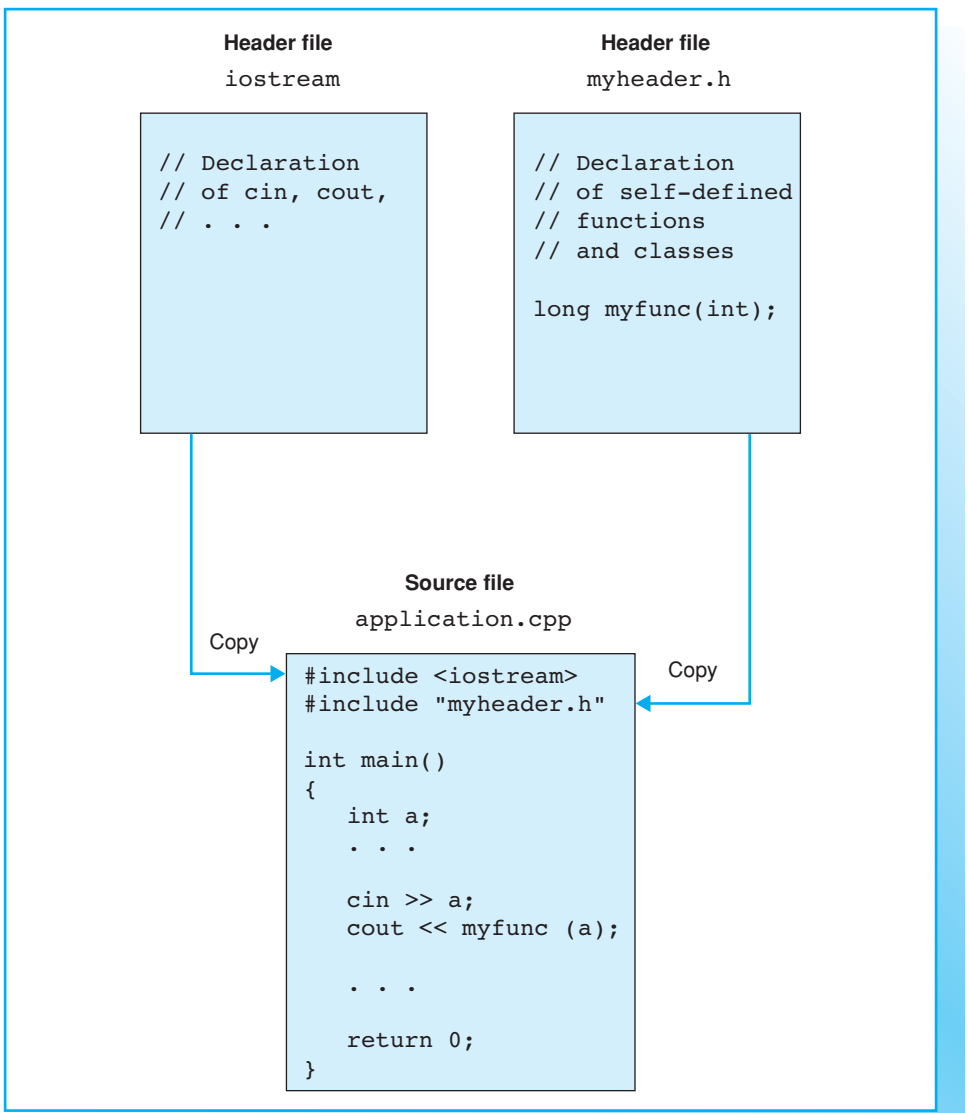
```
srand(1);
```

If you want to avoid generating the same sequence of random numbers whenever the program is executed, you must call `srand()` with a different value for the argument whenever the program is run.

It is common to use the current time to initialize a random number generator. See Chapter 6 for an example of this technique.

■ HEADER FILES

Using header files



☐ Using Header Files

Header files are text files containing declarations and macros. By using an `#include` directive these declarations and macros can be made available to any other source file, even in other header files.

Pay attention to the following points when using header files:

- header files should generally be included at the start of a program before any other declarations
- you can only name *one* header file per `#include` directive
- the file name must be enclosed in angled brackets `< ... >` or double quotes `" ... "`.

☐ Searching for Header Files

The header files that accompany your compiler will tend to be stored in a folder of their own—normally called `include`. If the name of the header file is enclosed by angled brackets `< ... >`, it is common to search for header files in the `include` folder only. The current directory is not searched to increase the speed when searching for header files.

C++ programmers commonly write their own header files and store them in the current project folder. To enable the compiler to find these header files, the `#include` directive must state the name of the header files in double quotes.

Example:
```
#include "project.h"
```

The compiler will then also search the current folder. The file suffix `.h` is normally used for user-defined header files.

☐ Standard Class Definitions

In addition to standard function prototypes, the header files also contain standard class definitions. When a header file is included, the classes defined and any objects declared in the file are available to the program.

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

Following these directives, the classes `istream` and `ostream` can be used with the `cin` and `cout` streams. `cin` is an object of the `istream` class and `cout` an object of the `ostream` class.

■ STANDARD HEADER FILES

Header files of the C++ standard library

algorithm	ios	map	stack
bitset	iosfwd	memory	stdexcept
complex	iostream	new	streambuf
dequeue	istream	numeric	string
exception	iterator	ostream	typeinfo
fstream	limits	queue	utility
functional	list	set	valarray
iomanip	locale	sstream	vector

NOTE

Some IDE's put the old-fashioned `iostream.h` and `iomanip.h` header files at your disposal. Within these header files the identifiers of `iostream` and `iomanip` are not contained in the `std` namespace but are declared globally.

Header files of the C standard library

assert.h	limits.h	stdarg.h	time.h
ctype.h	locale.h	stddef.h	wchar.h
errno.h	math.h	stdio.h	wctype.h
float.h	setjmp.h	stdlib.h	
iso646.h	signal.h	string.h	

The C++ standard library header files are shown opposite. They are *not* indicated by the file extension `.h` and contain all the declarations in their own namespace, `std`. Namespaces will be introduced in a later chapter. For now, it is sufficient to know that identifiers from other namespaces cannot be referred to directly. If you merely stipulate the directive

Example:
```
#include <iostream>
```

the compiler would not be aware of the `cin` and `cout` streams. In order to use the identifiers of the `std` namespace globally, you must add a *using* directive.

Example:
```
#include <iostream>
#include <string>
using namespace std;
```

You can then use `cin` and `cout` without any additional syntax. The header file `string` has also been included. This makes the `string` class available and allows user-friendly string manipulations in C++. The following pages contain further details on this topic.

☐ Header Files in the C Programming Language

The header files standardized for the C programming language were adopted for the C++ standard and, thus, the complete functionality of the standard C libraries is available to C++ programs.

Example:
```
#include <math.h>
```

Mathematical functions are made available by this statement.

The identifiers declared in C header files are globally visible. This can cause name conflicts in large programs. For this reason each C header file, for example `name.h`, is accompanied in C++ by a second header file, `cname`, which declares the same identifiers in the `std` namespace. Including the file `math.h` is thus equivalent to

Example:
```
#include <cmath>
using namespace std;
```

The `string.h` or `cstring` files must be included in programs that use standard functions to manipulate C strings. These header files grant access to the functionality of the C string library and are to be distinguished from the `string` header file that defines the `string` class.

Each compiler offers additional header files for platform dependent functionalities. These may be graphics libraries or database interfaces.

■ USING STANDARD CLASSES

Sample program using class `string`

```
// To use strings.

#include <iostream>     // Declaration of cin, cout
#include <string>       // Declaration of class string
using namespace std;

int main()
{
   // Defines four strings:
   string prompt("What is your name:  "),
          name,                    // An empty
          line( 40, '-'),          // string with 40 '-'
          total = "Hello ";        // is possible!

   cout << prompt;           // Request for input.
   getline( cin, name);      // Inputs a name in one line

   total = total + name;      // Concatenates and
                              // assigns strings.

   cout << line << endl       // Outputs line and name
        << total << endl;
   cout << " Your name is "   // Outputs length
        << name.length() << " characters long!" << endl;
   cout << line << endl;
   return 0;
}
```

NOTE

Both the operators `+` and `+=` for concatenation and the relational operators `<`, `<=`, `>`, `>=`, `==`, and `!=` are defined for objects of class `string`. Strings can be printed with `cout` and the operator `<<`.

The class `string` will be introduced in detail later on.

Sample screen output

```
What is your name: Rose Summer
---------------------------------------
Hello Rose Summer
Your name is 11 characters long!
---------------------------------------
```

Several classes are defined in the C++ standard library. These include stream classes for input and output, but also classes for representing strings or handling error conditions.

Each class is a type with certain properties and capacities. As previously mentioned, the properties of a class are defined by its *data members* and the class's capacities are defined by its *methods*. Methods are functions that belong to a class and cooperate with the members to perform certain operations. Methods are also referred to as *member functions*.

☐ Creating Objects

An *object* is a variable of a class type, also referred to as an *instance* of the class. When an object is created, memory is allocated to the data members and initialized with suitable values.

Example:
```
string s("I am a string");
```

In this example the object `s`, an instance of the standard class `string` (or simply a *string*), is defined and initialized with the string constant that follows. Objects of the `string` class manage the memory space required for the string themselves.

In general, there are several ways of initializing an object of a class. A string can thus be initialized with a certain number of identical characters, as the example on the opposite page illustrates.

☐ Calling Methods

All the methods defined as *public* within the corresponding class can be called for an object. In contrast to calling a global function, a method is always called for *one particular object*. The name of the object precedes the method and is separated from the method by a period.

Example:
```
s.length();                    // object.method();
```

The method `length()` supplies the length of a string, i.e. the number of characters in a string. This results in a value of 13 for the string `s` defined above.

☐ Classes and Global Functions

Globally defined functions exist for some standard classes. These functions perform certain operations for objects passed as arguments. The global function `getline()`, for example, stores a line of keyboard input in a string.

Example:
```
getline(cin, s);
```

The keyboard input is terminated by pressing the return key to create a new-line character, `'\n'`, which is not stored in the string.