



chapter

10

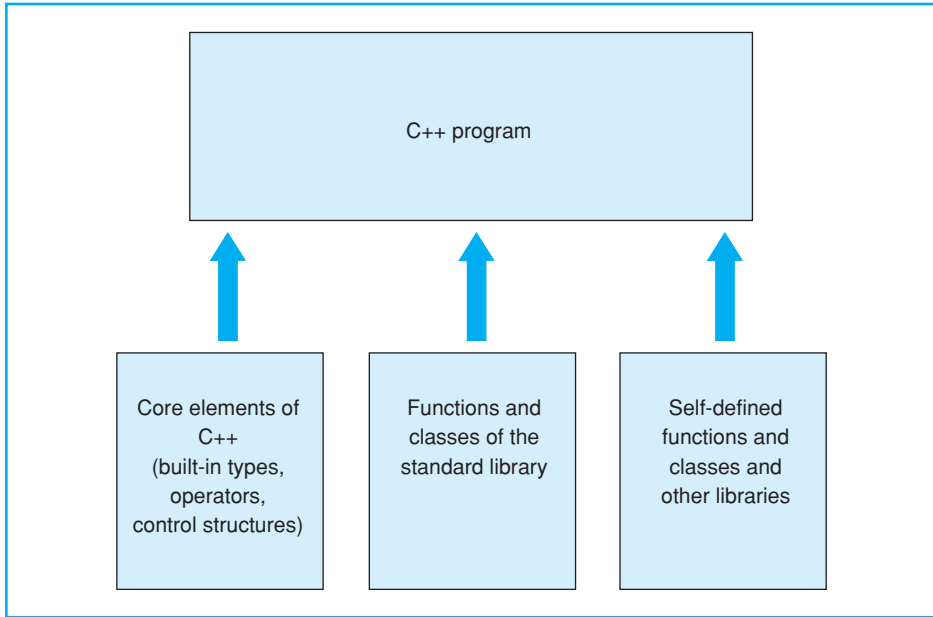
Functions

This chapter describes how to write functions of your own. Besides the basic rules, the following topics are discussed:

- passing arguments
- definition of `inline` functions
- overloading functions and default arguments
- the principle of recursion.

■ SIGNIFICANCE OF FUNCTIONS IN C++

Elements of a C++ program



C++ supports efficient software development on the lines of the top-down principle. If you are looking to provide a solution for a more complex problem, it will help to divide the problem into smaller units. After identifying objects you will need to define classes that describe these objects. You can use available classes and functions to do so. In addition, you can make use of inheritance to create specialized classes without needing to change any existing classes.

When implementing a class you must define the capacities of those objects, that is, the member functions, in your program. However, not every function is a member function.

Functions can be defined globally, such as the function `main()` for example. Functions of this type do not belong to any particular class but normally represent algorithms of a more general nature, such as the search or sort functions of the standard library.

□ Libraries

You will not need to program each “building block” yourself. Many useful global functions and classes are available from the C++ standard library. In addition, you can use other libraries for special purposes. Often a compiler package will offer commercial class libraries or graphical user interfaces. Thus, a C++ program will be made up of

- language elements of the C++ core
- global functions and classes from the C++ standard library
- functions and classes you have programmed yourself and other libraries.

Classes and functions that belong together are normally compounded to form separate source files, which can be compiled and tested independently. Using software components that you have already tested makes programming a complex solution much easier and improves the reliability of your programs. You can enhance the reusability of your source code by compiling your own libraries, but be sure to include comments for ease of readability.

Compiled source files, also known as *modules*, are compounded by the linker to an executable file by reference to the libraries you include. If you modify a source file, you may also need to recompile other files. In large scale projects it is recommended to use the *MAKE* utility for module management. An integrated developer environment will offer the functionality of this utility when you create a new project. This includes your own source files, the libraries used, and the compiler/linker settings for program compilation.

■ DEFINING FUNCTIONS

Example of a function definition

```
// func1.cpp
#include <iostream>
using namespace std;

void test( int, double );           // Prototype

int main()
{
    cout << "\nNow function test() will be called.\n";
    test( 10, -7.5);               // Call
    cout << "\nAnd back again in main()." << endl;

    return 0;
}

void test(int arg1, double arg2 )   // Definition
{
    cout << "\nIn function test()."
         << "\n 1. argument: " << arg1
         << "\n 2. argument: " << arg2 << endl;
}
```

General form of a function

```
[type] name([declaration_list]) // Function header
{
    // Beginning
    .
    .
    What will be done           // Function block
    .
    .
}                                // End
```

The following section describes how to program global functions. Chapter 13, *Defining Classes*, describes the steps for defining member functions.

□ Definition

Functions can be defined in any order, however, the first function is normally `main`. This makes the program easier to understand, since you start reading at the point where the program starts to execute.

The function `test()` is shown opposite as an example and followed by the general form of a function. The example can be read as follows:

<code>type</code>	is the function type, that is, the type of the return value.
<code>name</code>	is the function name, which is formed like a variable name and should indicate the purpose of the function.
<code>declaration_list</code>	contains the names of the parameters and declares their types. The list can be empty, as for the function <code>main()</code> , for example. A list of declarations that contains only the word <code>void</code> is equivalent to an empty list.

The *parameters* declared in a list are no more than local variables. They are created when the function is called and initialized by the values of the arguments.

Example: When `test(10, -7.5);` is called, the parameter `arg1` is initialized with a value of 10 and `arg2` with -7.5.

The left curved bracket indicates the start of a *function block*, which contains the statements defining what the function does.

□ Prototype and Definition

In a function definition the function header is similar in form to the prototype of a function. The only difference when a function is defined is that the name and declaration list are *not* followed by a semicolon but by a function code block.

The prototype is the declaration of the function and thus describes only the formal interface of that function. This means you can omit parameter names from the prototype, whereas compiling a function definition will produce machine code.

■ RETURN VALUE OF FUNCTIONS

Defining and calling the function `area()`

```
// area.cpp
// Example for a simple function returning a value.
//-----
#include <iostream>
#include <iomanip>
using namespace std;

double area(double, double);          // Prototype

int main()
{
    double x = 3.5, y = 7.2, res;

    res = area( x, y+1);              // Call

    // To output to two decimal places:
    cout << fixed << setprecision(2);
    cout << "\n The area of a rectangle "
         << "\n with width  " << setw(5) << x
         << "\n and length  " << setw(5) << y+1
         << "\n is          " << setw(5) << res
         << endl;
    return 0;
}

// Defining the function area():
// Computes the area of a rectangle.
double area( double width, double len)
{
    return (width * len);    // Returns the result.
}
```

Screen output:

```
The area of a rectangle
with width  3.50
and length  8.20
is          28.70
```

The program opposite shows how the function `area()` is defined and called. As previously mentioned, you must declare a function before calling it. The *prototype* provides the compiler with all the information it needs to perform the following actions when a function is called:

- check the number and type of the arguments
- correctly process the return value of the function.

A function declaration can be omitted only if the function is defined within the same source file immediately before it is called. Even though simple examples often define and call a function within a single source file, this tends to be an exception. Normally the compiler will not see a function definition as it is stored in a different source file.

When a function is called, an argument of the same type as the parameter must be passed to the function for each parameter. The arguments can be any kind of expressions, as the example opposite with the argument `y+1` shows. The value of the expression is always copied to the corresponding parameter.

□ Return Statement

When the program flow reaches a *return statement* or the end of a function code block, it branches back to the function that called it. If the function is any type other than `void`, the *return statement* will also cause the function to return a value to the function that called it.

Syntax: `return [expression]`

If *expression* is supplied, the value of the expression will be the return value. If the type of this value does not correspond to the function type, the function type is converted, where possible. However, functions should always be written with the *return value* matching the function type.

The function `area()` makes use of the fact that the *return statement* can contain any expression. The *return expression* is normally placed in parentheses if it contains operators.

If the expression in the *return statement*, or the *return statement* itself, is missing, the return value of the function is undefined and the function type must be `void`. Functions of the `void` type, such as the standard function `srand()`, will perform an action but not return any value.

■ PASSING ARGUMENTS

Calling function and called function

```

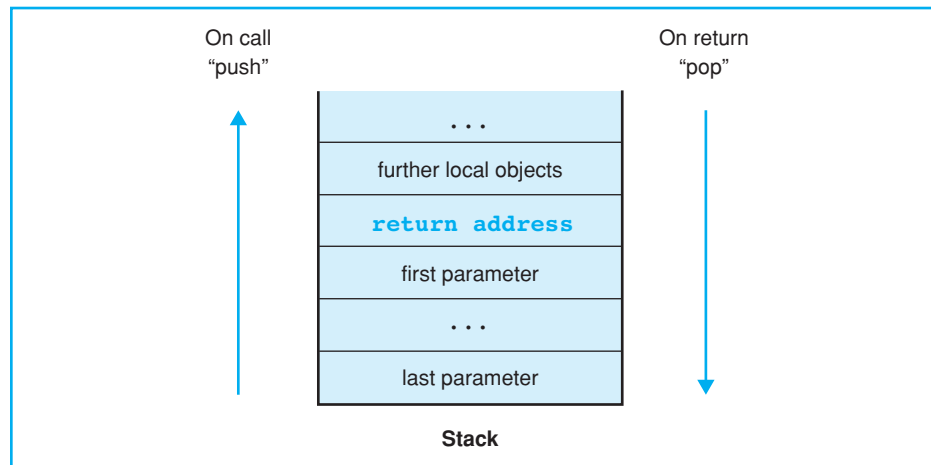
long func2(int, double);           // Prototype
// . . .
void func1()
{
    int x = 1.1;
    double y;
    . . .
    long a = func2(x, y);           // Call of func2().
    . . .
}

long func2(int a, double b) // Definition
{
    double x = 2.2;
    long result;
    .           // Here the result
    .           // is computed.
    .
    return result;
}

```

The diagram illustrates the call of `func2()` from `func1()`. The call `func2(x, y)` in `func1()` is annotated with `// Call of func2().` and `// Pass by value`. The definition of `func2` is annotated with `// Definition`. Arrows show the arguments `x` and `y` being passed to the parameters `a` and `b` in the function definition. Another arrow shows the `return result;` statement in the definition returning a value to the assignment `long a =` in the call.

Stack content after calling a function



□ Passing by Value

Passing values to a function when the function is called is referred to as *passing by value*. Of course the called function cannot change the values of the arguments in the calling function, as it uses copies of the arguments.

However, function arguments can also be *passed by reference*. In this case, the function is passed a reference to an object as an argument and can therefore access the object directly and modify it.

An example of passing by reference was provided in the example containing the function `time()`. When `time(&sek)` is called, the address of the variable `sek` is passed as an argument, allowing the function to store the result in the variable. We will see how to create functions of this type later.

Passing by value does, however, offer some important advantages:

- function arguments can be any kind of expression, even constants, for example
- the called function cannot cause accidental modifications of the arguments in the calling function
- the parameters are available as suitable variables within the functions. Additional indirect memory access is unnecessary.

However, the fact that copying larger objects is difficult can be a major disadvantage, and for this reason vectors are passed by reference to their starting address.

□ Local Objects

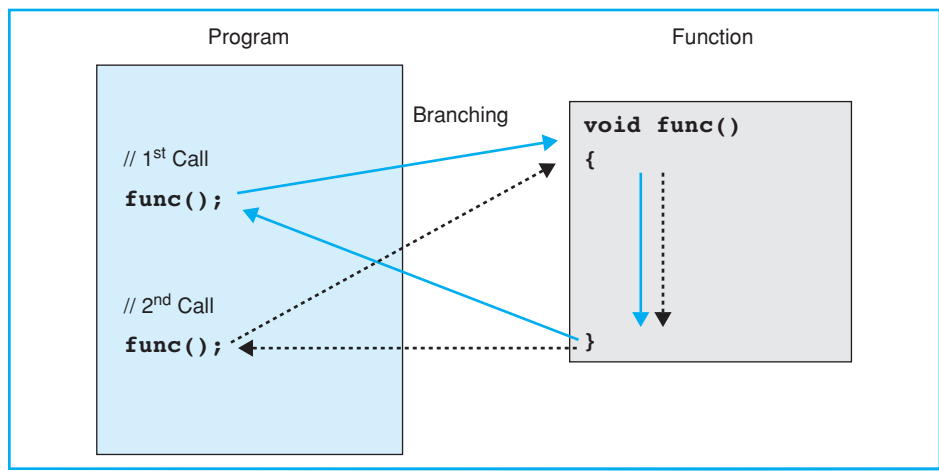
The scope of function parameters and the objects defined within a function applies only to the function block. That is, they are valid within the function only and not related to any objects or parameters of the same name in any other functions.

For example, the program structure opposite contains a variable `a` in the function `func1()` and in the function `func2()`. The variables do not collide because they reference different memory addresses. This also applies to the variables `x` in `func1()` and `func2()`.

A function's local objects are placed on the *stack*—the parameters of the function are placed first and in reverse order. The stack is an area of memory that is managed according to the LIFO (*last in first out*) principle. A stack of plates is a good analogy. The last plate you put on the stack has to be taken off first. The LIFO principle ensures that the last local object to be created is destroyed first.

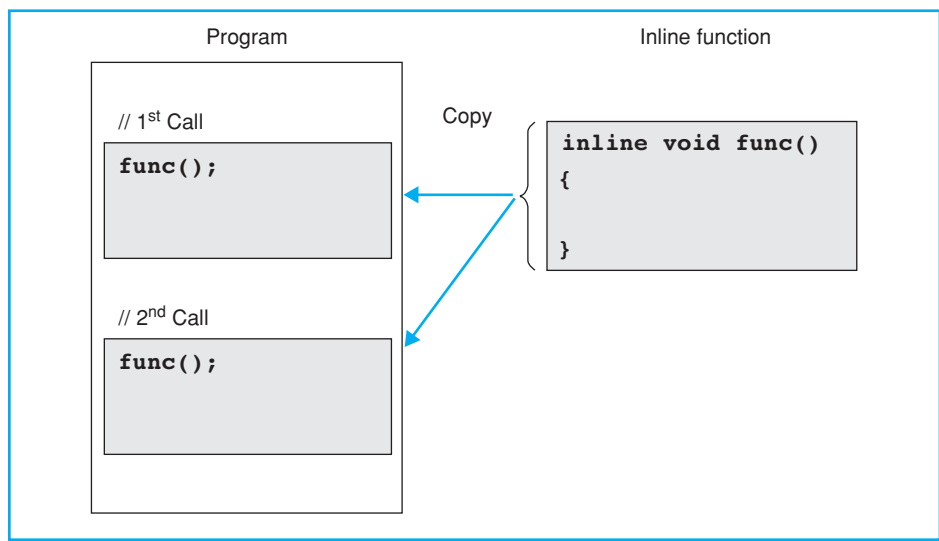
■ INLINE FUNCTIONS

Call to a function not defined as inline



HINT The executable file only contains one instance of the function's machine code.

Call to an inline function



HINT The machine code of the function is stored in the executable file wherever the function is called.

□ Jumping to Sub-Routines

When a function is called, the program jumps to a *sub-routine*, which is executed as follows:

- the function parameters are placed on the stack and initialized with appropriate arguments
- the so-called *return address*, that is, the place where the function was called, is stored on the stack and the program flow branches to the function
- after executing the function the program uses the return address it stored previously to return to the calling function. The part of the stack occupied by the function is then released.

All this jumping back and forth can affect the run time of your program, especially if the function contains only a few instructions and is called quite often. The time taken to branch to a small function can be greater than the time needed to execute the function itself. However, you can define `inline` functions to avoid this problem.

□ Inline Definition

The compiler inserts the code of an inline function at the address where the function is called and thus avoids jumping to a sub-routine. The *definition* of an inline function is introduced by the `inline` keyword in the function header.

Example:

```
inline int max( int x, int y)
{ return (x >= y ? x : y ); }
```

The program code will expand each time an `inline` function is called. This is why `inline` functions should contain no more than one or two instructions. If an inline function contains too many instructions, the compiler may ignore the `inline` keyword and issue a warning.

An `inline` function must be defined in the source file in which it is called. You cannot simply supply a prototype of the function. The code containing the instructions must also be available to the compiler. It therefore makes sense to define `inline` functions in header files, in contrast to “normal” functions. This means the function will be available in several source files.

□ Inline Functions and Macros

Inline functions are an alternative to macros with parameters. When a macro is called, the preprocessor simply replaces a block of text. In contrast, an `inline` function behaves like a normal function, although the program flow is not interrupted by the function branching. The compiler performs a type check, for example.

■ DEFAULT ARGUMENTS

Defining the function `capital()`

```
// Computes the final capital with interest and
// compound interest.
// Formula: capital = k0 * (1.0 + p/100)n
// where k0 = start capital, p = rate, n = run time
// -----
#include <math.h>
double capital( double k0, double p, double n)
{
    return (k0 * pow(1.0+p/100, n));
}
```

Possible calls

```
// Function capital() with two default arguments
// Prototype:
double capital( double k0, double p=3.5, double n=1.0);

double endcap;

endcap = capital( 100.0, 3.5, 2.5); // ok
endcap = capital( 2222.20, 4.8); // ok
endcap = capital( 3030.00); // ok

endcap = capital( ); // not ok
// The first argument has no default value.

endcap = capital( 100.0, , 3.0); // not ok
// No gap!

endcap = capital( , 5.0); // not ok
// No gap either.
```



NOTE

A function defined with default arguments is always called with the full number of arguments. For reasons of efficiency it may be useful to define several versions of the same function.

So-called *default arguments* can be defined for functions. This allows you to omit some arguments when calling the function. The compiler simply uses the default values for any missing arguments.

□ Defining Default Arguments

The default values of a function's arguments must be known when the function is called. In other words, you need to supply them when you declare the function.

Example: `void moveTo(int x = 0, int y = 0);`

Parameter names can be omitted, as usual.

Example: `void moveTo(int = 0, int = 0);`

The function `moveTo()` can then be called with or without one or two arguments.

Example: `moveTo (); moveTo (24); moveTo (24, 50);`

The first two calls are equivalent to `moveTo(0,0)`; or `moveTo(24,0)`;

It is also possible to define default arguments for only some of the parameters. The following general rules apply:

- the default arguments are defined in the function prototype. They can also be supplied when the function is defined, if the definition occurs in the same source file and before the function is called
- if you define a default argument for a parameter, all following parameters must have default arguments
- default arguments must not be redefined within the prototype scope (the next chapter gives more details on this topic).

□ Possible Calls

When calling a function with default arguments you should pay attention to the following points:

- you must first supply any arguments that do not have default values
- you can supply arguments to replace the defaults
- if you omit an argument, you must also omit any following arguments.

You can use default arguments to call a function with a different number of arguments without having to write a new version of the function.

■ OVERLOADING FUNCTIONS

Sample program

```

// random.cpp
// To generate and output random numbers.
//-----
#include <iostream>
#include <iomanip>
#include <cstdlib>    // For rand(), srand()
#include <ctime>      // For time()
using namespace std;

bool setrand = false;
inline void init_random() // Initializes the random
{                          // number generator with the
                          // present time.
    if( !setrand )
    {   srand((unsigned int)time(NULL));
        setrand = true;
    }
}
inline double myRandom() // Returns random number x
{                          // with 0.0 <= x <= 1.0
    init_random();
    return (double)rand() / (double)RAND_MAX;
}
inline int myRandom(int start, int end) // Returns the
{                                          // random number n with
    init_random();                          // start <= n <= end
    return (rand() % (end+1 - start) + start);
}

// Testing myRandom() and myRandom(int,int):
int main()
{
    int i;
    cout << "5 random numbers between 0.0 and 1.0 :\"
        << endl;
    for( i = 0; i < 5; ++i)
        cout << setw(10) << myRandom();
    cout << endl;
    cout << "\nAnd now 5 integer random numbers \"
        << "between -100 and +100 :\" << endl;
    for( i = 0; i < 5; ++i)
        cout << setw(10) << myRandom(-100, +100);
    cout << endl;
    return 0;
}

```

Functions in traditional programming languages, such as C, which perform the same task but have different arguments, must have different names. To define a function that calculated the maximum value of two integers and two floating-point numbers, you would need to program two functions with different names.

Example:

```
int    int_max( int x, int y);
double dbl_max( double x, double y);
```

Of course this is detrimental to efficient naming and the readability of your program—but luckily, this restriction does not apply to C++.

□ Overloading

C++ allows you to overload functions, that is, different functions can have the same name.

Example:

```
int    max( int x, int y);
double max( double x, double y);
```

In our example two different function share the same name, `max()`. The function `max()` was overloaded for `int` and `double` types. The compiler uses a function's signature to differentiate between overloaded functions.

□ Function Signatures

A function signature comprises the number and type of parameters. When a function is called, the compiler compares the arguments to the signature of the overloaded functions and simply calls the appropriate function.

Example:

```
double maxvalue, value = 7.9;
maxvalue = max( 1.0, value);
```

In this case the `double` version of the function `max()` is called.

When overloaded functions are called, implicit type conversion takes place. However, this can lead to ambiguities, which in turn cause a compiler error to be issued.

Example:

```
maxvalue = max( 1, value); // Error!
```

The signature does not contain the function type, since you cannot deduce the type by calling a function. It is therefore impossible to differentiate between overloaded functions by type.

Example:

```
int    search(string key);
string search(string name);
```

Both functions have the same signature and cannot be overloaded.

■ RECURSIVE FUNCTIONS

Using a recursive function

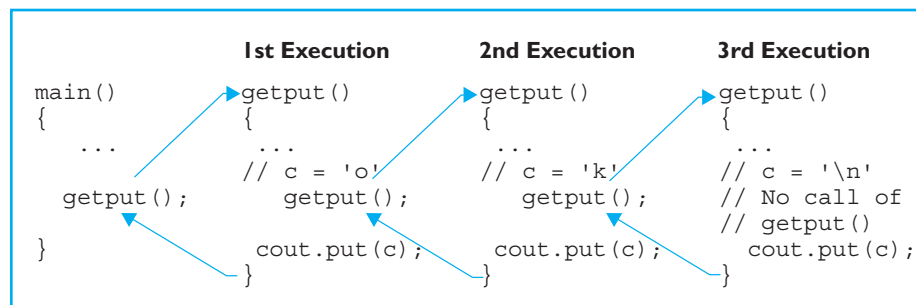
```
// recursive.cpp
// Demonstrates the principle of recursion by a
// function, which reads a line from the keyboard
// and outputs it in reverse order.
// -----
#include <iostream>
using namespace std;

void getput(void);

int main()
{
    cout << "Please enter a line of text:\n";
    getput();
    cout << "\nBye bye!" << endl;
    return 0;
}

void getput()
{
    char c;
    if( cin.get(c)  &&  c != '\n')
        getput();
    cout.put(c);
}
```

Program flow after typing ok<return>



□ Recursion

A function that calls itself is said to be *recursive*. This process can also be performed indirectly if the function first calls another function or multiple functions before it is called once more. But a break criterion is always necessary to avoid having the function call itself infinitely.

The concept of local objects makes it possible to define recursive functions in C++. Recursion requires local objects to be created each time the function is called, and these objects must not have access to any other local objects from other function calls. What effectively happens is that the local objects are placed on the stack, and thus the object created last is destroyed first.

□ A Sample Program

Let's look at the principle of recursion by referring to the sample program opposite. The program contains the recursive function `getput()` that reads a line of text from the keyboard and outputs it in reverse order.

The function `getput()` is first called by `main()` and reads a character from the keyboard, storing it in the local variable `c`. If the character is not `'\n'`, the function `getput()` calls itself again and thus reads a further character from the keyboard before storing it in the local variable `c`.

The chain of recursive function calls is terminated by the user pressing the Return key. The last character to be read, `'\n'` (line feed), is output and the program flow branches to the previous `getput()` instance. This outputs the second to last character, and so on. When the first character to have been read has finally been output, the program flow is handed back to `main()`.

□ Practical Usage

The logic of various solutions to common problems results in a recursive structure, for example, browsing directory trees, using binary trees for data management, or some sorting algorithms, such as the quick sort algorithm. Recursive functions allow you to formulate this kind of logic in an efficient and elegant manner. However, always make sure that sufficient memory is available for the stack.

exercises

■ EXERCISES

Hint for exercise 1**Working with several source files:**

Within an integrated development environment a project, containing all source files of the program, first has to be created. This ensures that all the source files will be compiled and linked automatically.

However, when calling the compiler/linker from the command line, it is sufficient to declare the source files, for example:

```
cc sum_t.cpp sum.cpp
```

Screen output for exercise 3

n	Factorial of n
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
.	...
.
.
19	121645100408832000
20	2432902008176640000

Exercise 1

- a. Write the function `sum()` with four parameters that calculates the arguments provided and returns their sum.

Parameters: Four variables of type `long`.

Returns: The sum of type `long`.

Use the default argument 0 to declare the last two parameter of the function `sum()`. Test the function `sum()` by calling it by all three possible methods. Use random integers as arguments.

- b. Now restructure your program to store the functions `main()` and `sum()` in individual source files, for example, `sum_t.cpp` and `sum.cpp`.

Exercise 2

- a. Write an inline function, `Max(double x, double y)`, which returns the maximum value of `x` and `y`. (Use `Max` instead of `max` to avoid a collision with other definitions of `max`.) Test the function by reading values from the keyboard.

Can the function `Max()` also be called using arguments of the types `char`, `int`, or `long`?

- b. Now overload `Max()` by adding a further inline function `Max(char x, char y)` for arguments of type `char`.

Can the function `Max()` still be called with two arguments of type `int`?

Exercise 3

The *factorial* $n!$ of a positive integer n is defined as

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

Where $0! = 1$

Write a function to calculate the factorial of a number.

Argument: A number n of type `unsigned int`.

Returns: The factorial $n!$ of type `long double`.

Formulate two versions of the function, where the factorial is

- calculated using a loop
- calculated recursively

Test both functions by outputting the factorials of the numbers 0 to 20 as shown opposite on screen.

Exercise 4

Write a function `pow(double base, int exp)` to calculate integral powers of floating-point numbers.

Arguments: The base of type `double` and the exponent of type `int`.

Returns: The power `baseexp` of type `double`.

For example, calling `pow(2.5, 3)` returns the value

$$2.5^3 = 2.5 * 2.5 * 2.5 = 15.625$$

This definition of the function `pow()` means overloading the standard function `pow()`, which is called with two `double` values.

Test your function by reading one value each for the base and the exponent from the keyboard. Compare the result of your function with the result of the standard function.



NOTE

1. The power x^0 is defined as 1.0 for a given number x .
2. The power x^n is defined as $(1/x)^{-n}$ for a negative exponent n .
3. The power 0^n where $n > 0$ will always yield 0.0.

The power 0^n is not defined for $n < 0$. In this case, your function should return the value `HUGE_VAL`. This constant is defined in `math.h` and represents a large `double` value. Mathematical functions return `HUGE_VAL` when the result is too large for a `double`.

solutions

■ SOLUTIONS

Exercise I

```
// -----
// sum_t.cpp
// Calls function sum() with default arguments.
// -----

#include <iostream>
#include <iomanip>
#include <ctime>
#include <cstdlib>
using namespace std;

long sum( long a1, long a2, long a3=0, long a4=0);

int main()                // Several calls to function sum()
{
    cout << " **** Computing sums ****\n"
          << endl;

    srand((unsigned int)time(NULL)); // Initializes the
                                     // random number generator.
    long res, a = rand()/10, b = rand()/10,
          c = rand()/10, d = rand()/10;

    res = sum(a,b);
    cout << a << " + " << b << " = " << res << endl;

    res = sum(a,b,c);
    cout << a << " + " << b << " + " << c
          << " = " << res << endl;

    res = sum(a,b,c,d);
    cout << a << " + " << b << " + " << c << " + " << d
          << " = " << res << endl;

    return 0;
}

// -----
// sum.cpp
// Defines the function sum()
// -----

long sum( long a1, long a2, long a3, long a4)
{
    return (a1 + a2 + a3 + a4);
}
```

Exercise 2

```
// -----  
// max.cpp  
// Defines and calls the overloaded functions Max().  
// -----  
  
// As long as just one function Max() is defined, it can  
// be called with any arguments that can be converted to  
// double, i.e. with values of type char, int or long.  
// After overloading no clear conversion will be possible.  
  
#include <iostream>  
#include <string>  
using namespace std;  
  
inline double Max(double x, double y)  
{  
    return (x < y ? y : x);  
}  
  
inline char Max(char x, char y)  
{  
    return (x < y ? y : x);  
}  
  
string header(  
    "To use the overloaded function Max().\n"),  
    line(50, '-');  
  
int main()    // Several different calls to function Max()  
{  
    double x1 = 0.0, x2 = 0.0;  
  
    line += '\n';  
    cout << line << header << line << endl;  
  
    cout << "Enter two floating-point numbers:"  
        << endl;  
    if( cin >> x1  &&  cin >> x2)  
    {  
        cout << "The greater number is " << Max(x1,x2)  
            << endl;  
    }  
    else  
        cout << "Invalid input!" << endl;  
  
    cin.sync(); cin.clear();    // Invalid input  
                                // was entered.
```

```

cout << line
    << "And once more with characters!"
    << endl;

cout << "Enter two characters:"
    << endl;

char c1, c2;
if( cin >> c1 && cin >> c2)
{
    cout << "The greater character is " << Max(c1,c2)
        << endl;
}
else
    cout << "Invalid input!" << endl;

cout << "Testing with int arguments." << endl;
int a = 30, b = 50;
cout << Max(a,b) << endl;    // Error! Which
                            // function Max()?

return 0;
}

```

Exercise 3

```

// -----
// factorial.cpp
// Computes the factorial of an integer iteratively,
// i.e. using a loop, and recursively.
// -----
#include <iostream>
#include <iomanip>
using namespace std;

#define N_MAX 20

long double fact1(unsigned int n); // Iterative solution
long double fact2(unsigned int n); // Recursive solution

int main()
{
    unsigned int n;

    // Outputs floating-point values without
    // decimal places:
    cout << fixed << setprecision(0);
}

```

```

// --- Iterative computation of factorial ---

cout << setw(10) << "n" << setw(30) << "Factorial of n"
    << "          (Iterative solution)\n"
    << "          -----"
    << endl;

for( n = 0; n <= N_MAX; ++n)
    cout << setw(10) << n << setw(30) << fact1(n)
        << endl;

cout << "\nGo on with <return>";
cin.get();

// --- Recursive computation of factorial ----

cout << setw(10) << "n" << setw(30) << "Factorial of n"
    << "          (Recursive solution)\n"
    << "          -----"
    << endl;

for( n = 0; n <= N_MAX; ++n)
    cout << setw(10) << n << setw(30) << fact2(n)
        << endl;

cout << endl;

return 0;
}

long double fact1(unsigned int n)          // Iterative
{                                          // solution.
    long double result = 1.0;
    for( unsigned int i = 2; i <= n; ++i)
        result *= i;

    return result;
}

long double fact2(unsigned int n)          // Recursive
{                                          // solution.
    if( n <= 1)
        return 1.0;
    else
        return fact2(n-1) * n;
}

```

Exercise 4

```
// -----  
// power.cpp  
// Defines and calls the function pow() to  
// compute integer powers of a floating-point number.  
// Overloads the standard function pow().  
// -----  
#include <iostream>  
#include <cmath>  
using namespace std;  
  
double pow(double base, int exp);  
  
int main() // Tests the self-defined function pow()  
{  
    double base = 0.0;  
    int exponent = 0;  
  
    cout << " **** Computing Integer Powers ****\n"  
         << endl;  
  
    cout << "Enter test values.\n"  
         << "Base (floating-point): "; cin >> base;  
    cout << "Exponent (integer): "; cin >> exponent;  
  
    cout << "Result of " << base << " to the power of "  
         << exponent << " = " << pow( base, exponent)  
         << endl;  
  
    cout << "Computing with the standard function: "  
         << pow( base, (double)exponent) << endl;  
    return 0;  
}  
double pow(double base, int exp)  
{  
    if( exp == 0) return 1.0;  
    if( base == 0.0)  
        if( exp > 0) return 0.0;  
        else return HUGE_VAL;  
    if( exp < 0)  
    {  
        base = 1.0 / base;  
        exp = -exp;  
    }  
    double power = 1.0;  
    for( int n = 1; n <= exp; ++n)  
        power *= base;  
    return power;  
}
```

