

# LECTURE 7: FUNCTIONS

## Contents

6.1 Introduction .....	1
6.2 What Is a Function .....	2
6.2.1 Calculate average example .....	2
6.3 Calling a Function .....	3
6.4 Several return Statements .....	5
6.4.1 Example: Least of Three Numbers .....	5
6.5 Array As Parameter .....	7
6.6 Function and Subfunction .....	9
6.6.1 Example.....	9
6.6 Function without Return Value.....	12
6.7 Replacing Characters in a String .....	13
6.8 Override Functions.....	15
6.9 Declaration - Definition.....	16
6.10 Header Files.....	17
6.11 Recursive Functions .....	19
6.12 Summary .....	20
6.13 Revision Questions.....	21

## 6.1 Introduction

When working with bigger programs, or programs performing many different tasks, it is important to divide the code into small limited pieces, which makes it easier to grasp and maintain. Functions are used for this purpose. A function is a part of the program well marked off that performs a particular task. A function can be reused several times in a program or in many different programs.

A function, when completed and tested, is like a black box that always works as expected. You don't need to bother any more about what is inside. You give it a number of input values, and it does its job.

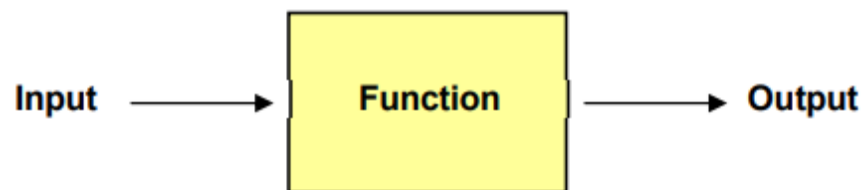
Think for instance of a car driver who wants to increase the speed. He pushes the accelerator and the car accelerates. He doesn't have to bother about fuel injection, gear ratio, engine compression and the like. He knows that the same thing always happens when pushing the accelerator. It is the same with functions. The programmer initiates the function from his code and the function always performs the same task without having to bother about the details.

To summarize, the advantages with functions are:

- Small well marked off program sections
- Well-structured programs
- Easier to debug and maintain
- Reusage of code

## 6.2 What Is a Function

A function is written to perform a specific task. It might need input values and it returns the result from the task being performed:



The programmer to make the function perform its task must supply required input and receive the supplied output

Examples of function tasks are:

- Calculate average – here you will have to supply the detailed values for the average calculation, and receive the average value delivered by the function.
- Sort an array – here the function must know which array to sort. It returns the sorted array
- Search for an item in an array – here the function also must know which array to search. It returns the item found.
- Calculate order price – here the input may consist of quantity, unit price and a discount factor. The output is the calculated order price

### 6.2.1 Calculate average example

We will write a function which calculates the average of two numbers. The two numbers form the input, and the calculated average is the output.

The task is to add the two numbers and divide the sum by 2. Suppose that the two variables x1 and x2 contain the input values. The statement to calculate the average is then:

$$av = (x1 + x2) / 2;$$

The variable av now contains the calculated average.

The entire function is coded in this way:

```
double dAverage(double x1, double x2)  
{
```

```
double av;  
  
av = (x1 + x2)/2;  
  
return av;  
  
}
```

The function has a name, dAverage. The required input values are enumerated within parenthesis after the function name. They are called **formal parameters** and are named x1 and x2. Furthermore, you specify the data type in question. Both x1 and x2 are of the double type. The data type is given in front of each parameter, and the parameters are separated by comma.

At the first line we also specify the data type for the result value in front of the function name. In our case the result is of the double type.

Thus, the first line specifies the name of the function, the required input and the output. The first line is called **function header**.

The task to be performed by the function is described by the code inside the curly brackets. This section is called the **function body** and consists of three statements.

- First we declare a variable av, which is required inside the function.
- At the second line we perform the average calculation. The average is stored in the variable av.
- At the third line we return that value to the caller. The statement

```
return av;
```

means that the value in the variable av is delivered as output. A return statement also has the effect that the function is terminated. The task has been completed

## 6.3 Calling a Function

The function dAverage is complete, but it does not perform anything yet. Furthermore, the parameters x1 and x2 don't have any values, so the statement with the average calculation is meaningless so far. We must make the function start.

Calling a function means three things:

- Write the function name
- Supply input to the function
- Receive the return value

The following statement calls the function dAverage:

```
dAvg = dAverage(dNo1, dNo2);
```

This statement implies that the function dAverage is initiated and that the values contained by the variables dNo1 and dNo2 are supplied as input to the function. A prerequisite for this is of course that the variables dNo1 and dNo2 have been assigned values prior to the function call. The variables dNo1 and dNo2 are called actual parameters, since they contain actual values.

The execution is now passed over to the function. The function header:

```
double dAverage(double x1, double x2)
```

tells us that there are two formal parameters x1 and x2. The actual parameter values are copied to the formal parameters in the specified sequence. x1 will get the value of dNo1 and x2 the value of dNo2. Now that the formal parameters have got their values the statements of the function body are meaningful and can be executed.

In the function body the variable av is declared, the average is calculated and stored in av, which is returned. Then the function has completed.

The execution is now passed over to the calling statement:

```
dAvg = dAverage(dNo1, dNo2);
```

The entire right part is now replaced by the returned value, which is assigned to the variable dAvg. At completion of the statement the variable dAvg has got the value returned by the function, namely the average of dNo1 and dNo1.

Here is the entire program

```
#include <iostream.h>  
  
double dAverage(double x1, double x2)  
{  
  
double av;  
  
av = (x1 + x2)/2;  
  
return av;  
}  
  
void main()  
{  
  
double dNo1, dNo2, dAvg;  
  
cout << "Enter two numbers: ";  
  
cin >> dNo1 >> dNo2;  
  
dAvg = dAverage(dNo1, dNo2);  
  
cout << "The average is " << dAvg;  
  
}
```

The execution of a program always starts with the main() function, and here the variables dNo1, dNo2 and dAvg are declared.

Then the user is prompted for two numbers to be stored in the variables dNo1 and dNo2. Then comes the statement:

```
dAvg = dAverage(dNo1, dNo2);
```

which calls the function dAverage and supplies the two entered values. The function does its job and returns the average, which is stored in the variable dAvg. The last statement prints the calculated average.

You may ask why we don't write the main() function first, since it is the starting point of the execution. The reason is that the compiler must know about the function dAverage before it is called from the main() function. Therefore the compiler must first process dAverage. We will discuss this more later in this chapter.

## 6.4 Several return Statements

A function might need to return different values depending on the circumstances. Consequently, a function can contain several return statements. However, it is always only one return statement performed by the function at execution. As soon as a return statement is executed, the function completes.

Here is an example of a function with two return statements:

```
int min(int x, int y)  
{  
  
6 Functions  
  
if (x<y)  
  
return x;  
  
else  
  
return y;  
  
}
```

The function returns the least of two integers. By examining the function header (the first line) we figure out that the function takes two integers stored in the formal parameters x and y and that it returns an integer.

The function body contains an if statement that checks whether x is less than y. If so, x is returned, otherwise y is returned.

This implies that the function always returns the least of the two numbers.

### 6.4.1 Example: Least of Three Numbers

We will now use the function min() in a program that reads three integers from the keyboard and prints the least of them.

Since the function `min()` only can compare two numbers, we will have to call it twice. We compare the two first integers entered by the user and get as a result the least of these two. That integer is then compared to the third number, which again gives the least of them. As a result we get the least of all three integers.

Here is the code:

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

cout << "Enter three integers: ";

cin >> a >> b >> c;

m = min(a,b);

m = min(m,c);

cout << m;
```

We declare the three variables `a`, `b` and `c` for storing of the input values, and a variable `m` used for storing of the value returned from the function `min()`.

The function `min()` is called with `a` and `b` as actual parameters. The least of these two is returned and stored in the variable `m`. The function `min()` is again called with `m` and `c` as actual parameters. The least of these two is returned and stored in the variable `m`, which now contains the least of the three integers. Finally that value is printed.

Here is the entire program:

```
#include <iostream.h>

int min(int x, int y)

{

    if (x<y)

        return x;

    else

        return y;

}

void main()

{

    int a, b, c, m;

    cout << "Enter three integers: ";

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

```

    m = min(a,b);

    m = min(m,c);

    cout << m;

}

```

As mentioned, the execution starts in main(). The reason for placing the function min() before main() is for the compiler to recognize it when calling it from main().

An alternative to calling the function min() in two different statements is given here:

```

    cout << min(min(a,b),c) << " is the least";

```

This statement replaces the three last statements of the preceding program. First the program tries to execute the outer min() call. But since the first actual parameter is not an ordinary variable value, the program must calculate it, and is then forced to execute the inner min() call. As a result it returns the least of a and b, which now can be used as actual parameter to the outer min(). The second actual parameter to the outer min() call is the variable c. The outer min() returns the least of the three integers, which is printed by the cout statement.

## 6.5 Array As Parameter

Sometimes you want to send an entire array as parameter to a function, especially when working with strings. The function header for such a function could look like this:

```

int iLgth(char s[])

```

The function iLgth takes a parameter of char type. The formal parameter name is s. The empty square bracket indicates that it is an array. Note that the square bracket does not contain any value indicating the number of items of the array.

The reason is that the function should be able to manage arrays of any size, so we don't want to lock the function to any fixed array size.

The function header also shows that the function returns a value of integer type, namely the length of the string array.

The calculation of the length is made by the function body:

```

int iLgth(char s[])
{
    int n=0;
    while (s[n] != '\0')
        n++;
    return n;
}

```

First a variable `n` is declared initialized to 0, which is to count the number of characters in the string `s`. The while loop has the condition that the `n`th character must not be the null character, i.e. it proceeds until the end of the string. When the `n`th character equals the null character, the string end has been reached and `n` then contains the number of characters in the string, which is returned.

When the function `iLgth` is called with an array as actual parameter, you don't specify any square brackets:

```
iLen = iLgth(cWord);
```

Here, `cWord` is a string array that contains a number of characters. The returned length of the string is stored in the variable `iLen`.

Below is a program that tests the function `iLgth`:

```
#include <iostream.h>  
  
int iLgth(char s[])  
{  
    int n=0;  
    while (s[n] != '\0')  
        n++;  
    return n;  
}  
  
void main()  
{  
    char cWord[30];  
    int iLen;  
    cout << "Enter a word: ";  
    cin.getline(cWord,29);  
    iLen = iLgth(cWord);  
    cout << "The length of the word is " << iLen << endl;  
}
```

In `main()` we declare the string array `cWord` with max 30 characters, and the integer variable `iLen`, which later will contain the string length. A text is read from the keyboard. Then the function `iLgth()` is called with the entered string as actual parameter. The function returns the length of the string which is stored in the variable `iLen` and printed.



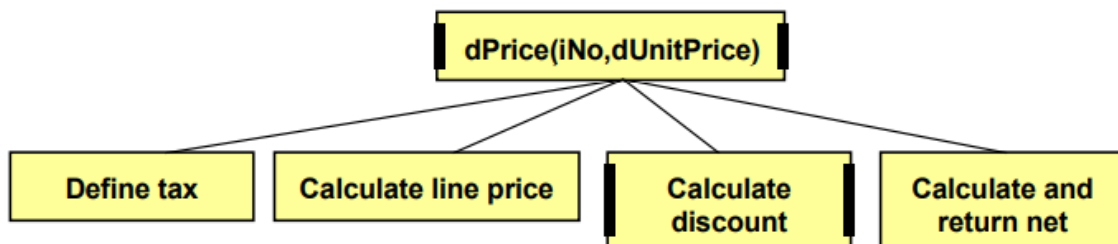
## 6.6 Function and Subfunction

It is possible to write functions to be called by other functions providing a hierarchy of functions and subfunctions. This is rather common in bigger programs and provides well-structured programs, where each function performs a limited and well-defined task

### 6.6.1 Example

We will now write a function, `dPrice()`, which calculates the price of a product and also calls another function, `dDiscount()`, which calculates and returns a discount percent. The percent is then used by `dPrice()` to calculate the discounted price.

First we write a JSP graph for the function `dPrice()`:

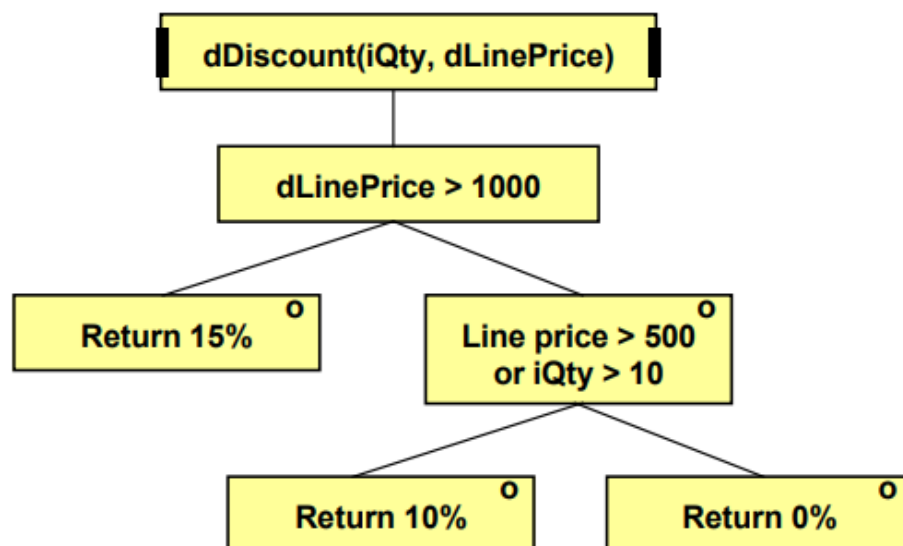


The function takes two parameters, number of units and the unit price.

The function defines the tax factor to 0.25. The line price is calculated as number of units times unit price. The discount is then calculated by sending the quantity and line price. Later, the discount will be dependent of both the quantity and line price. The function `dDiscount` returns a discount percent to be used for calculation of the discounted net price, which is returned by `dPrice()`.

There are some thicker side lines indicating functions.

The JSP graph for the `dDiscount()` function looks like this:



The function `dDiscount()` takes the quantity and line price as parameters. If the line price exceeds 1000, the discount 15 % is returned. Otherwise (i.e. the line price is less than or equal to 1000), we check if the line price exceeds 500 or the quantity exceeds 10. Then 10 % is returned. In other cases there will be no discount and 0 % is returned.

We begin the coding with the `dDiscount()` function. When writing programs with functions it is convenient to start with the functions at the bottom of the hierarchy.

```
double dDiscount(int iQty, double dLinePrice)  
{  
    if (dLinePrice>1000)  
        return 0.15;  
    else if (dLinePrice>500 || iQty>10)  
        return 0.10;  
    else  
        return 0;  
}
```

The function `dDiscount()` takes the quantity as parameter, which is stored in the formal parameter `iQty`, and the line price stored in the parameter `dLinePrice`. The if statement checks the values and returns a percentage of the double type.

Below is the code for the `dPrice()` function:

```
double dPrice(int iNo, double dUnitPrice)  
{  
    const double dTax = 0.25;  
    double dLinePr, dDiscPerc;  
    dLinePr = iNo * dUnitPrice;  
    dDiscPerc = dDiscount(iNo, dLinePr);  
    return dLinePr * (1-dDiscPerc)*(1+dTax);  
}
```

The function `dPrice()` takes the quantity and unit price as parameters. First a constant named `dTax` is declared. Then variables for the line price and discount percent are declared. The line price is calculated as quantity times unit price.

Then the function `dDiscount()` is called with quantity and line price as actual parameters. The returned discount percentage is stored in the variable `dDiscPerc`. Finally, the discounted net price of double type is returned, where we deduct the discount percentage and add the tax percentage.

Here is an entire program that tests the functions:

```
#include <iostream.h>

double dDiscount(int iQty, double dLinePrice)
{
    if (dLinePrice>1000)
        return 0.15;
    else if (dLinePrice>500 || iQty>10)
        return 0.10;
    else
        return 0;
}

double dPrice(int iNo, double dUnitPrice)
{
    const double dTax = 0.25;
    double dLinePr, dDiscPerc;
    dLinePr = iNo * dUnitPrice;
    dDiscPerc = dDiscount(iNo, dLinePr);
    return dLinePr * (1-dDiscPerc)*(1+dTax);
}

void main()
{
    int iQuantity;
    double dUnitPrice;
    cout << "Enter quantity and unit price: ";
    cin >> iQuantity >> dUnitPrice;
    cout << "To be paid: "
```

```

    << dPrice(iQuantity, dUnitPrice)<<endl;
}

```

In main() we declare the variables iQuantity and dUnitPrice, which store the entered values. The cout statement calls the dPrice() function with the entered values as actual parameters. As return value we get the discounted net price, which is printed by the cout statement

## 6.7 Function without Return Value

Some functions are supposed to perform a task but don't need to return any value. An example could be a function that prints a message or performs some string manipulation.

We will create a function that prints the character '=' a specific number of times. It can be used to print a number of equality signs acting as underlining a text. We use the name underline for the function:

```

void underline(int n)
{
    for (int i=1; i<=n; i++)
        cout << "=";
}

```

A function that does not return any value has 'void' in front of the function name. Compare the function main() which does not have to return any value to the operating system, and which consequently has been defined as 'void main()'.

The function underline() takes an integer as parameter. It has a for-loop that prints the character '=' as many times as given by the integer.

We can now use this function in a program to underline a text:

```

char s[7] = "Prices";

cout << s << endl;

underline(strlen(s));

```

This code section defines a string array s with the string 'Prices'. The string is printed at the second code line and then we call the function underline() with the length of the string as actual parameter. This gives the output:

**Priser**

=====

An alternative to the function underline is given here:

```

void underline(char text[])
{

```

```

    for (int i=1; i<=strlen(text); i++)

    cout << "=";

}

```

Here, the function instead takes the string itself as parameter. The for-loop goes from 1 to the number of characters in the string and prints as many equality signs. The call to this variant of the function should be:

```

char s[7] = "Prices";

cout << s << endl;

underline(s);

```

Here we send the string variable as actual parameter instead of the length of the string.

## 6.8 Replacing Characters in a String

We will create still another void function that replaces an arbitrary character in a string and prints the modified string.

We call it replace:

```

void replace(char s[], char c, char cnew)
{
    int n = 0;
    while (s[n] != '\0')
    {
        if (s[n] == c)
            s[n] = cnew;

        n++;
    }

    cout << s << endl;
}

```

The function takes three parameters, a string array - s, the character to be replaced – c, and the replacing character – cnew.

The function first defines a variable n to be used for indication of one character at a time in the string. The while loop has the condition that the character must not be the null character, i.e. we proceed from the first to the last character of the string.

The if statement checks whether the character in the string equals the character to be replaced (c). If so, that character is replaced by cnew. In the while loop we finally increase n by 1 to point out the next character in the string.

Finally the modified string is printed by the cout statement.

Here is an entire program that tests the replace() function:

```
#include <iostream.h>

void replace(char s[], char c, char cnew)
{
    int n= 0;
    while (s[n] != '\0')
    {
        if (s[n] == c)
            s[n] = cnew;
        n++;
    }
    cout << s << endl;
}

void main()
{
    char a[100] = "C:/Mydocuments/Sheets";
    cout << a << endl;
    replace(a, '/', '-');
}
```

In main() we define a string array a which is printed. Then the replace() function is called with the actual parameters a,

the character '/' and the character '-'. The printout will be:

**C:/Mydocuments/Sheets**

**C:-Mydocuments-Sheets**

## 6.9 Override Functions

A function can appear in different shapes. For instance, it can perform a task with different data sets. What differs between the various shapes is the parameter set. If the number of parameters or the data types of the parameters are different, it is considered different function shapes, or override functions. The function header defines the difference. Here are two examples of override functions:

```
void prt(int i, int width)
```

```
void prt(char s[])
```

The functions have the same name (prt), but different parameter sets. The purpose of the functions is to print something on the screen. The first function should print the number i with as many positions as given by the width parameter (using the setw() function). The second function should print the string s.

Here is the code for the two functions:

```
void prt(int i, int width)  
{  
    cout << setw(width) << i;  
}  
  
void prt(char s[])  
{  
    cout << s;  
}
```

The first function uses the number width as parameter to the setw() function, which assigns that number of positions on the screen for the number i.

The second function prints the string sent to the function.

We can now use the function prt() in a program to print a number or a text. Depending on the actual parameters sent to the function, the program will select the appropriate function variant.

For instance, the statement:

```
prt("The number:");
```

will select the second function, while the statements:

```
int k = 8;  
prt(k, 3);
```

will select the first function. If we combine these statements:

```
prt("The number:");
```

```
int k = 8;

prt(k, 3);
```

we will get the printout:

**The number: 8**

with two blanks in front of the 8, since we set width to 3, totally three positions.

Writing override functions in this way provides a flexibility to programming, where the program selects the function variant applicable for the moment.

## 6.10 Declaration - Definition

We have previously stated that a function should be positioned in front of main() in the program to make the compiler be able to recognize it when called from main(). Here is an example of this:

```
double dAvg(double x1, double x2)
{
    return (x1 + x2)/2;
}

main()
{
    //...

    mv = dAvg(no1, no2);

    //...
}
```

Here, the function dAvg() calculates the average of the two numbers sent to the function. In main() we call dAvg() with the actual parameters no1 and no2, which we assume to have been assigned values previously in the program.

An alternative is to write only the function declaration before main() and let the definition of the function appear after

```
main():

double dAvg(double x1, double x2);

main()
{
    //...
```



```

    mv = dAvg(no1, no2);

    //...
}

double dAvg(double x1, double x2)
{
    return (x1 + x2)/2;
}

```

The first line declares the function `dAvg()`. It is exactly identical to the function header, followed by a semicolon. Having declared the function, the compiler knows about it and is recognized when called from `main()`.

The definition of the function, i.e. the function header plus the function body, can then be positioned anywhere in the program. If you have several functions declared before `main()`, you can write the function declarations in any order after `main()`.

This way of first declaring functions and placing the definitions afterwards is common by programmers and provides the advantage of having `main()` first, which is logical since the execution starts there.

You can declare a function in an abbreviated way:

```

double dAvg(double, double);

```

Here we exclude the formal parameter names and specify only their data types. However, the function header of the function definition must be complete with formal parameter names.

## 6.11 Header Files

Function declarations are often stored in a separate header file and the function definitions in the corresponding `cpp` file.

The header file must then be included in the program using the functions.

For instance, you can create a `cpp` file with all your function definitions. Suppose we name it `myfunc.cpp`. The function declarations are stored in the header file `myfunc.h`.

When writing the program in still another `cpp` file, which will call the functions, `myfunc.h` must be included in the program file.

Here is a set of functions that we have used previously in this chapter:

```

// myfunc.cpp

#include <iostream.h> // Necessary for cout

void underline(int n)
{

```

```

    for (int i=1; i<=n; i++)

    cout << "=";

}

double dDiscount(int iQty, double dLinePrice)
{
    if (dLinePrice>1000)

    return 0.15;

    else if (dLinePrice>500 || iQty>10)

    return 0.10;

    else

    return 0;

}

double dPrice(int iNo, double dUnitPrice)
{
    const double dTax = 0.25;

    double dLinePr, dDiscPerc;

    dLinePr = iNo * dUnitPrice;

    dDiscPerc = dDiscount(iNo, dLinePr);

    return dLinePr * (1-dDiscPerc)*(1+dTax);

}

// myfunc.h

void underline(int n);

double dDiscount(int iQty, double dLinePrice);

double dPrice(int iNo, double dUnitPrice);

// price.cpp

#include <iostream.h>

#include "myfunc.h"

void main()

{

```

```

//...

cout << "To be paid: " << dPrice(iQty, dUnitPr)<<endl;

//...

}

```

In the price.cpp program we have included the file myfunc.h. The compiler will look in myfunc.h to ensure that all functions called by the program are declared in myfunc.h.

Note that we include own header files with double quotes instead of the characters < and >. That implies that the compiler looks in different folders to find the header files. The files myfunc.h and myfunc.cpp should be stored in the same folder as price.cpp, while iostream.h is stored in a particular folder created at installation of Visual C++.

## 6.12 Recursive Functions

A recursive function is a function that calls itself, i.e. from inside of the function body you call the same function in a program statement. This may sound as an infinite loop. The code must of course be constructed with a condition to make the series of calls be interrupted.

Here is the basic logic for a recursive function:

```

func()
{
    //misc code
    if (...)
        return func();
    else
        return 1;
}

```

The function func() has a call to itself in the first return statement. This call will be performed as long the if condition is true, repeatedly. But some time the statements before the if statement must imply that the if condition is false. Then the value 1 is returned and the recursive function calls are interrupted.

Recursive functions are mostly used in mathematical applications. We will create a recursive function which calculates the faculty of the number sent to the function.

Some repetition of your math knowledge. Faculty is identified by !. For instance 5! (faculty of 5) = 5 x 4 x 3 x 2 x 1. You start with the number and repeatedly multiply by the number that is 1 less until you arrive at 1.

The function has the following header:

```

int nfac(int n)

```

The function name is `nfac` and it takes an integer as parameter. We will consequently multiply `n` by `n-1`, `n-2`, `n-3` etc. down to 1. We will call `nfac()` with the number that is 1 less than the number used in the previous call. So the function must have an if statement which checks if the parameter `n` is = 1. If so, the number 1 should be returned, otherwise the function should be called again, this time with `n-1` as actual parameter.

Here is the code:

```
int nfac(int n)
{
    if (n<=1)
        return 1;
    else
        return n * nfac(n-1);
}
```

The function has an if statement which checks if the supplied parameter is 1 or less. If so, 1 is returned. Otherwise the product of `n` and the result of the call to the same function with parameter `n-1` is returned.

Suppose we use this call:

```
iFaculty = nfac(4);
```

The number 4 is sent as parameter.

Since 4 is greater than 1, this multiplication is performed: `4 * nfac(3)`.

The call `nfac(3)` accordingly gives the multiplication `3 * nfac(2)`.

The call `nfac(2)` gives the multiplication `2 * nfac(1)`.

The call `nfac(1)` now has the actual parameter 1 and since it provides a true condition in the if statement, 1 is returned, and no more calls are made.

Thus, the resulting multiplication is `4 * 3 * 2 * 1`, which is stored in the receiving variable `iFaculty`.

## 6.13 Summary

In this chapter we have learnt to write functions. Functions are used in professional programs to split up the code into well-defined sections and to achieve a structure easy to grasp, which facilitates program maintenance.

We have learnt how to send values to, and receive the result from a function. We have also learnt how to use header files in connection with declaration and definition of functions.

We have made a brief introduction to reference parameters – an efficient tool to save memory and improve program performance. You have also seen how to write recursive functions to make the code more efficient.

## 6.14 Revision Questions

1. Write a function which calculates and returns the average of three numbers. Call the function from `main()` and create convenient printouts, so you can check the correctness of the function.
2. Write a function `max()` which returns the greatest of two numbers. Test the function with a call from `main()` and complete with suitable printouts.
3. Complete the previous program so that it can calculate the greatest of three numbers by means of the function `max()`.
4. Start from the program 'Least Item of an Array' and complete it with printing of the greatest item of the array by means of the function `max()`.
5. Write three functions which calculate
  - circumference of a rectangle with the sides as parameters
  - area of a rectangle with the sides as parameters
  - price for building a fence around a rectangular field, where the price per meter is 145:- and a gate of 650:-

Then, write a program that reads the sides of the rectangle from the user and displays the circumference, area and fence price by means of the three functions.

6. Write a function that takes an integer `n` and prints a list of the squares and cubes of the numbers 1-`n`. From `main()`, read the number `n` from the user.
7. Write a function `printLine(int n, char c)` which prints the character `c` at a line as many times as specified by the integer `n`. Use the function in a program which prints a frame consisting of asterixes (\*) on the screen.
8. Write a function `expandWord(char cWord[])` which prints the text in the parameter 'cWord' with one blank between each character. E.g. the word Data is printed as D a t a. Also write a `main()` program which reads a text from the user and calls the function.
9. Write the following functions:
  - `void initial(char str[])` which prints the initials of the name `str`.
  - `void revers(char str[])` which interchanges the first and surname of `str` and prints it.
  - `int lgth(char str[])` which returns the length of `str`.
  - `void back(char str[])` which prints the name backwards.
  - `void upper(char str[])` which prints the name in upper case.

Use the functions in a `main()` program.

10. Start from the program with the price calculation in the section 'Function and Subfunction'. Write one more subfunction which is used by the function `dPrice()` and which reads a customer category from the user and returns still another discount percent (A=5%, B=7%, C=9%). Ensure that the function takes erroneous entry into account. Modify the function `dPrice()` to provide a correct price calculation.