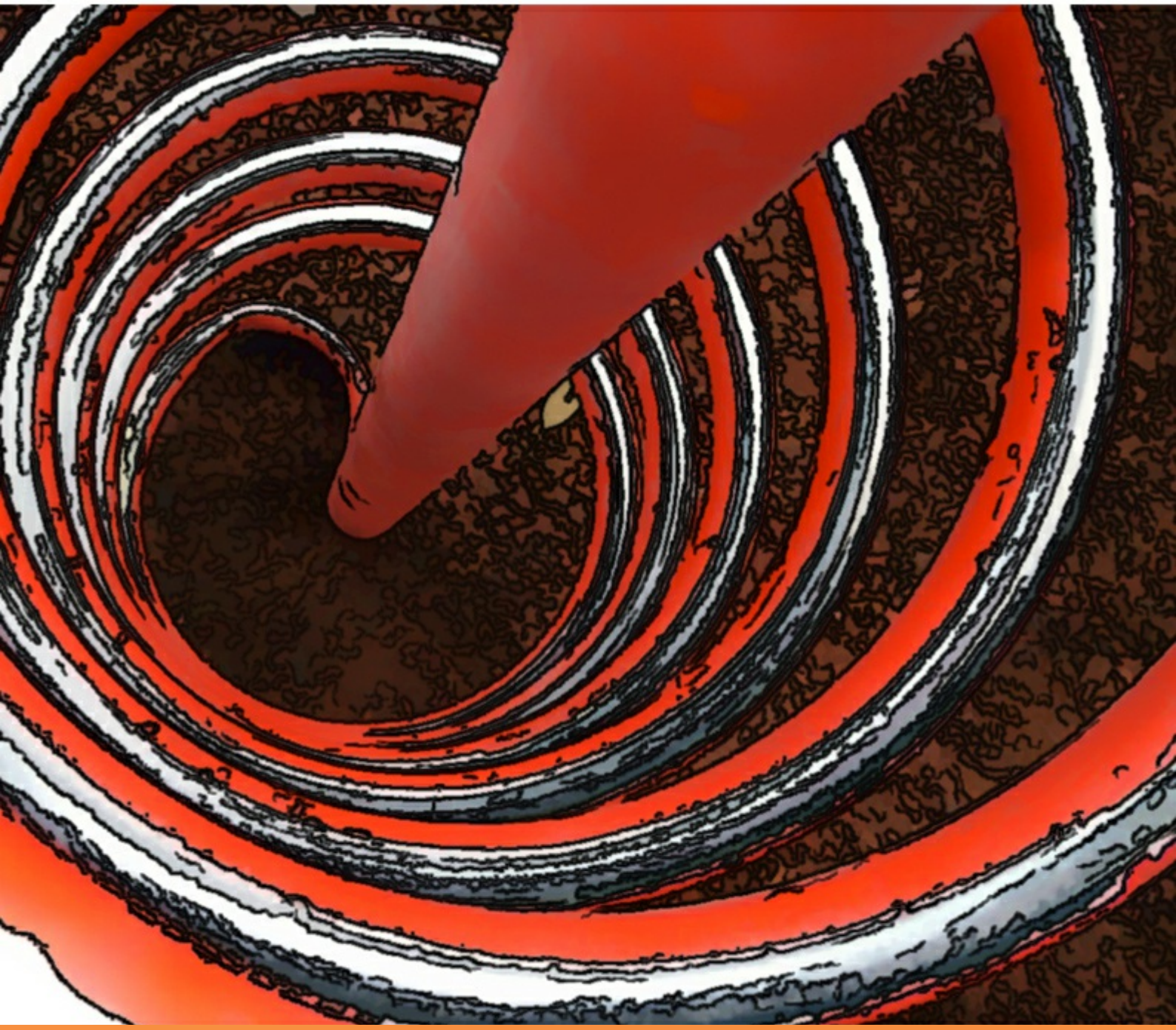
Structured Programming with C++

Kjell Bäckman



bookboon
The eBook company

Kjell Bäckman

Structured Programming with C++

Structured Programming with C++

ISBN 978-87-403-0099-4

Contents

	About the Book and the Course	**11**
1	**Introduction to Programming**	**13**
1.1	What Does It Mean to Program	13
1.2	Coding	15
1.3	Compiling and linking	16
1.4	The First Steps with Visual C++	17
2	**Variables**	**24**
2.1	Introduction	24
2.2	Why Variables	24
2.3	Declaring Variables	25
2.4	Assignment	25
2.5	Initiating Variables	26
2.6	Constants	26
2.7	More about Assignment of Values	27
2.8	The main function	28
2.9	Input and Output	29
2.10	An Entry Program	31

2.11	Formatted Output	33
2.12	Invoice Program	35
2.13	Time Conversion Program	37
2.14	Type Conversion	39
2.15	The Random Number Generator	40
2.16	Game Program	41
2.17	Summary	42
2.18	Exercises	42
3	**Selections and Loops**	**45**
3.1	Introduction	45
3.2	Selection	45
3.3	if statement	45
3.4	Price Calculation Program	46
3.5	Comparison Operators	48
3.6	Even or Odd	49
3.7	else if	49
3.8	and (&&), or (||)	50
3.9	Conditional Input	51
3.10	The switch statement	52
3.11	Menu Program	52
3.12	Loops	55

3.13	The while Loop	57
3.14	The for Loop	57
3.15	Addition Program	58
3.16	Double Loop	60
3.17	Roll Dice	61
3.18	Two Dice Roll	63
3.19	Breaking Entry with Ctrl-Z	64
3.20	Pools	65
3.21	Equation	67
3.22	Interrupting a Loop - break	69
3.23	Summary	70
3.24	Exercises	70
4	**Arrays**	**73**
4.1	Introduction	73
4.2	Why Arrays	73
4.3	Declaring an Array	74
4.4	Initiating an Array	75
4.5	Copying an Array	76
4.6	Comparing Arrays	76
4.7	Average	77
4.8	Sales Statistics	80

4.9	Product File, Search	85
4.10	Two-Dimensional Array	85
4.11	Sorting	87
4.12	Searching a Sorted Array	90
4.13	Summary	94
4.14	Exercises	94
5	**Strings**	**96**
5.1	Introduction	96
5.2	Data Type char	96
5.3	Menu Program	96
5.4	Menu Program with Loop	98
5.5	Christmas Tree	100
5.6	int - char	103
5.7	Å, Ä, Ö	103
5.8	String Array, char[]	103
5.9	Length of a String	105
5.10	Upper/Lower Case	106
5.11	Initials	106
5.12	Comparing Two Strings	108
5.13	Copying Strings	109
5.14	Array with String Arrays	109

5.15	Sorting Strings	110
5.16	Substring	112
5.17	Concatenating Strings	112
5.18	Interchanging First Name and Surname	112
5.19	Encryption	115
5.20	Random Password	116
5.21	Translation Table	117
5.22	Summary	120
5.23	Exercises	120
6	**Functions**	**123**
6.1	Introduction	123
6.2	What Is a Function	123
6.3	Average	124
6.4	Calling a Function	124
6.5	Several return Statements	126
6.6	Least of Three Numbers	127
6.7	Least Item of an Array	129
6.8	Array As Parameter	130
6.9	Function and Subfunction	132
6.10	Function without Return Value	135
6.11	Replacing Characters in a String	136

6.12	Declaration Space	138
6.13	The Word Program	138
6.14	Override Functions	140
6.15	Declaration - Definition	141
6.16	Header Files	143
6.17	Reference Parameters	145
6.18	Parameters with Default Values	147
6.19	Recursive Functions	148
6.20	Summary	150
6.21	Exercises	150
7	**Files**	**153**
7.1	Introduction	153
7.2	Streams	154
7.3	Reading from a Stream	154
7.4	Writing to a Stream	155
7.5	Attaching a File to a Stream	155
7.6	A Complete Write Program	157
7.7	A Complete Reading Program	158
7.8	New Item at the End of the File	160
7.9	Products and Prices	161
7.10	Search for a Product Price	163
7.11	Sorting a File in Memory	165
7.12	Updating File Content	168
7.13	Copying Files	171
7.14	Summary	172
7.15	Exercises	172
8	**Pointers**	**174**
8.1	Introduction	174
8.2	What Is a Pointer	174
8.3	Declaring a Pointer	175
8.4	Assigning Values to Pointers	175
8.5	Addresses and char Pointers	177
8.6	cout and char Pointers	178
8.7	Price Program with Pointers	178
8.8	Pointer Arithmetics	179
8.9	Tax Program	181
8.10	Functions and Pointers	182
8.11	Dynamic Memory	186
8.12	Summary	190
8.13	Exercises	190

9	**Structures**	**192**
9.1	Introduction	192
9.2	What Is a Structure	192
9.3	Defining a Structure	192
9.4	Declaring and Initiating Structure Variables	193
9.5	Assigning Values to Structure Members	193
9.6	A Structure Program	194
9.7	Array with Structure Variables	196
9.8	Pointer to Structure	197
9.9	Structures in the Dynamic Memory	198
9.10	Structure As Function Parameter	199
9.11	Summary	206
9.12	Exercises	206
10	**Answers**	**207**
10.1	Variables	207
10.2	Selections and Loops	219
10.3	Arrays	225
10.4	Strings	229
10.5	Functions	234
10.6	Files	240
10.7	Pointers	241
10.8	Structures	245

About the Book and the Course

This book is intended as course material for the course Structured Programming with C/C++ at university level. It contains eight chapters, one for each lecture of the course. The chapters are:

1. **Introduction to programming.** Here we go through general principles about what programming means. You will be introduced to the development tool Microsoft Visual C++ and build your first programs.
2. **Variables.** Here we start from the beginning and explain all details in the first programs. You will learn what variables are and how they are used for storing of values needed in the program.
3. **Selections and loops.** This chapter will teach you to include intelligence into the programs, which will be capable of doing different things depending on given conditions. The programs will also be able to repeat operations any number of times.
4. **Arrays.** Arrays are very useful tools for storing of and working with information items of the same kind, like for instance sampled measurement values, product files with prices, or customer data.
5. **Strings.** Texts (strings) are handled in a special and tricky way in C++. Because of that a special chapter is dedicated for this subject.
6. **Functions.** In this chapter, when our programs are getting to some size, we divide the code into subroutines or functions.
7. **Files.** Many times data needs to be stored or accessed. Here we will learn the basics of file management.
8. **Pointers.** This chapter goes deeper into the more advanced aspects of C++ programming and implies a springboard to professional level.
9. **Structures.** Data is often organized into structures, which means easier input and output of structured data. Here we also use pointers together with structures.

Each chapter contains theoretical parts and programming examples. At the end of each chapter there is a bunch of exercises for your practice. At the end of the book you will find solutions to the exercises. Remember, though, that each problem could in general be solved in different ways, and maybe your own solution is as good as, or even better than the one presented. Therefore, my recommendation is that you as far as possible try to manage without the solutions.

The purpose of the course Structured Programming with C/C++ is primarily to teach how to ”think programming” and secondarily to teach C++ code. Therefore, I will emphasize how to focus on the problem solution and prepare the coding. **JSP** (Jackson Structured Programming) is a common tool within programming and is used to structure a problem. You will learn how to use JSP to build your solution. **Flow charts** is an alternate tool to JSP, which we also will make some notice to.

Primarily, this is a beginner’s course in C, but we will utilize some C++ tools for e.g. input and output.

Learning to program is not made in short time. It requires long-term and patient work with reading, coding, testing and debugging. There is no shortcut, but if you work with endurance, you will have many times of inspiration and nice experiences.

I would also like to stress that, when you write your programs, you will make many mistakes. This is normal. No programmer writes the correct code already from the beginning. Error tracing and correction is a natural ingredience of the development process.

Finally I would like to thank all who has encouraged me and supplied positive feed-back to make this book as pedagogic as possible.

Kjell Bäckman
Department of Economy and Informatics
University West, Trollhättan
Sweden

1 Introduction to Programming

1.1 What Does It Mean to Program

1.1.1 Algorithm

Programming is not only coding. Primarily it implies structuring of the solution to a problem and then refine the solution step by step. When refined to a level deep enough, you have created an **algorithm**. Then it is time to translate each step of the algorithm to program code.

Suppose you have a problem that needs to be solved. Then you begin with writing a sequence of operations at an overview level that need to be performed to solve the problem. Then you start from the beginning again and focus on one operation at a time and find out whether the operation needs to be refined to more detailed steps. Then you proceed to the next level and refine further. This refinement process goes on until you arrive at a level deep enough to start coding.

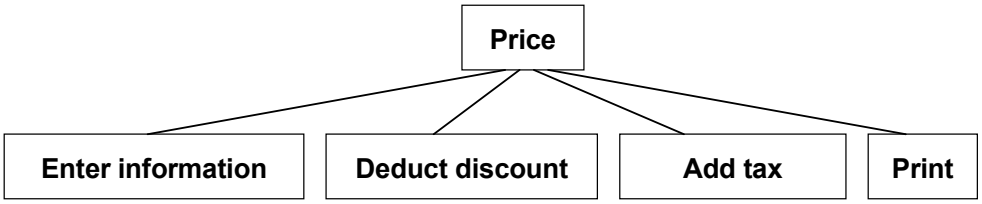
Creating an algorithm to solve a problem is in general the most laboursome task of the programming work. Many people do the mistake of starting to code at once, which makes you focus on code details and forget the actual problem to be solved. That gives an unstructured and inefficient code hard to understand and maintain.

That's why we emphasize that you structure your logic train of thought and construct a good algorithm before starting to code.

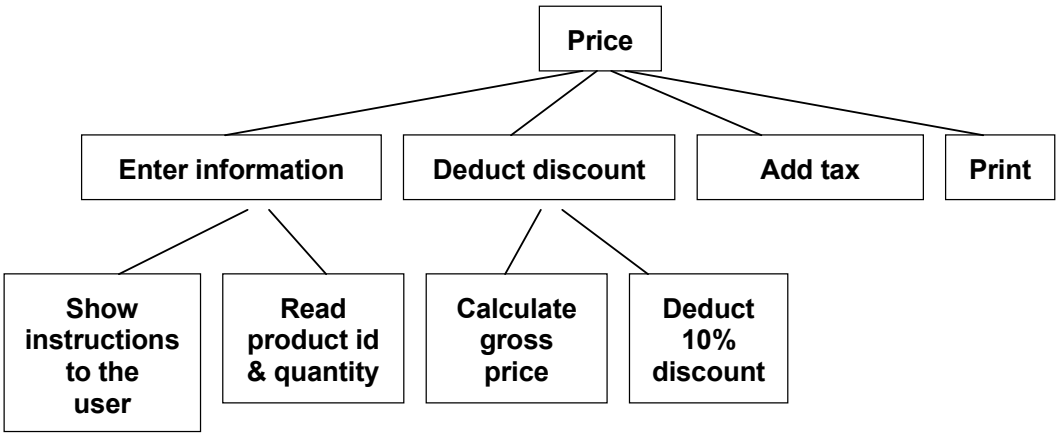
1.1.2 JSP

A JSP graph is a tool to create an algorithm. JSP is an abbreviation for Jackson Structured Programming and is a commonly used instrument for logic structuring. Let's take an example.

You are supposed to create a program that calculates the price of a product to be bought by a customer. The customer specifies the product id and the requested quantity. The program should then calculate the relevant discount, add tax and show the customer price. A JSP graph could look as follows:

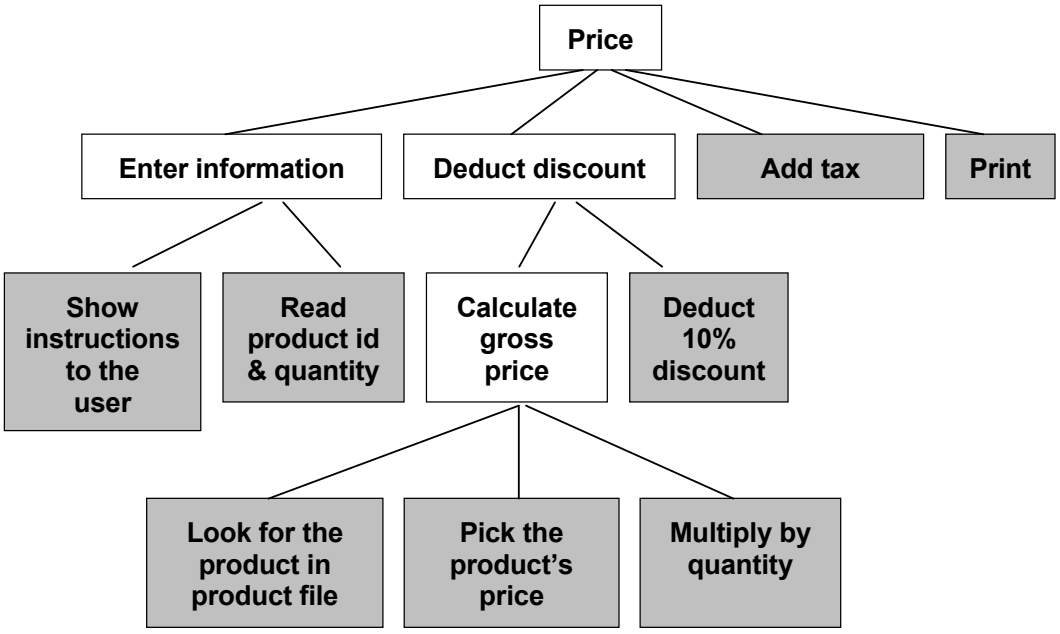


The upper box is the name of the program. We have split the solution into four steps at an overview level. You read the steps from left to right. As you probably realize the algorithm is too rough to be able to write code. So we proceed by refining the solution to the next level:



The box ”Enter information” has been split into two steps, “Show instructions to the user” and “Read product id & quantity”. In the same way we have refined the box “Deduct discount”.

We could break down the box ”Calculate gross price” further:



We could refine the algorithm further, but let us say that we are satisfied with the detail level. The shadowed boxes in the graph are the end points at the lowest level, the “leaves of the tree”. These are the boxes to be used for coding, from left to right.

We will work a great deal with JSP graphs in the program examples of the course.

1.1.3 Sequence - Selection - Iteration

Each program is logically built up from three basic logic principles:

- **Sequence** – the program performs instructions in sequence, one after the other.

- **Selection** – the program selects one of several operations depending on the prerequisites. The program thus makes a selection based on some condition.
- **Iteration** – the program repeats a series of instructions a certain number of times.

The logic principles can also be combined. For instance, a sequence of instructions can be repeated a number of times if a specific condition is satisfied, otherwise another sequence of instructions should be performed a specific number of times.

All programming languages use these three logic principles. If you have built up your algorithm in a correct way, it is only a question of selecting a programming language when the coding is to take place. The price calculation algorithm above should consequently give the same result irrespective of whether the code is written in C++, Java or VisualBasic.

In the JSP graph above the box "Calculate gross price" is refined in a *sequence* of three operations, from left to right:

- Look for the product in product file
- Pick the product's price
- Multiply by quantity

The box "Look for the product in product file" could suggest an *iteration*, e.g. "Read next product id until we find the product id stated by the user".

The discount calculation in the price program above could imply a more differentiated discount situation:

- If the gross price is between 100:- and 500:- the customer will get 5% discount.
- If the gross price is between 500:- and 1000:- the customer will get 8% discount.
- If the gross price is above 1000:- the customer will get 10% discount.

Here the program must do a *selection*.

1.2 Coding

When you have refined your algorithm to a level detailed enough, it is time to write code. This written code is called **source code**. The code must of course follow the rules in effect for the programming language in question, it must follow the syntax. Each programming language has its own rules.

You can in principle use any word processor or text editor you like, such as the program Notepad, Wordpad or Word. If you use word processors like Wordpad or Word, you must save the file as pure text file (.txt).

It is however recommended to use the text editor present in the program development package you are using. The advantage is that you will get some support when coding. Microsoft Visual C++, which is the program development package used in this course, contains an editor which:

- Shows key words in C++ in blue colour and comments in green colour,
- Provides IntelliSense, i.e. proposes code alternatives in certain situations,
- Supports context sensitive help, i.e. shows an explanation of a certain code item if you put the cursor on the item and press F1,
- Provides extensive support at debugging by allowing you to execute the code up to a certain breakpoint, where you can examine variable values at this particular position.

There are other development tools for C++ like Borland and Dev C++. The tools differ somewhat as concerns small details of the code. You can use any tool, the important thing is that you learn to "think" structured programming. In this course we have used Microsoft Visual C++ 2008, and all program examples are tested in this environment.

C++ is a very extensive language that can be used both within basic structured programming and object oriented programming. Furthermore, windows programming in the graphic Windows environment is supported. C++ has, thanks to its level of detail, its strength in digging deeply into the most obscure corners of the computer, control the operating system, communicate with hardware, circuit boards and external equipment like measurement units and communication devices. In this course, however, we will stick to basic structured programming.

C++ is a compact language with many symbols having their own meanings. This means that C++ code looks complicated to the novice. On the other hand it provides many tools for efficient coding. Programs written in C++ are very rapid due to the fact that the compiler optimizes them to each specific processor type. That is the reason why you mostly use C++ in situations where processor time and performance are ultimate.

1.3 Compiling and linking

When you have written the code of your program, it should be compiled, i.e. translate it to machine code consisting of 1's and 0's. That is the level understood by the processor. Before reaching this level, there is an interim level of code, called object code. Thus the compilation is made in two steps, first from source code to object code, and then from object code to machine code. In Microsoft Visual C++ you don't have to bother about these two steps, because the compilation from source code to machine code is taken care of by just clicking a button.

When writing a program, you often split the code into several files of source code. The different files contain references to each other. When all source code has been compiled the different parts of the program must be linked together to one single executable program (exe file). In some environments the linking must be initiated manually. In Visual C++ the linking is taken care of automatically directly after the compilation.

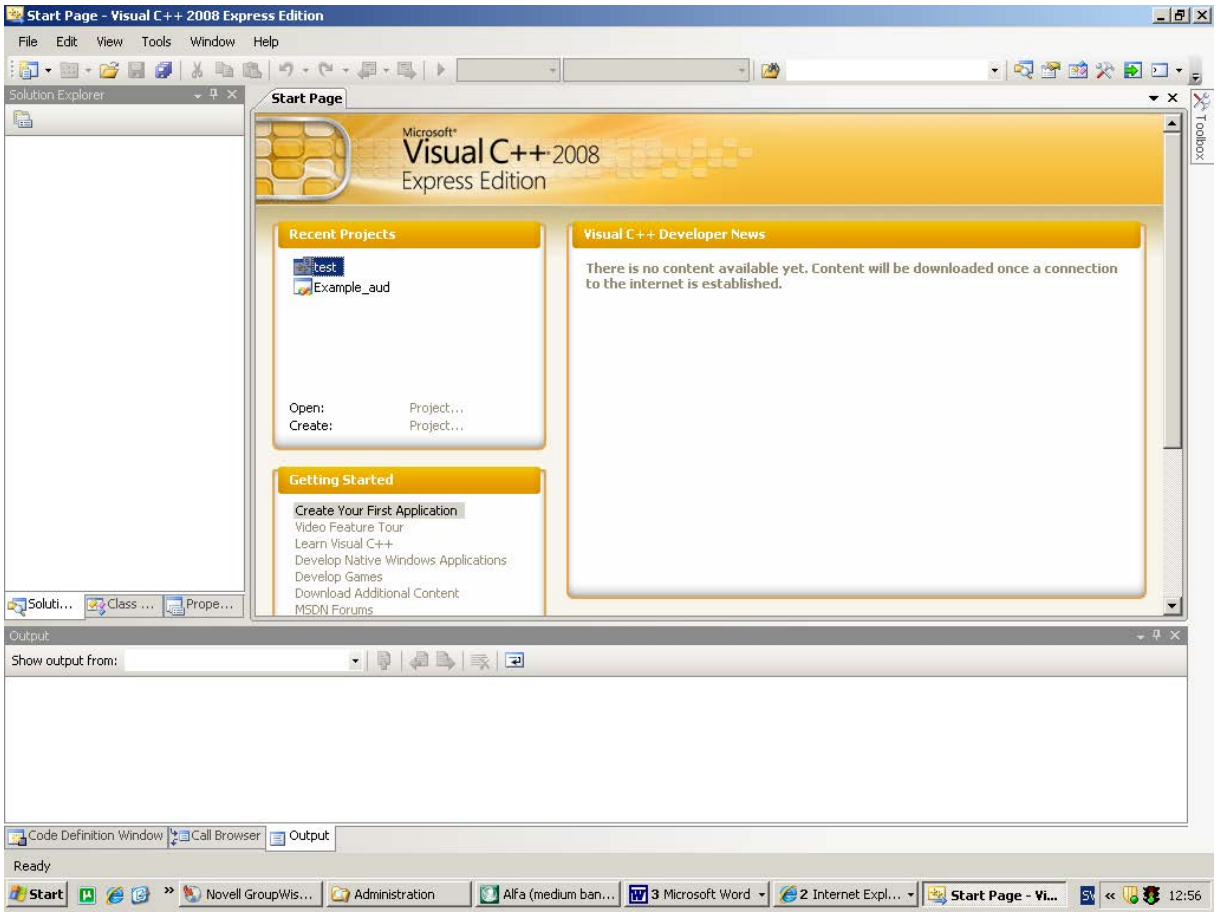
1.4 The First Steps with Visual C++

We will start with creating a program that prints 'Hello World' on the screen.

Click the Start button and select:

1.4.1 All programs – Visual C++ 2008 Express Edition -Microsoft Visual C++ 2008 Express Edition

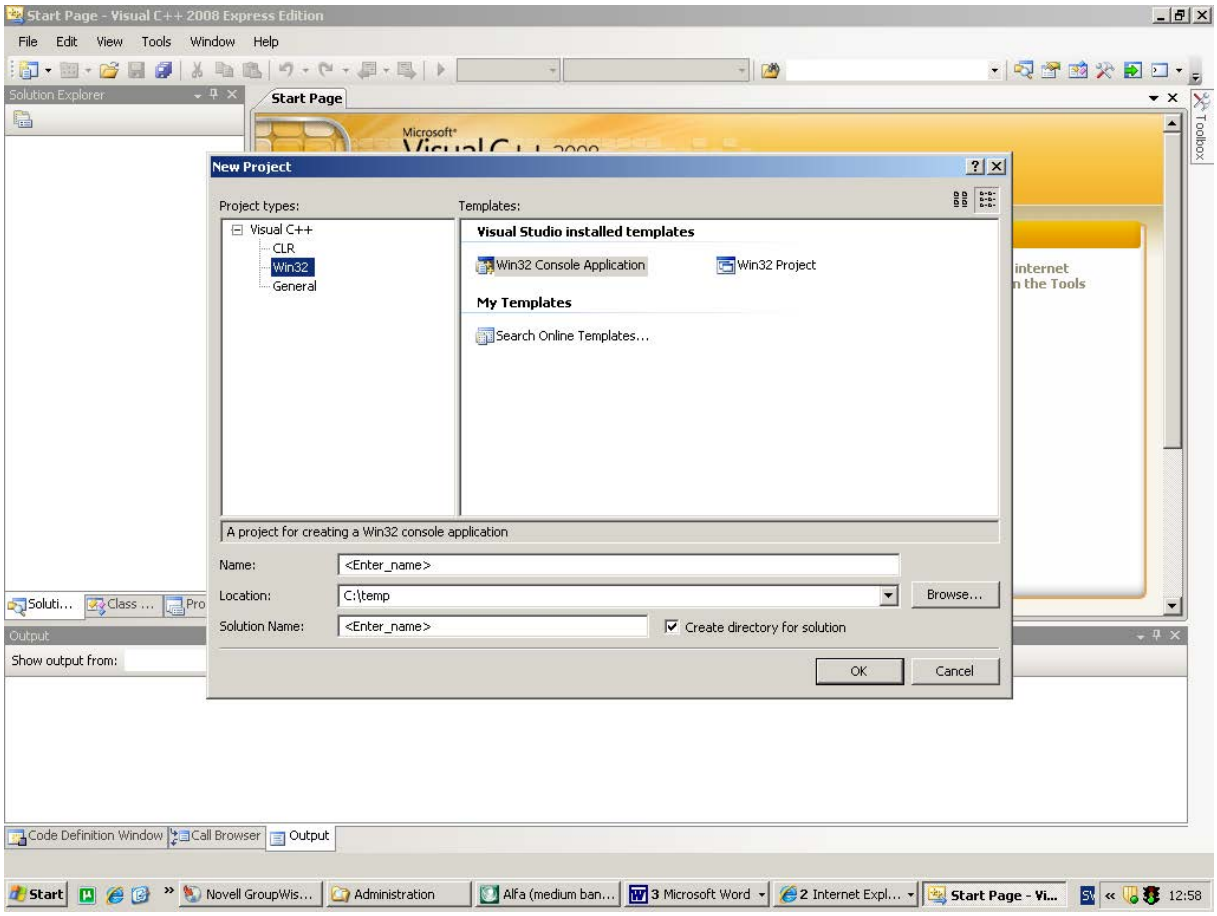
The Visual C++ Start Panel is displayed:



To create a new program, select from the menu:

1.4.2 File – New – New project

A window is displayed:

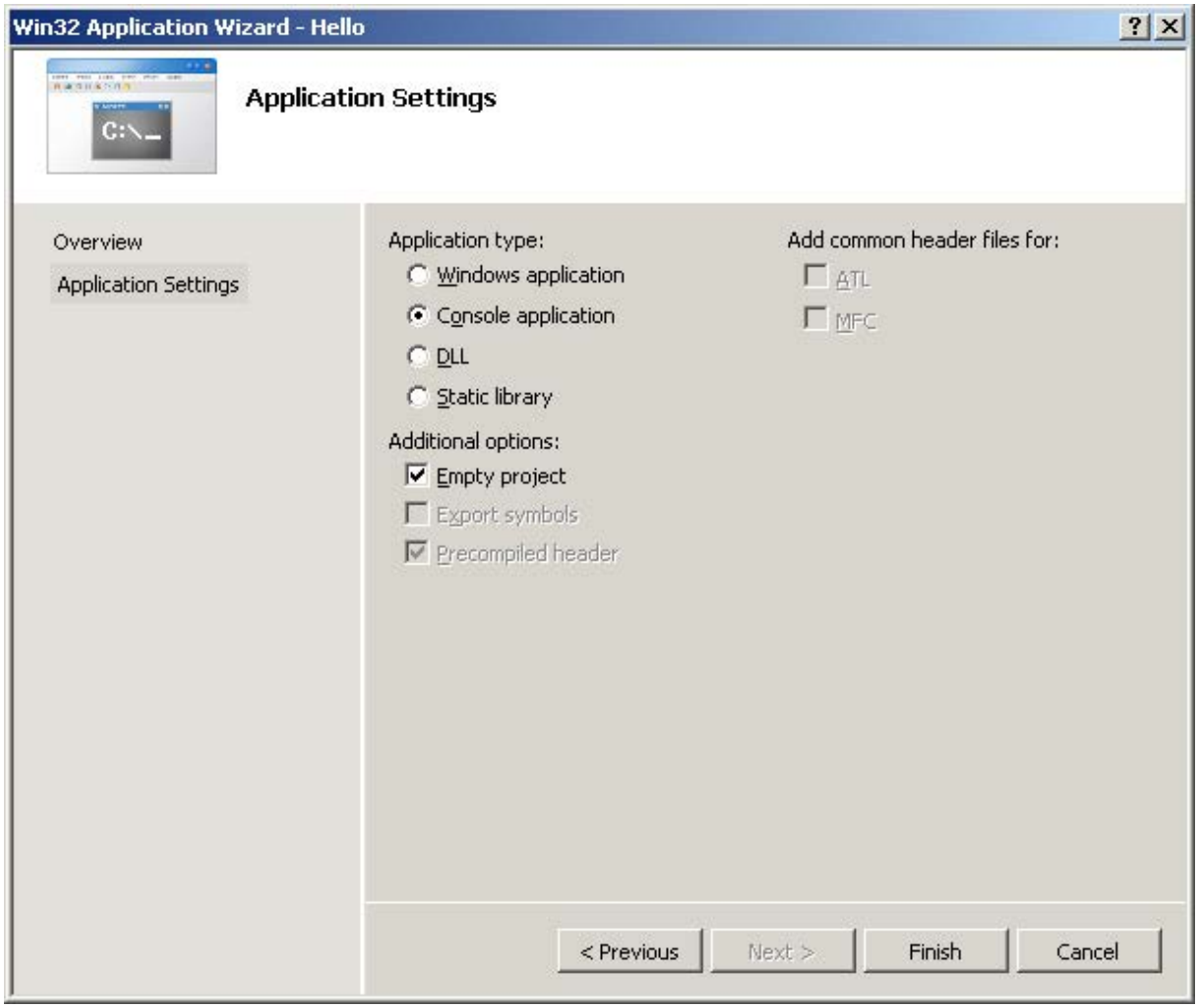


Select as shown by the window above.

1.4.3 Win32 – Win32 Console Application

Also specify the name of the program, for instance Hello, in the box after 'Name', and indicate a suitable folder with the 'Browse' button, where the programme is to be stored. A particular sub-folder will be created with that name.

Click 'OK'. A new window will be displayed:

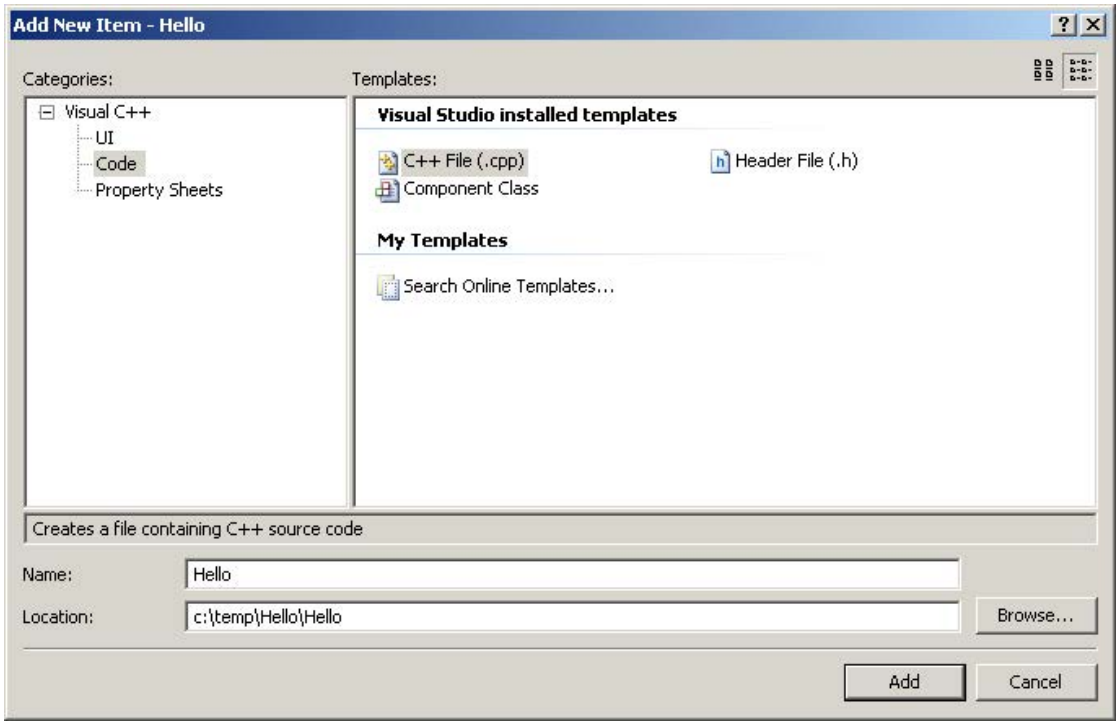


Click on 'Application Settings', check the box 'Empty project' and click on 'Finish'.

The project is now created but contains no code files yet. Add one by selecting from the menu:

1.4.4 Project – Add New Item

A window is displayed:

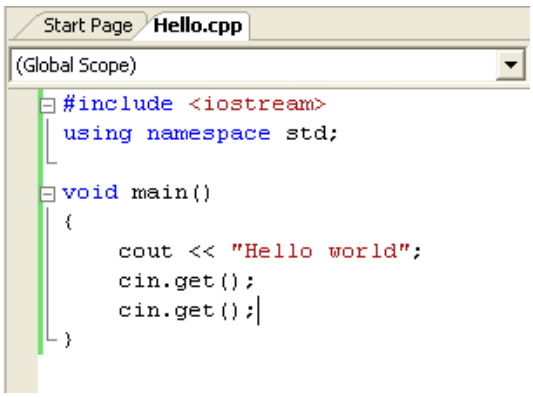


Select 'Code' in the left part and 'C++ File (cpp)' in the list of icons.

Enter a name of the file to be created in the box after 'Name'.

Click 'Add'.

A code window is displayed where you can enter code:

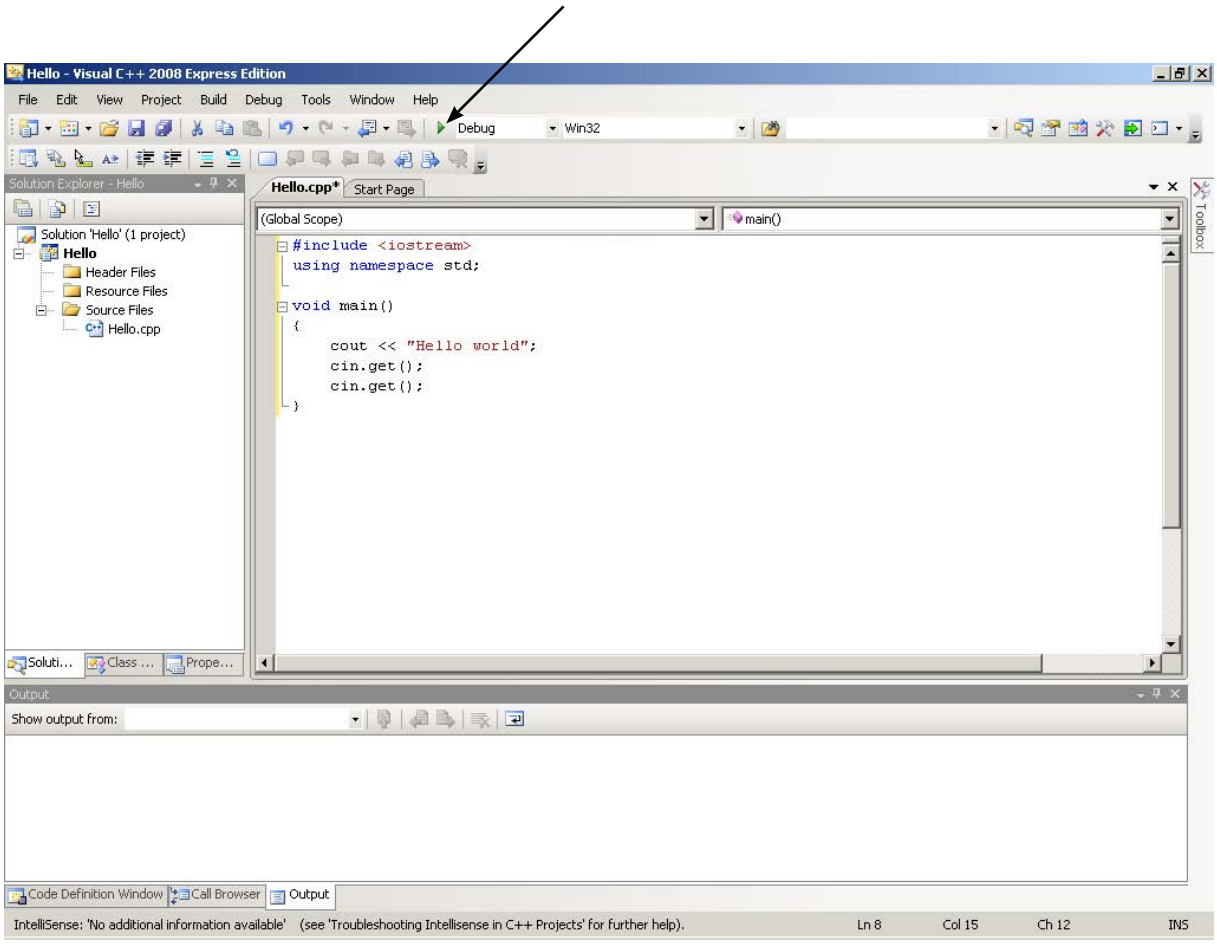


We will not explain all details in this program. That is done in the next chapter. The main thing is that you get started with the system and are able to write and compile code and run programs.

The changes necessary in the code compared to previous versions of Visual C++ are:

- Do not use .h in include statements, it should be:
  `#include <iostream>`
  However, in some include statements the .h should be kept
- Insert the statement
  `using namespace std;`
  which indicates to the system where the standard library is
- Insert the following statement as the last statement in your programs:
  `cin.get();`
  which makes the program stop and you will get an opportunity to view the console window with the displayed output. Sometimes two cin.get() –statements are needed:
  `cin.get();`
  `cin.get();`

Compile and run the program by clicking the arrow icon:



When you see the output in the console window you can stop the program by pressing Enter once or twice. It is the cin. get () –statements at the end of the program that waits for this Enter press.

We will create another program that asks the user for the unit price of a product and the quantity, and then calculates the total price.

Before you create a new program, you should close the project of the previous program, otherwise it will trouble the new program Select:

File - Close Solution

and answer 'Yes' to the question of closing all windows.

One thing you should remember is that each program takes 5-6 Mbyte disk space, due to that a lot of extra files and a Debug folder is created, which is necessary for using the debug function. These extra files and the Debug folder is recommended to be deleted after completion of a program, otherwise you might soon run out of disk space. The only thing to be saved is the cpp file where your source code is stored, which in our example has the name Hello.cpp.

Start a new program like in the Hello example and enter the following code:

```
#include <iostream.h>
void main()
{
    int iNo;
    double dblPrice, dblTotal;
```

```
    cout << "Enter price per unit ";
    cin >> dblPrice;
    cout << "Enter quantity ";
    cin >> iNo;
    dblTotal = dblPrice * iNo;
    cout << "The total price is " <<dblTotal<< endl;
}
```

We will not explain all details in this program. That is done in the next chapter. We will only touch the main steps of the program.

After the 'void main' line there are two lines where we declare some variables needed for storing of entered and calculated values.

The cout-line outputs a text to the screen. The subsequent cin-line makes the program stop and wait for the user to enter the price per unit and press Enter. This is repeated for the quantity of the product.

The dblTotal-line calculates the total price by multiplying the unit price by the quantity.

The last line outputs the total price.

When you compile the program it might happen that you have typed wrong. Then you will get a list of compilation errors in the window at the bottom of the screen. Double-click the first error to make Visual C++ indicate the erroneous line. Correct the error. Go on with the other errors and compile again. You might need to recompile a number of times.

Finally, when all errors have been corrected, run the program by clicking on the exclamation character button.

1.4.5 Where Is the Program

You can in Explorer examine the folder where you saved your program. During the compilation a sub-folder is automatically created called Debug. If you open it you will find the exe-file. You can now run the program by double-clicking the exe-file.

You can also put your program at the start menu. This is preferably done with the drag and drop method. Push the mouse button on the exe-file and hold it down, draw the mouse pointer to the Start button and release the mouse button. When you then click the Start button you will find your program on the Start menu.

2 Variables

2.1 Introduction

In this chapter you will learn what a variable is, how to declare a variable, i.e. tell the computer that there is a variable in the program, and how to assign values to variables. You will also learn how to perform simple mathematic calculations, how to read values from the keyboard, how to display information on the screen and control where on the screen the information will be displayed. We will also present a number of programming examples with JSP graphs.

2.2 Why Variables

A variable is used by the program to store a calculated or entered value. The program might need the value later, and must then be stored in the computer's working memory. Example:

Variable name	**Variable value**
dTaxpercent	0.25

Here we have selected the name 'dTaxpercent' to hold the value 0.25. You can in principle use any variable name, but it is recommended to use a name that corresponds to the use of the variable. When the variable name appears in a calculation the value will automatically be used, for example:

```
1500 * dTaxpercent
```

means that 1500 will be multiplied by 0.25.

2.3 Declaring Variables

The purpose of declaring a variable is to tell the program to allocate space in the working memory for the variable. The declaration:

```
int iNo;
```

tells that we have a variable with the name iNo and that is of integer type (int). You must always specify the data type to allocate the correct memory space. An integer might for instance require 4 bytes while a decimal value might require 16 bytes. How many bytes to allocate depends on the operating system. Different operating systems use different amounts of bytes for the different data types.

The variable name should tell something about the usage of the variable. Also, a standard used by many people is to allocate 1-3 leading characters to denote the data type (i for integer).

Note that each program statement is ended by a semicolon.

Below we declare a variable of decimal type:

```
double dUnitPrice;
```

`double` means decimal number with double precision. Compare to `float` which has single precision. Since `double` requires twice as many bytes, a `double` variable can of course store many more decimals, which might be wise in technical calculations which require high precision.

The most common data types:

short	integer	Usually 2 bytes
int	integer	Usually 4 bytes
float	decimal	Usually 4 bytes
double	decimal	Usually 8 bytes
bool	true or false	Usually 1 byte

You can declare several variables of the same type in one single statement:

```
double dUnitPrice, dTotal, dToBePaid;
```

The variables are separated by commas.

Note that C++ is case sensitive, i.e. a 'B' and 'b' are considered different characters. Therefore the variables:

```
dTobepaid
dToBePaid
```
are two different variables.

2.4 Assignment

Now we have explained how to declare variables, but how do the variables get their values? Look at the following code:

```
dTaxpercent = 0.25;

iNo = 5;

dUnitprice = 12;
```

Here the variable dTaxpercent gets the value 0.25, the variable iNo the value 5 and the variable dUnitprice the value 12. The equal character (=) is used as an assignment operator. Suppose that the next statement is:

```
dTotal = iNo * dUnitprice;
```

In this statement the variable iNo represents the value 5 and dUnitprice the value 12. The right part is first calculated as 5 * 12 = 60. This value is then assigned to the variable dTotal. Note that it is not the question of an equality in normal math like in the equation x = 60, where x has the value 60. In programming the equal sign means that something happens, namely that *the right part is first calculated, and then the variable to the left is assigned that value.*

C++ performs the math operations in correct order. In the statement:

```
dToBePaid = dTotal + dTotal * dTaxpercent;
```

the multiplication `dTotal * dTaxpercent` will first be performed, which makes 60 * 0.25 = 15. The value 15 will then be added to `dTotal` which makes 60 + 15 = 75. The value 75 will finally be assigned to the variable `dToBePaid`.

If C++ would perform the operations in the stated order, we would get the erroneous value 60 + 60, which makes 120, multiplied by 0.25, which makes 30.

If you need to perform an addition before a multiplication, you must use parentheses:

```
dToBePaid = dTotal * (1 + dTaxpercent);
```

Here the parenthesis is calculated first, which gives 1.25. This is then multiplied by 60, which gives the correct value 75.

Priority rules:

( )
* /
+ -

2.5 Initiating Variables

It is possible to initiate a variable, i.e. give it a start value, directly in the declaration:

```
double dTaxpercent = 0.25;
```

Here we declare the variable dTaxpercent and simultaneously specify it to get the value 0.25.

You can mix initations and pure declarations in the same program statement:

```
double dTaxpercent = 0.25, dTotal, dToBePaid;
```

In addition to assigning the dTaxpercent a value, we have also declared the variables dTotal and dToBePaid, which not yet have any values. In the statement:

```
int iNo = 5, iNox = 1, iNoy = 8, iSum;
```

we have initiated several variables and declared the variable iSum.

2.6 Constants

Sometimes a programmer wants to ensure that a variable does not change its value in the program. A variable can of course not change its value if you don't write code that changes its value. But when there are several programmers in the same project, or if a program is to be maintained by another programmer, it might be safe to declare a variable as a constant. Example:

const double dTaxpercent = 0.25;

The key word `const` ensures that the constant `dTaxpercent` does not change its value. Therefore, a statement like this is forbidden:

```
dTaxpercent = 0.26;
```

A constant must be initiated directly by the declaration, i.e. be given a value in the declaration statement. Consequently the following declaration is also forbidden:

```
const double dTaxpercent;
```

2.7 More about Assignment of Values

We have seen how a variable can be initiated in the declaration and how the variable can be assigned a value in other parts of the program. A variable can also get new values several times in the program.

A variable can furthermore be changed by originating from the current value of the variable. The following example shows how the variable `iNo` is decreased by 2:

```
iNo = iNo - 2;
```

As we have previously said the right part will first be calculated and then be assigned to the variable on the left side. Suppose that the variable `iNo` from the beginning has the value 5. The right part will then be 5-2 = 3. 3 is then assigned to the variable to the left, i.e. `iNo`. The effect of this statement is thus that `iNo` changes its value from 5 to 3, i.e. is decreased by 2.

A more compact way of coding giving the same result is:

```
iNo -= 2;
```

The operator -= means that the variable on the left side is decreased by the value on the right side. The operator += works in the same way. Example:

```
dPrice += 10;
```

Here the value of the variable `dPrice` is increased by 10.

The operator *= implies that the variable on the left side is multiplied by the value on the right side. In the following statement the variable `dDiscount` is multiplied by 1.10, i.e. it is increased by 10%:

```
dDiscount *= 1.10;
```

The operator /= works in the same way:

```
dNumber /= 2;
```

Here the variable `dNumber` is divided by 2.

In many situations the value of a variable should be increased by 1. We will give many examples of this in the following. Here are two variants of code how this can be made:

```
iNo = iNo + 1;
iNo += 1;
```

Still another way:

```
iNo++;
```

Here we use the operator ++ which increases the value of the variable by 1. It is from this operator that C++ has got its name.

Increasing a value by 1 is called incrementation. In the same way you can use the operator -- for decrementation, i.e. decrease a value by 1. Example:

iNo--;

2.8 The main function

So far we have described code details needed to be able to construct a program. Now we will step back a little and look at the entire program. Look at the following program skeleton:

```
void main()
{
    …
    … Various program statements
    …
}
```

To be able to run (execute) a program, a function called `main()` must exist. A function is a section of code that performs a specific task. Usually a program consists of several functions, but one of them must have the name `main()`, and the very execution is started in `main()`. In our first programs we only use one function in each program, and consequently it must be named `main()`. The parenthesis after main indicates that it is a function. Each function has a parenthesis after the function name, sometimes it is empty and sometimes it contains values or parameters.

A function is supposed to return a value, which could be the result of calculations or a signal that the function turned out successfully or failed. The return value can then be used by the program section calling the function. In our environment it is the operating system Windows that starts the function `main()`. Windows does not need any return value from our programs, and as a consequence we use the key word `void` in front of `main()`. `void` means that the function will not return any value.

We will discuss functions in a later chapter and will then go deeper into these details. So for now you don't need to bother about all details, only that you write `void main()` in the beginning of your programs.

All code belonging to a function must be enclosed by curly brackets, one left curly bracket ({) as the first character and one right (}) as the last. It is also good programming conventions to **indent** the code between the curly brackets as shown by the example above. The editor in Visual C++ will assist you with the indentation. When you have typed a left curly bracket and pressed Enter, the next and subsequent lines will be automatically indented.

2.9 Input and Output

A program mostly needs data to be entered from the keyboard and results to be displayed on the screen. We will write a little program which displays a request for a value from the user and then reads this value:

```
#include <iostream>
using namespace std;
void main()
{
    int iNo;
    cout << "Specify quantity: ";
    cin >> iNo;
}
```

We will soon talk about the first #include statement.

The first thing to be done in this program is that the integer variable iNo is declared. We will need it later in the program.

Then, the text

```
Specify quantity:
```

will be displayed on the screen. cout is an abbreviation of console out. With console we mean the keyboard and screen together. The text to be displayed on the screen (Specify quantity) must be surrounded by quote marks (" "). The characters << are called stream operator and indicates that each character is streamed to the console. We will return to streams when we work with files in a later chapter.

It can be hard to remember in which direction to write the stream operator. You can regard it as an arrow directed towards the console, i.e. the characters are streamed out to the console.

The next program statement (cin statement) implies that the program wants something from the console (cin = console in). The program stops and waits for the user to enter a value and press Enter. The value is stored in the variable iNo. The stream operator >> is here in the opposite direction and indicates that the entered value is streamed the other way, i.e. in to the program.

You may have noticed that the text ”Specify quantity: ” in the cout statement contains an extra space after the colon. This implies that also the space character is streamed to the screen. The visual effect of this is that, when the user enters the number 24, it is not shown close to the text “Specify quantity: “ but appears at a distance of one space character. Write the program and run it, and experiment with space characters and different texts.

2.9.1 #include

cout and cin are functions not automatically available. Therefore we must tell the compiler that these functions are defined in an external file called iostream. When you compile the program, the compiler does not understand the words cout and cin. It will then read iostream and insert the code for how cout and cin should be executed. This is the reason why you the statement

```
#include <iostream>
```

must be present first in your program.

Some external files use the extension .h, which is an abbreviation of **header file**. Another name of such a file is **include file**. We will use other header files later on when we need particular functions.

2.9.2 namespace

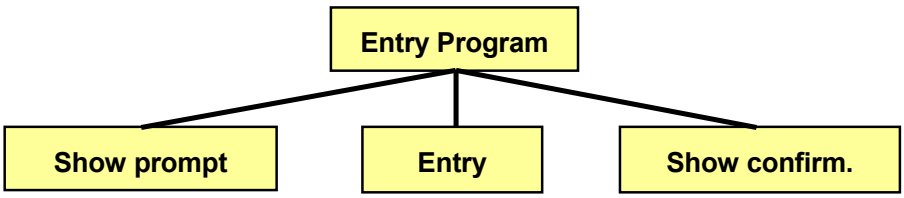
Some development tools, like for instance modern versions of Visual C++, store their include files and classes with code in namespaces. In object oriented programming (which is outside of this course) it is possible to store your own classes in different namespaces. To notify Visual C++ about the include files to be used, reside in the standard namespace, we use the statement

```
using namespace std;
```

2.10 An Entry Program

We will now write a program that prompts the user for a number and then shows a confirmation on the screen about which number the user wrote.

To practice algorithm creation we will first write a JSP graph that shows the program steps before writing the code:



The upper box shows the name of the program (Entry Program). The program contains three steps corresponding to the three boxes which are read from left to right. First we will show the user prompt. The user will then have the opportunity to enter a number. Last, we will show a confirmation about which number the user entered. The program will look like this:

```
#include <iostream>
using namespace std;
void main()
{
   int iNo;
   cout << "Specify quantity: ";
   cin >> iNo;
   cout << "You entered: ";
   cout << iNo;
}
```

First we must include iostream since we are going to read from and write to the console, and indicate the standard namespace to be used. In the main() function we declare the integer variable iNo and then show the text “Specify quantity: “. The cin statement implies that the program halts and waits for keyboard entry. When the user has entered a number and pressed Enter, the number is stored in the variable iNo. Then the program continues with displaying the text “You entered: “ followed by the value of the variable iNo. If the user for instance entered the value 24, the printed text will be “You entered: 24”. Write the program and run it. Experiment with different entries and other texts.

Note that, even if the program uses two cout statements for the printed confirmation, the result still is one printed line on the screen.

We will now expand the program so that the user can enter a quantity and a price:

```
#include <iostream.h>
void main()
{
   int iNo;
   double dPrice;
   cout << "Specify quantity: ";
   cin >> iNo;
   cout << "Specify unit price: ";
   cin >> dPrice;
   cout << "You entered the quantity " << iNo <<
            " and the price " << dPrice;
}
```

We have used an integer variable for the quantity and a double variable for the unit price, since the price could require decimals.

Note that, when entering a decimal value, you must use a decimal *point*, not decimal comma.

The last cout statement contains some news; you can combine several texts and variable values into one single statement, provided that you use the stream operator between every text and variable. We have split the statement on two lines, but you can write the whole statement on one single line. Blanks or line breaks in the code has no effect on the displayed result.

You could also combine the entry of quantity and price in the following way:

```
cout << "Specify quantity and unit price: ";
cin >> iNo >> dPrice;
```

First the text prompt is displayed to the user. When the program halts and waits for entry, you can enter a quantity and unit price with a space character in between, or press Enter. The first entered value is stored in the variable iNo and the second in dPrice.

If you press Enter after the first entered number, the program will still be waiting for yet another value. You must also enter the unit price before the program can continue the execution.

If you want a line break in the displayed result, you can use the keyword `endl`:

```
cout << "You entered the quantity " << iNo <<
         endl << "and the price " << dPrice;
```

The statement implies that the printed result will look like this:

```
You entered the quantity 5
and the price 12.45
```

2.11 Formatted Output

When programming for a DOS window, which we have done so far and will do during the rest of the course, there are very limited possibilities for a nice layout of printed information, especially if you compare to common Windows environment. C++ however offers a few functions for improved control of printed information. We will discuss the following:

- Fixed / floating decimal point. When printing large numbers, it might happen that C++ uses floating decimal point. For instance the number 9 560 000 000 (fixed) might be printed as 9.56E+9, which is interpreted as 9.56 x 10^9 (floating decimal point). C++ itself controls when to use either of these representations. Many times we want to control this ourselves.
- Number of decimals for numeric data.
- Number of screen positions allocated for data.

To be able to use these possibilities we will have to include the file iomanip.h.

2.11.1 Fixed Decimal Point

To instruct the program to output values with fixed decimal points the function setiosflags() is used. It must reside in a cout statement:

```
cout << setiosflags(ios::fixed);
```

The characters io in setiosflags represent input/output. A flag is a setting that can be switched on or off (0 or 1). The double colon :: is a class operator and is used to refer to a value defined in a certain class. “fixed” is such a value. We will not go further into classes in this course, but it does not hurt to have basic knowledge about classes.

To return to letting C++ decide when to use fixed or float format, use the following statement:

```
cout << resetiosflags(ios::fixed);
```

2.11.2 Number of Decimals

To instruct the program to use a specific number of decimals, use the following statement:

```
cout << setprecision(2);
```

which specifies two decimals to be used in subsequent outputs.

Many times the statements are combined:

```
cout << resetiosflags(ios::fixed) <<
    setprecision(2);
```

2.11.3 Number of Positions

An efficient way of having numeric data printed in columns with the 1 unit digit straight below each other, is to use the setw() function. 'w' represents width. Example:

```
cout << setw(8) << dAverage;
```

This statement allocates eight positions for the value of the variable dAverage, and the value is printed right-aligned within this space.

Remember that setw() only applies to the next value, which implies that it must be repeated for each value to be printed. Example:

```
cout << setw(8) << dValue1 << endl;
cout << setw(8) << dValue2 << endl;
cout << setw(8) << dAverage << endl;
```

Here the three variable values will be printed below each other right aligned within eight screen positions, which implies a nice column with the 1 unit digits right below each other.

Here is an example of how you can combine printed data with heading texts:

```
cout << "Number of units: " << setw(5) <<
   iNo << endl;
cout << setiosflags(ios::fixed) <<
    setprecision(2);
cout << "Price per unit:   " << setw(8) <<
   dPrice << endl;
cout << "Total price:      " << setw(8) <<
   dTotal;
```

This code will render an output like this:

```
Number of units:      10
```

```
Price per unit:       12.25
Total price:         122.50
```

We presume that the variable iNo is declared as int, while dPrice and dTotal are double variables. That requires that the number of positions for iNo is 3 less than for the others, which correspond to the decimal point and the two decimals.

Note that we in the cout statements have inserted blanks to make the heading texts be equal in length. That makes it easier to calculate the number of positions required in the setw() function.

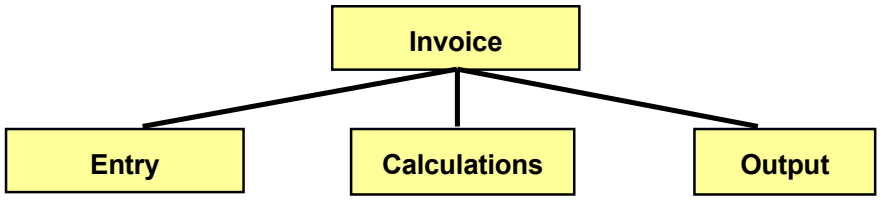
2.12 Invoice Program

We will now write a program where the user is prompted for quantity and unit price of a product, and the program should respond with an invoice receipt like this:

```
INVOICE
=======
Quantity:             30
Price per unit:       42.50
Total price:        1593.75
Tax:                 318.75
```

The program should thus calculate the total price and tax amount.

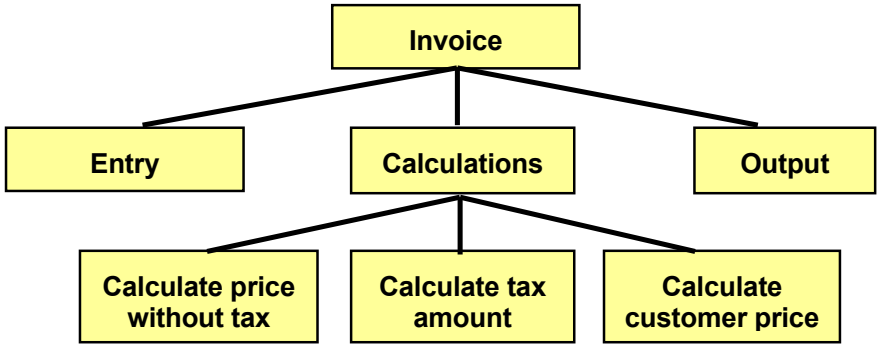
We start with a JSP graph:



First, the user will enter quantity and unit price of the product (the Entry box). Then we will calculate the total price and tax amount (the Calculations box). Last, the information will be printed (the Output box).

The first and last boxes are pretty uncomplicated, but the Calculations box requires that we go deeper before starting to code. Input data is quantity and unit price. We multiply these, which gives the price without tax. We then multiply this amount by the tax percent, which gives the tax amount. Finally we add these amounts to get the customer price.

The detailed JSP graph will then look:



Here is the code:

```
/*Invoice program
The file iomanip.h is needed to be able to
format the output on the screen*/

#include <iostream>
#include <iomanip>
void main()
{
  //Declarations
  int iNo;
  double dUnitPr, dPriceExTax, dCustPrice, dTax;
  const double dTaxPerc = 25.0;

  //Entry of quantity and unit price
  cout<< "Specify quantity and unit price: ";
  cin >> iNo >> dUnitPr;

  //Calculations. First the price without tax
  dPriceExTax = dUnitPr * iNo;
  //then the tax amount
```

```
    dTax = dPriceExTax * dTaxPerc / 100;
    //and finally the customer price
    dCustPrice = dPriceExTax + dTax;

    //Output
    cout << endl << "INVOICE";
    cout << endl << "=======" << endl;
    cout << "Quantity:       " <<  setw(5) << iNo << endl;
    cout << setprecision(2) << setiosflags(ios::fixed);
    cout << "Price per unit:" << setw(8) << dUnitPr << endl;
    cout << "Total price:    " << setw(8) << dCustPrice <<endl;
    cout << "Tax:             "  <<setw(8) << dTax << endl;
}
```

As you can see we have inserted comments in the code. Comments don't affect the final size of the program or performance. Therefore, use comments frequently, partly to explain the operations to others, and partly as a check list at maintenance of the program some years later.

Comments are surrounded by the characters /* and */. All text between these delimiters is treated as comments. You can have as many lines as you want between these delimiters. Another way is to begin a comment line with //, Then only that line will be treated as comment.

After the include statements the required variables are declared. The variable iNo is an integer, while all variables capable of storing an amount have been declared as double. The tax percent is declared as constant, since it should not be amended in the program.

Compare the code to the JSP graph and you will discover that we have followed the sequence of the boxes when coding.

2.13 Time Conversion Program

We will now examine a common technique, namely to use the modulus operator (%) to get the decimals of a division, to check whether a number is evenly dividable by another number, to get an interval for random numbers and much more.

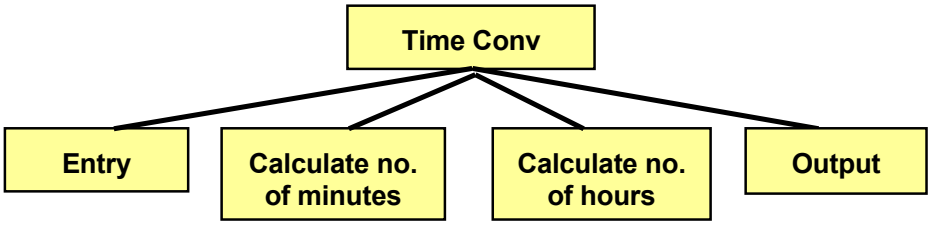
The modulus operator % gives the remainder at integer division

For instance, if you divide 6 by 3, the division is even and there will be no remainder. The remainder is zero, i.e. 6%3 equals 0. If you divide 7 by 3 the result is 2 with the remainder 1. 7%3 equals 1.

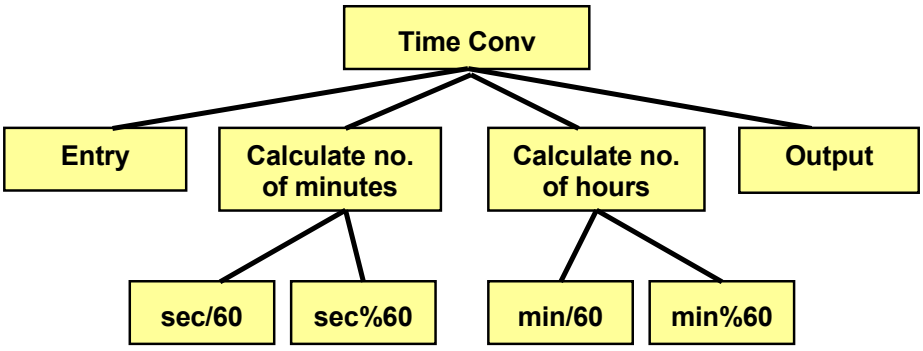
A peculiarity at division with the / operator is that, if both the numerator and denominator are of the int type, then also the quotient is an integer, i.e. the decimals will be discarded. For instance 7/3 equals 2.

We will use this in a program where the user enters a number of seconds, which the program converts to hours, minutes and seconds.

We start with a JSP graph:



Entry and output are pretty uncomplicated, while the calculation of no. of minutes and hours requires more details. Suppose the user enters 63 seconds. First we divide the entered number by 60, i.e. 63/60. Since both are integers, the quotient is an integer = 1. Then we use the modulus operator. 63%60 gives the remainder 3. We have calculated that 63 seconds makes 1 minute and 3 seconds. If the user enters a large number, we might have got so many minutes that hour calculation could be done. We would then originate from the number of minutes and in the same way divided by 60. The calculation is shown in the JSP graph:



Here is the code:

```
#include <iostream>
using namespace std;
void main()
{
     //Declarations
       int iNoOfSec, iSecLeft, iNoOfMin, iMinLeft, iNoOfHours;

     //Entry of no. of seconds to be converted
       cout << "Specify no. of seconds: ";
       cin >> iNoOfSec;

     //Number of entire minutes:
       iNoOfMin = iNoOfSec / 60;
     //Number of seconds left:
       iSecLeft = iNoOfSec % 60;

     // iNoOfMin is now the origin of the hours calculation:
```

```
    iNoOfHours = iNoOfMin / 60;
  //and no. of minutes left:
    iMinLeft = iNoOfMin % 60;

  //Output
    cout << "Number of hours   = " << iNoOfHours << endl;
    cout << "Number of minutes = " << iMinLeft << endl;
    cout << "Number of seconds = " << iSecLeft << endl;
}
```

Compile and run the program with different input.

2.14 Type Conversion

A problem with the division operator / is that it discards the decimals from the quotient if both the numerator and the denominator are of integer type. For instance if you have summed some integers and want to calculate the average by dividing by the number of integers. The precision will then be bad since the decimals will get lost:

```
int iNumerator = 7, iDenominator = 3;
cout << iNumerator / iDenominator;
```

This code section will give the output 2 and not 2.333 as expected.

Compare with the following code:

```
double dNumerator = 7;
int iDenominator = 3;
cout << dNumerator / iDenominator;
```

Here the numerator is of double type, and then the compiler understands that it should perform a normal division and include the decimals. The output will in this case be 2.333.

Many times you might have declared variables as int, but still want a division with the decimals kept. The solution is to do a type conversion (type cast) to double for the numerator before the division is executed:

```
int iNumerator = 7, iDenominator = 3;
cout << (double) iNumerator / iDenominator;
```

Type cast is performed by specifying the new data type within parentheses immediately before the variable. Here we have put double within parentheses before the variable numerator, which makes a decimal division to be performed with the result 2.333.

2.15 The Random Number Generator

In situations where an unpredictable result is required, the random number generator is used. Examples of such situations are game programs, pools, dice rolling etc.

The random number generator provides random numbers in the interval 0-32768. Why just 32768? It has to do with the binary storage of numbers. Two bytes can contain 65536 different numbers (2^{16}). Half of these are dedicated for negative numbers and half to positive = 32768.

The function `rand()` gives a number in the interval 0-32768. rand is an abbreviation of random. We use this number with the modulus operator: `rand()%6` which gives the remainder at integer division with 6, i.e. a number in the interval 0-5. The reason why it can't be greater is that, if for instance the remainder had been 7, then 6 could be divided once more and the remainder had been 1.

We now add 1:

```
rand()%6 + 1
```

which gives an integer in the interval 1-6, i.e. a random dice roll.

Each time you run a program using random numbers, it will start from the same "location", i.e. you will always get the same series of numbers. That is of course not acceptable. Therefore you must tell the generator to start at a random position, which is done with the function:

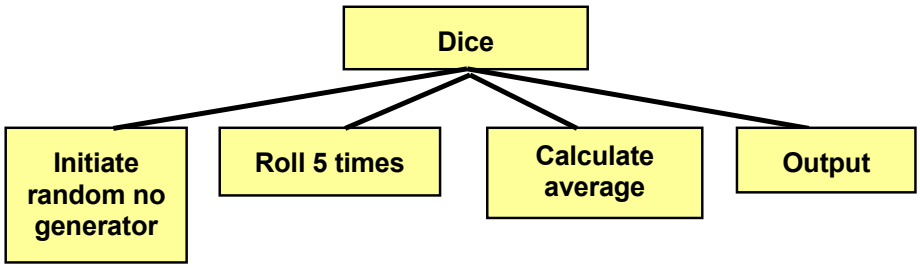
```
srand(time(0))
```

srand means "start random". The function uses the system clock (time) as origin for the calculation. The system clock contains the number of milliseconds since Jan 1st 1970 (different for different processors). We can't predict the millisecond to be used when the generator gets its starting point.

To make this work, you must include the header files stdlib.h och time.h. Note that these include files require '.h'.

2.16 Game Program

We will now write a program that uses the random number generator and rolls a dice 5 times. We will also calculate the average score. First we will create a JSP graph:



The code will be:

```
#include <iostream>
#include <iomanip>    //for formatting of output
#include <stdlib.h>     //for random generator
#include <time.h>       //for system clock
using namespace std;
void main()
{
   //Declarations
   int iRoll1, iRoll2, iRoll3, iRoll4, iRoll5;
   double dAverage;
   const int iNo = 5;
   //Initiate random number generator
   srand(time(0));
   //Roll 5 times
   iRoll1 = rand()%6+1;
   iRoll2 = rand()%6+1;
   iRoll3 = rand()%6+1;
   iRoll4 = rand()%6+1;
   iRoll5 = rand()%6+1;
   //Calculate average
   dAverage = (double)( iRoll1 + iRoll2 + iRoll3 + iRoll4
               + iRoll5) / iNo;
   //Output
   cout << "Number of rolls: " << iNo << endl;
   cout << setprecision(1) <<  setiosflags(ios::fixed);
   cout << "Average score: " << dAverage;
}
```

In the statement where we calculate the average we have first added the five rolls and then made a type cast to double before we divide with iNo to not loose decimals.

In the second last cout statement we have requested 1 decimal and fixed decimal point.

Run the programs several times. You will get different results each time. Also try to extend the program to also print the 5 rolls.

2.17 Summary

In this chapter we have taken our first stumbling steps in C++ programming. We have learnt what a variable is, how it is declared and assigned a value. We have also learnt to read and write data, and in connection to that, also present data in a more user-friendly way by means of formatted output. We have learnt how to include header files and we have written some example programs utilizing specialties like the modulus operator, type casting and random number generation.

But above all we have practiced how to build a solution to a problem by means of algorithms and JSP graphs. And this will be still more important when we enter into the subject of the next chapter, selections and loops.

2.18 Exercises

1. Originate from the Entry program and extend it so that the user can enter two numbers. Both numbers should then be printed on the screen by the program.
2. Write a program that prompts the user for two numbers and prints their sum.

3. Extend the previous excercise so that the program prints the sum, difference, product and quotent of the two numbers.
4. Write a program that prompts the user for 3 decimal numbers and then prints them on the screen with the decimal points right below each other.
5. Originate from the Invoice program and amend it so that:
   a) the user can enter a tax percent.
   b) a 10% discount is deducted before the tax calculation
   c) the last cout statement is organized in a more structured way
   d) The price with tax excluded is printed
   e) the discount amount is printed.

6. Write a program that prompts the user for the gas quantity and the gas price per litre. The program should then print a gas receipt like this:

```
          RECEIPT
          =======
Volume:         45.24 l
Litre price:     9.56 kr/l
-----------
To be paid:   432.49 kr
```

7. Write a program that prompts the user for current and previous electricity meter value in kWh and the price per kWh. The program should then calculate the total price of the current consumption.
8. Write a program that prompts the user for five integers. The program should then print:
   a) the sum of the integers
   b) average
   c) the sum of the squared numbers
   d) the sum of the cube of the numbers

9. Write a program that prompts the user for a number, divides it by 3 and prints the result in the form: "4 and remainder 2".
10. You want a program that converts a temperature in Celsius to Fahrenheit according to the formula:
    `tempF = 1.8 * tempC + 32`
    Create the conversation with the user in your own way.
11. Originate from the TimeConv program which converts a given number of seconds to hours, minutes and seconds. Change it so that the user can enter a number of minutes and the program responds with a number of hours and minutes.
12. Write a program that prompts the user for a number of days and responds with number of years, months and days. For simplicity, you can treat all months as having 30 days.
13. Write a program that prompts the user for the distance between two cities and in what speed you intend to drive. The program should print the time for the trip.
14. Change the previous program so that, instead of speed, it prompts for the time allocated for the trip. The program should respond with the speed required for the trip.
15. Change the previous program so that you can enter the speed and the time for the trip. The program should then respond with the driving distance.

16. Write a program that converts a given number of Swedish ”öre” to the number of 50-öre coins, crowns, 5-crowns, 10-crowns, 20-crown notes, 50-crown notes and 100-crown notes.
17. A farmer wants to build a wooden fence around a rectangular field. He measures the length and the width of the field and decides how high the fence should be. He also decides how wide the space between each board of the fence should be. Each board is 10 cm wide. Help him with a program that calculates the total length of all boards required to be bought.
18. Improve the previous program so that it also takes into account the amount of board waste (10%) at cutting the boards to suitable length.
19. Extend the previous program so that you also can enter a price per meter of the boards and have the total price printed.
20. Originate from the Game program which creates random rolls of a dice. Extend it so it also printes the individual rolls.
21. Modify the previous program so that it rolls two dice at a time and prints the score sum of the two rolls. Five such double-rolls should be made.
22. The ”lotto” game creates random numbers in the interval 1-35. A simple lotto game contains seven such numbers. Create a program that provides a simple lotto game set of numbers. Don’t pay attention to repetition of a single number.
23. So far we have used the random number generator to produce integers. Figure out how we could get random numbers with one decimal. Then write a program that produces five temperatures in the interval 18.0 - 23.5.

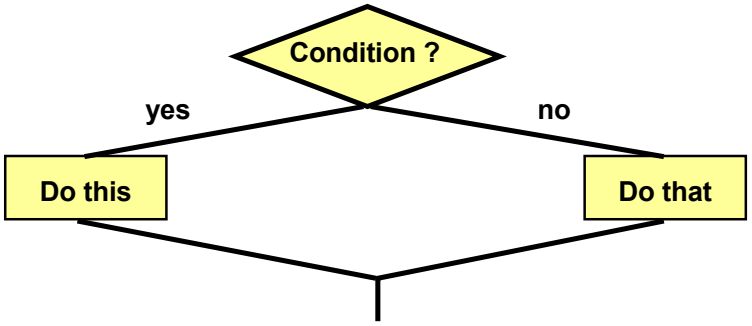
3 Selections and Loops

3.1 Introduction

In this chapter you will learn to incorporate intelligence into your programs, i.e. the program can do different things depending on different conditions (selections). You will also learn how to repeat certain tasks a specific number of times or until a specific condition is fulfilled (iteration, loop). We will introduce new symbols in our JSP graphs to illustrate selections and loops.

3.2 Selection

A selection situation can be illustrated by the following figure:



If the condition is fulfilled (yes option) the program will do one thing, else (no option) another thing.

3.3 if statement

The selection situation is in C++ coded according to the following syntax:

```
if (condition)
   statement1;
else
   statement2;
```

The keyword `if` introduces the if statement. The condition is put within parentheses. If the condition is true statement1 will be performed, otherwise statement2. Here is a code example:

```
if (a>b)
   greatest = a;
else
   greatest = b;
```

The values of two variables are compared. If a is greater than b, the variable greatest will get a's value. Otherwise, i.e. if b is greater than or equal to a, greatest will get b's value. The result from this code section is that the variable greatest will contain the greatest of a and b.

Sometimes you might want to perform more than one statement for an option. Then you must surround the statements with curly brackets:

```
if (condition)
{
   statements
   …
}
else
{
   statements
   …
}
```

If the condition is true all statements in the first code section will be executed, otherwise all statements in the second code section will be executed. Example:

```
if (a>b)
{
   greatest = a;
   cout << "a is greatest";
}
else
{
   greatest = b;
   cout << "b is greatest";
}
```

If a is greater than b, the variable greatest will get a's value and the text "a is greatest" will be printed. Otherwise the variable greatest will get b's value and the text "b is greatest" will be printed.

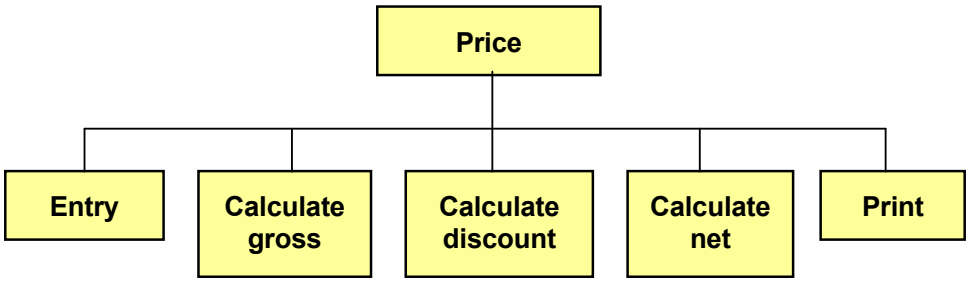
Sometimes you don't want to do anything at all in the else case. Then the else section is simply omitted like in the following example:

```
if (sum>1000)
{
   dDiscPercent = 20;
   cout << "You will get 20 % discount";
}
```

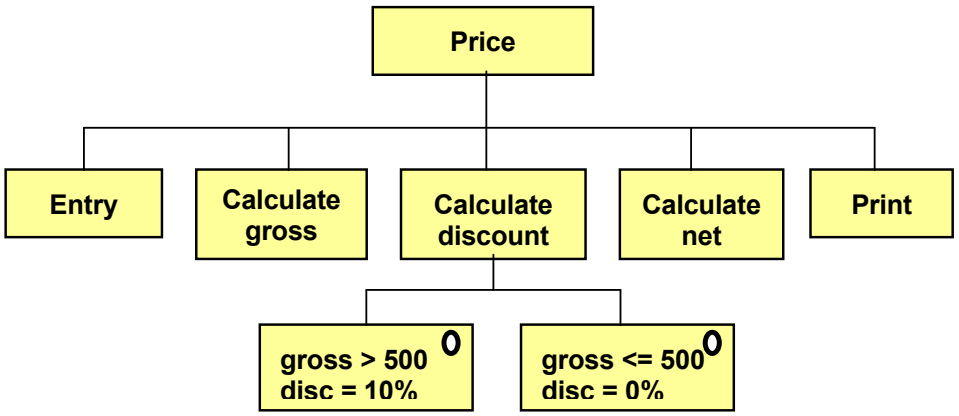
If the variable sum is greater than 1000 the variable dDiscPercent will get the value 20 and the text "You will get 20% discount" will be printed. Otherwise nothing will be executed and the program goes on with the statements after the last curly bracket.

3.4 Price Calculation Program

We will now create a program that calculates the total price of a product. The user is supposed to enter quantity and price per unit of the product. If the total exceeds 500:- you will get 10 % discount, otherwise 0 %. We start with a JSP graph:



All boxes except "Calculate discount" are rather simple to code. "Calculate discount" requires a closer examination. It has a condition included which says that the discount is different depending on whether gross is less or greater than 500. We'll break down that box:



A conditional situation in JSP is identified by a ring in the upper right corner of the box. That implies that only one of the boxes will be executed. Here is the code:

```
#include <iostream.h>
void main()
{
   const double dLimit = 500;
   int iNo;
   double dUnitPrice, dGross, dNet, dDisc;
   cout << "Specify quantity and unit price";
   cin >> iNo >> dUnitPrice;
   dGross = iNo * dUnitPrice;
   if (dGross > dLimit)
      dDisc = 10;
      else
      dDisc = 0;
   dNet = (100- dDisc) * dGross / 100;
   cout << "Total price: " << dNet;
}
```

The declaration shows a constant dLimit, which later is used to check the gross value. The variable iNo is used to store the entered quantity and dUnitPrice is used for the entered unit price.

It is common among programmers to use one or a few characters in the beginning of the variable name to signify the data type of the variable. The variable iNo has first character I (integer), and the variable dUnitPrice has d (double).

After data entry the gross is calculated by multiplying the two entered values (quantity * unit price). That value is stored in the variable dGross.

The if statement then checks the value of dGross. If greater than dLimit (i.e. 500) the variable dDisc will get the value 10, otherwise 0. dDisc contains the discount percent to be applied.

The net is then calculated by subtracting the discount percent from 100, which then is multiplied by dGross and divided by 100 (to compensate for the percent value).

Finally the total price is printed.

3.5 Comparison Operators

In the if statements in previous example codes we have so far only used the comparison operator > (greater than). Here is a list of all comparison operators:

```
<   less than
>   greater than
<=  less than or equal to
```

```
>=  greater than or equal to
==  equal to
!=  not equal to
```

3.6 Even or Odd

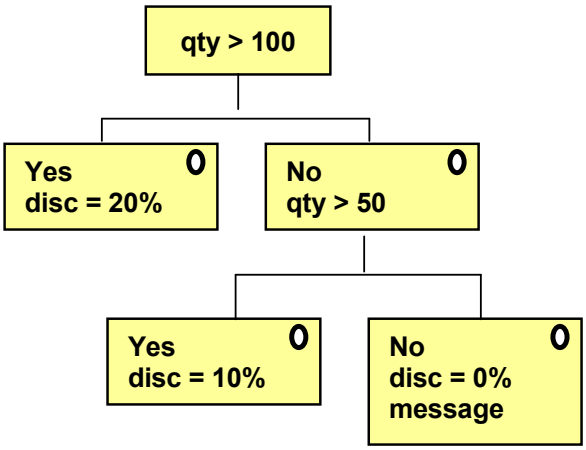
In some situations you will need to check whether a number is evenly dividable by another number. Then the modulus operator % is used. Below are some code examples of how to check whether a number is odd or even, i.e. evenly dividable by 2.

```
//If iNo is even, the remainder of the integer
//division by 2 equals 0:
if (iNo%2 == 0)
   cout >> "The number is even";

//If the remainder of the integer division by 2
//does not equal 0, the number is not dividable
//by 2:
if (iNo%2 != 0)
   cout >> "The number is odd";

//Short way of codeing. An expression not equal
//to 0 is regarded as false, otherwise true.
//If iNo is odd, iNo%2 gives a non zero value:
if (iNo%2)
   cout >> "The number is odd";
```

3.7 else if

We will now study an example of a more complicated situation. Suppose the following conditions prevail:

If a customer buys more than 100 pieces, he will get 20% discount. Otherwise if the quantity exceeds 50, i.e. lies in the interval 50-100, he will get 10%. Otherwise, i.e. if the quantity is below 50, no discount is given. The situation is shown by the following JSP graph:



The code for this will be:

```
if (iNo>100)
   dDisc = 20;
else if (iNo>50)
   dDisc = 10;
else
{
   dDisc = 0;
   cout << "No discount";
}
```

Here we use the keyword else if.

You can use any number of else if-s to cover many conditional cases.

3.8 and (&&), or (||)

The situation with different discount percentages for different quantity intervals can be solved in another way, namely by combining two conditions. In common English it can be expressed like this:

If the quantity is less than 100 and the quantity is greater than 50, the customer will get 10% discount.

Here we combine two conditions:
- If the quantity is less than 100

and
- and the quantity is greater than 50

The combination of the conditions means that the quantity lies in the interval 50-100. Both conditions must be fulfilled in order to get 10%. The conditions are combined with "and" which is a logical operator. It is written && in C++. The code will then be:

```
if (iNo<100 && iNo>50)
   dDisc = 10;
```

Suppose the situation is this:

If the quantity is greater than 100 or the total order sum is greater than 1000, the customer will get 20% discount.

Here we combine the conditions:
- If the quantity is greater than 100
eller
- or the total order sum is greater than 1000

In both cases the customer has bought so much that he will get 20% discount. One of the conditions is sufficient to get that discount. The conditions are combined with the logic operator "or", which is written || in C++. The code for this situation will be:

```
if (iNo>100 || dSum>1000)
   dDisc = 20;
```

3.9 Conditional Input

In many situations you cannot predict what a user is going to enter. It might happen that the user enters characters when the program expects integers, or that he does not enter anything at all but just press Enter. Then you can use conditional input:

```
if (cin >> iNo)
   ...
```

To understand how this code works you must know that cin is a function that returns a value. If reading of the value to the variable iNo succeeded, the return value from cin is true, otherwise false. Here is a code section that shows how it can be used:

```
cout << "Specify quantity: ";
if (cin >> iNo)
   dTotal = iNo * dUnitPrice;
else
{
   cout << "Input error";
   cin.clear();
   cin.get();
}
```

First we prompt the user for a quantity. The the program halts (cin) and waits for a vlue. If data entry turned out well, the whole condition is true, and the total price is calculated.

If the data entry failed, i.e. if the user entered letters or just pressed Enter, the condition is false and the statements after else are executed. The user will get a message about input error, the keayboard buffer is cleared (cin.clear()) and the next character in the queue is read (cin.get()). This clean-up procedure must be performed to be able to enter new values to the program.

3.10 The switch statement

In addition to the if statement there is another tool that allows you to perform different tasks depending on the circumstances. The tool is called switch statement and is best accommodated to the situation when checking a value against several alternatives. A good example is a menu where the user enters a menu option (1, 2, 3 ... or A, B, C ...) to make the program do different things depending on the user's choice.

The switch statement has the following syntax:

```
switch (opt)
{
   case 'A':
      //statements
      break;
   case 'B':
      //statements
      break;
   …
   default:
      cout << "Wrong choice";
      break;
}
```

First comes the keyword switch. Within parenthesis after switch there is the variable to be checked. It is checked against the values after the different case keywords. If for instance opt has the value 'A', i.e. opt is a char variable in the example above, then the statements below case 'A' are executed. Note that the keyword break must be found at the end of each case block. If break is omitted, the program will continue into the next case block. Note also that there must be a colon (:) after each case line. The default block takes care of all other options, i.e. if the variable opt does not contain any of the values 'A', 'B' etc. then the statements in the default block will be executed. The entire switch block should be surrounded by curly brackets.

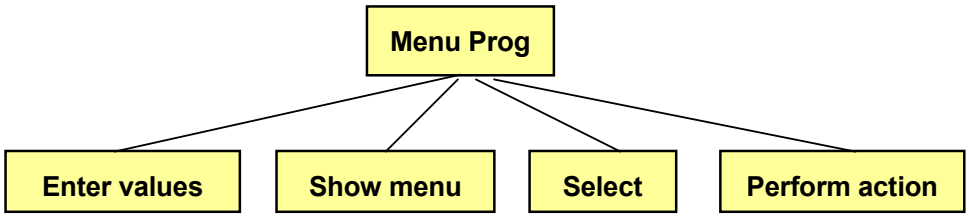
3.11 Menu Program

We will now write a menu program that illustrates how the switch statement can be used. First, the user is prompted for two numbers, and then a menu is displayed where the user can select whether to view the greatest, the least, or the average of the two numbers. The screen will look like this:

```
Enter 2 numbers:  7 5
1.  Greatest
2.  Least
3.  Average
Select:
```

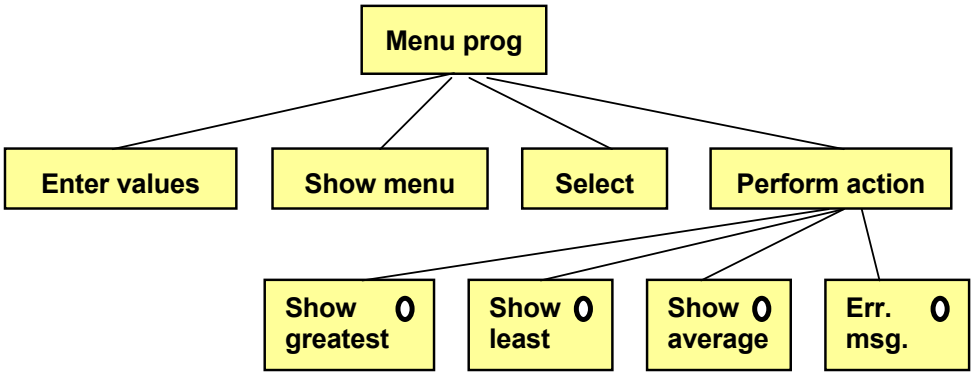
First, the user has entered the numbers 7 and 5. Then a menu is displayed where the user has to select 1, 2 or 3, depending on what he wants to view.

We will first draw a JSP graph that explains the process:



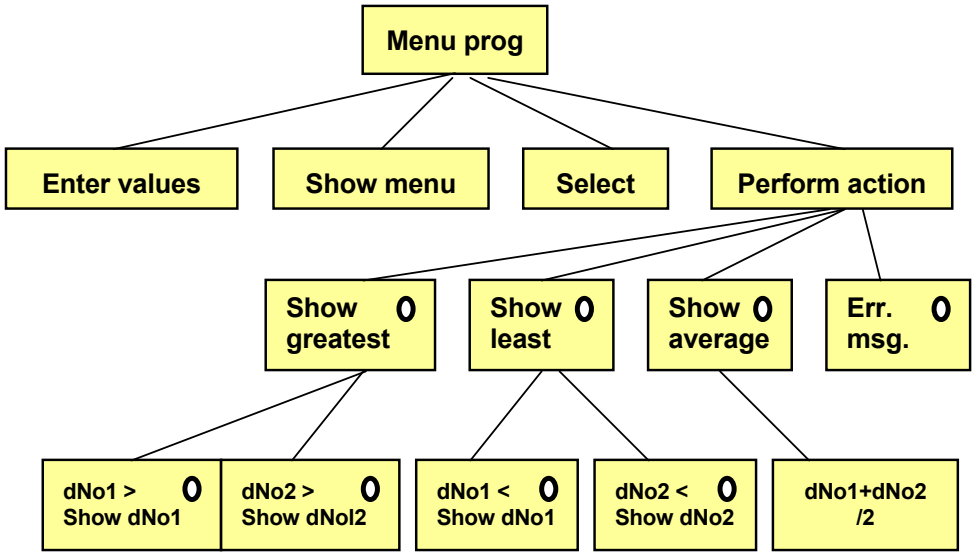
First, the user enters two numbers. Then the menu is displayed on the screen and the user selects an option. Finally the requested action is performed.

The requested action can be one of four options, so we break down the box ”Perform action”:



Since we have a selection situation where only one of the options should be performed, we indicate this with a circle in the upper right corner in each selection box.

The four options contain some logic, so we break down the JSP graph further:



In the "Show greatest" case we perform a check: if iNo1 is the greatest, we print it, otherwise we print iNo2. The "Show least" is analoguous. In the "Show average" case we add the two numbers and divide by 2.

The code will be this:

```
#include <stdlib.h>
#include <iostream.h>
void main()
{
   int iOpt;
   double dNo1, dNo2;
   cout << "Enter 2 numbers: ";
   cin >> dNo1 >> dNo2;
   system("cls");
   cout << "1.  Greatest" << endl;
   cout << "2.  Least" << endl;
   cout << "3.  Average" << endl;
   cout << endl << "Select: ";
   cin >> iOpt;
   switch (iOpt)
   {
      case 1:
         if (dNo1>dNo2)
```

```
          cout << dNo1;
          else
          cout << dNo2;
        cout << " is the greatest";
        break;
      case 2:
        if (dNo1<dNo2)
          cout << dNo1;
          else
          cout << dNo2;
        cout << " is the least";
        break;
      case 3:
        cout << "The average is " << (dNo1+dNo2)/2;
        break;
      default:
        cout << "Wrong choice";
        break;
   }
}
```

The header file stdlib.h is needed to be able to clean the screen with system("cls"), which is done after the user has entered the two values. Then we print the menu on the screen and the user enters his choice (1, 2 eller 3) to the variable iOpt.

The switch statement will then check the variable iOpt. If it is 1, the statements after "case 1" are executed. There we check which of the two numbers are the greatest and print it. In the same way the least number is printed under "case 2". In case of 3, the average is calculated and printed. If the user has entered anything else, the default statements are executed.

3.12 Loops

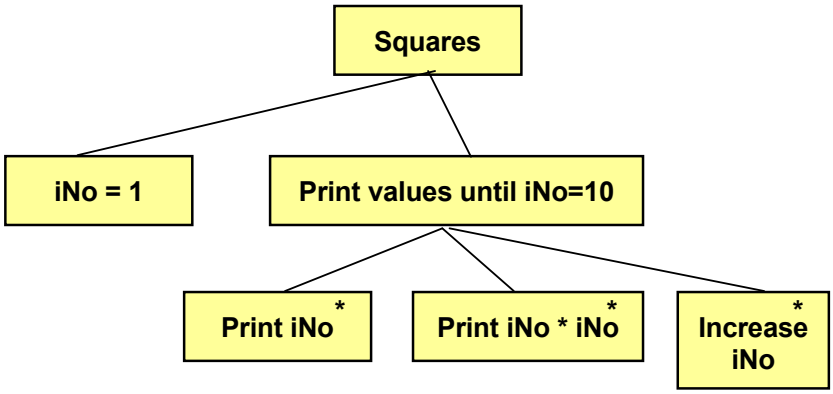
We will now continue with another powerful tool within programming, that can make the program perform a series of operations a specific number of times. Sometimes, the number of times, or the number of iterations, decided from start, sometimes it depends on the circumstances. We begin with an example:

We want to print a list of the numbers 1-10 and their squares:

1 1
2 4
3 9
etc.

We have a variable, iNo, which first has the value 1. We print it and the square of it. Then we increase the value of iNo by 1 and repeat the process, i.e. we print iNo and the square of iNo. Then we increase iNo again etc. This goes on until iNo = 10.

Thus, we have a series of operations (print iNo, print the square of iNo) which is repeated 10 times. A repetition is called a loop. It is illustrated by the following JSP graph:



First the variable iNo is set = 1. Then a loop "Print values until iNo=10" is started. The fact that it is a loop is shown by the subordinate boxes having an asterix in the upper right corner.

The loop consists of three boxes, which in turn print the value of iNo, the value of iNo * iNo (i.e. the square of iNo), and increase the value of iNo by 1. The loop goes on until iNo has reached the value 10.

3.13 The while Loop

Here is a code section that performs the task:

```
iNo = 1;
while (iNo <= 10)
{
   cout << iNo << " " << iNo * iNo << endl;
   iNo++;
}
```

First the variable iNo gets the value 1. Then the loop follows starting with the keyword while, followed by a condition within parenthesis. The operations to be repeated are given within the curly brackets immediately after the while condition.

The while line can be read: ”As long as iNo is less than or equal to 10”. For each turn of the loop iNo and iNo*iNo are printed on the screen, separated by a space and followed by a line break. At the end of the loop iNo is increased by 1.

3.14 The for Loop

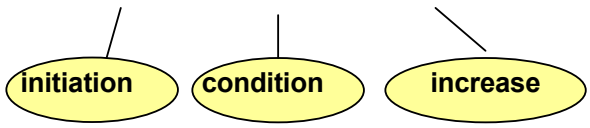
The task was solved by the while loop above. The for loop is another type of loop:

```
for (iNo=1; iNo <= 10; iNo++)
{
   cout << iNo << " " << iNo * iNo << endl;
}
```

This loop does exactly the same thing, namely prints the numbers 1-10 and their squares. The code block however contains only one statement. The actual increase of the iNo value is managed by the parenthesis after the keyword for.

The parenthesis contains three parts, separated by semicolons:

```
for (iNo=1; iNo <= 10; iNo++)
```



The initiation part sets a start value of a variable, often called loop variable, since it controls when to interrupt the loop. The condition part is checked for each turn of the loop. When the condition is false, the loop is interrupted. The increase part changes the value of some variable; mostly it is the loop variable that is increased by 1.

However, you don’t have to start with 1 or increase by 1 for each turn of the loop. The following code example shows how the variable iNo from start is set to 2. The increase part will add 2 for each turn of the loop:

```
for (iNo=2; iNo <= 10; iNo=iNo+2)
{
   cout << iNo << " " << iNo * iNo << endl;
}
```

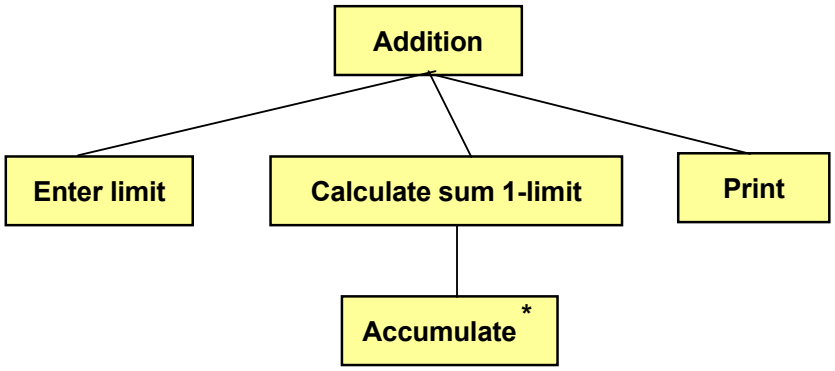
3.14.1 while or for

When should you use the while loop and when the for loop? Many times you can solve the problem with both loop types, and many times it is a question about personal preference. In general, however, if you can predict the number of turns of the loop, the for loop is the best one. If there is an unpredictable situation, e.g. if the loop goes on until the user enters a specific value, or that the random number generator provides a specific number, use the while loop. We will use both alternatives.

3.15 Addition Program

We will create a program that adds the integers 1 + 2 + 3 + 4 + … up to the limit specified by the user. The user should first enter the requested limit. Then we will use a loop that goes from 1 to that limit and sums the numbers. We will then need a variable, which is a kind of accumulater, which stores the sum.

We begin with a JSP graph:



First the user is prompted for a limit. The following loop goes from 1 to limit with the loop variable i. For each turn of the loop we add the value of i to the sum, i.e. we accumulate the numbers. Finally we print the accumulated sum. Note that the operation to be repeated (Accumulate) in the loop is indicated by an asterix in the JSP graph.

```
#include <iostream.h>
void main()
{
   int i, iLimit, iSum = 0;
     cout << "Enter limit: ";
     cin >> iLimit;
     for (i=1; i<=iLimit; i++)
           iSum += i;
     cout << "The sum = " << iSum << endl;
}
```

First, a number of variables are declared. The variable i is used as loop variable, iLimit is used for storage of the user specified limit, and iSum the accumulated sum. Note that, since iSum is increased by a value all the time, it must have a

start value. A declared variable does not automatically get the value 0 or any other value. That is why we must initiate it with 0. The other variables will get fix values during the execution of the program, so they need not be initiated.

The user will then enter a value to be stored in the variable iLimit. The loop then sets a start value = 1 to the loop variable i. The for-condition is that i is not allowed to exceed iLimit. For each turn of the loop the loop variable is increased by 1. That means that the value stated by the user controls the number of turns of the loop.

The repetition code block of the loop contains only one statement. Therefore we don't need any curly brackets surrounding the loop. If, however, the repetition code block contains several statements, they must be surrounded by curly brackets. Compare the if statement, which works in the same way.

The repetition code block contains this statement:

```
iSum += i;
```

which implies that the variable iSum is increased by i for each turn of the loop. This means that the variable iSum will contain the sum 1 + 2 + 3 + …

Finally the value of iSum is printed.

An optional way of writing the for line:

```
for (int i=1; i<=iLimit; i++)
```

Here we declare the variable i inside the for statement. The variable should then not be declared earlier in the program.

Still another way of coding:

```
for (int i=1; i<=iLimit; iSum+=i++);
```

Here the increase part of the for statement contains the code `iSum+=i++.` Here two things happen, namely that the variable iSum is increased by the value of i, and then i is increased by 1 (i++). This means that we don't need any repetition code block, so we put a semicolon directly after the parenthesis. Consequently the loop consists of one single line.

3.16 Double Loop

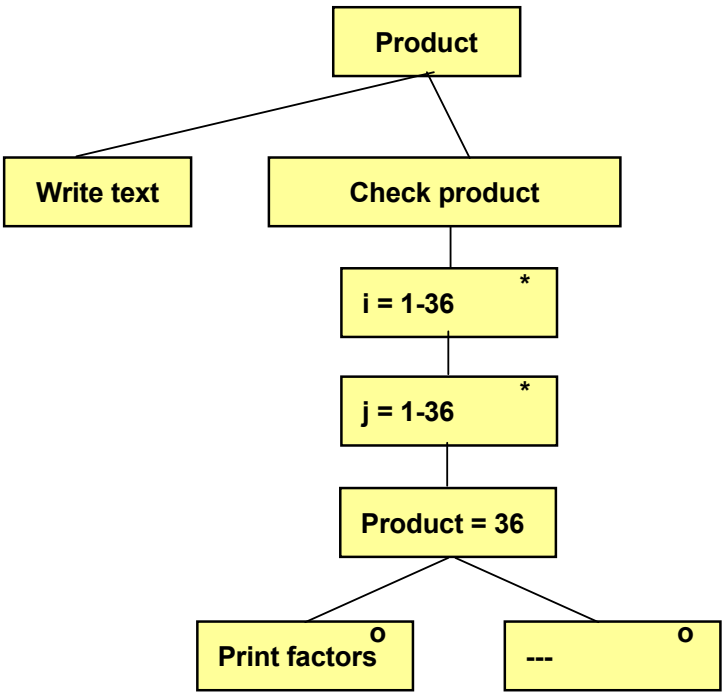
We will now how to use a double loop, i.e. a loop inside another loop. The inner loop will then do all its loop turns for each turn of the outer loop. Here is an example.

We will write a program that figures out all combinations of two integers whose product is 36:

1 x 36
2 x 18
3 x 12
etc.

We let the outer loop control the first factor, which runs from 1 to 36. For each value of the first factor we will go through the values 1-36 for the second factor and check if the product equals 36. If so, the factors are printed.

First we give a JSP graph:



Under "Check product" we have an outer loop with the loop variable i and an inner loop with loop variable j. Inside the loop we check if the product of i and j makes 36. If so, i and j are printed.

Here's the code:

```
#include <iostream.h>
void main()
{
   int i, j;
     cout << "Calculation of produdt" << endl;
     for (i=1; i<=36; i++)
     {
        for (j=1; j<=36; j++)
        {
          if (i*j == 36)
          cout << i << " and " << j << endl;
        }
     }
}
```

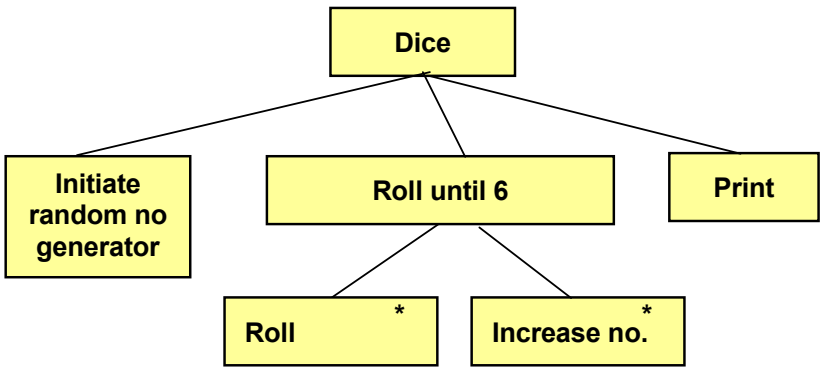
In the double for loop the variable i gets the value 1. The inner loop starts and lets j run through the values 1-36. For each value of j we check if i*j makes 36. If so, we print the values of i and j. When the inner loop has finished, the next turn of the outer loop will start where i is set =2, and the inner loop starts once again and lets j run from 1 to 36.

3.17 Roll Dice

So far we have mainly used the for loop. We will now look at a few situations where the while loop is preferred. We will write a program that rolls a dice until we get 6. Then the number of rolls is printed.

Here we cannot predict how long the loop will run. That depends on the numbers being generated. Therefore, the while loop is perfect.

Let us first create a JSP graph:



We begin with initiating the random number generator to a randomly selected start position. Then the loop is repeated until we get 6. For each turn of the loop we roll the dice once more and increase the number of rolls by 1. When 6 has been achieved, the loop is terminated and the number of rolls is printed. Here's the code:

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
void main()
{
   int iRoll=0, iNoOfRolls=0;
   srand(time(0));
   while (iRoll != 6)
   {
      iRoll = rand()%6+1;
      iNoOfRolls++;
   }
   cout << iNoOfRolls;
}
```

The header file stdlib.h is needed for the random number functions, and time.h is needed for the function time(0) at initiation of the generator.

The variable iRoll is used to store each roll. The reason for initiating it with 0 at the declaration is that it must hold a value when the while loop starts. The value must be something else than 6, otherwise the loop will not start. Any other value will do.

The variable iNoOfRolls is used to count the number of rolls. It must be initiated to 0, since it is increased by 1 all the time.

The function srand() initiates the generator to a randomly selected start position.

The while loop contains the condition that iRoll must not be 6. As long as no 6 is achieved the loop runs one more turn. For each turn we make another roll stored in iRoll, and the variable iNoOfRolls is increased by 1.

When we get 6, the while condition is false and the loop is terminated. The variable iNoOfRolls then contains the number of rolls, which is printed.

A variant of the program looks like this:

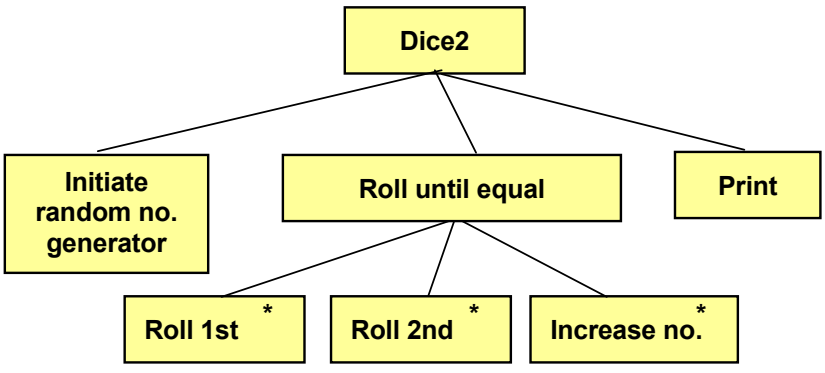
```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
void main()
{
   int iRoll, iNoOfRolls=0;
   srand(time(0));
   do
   {
      iRoll = rand()%6+1;
      iNoOfRolls++;
   } while (iRoll != 6);
   cout << iNoOfRolls;
}
```

The big difference is that the loop has its condition *after* the loop body instead of before. The effect of this is that at least one turn of the loop is executed before the condition is tested. This also means that the variable iRoll needs not be initialized to 0. It will anyway get a new value during the first turn of the loop.

The keyword 'do' is before the loop body, and 'while' followed by the condition right after the ending curly bracket. You must have a semicolon immediately after the condition.

3.18 Two Dice Roll

We will now write a program which repeatedly rolls two dice and checks if the two rolls are equal. When two equal rolls have been achieved, the process is terminated and the number of "double" rolls is printed. We start with a JSP graph:



When having initiated the random number generator to a random start position, the loop begins. For each turn, we roll the 1st and then the 2nd dice, and then increase the counter by 1. When the two rolls are equal the loop is terminated and the number of "double" rolls is printed. Here is the code:

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
void main()
{
   int iRoll1, iRoll2, iCounter=0;
       srand(time(0));
       do
       {
             iRoll1 = rand()%6+1;
             iRoll2 = rand()%6+1;
             iCounter++;
       } while (iRoll1 != iRoll2);
       cout << "The rolls were " << iRoll1 << endl;
       cout << "Number of attempts = " << iCounter << endl;
}
```

The program is similar to the previous with the difference that here we have two variables which store the rolls. Inside the loop iRoll1 and iRoll2 get their values and the number of "double" rolls is increased by 1.

Since the while condition comes after the loop body, at least one loop turn will be executed. The condition is that iRoll1 is not equal to iRoll2. If they are equal the loop is terminated and the dice score and the number of rolls are printed.

3.19 Breaking Entry with Ctrl-Z

We will now use the while loop condition to contain a user input with cin. The program will prompt the user for repeated entry of numbers. The entered numbers are summed. When the user presses Ctrl-Z the entry of numbers is interrupted and their average is printed. Here is the code:

```
#include <iostream.h>
void main()
{
   int iSum=0, i=0, iNo;
       cout << "Enter a number: ";
       while (cin >> iNo)
       {
             iSum += iNo;
             i++;
             cout << "Enter one more number: ";
```

```
    }
    cout << "Average = " << (double)iSum/i << endl;
}
```

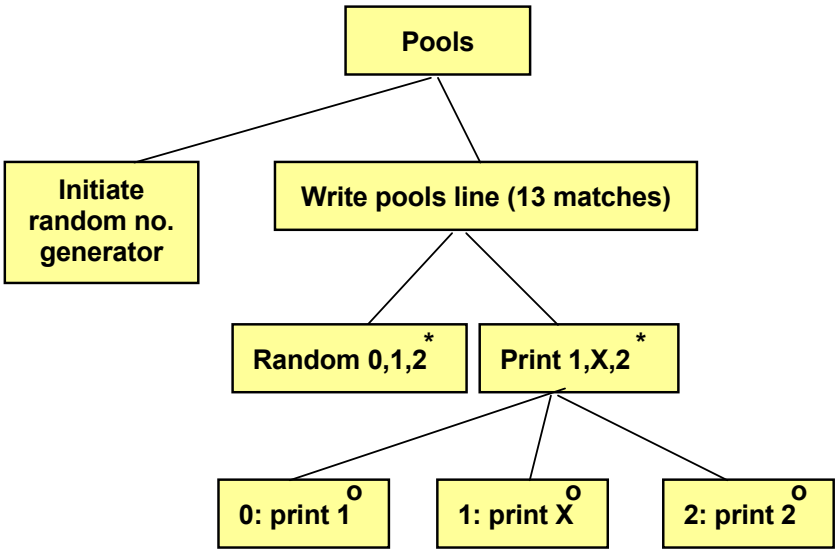
The variable iSum is used to store the sum of the entered numbers. The variable i counts the number of numbers.

The while condition contains input of a number from the user. If the input succeeds, the cin function will return a true value. One turn of the loop is then executed. In the loop the variable iSum is increased by the entered value and the variable i, which counts the values, is increased by 1. At the end of the loop the user is prompted for yet another value.

When one turn of the loop has been run, the condition is tested again, i.e. the program halts and waits for a new entry. As long as the user enters numbers, a new loop turn is run. If the user presses Ctrl-Z the function cin returns a false value, which makes the loop to be terminated. Then the average is printed, which is calculated by dividing the sum by the number of values. Since the variable iSum is an integer we must type cast it to double before the division to not loose the decimals.

3.20 Pools

Programming a pools line (1, X or 2) with 13 football matches is another example of how to use the random number generator in a loop. Since we know that a pools line contains 13 matches, we use a for loop. First we create a JSP graph:



When having intitated the random number generator, the loop begins which should create 1, X or 2 for 13 matches. We do that by create a random number which is 0, 1 or 2. If it is 0, we print '1'. If it was 1, we print 'X'. If it was 2, we print '2'. Here is the code:

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
void main()
{
	int iNo;
	srand(time(0));
	for (int i=1; i <= 13; i++)
	{
		iNo = rand()%3;
		switch (iNo)
		{
		case 0:
			cout << "1" << endl;
			break;
		case 1:
			cout << "  X" << endl;
			break;
		case 2:
			cout << "    2" << endl;
			break;
		}
	}
}
```

When having initiated the random number generator with srand(), the for loop runs from 1 to 13.

For each loop turn we create a random number in the interval 0-2, which is stored in the variable iNo. It is then checked by the switch statement. The different case blocks takes care of the cases 0, 1 and 2. For the value 0, we print '1'. For the value 1, we print 'X' preceded by some blanks which makes the X's appear in a separate column. For the value 2, we print '2' preceded by some more blanks.

3.21 Equation

We will now solve a math problem, namely to solve an equation of 2nd degree. To simplify, we assume that the equation has only integer roots. The equation is:

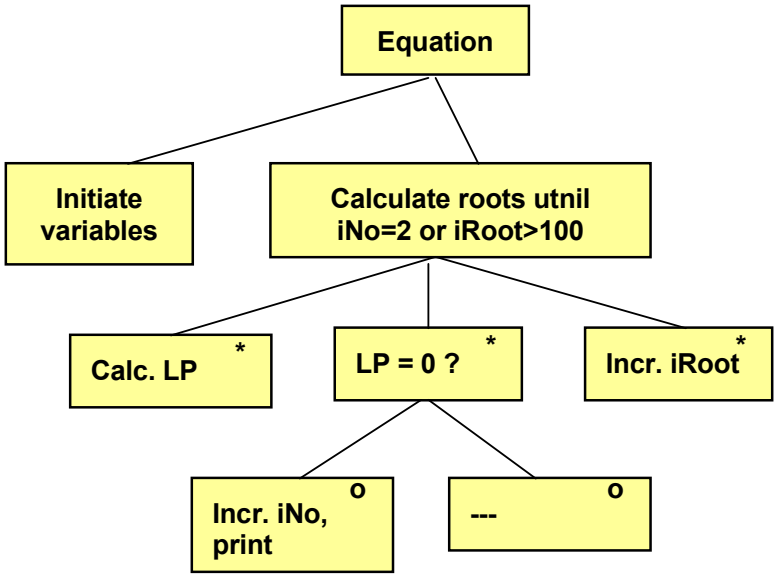
$$x^2 - 6x + 8 = 0$$

Since it is of the 2nd degree it has two roots.

Finding the solution to an equation means to find x values such that the left part (LP) equals the right part (RP), i.e. equal to 0 in our equation.

We create a loop which in turn tests the values 1, 2, 3 … up to 100. The test procedure is to replace x by 1 in LP and calculate if LP equals 0. Then we replace x by 2 and repeat the process until we find two values that match the equation or until 100 has been reached.

First, we create a JSP graph:



The variable iNo is used to count the number of roots to the equation that we have got. The variable iRoot is the x value in the equation which in each loop turn is used to calculate LP. The value of iRoot starts with 1 and is increased by 1 for each loop turn. If the value of LP = 0, we increase the value of iNo and print the root (x value).

Here is the code:

```
#include <iostream.h>
void main()
{
   int iNo=0, iRoot=1, LP;
       while ((iNo<2) && (iRoot<=100))
       {
             LP = iRoot * iRoot - 6 * iRoot + 8;
             if (LP==0)
             {
                   iNo++;
                   cout << iRoot << endl;
             }
             iRoot++;
       }
}
```

First, the value of iNo is set to 0, since it later will be increased by 1 for each found root. The first root value to be tested is 1 (iRoot=1)..

The loop has the condition that iNo should be less than 2, since the number of roots to an equation of 2nd degree is not greater than 2, and iRoot must not exceed 100, since we don't examine roots over 100.,

The first statement in the loop body calculates the value of the left part (LP). This is the actual definition of the equation (x^2 - 6x + 8). If this value is = 0, we increase iNo by 1 and the root is printed.

Then we increase the root value by 1.

3.22 Interrupting a Loop - break

Many times you don't want to set an upper limit on the number of loop turns. Then you can use a condition for the while loop which always is true, for example:

```
while (1==1)
```

1 is obviously always equal to 1, so the loop will run an infinite number of turnes. Therefore we need a possibility to, from inside the loop body, interrupt it, i.e. jump out of it and continue with the first statement after the loop. That is accomplished with the keyword:

```
break;
```

We will give a little program example of this. We will write a program where the user repeatedly is prompted for a number, and the program will respond with the square root of the number. Since you cannot calculate the square root of a negative number, we will inside the loop body check whether the user has entered a negative value. If so, the loop is interrupted. Here is the code:

```
#include <math.h>
#include <iostream.h>
void main()
{
   double dNo;
   while (1==1)
   {
      cout << "Enter a number ";
      cin >> dNo;
      if (dNo<=0)
        break;
      cout << "The square root of the number is " << sqrt(dNo)
           << endl;
   }
}
```

To be able to calculate the square root, we must include math.h, which contains code for a large number of math functions.

The while condition is that 1 equals 1, which always is true, i.e. we have created an infinite loop. Inside the loop body the user is first prompted for a number. If the number is less than 0, the loop is interrupted with break. If the number is 0 or positive, the loop goes on with calculating and printing the square root. The function sqrt() is used for this calculation.

Of course you could solve this problem without using an infinite loop, but regard this as an alternative to create loops.

3.23 Summary

In this chapter we have learnt to incorporate intelligence into our programs by means of selections. We have also learnt how if statements can be used to check different situations and perform different tasks depending on the circumstances. We have showed how to combine various conditions in complex situations with the operators && and ||. We have also learnt how to use the modulus operator % and how to perform conditional input of values from the user by placing the input statement with cin inside the if condition.

An alternative to the if statement is the switch statement which is often used in connection with menu programs.

We have in this chapter also introduced loops, which are used to perform a series of operations a repeated number of times. The main loops are the for loop and the while loop.

We have also extended our knowledge about the random number generator, which has been used to roll a dice and play pools game. Finally we have spent some effort by solving mathematical equations by means of loops.

3.24 Exercises

1. Write a program that prompts the user for two values and prints the least of them.
2. Write a program that prompts the user for his age. If he is younger than 15, the text “You’ll got to stick to the bike some more time” should be printed. Otherwise the text ”You are allowed to drive moped” should be printed.
3. Improve the previous program so that it also pays attention to the driving license age of 18.
4. Write a program that prompts the user for three numbers and prints the greatest of them.
5. Start from the Price Calculation program earlier in this chapter and apply a new discount of 5% if the gross value exceeds 250:- .
6. Continue with the previous program and write code for tax calculation, which is performed so that the user is asked for whether it is food or other products that he has bought. Let the user enter 1 for food and 2 for other products. The program should then add 12% tax for food, or 25% for other products. The tax amount and the final customer price should also be printed.
7. Suppose that the following taxing rules apply:
   a) Income below 10 000:- is not taxable.
   b) For income of 10 000 and more the base tax is always 50%.
   c) For income below 50 000 a tax reduction of 5 000:- is given.
   d) For income over 100 000 there is an extra tax addition of 20% of the portion exceeding 100 000.

   Write a program that prompts the user for his income and calculates the total tax.
8. Write a program that defines whether an entered number is odd or even.
9. Improve the previous program so that it also defines whether the number could be evenly divided by 3.
10. Write a program that prompts the user for how many coins of values 0,50-crowns, 1-crown, 5-crowns and 10-crowns he has in his wallet. The program should then print the total value.
11. Write a program that prompts the user for a price. A discount percent should then be printed according to the following table:

0-100	0%
100-500	5%

500-1000	8%
1000-2000	10%
2000-5000	15%
over 5000	18%

12. Write a program that prompts the user for a quantity and a unit price of a product. If the quantity exceeds 20 and the total price exceeds 1000 kr, the user will get 20% discount. Otherwise, if either the quantity exceeds 20 or the total price exceeds 1000, he will get 10% discount. In all other cases no discount will be given. The total price and the discount should be printed.
13. Use the menu program with the switch statement earlier in this chapter and add the option:
    9. Exit
14. Extend the previous program with the option:
    4. Product
    i.e. the numbers should be multiplied.
15. Write a menu program that prompts the user for three numbers and then displays the following menu:
    1. Least
    2. Greatest
    3. Sum
    The program should also print the requested information.
16. Write a program that prints the numbers 1-10 and their squares.
17. Extend the previous program to also print the cubes of the values.
18. Write a program that prompts the user for integers until the entered number = 0.
19. Extend the previous program so that it also prints the sum of all entered numbers.
20. Write a program that prompts the user for integers and prints a message for each integer whether it is positive or negative. This is repeated until the entered value equals 0.
21. Write a program that prompts the user for integers until the entered value is evenly dividable by 3.
22. In many sports the competitors get scores which are the sum of the scores given by each judge after the highest and lowest score has been deducted. Write a program that prompts for one score from a judge at a time, adds the score and keeps track of the highest and lowest score. The entry is interrupted with Ctrl-Z. Then the competitor's total score should be printed after having deducted the highest and lowest score.
23. Start from the Double Loop program earlier in this chapter, which calculates pair of numbers whose product equals 36. Change it to let the user enter the product to be used.
24. Write a program that calculates the quotient of two numbers. If the quotient = 5, the numbers should be printed. All numbers up to and including 100 should be examined.
25. Start from the Roll Dice earlier in this chapter. Complete it with a printout of all rolls.
26. Change the previous program to roll the dice until it shows 5 or 6.
27. Start from the Two Dice Roll program that rolls two dice at a time. Complete it with a printout of all pair or rolls.
28. Change the previous program to roll the dice until the sum of two rolls is 12.
29. Start from the Pools program earlier in this chapter. Extend it to print 5 lines of pools beside each other, e.g.:

```
1          X          2        X        1
 X          2       1         1          X
1         1          X          2         2
etc.
```

30. Write a program that randomly prints the numbers 0 and 1. It can for instance illustrate tossing of a coin, where 0 and 1 represent the two sides of the coin.
31. Write a program that randomly pulls cards from a pack of cards and prints both colour (spades, diamonds, clubs, hearts) and value (2-10, jack, queen, king, ace). The colour and value of the card should be printed.
32. Start from the equation program earlier in this chapter. Solve the equation:
    $x^2 - 8x + 15 = 0$
33. Write a program that solves the equation of 3rd degree (3 roots):
    $x^3 - 9x^2 + 23x - 15 = 0$

4 Arrays

4.1 Introduction

In this chapter you will learn what an array is, namely a method of storing many values under a single variable name, instead of using a specific variable for each value. We will begin by declaring an array and assign values to it.

In connection with arrays you will have great use for loops, by means of which you can efficiently search for a value in the array and sort the values.

Arrays is a fundamental concept within programming which will frequently be used in the future.

4.2 Why Arrays

An array is, as already mentioned, a method of storing many values of the same data type and usage under a single variable name. Suppose you want to store temperatures measured per day during a month:

```
12.5
10.7
13.1
11.4
12.1
...
```

If you didn't know about arrays, you would need 30 different variable names, for instance:

```
tempa = 12.5
tempb = 10.7
tempc = 13.1
tempd = 11.4
tempe = 12.1
...
```

This is a bad option, especially if you want to calculate the average temperature or anything else. Then you would need to write a huge program statement for the sum of the 30 variables.

Instead, we use an array, i.e. one single variable name followed by an **index** within square brackets that defines which of the temperatures in the array that is meant:

```
temp[1] = 12.5
temp[2] = 10.7
temp[3] = 13.1
temp[4] = 11.4
temp[5] = 12.1
...
```

The name of the array is temp. The different values in the array are called **elements**.

In this way we can use a loop, where the loop variable represents the index, and do a repeated calculation on each of the temperatures:

```
for (i=1; i<=30; i++)
{
   //Do something with temp[i];
}
```

The loop variable i goes from 1 to 30. In the first turn of the loop i has the value 1, which means that temp[i] represents temp[1], i.e. the first temperature. In the second turn of the loop i has the value 2 and temp[i] represents the second temperature.

By using a loop the amount of code required will not increase with the number of temperatures to handle. The only thing to be modified is the number of turns that the for loop must execute.

In the code below we calculate the average of all the temperatures:

```
iSum = 0;
for (i=1; i<=30; i++)
{
   iSum += temp[i];
}
dAvg = iSum / 30;
cout << dAvg;
```

The variable iSum is set to 0 since it later on will be increased by one temperature at a time. The loop goes from 1 to 30, i.e. equal to the number of elements in the array. In the loop body the variable iSum is increased by one temperature at a time. When the loop has completed, all temperatures have been accumulated in iSum. Finally we divide by 30 to get the average, which is printed.

4.3 Declaring an Array

Like for all variables, an array must be declared. Below we declare the array temp:

```
double temp[31];
```

The number within square brackets indicates how many items the array can hold, 31 in our example. 31 positions will be created in the primary memory each of which can store a double value. The indeces will automatically be counted from 0. This means that the last index is 30. If you need temperatures for the month April, which has 30 days, you have two options:

1. Declare temp[30], which means that the indeces goes from 0 to 29. 1st of April will correspond to index 0, 2nd of April to index 1 etc. 30th of April will correspond to index 29. The index lies consequently one step after the actual date.
2. Declare temp[31]. Then 1st of April can correspond to index 1, 2nd of April to index 2 etc. 30th of April will correspond to index 30. The date and index are here equal all the time. This means that the item temp[0] is created ”in vain” and will never be used.

It is no big deal which of the methods you use, but you will have to be conscious about the method selected, because it affects the code you write. We will show examples of both methods.

Note that, in the declaration:

```
double temp[31];
```

all elements are of the same data type, namely double. For arrays *all elements all items always have the same data type*.

4.4 Initiating an Array

You can assign values to an array already at the declaration, e.g.:

```
int iNo[5] = {23, 12, 15, 19, 21};
```

Here the array iNo will hold 5 items, where the first item with index 0 gets the value 23, the second item with index 1 the value 12 etc.

The enumeration of the values must be within curly brackets, separated by commas.

As a matter of fact it is redundant information to specify the number of items to 5 in the declaration above, since the number of enumerated values is 5. Therefore you could as well write:

```
int iNo[] = {23, 12, 15, 19, 21};
```

An enumeration within curly brackets can only be written in the declaration of an array. For instance, the following is erroneous:

```
double dTemp[4];
dTemp = {12.3, 14.1, 11.7, 13.8};
```

In the following code section we declare an array of integers and assign values to the array in the loop:

```
int iSquare[11];
for (int i=0; i<=10; i++)
{
  iSquare[i] = i*i;
}
```

The array iSquare is declared to hold 11 items of integer type. The loop then goes from 0 to 10. In the first turn of the loop i is =0 and the item iSquare[0] gets the value 0*0, i.e. 0. In the second turn of the loop i is =1 and iSquare[1] gets the value 1*1, i.e. 1. In the third turn of the loop the item iSquare[2] gets the value 2*2, i.e. 4. Each item will contain a value equal to the square of the index.

4.4.1 Index outside the Interval

As a C++ programmer you must yourself keep track of the valid index interval. The result could be disastrous if you wrote:

```
temp[35] = 23.5;
```

This means that we store the value 23.5 in the primary memory at an adress that does not belong to the array, but might belong to a memory area used for other data or program code. If you run such a code the system might in worst case break down and you will have to restart the computer.

4.5 Copying an Array

Suppose we want to copy the temperatures from the array with April's values to an array with June's values. You cannot copy an entire array in this way:

```
dblTempJune = dblTempApr;
```

You will have to copy the values item by item by means of a loop:

```
for (int i=1; i<=30; i++)
{
  dblTempJune[i] = dblTempApr[i];
}
```

Here the loop goes from 1 to 30 and we copy item by item for each turn of the loop.

4.6 Comparing Arrays

What is meant by comparing whether two arrays are equal? They must contain item values that are equal in pairs. In the following code section we compare the two arrays with April's and June's temperatures:

```
int eq = 1;
for (int i=1; i<=30; i++)
```

```
{
   if (dblTempJune[i] != dblTempApr[i])
      eq = 0;
}
```

Here we let the variable eq reflect whether the two arrays are equal, where the value 1 corresponds to ”equal” and 0 ”not equal”. From the beginning we assign eq the value 1, i.e. we presume the arrays to be equal. Then in the loop we go through item by item in the two arrays and checks if they are equal in pairs. The if statement checks if they are different. If so, the variable eq is set to 0, otherwise nothing is changed. If two items happen to be different, the variable eq will have the value 0 after the loop has completed. If however all pairs of items are equal, the statement:

```
eq = 0;
```

will never be executed, and the variable eq will remain =1. We could then complete our program with output about the result:

```
if (eq == 1)
   cout << "The arrays are equal";
   else
   cout << "The arrays are different";
```

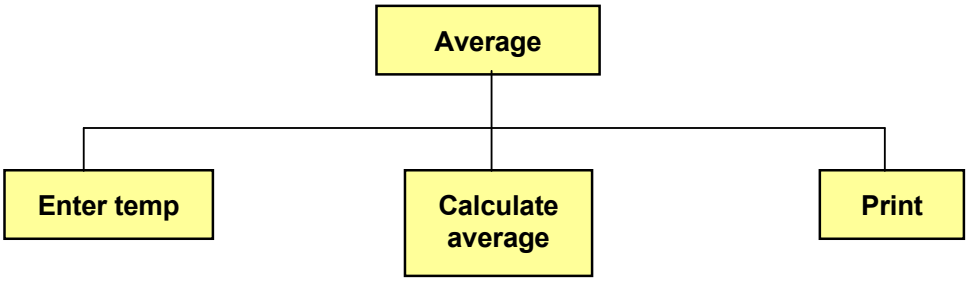
It is *not* possible to in one single statement check whether the arrays are equal:

```
if (dblTempJune == dblTempApr)
```

You must compare item by item like in the code above.

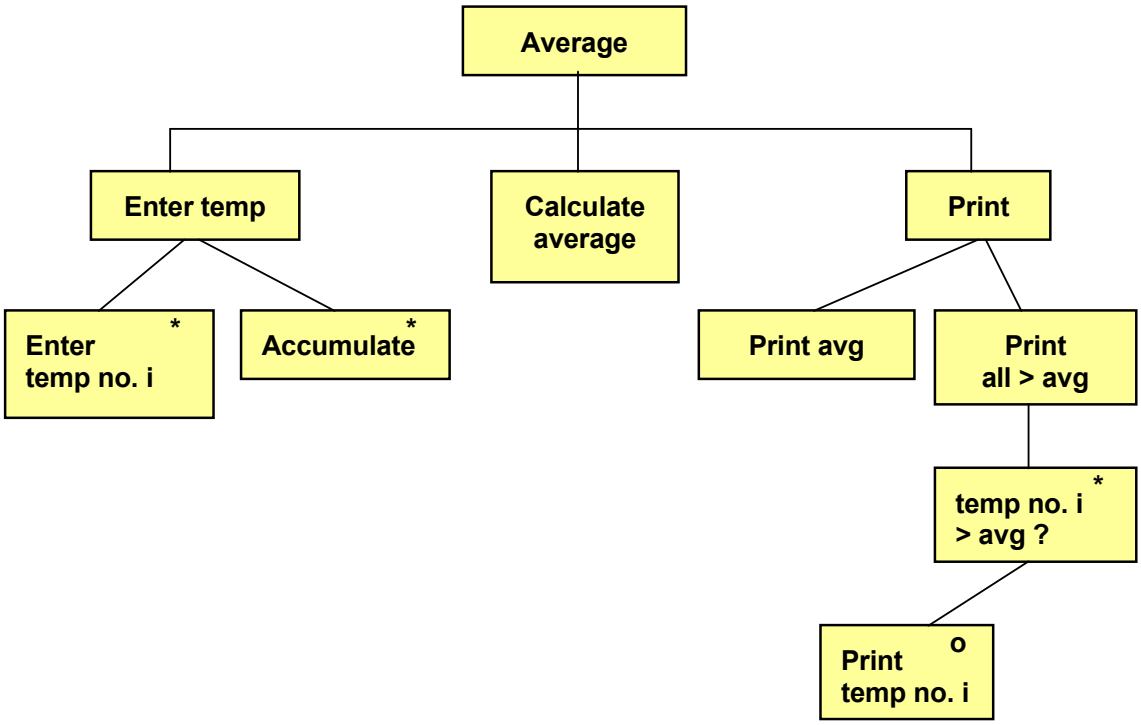
4.7 Average

We will now write a program that reads temperatures to an array from the user and then calculates the average of all temperatures. The program should then print the average and all temperatures exceeding the average. We begin with a JSP graph:



We have made an overview JSP that mainly describes the procedure.

Since we will calculate an average, we need the sum of all temperatures. We choose to sum the temperatures at the time of entry, which is made in a loop:



The average calculation is simple. We don't have to detail it. However the output is a little more complicated. First we print the calculated average. Then we write a loop which in turn checks each item of the array against the calculated average. If temperature number i is greater than the average, we print temp no. i.

Here is the code:

```
#include <iostream.h>
void main()
{
   // Deklarations
   const int iNoOfDays = 30;
   double dAvg, dSum = 0;
   double dblTempApr[iNoOfDays + 1];
   int i;
   // Entry and calculation
   for (i=1; i<= iNoOfDays; i++)
   {
      cout << "Temperature day " << i;
      cin >> dblTempApr[i];
      dSum += dblTempApr[i];
   }
   dAvg = dSum / iNoOfDays;
   // Printout
   cout << "Average temperature: " << dAvg << endl;
   cout << "Temperatures exceeding average: " << endl;
   for (i=1; i<= iNoOfDays; i++)
   {
      if (dblTempApr[i] > dAvg)
         cout << "Day no.: "<<i<<" temp:
           "<<dblTempApr[i]<<endl;
   }
}
```

First we declare a constant iNoOfDays which is set to 30 and is used later in loops and average calculation. The variable dAvg is used for storing of the calculated average. The variable dSum is initiated to 0 since it will be increased by the value of each entered temperature. The array dblTempApr is declared to hold 31 items, which means that we can let the index values correspond to the day numbers of the month. The item with index 0 will consequently not be used. Finalyy we declare the variable i, which is used as loop counter.

The first loop takes care of entry of the temperatures. The loop counter goes from 1 to 30 and each entered temperature is stored in the array. The variable dSum is increased by the entered temperature.

At loop completion the variable dSum contains the accumulated total of all temperatures, which is divided by the number of days, which gives the average.

The printout starts with the average. Then comes the last loop which goes from 1 to 30. For each turn of the loop we check whether temperature number i exceeds the average. If so, the day number is printed, which is equal to the index value, together with the corresponding temperature.

4.8 Sales Statistics

We will now give an example that shows how to use arrays and conditional input in a while statement. The situation is this:

A company has a number of salesmen, each with a salesman number in the interval 1-100. When a salesman has sold for a specific amount, he enters his salesman number and the sales amount. This goes on until you terminate the entry with Ctrl-Z. Then a summary should be printed with one line per salesman showing total sales amount.

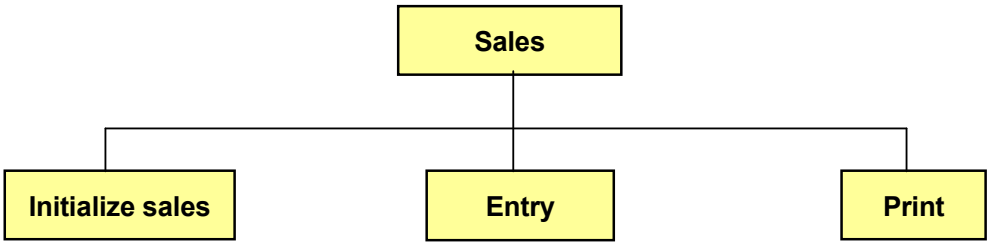
Furthermore, a fee per salesman should be calculated. If the sales amount is below 50000:- the fee is 10% of the sales amount. If the amount is greater, the salesman will get a fee which is 10% of the first 50000:- plus 15% of the amount exceeding 50000:-. If for instance the sales amount is 70000:- the fee is 10% of 50000:- which gives 5000:- plus 15% of the exceeding 20000:- which is 3000:-. The total fee will in this case be 8000:-.

An entry from different salesmen could look like this:

```
78  10000
32  500
2  12000
100  25000
78  60000
2  1000
5  60000
```

```
The printout will then be:

Number      Amount        Fee
======      ======      =====
     2       13000       1300
     5       60000       6500
    32         500         50
    78       70000       8000
   100       25000       2500
```

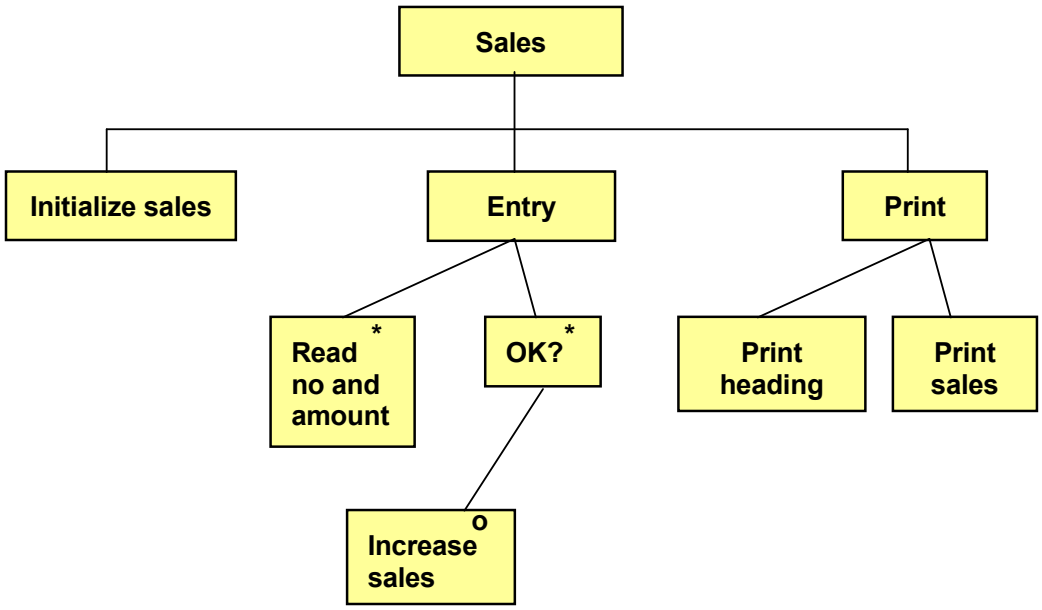
We begin with a JSP graph:



We will use an array called sales with 100 items, where each item corresponds to a certain salesman. For each entered sales amount the array item corresponding to the salesman number should be increased by the entered amount. Therefore we must initialize the entire array, i.e. set all its items = 0 so each salesman's accumulated amount starts with 0. Note that in C++ the items of an array are not automatically set to 0 at the declaration. The declaration only allocates memory space.

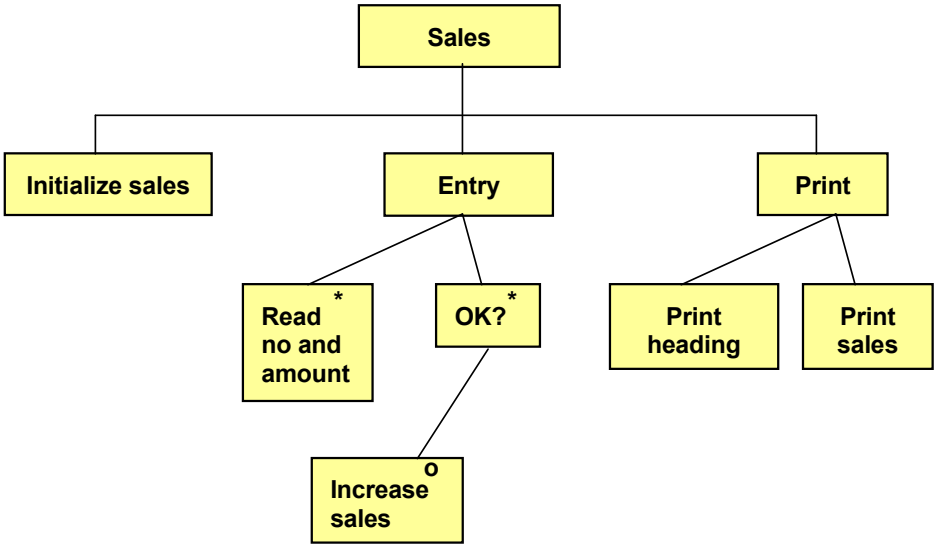
Any unpredictable values present in these memory addresses will be retained until you initialize them.

Then we read salesman number and sales amount. This is made in a loop so that we can go on with entry of values as long as we want. We break down the "Entry" box:

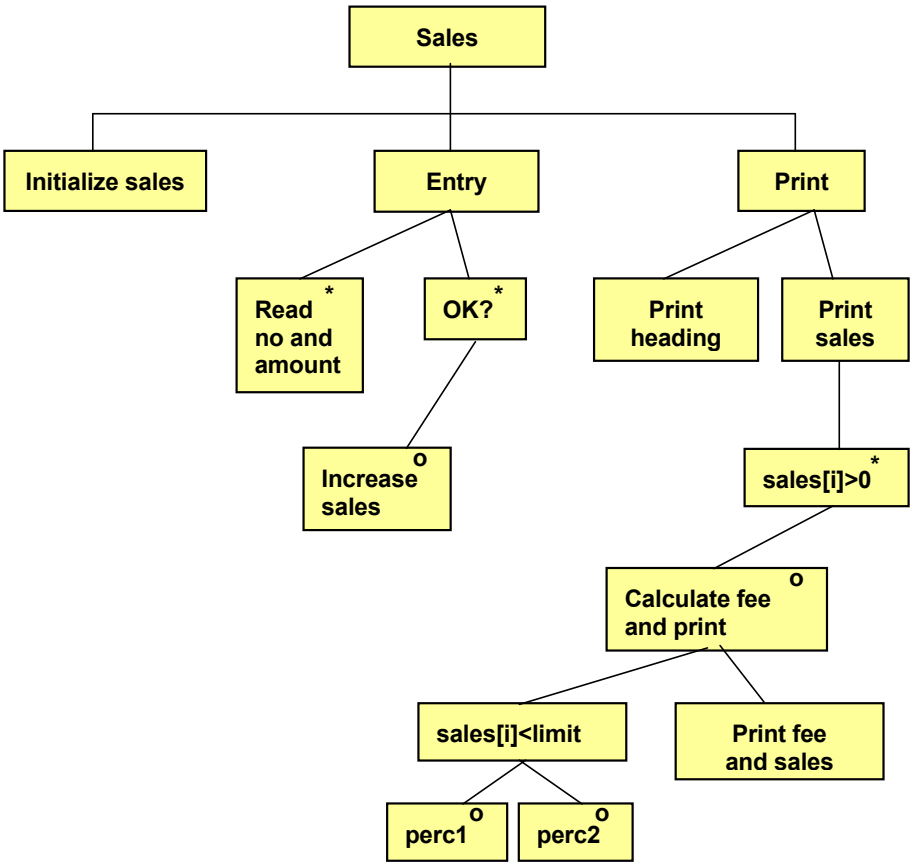


We read a salesman number and sales amount, one at a time. In the box "OK?" we check that the salesman number is between 1 and 100 and that the sales amount is not negative. If OK, we increase the corresponding item in the sales array.

The box ”Print” has been detailed by first printing the heading and then the sales values. In connection with the printing we check that the sales amount is not 0. Salesmen having sold nothing should not be included in the printed summary. We break down the box “Print sales” further:



If the sales amount exceeds 0, we calculate the fee and print one line in the summary. At fee calculation we will now pay attention to whether the amount exceeds the limit 50000:-, which gives still another detailed level:



If the sales amount is below the limit 50000:- the lower percent 10% will be applied. Otherwise the greater percent 15% will be applied to the exceeding amount. When the fee calculation is finished, the fee and sales amount are printed.

Here is the code:

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
  const int iMaxNo=100;
  const double dLimit=50000, perc1=0.1, perc2=0.15;
  double sales[iMaxNo], dAmount, dFee;
  int i, nr;
  //Initialize array
  for (i=0;i<iMaxNo; i++)
    sales[i] = 0;
  //Enter salesman info
  while (cin>>nr && cin>>dAmount)
  {
    if (nr<1 || nr>iMaxNo || dAmount<0)
      cout << "Input error" << endl;
    else
      sales[nr-1] += dAmount;
  }
  //Print summary
  cout << endl
    << "Number      Amount         Fee" << endl
    << "======      ======      =====" << endl;
  for (i=0; i<iMaxNo; i++)
  {
    if (sales[i] > 0)
    {
      if (sales[i] <= dLimit)
        dFee = perc1 * sales[i];
      else
        dFee = perc1*dLimit + perc2*(sales[i]-dLimit);
      cout << setw(4) << (i+1) <<
        setprecision(0) << setiosflags(ios::fixed) <<
        setw(13) << sales[i] << setw(10) <<
        dFee << endl;
    } // end if
  } // end for-loop
} // end main
```

The constant iMaxNo = 100 represents the number of salesmen and is used for loop control. The constand dLimit = 50000 is used for the fee calculation. The constants perc1 and perc2 are the two different percentages used for the fee.

The array is declared to contain 100 items with the index 0-99. Here, salesman no.1 will correspond to index 0, salesman no. 2 index 1 etc.

The variable dAmount is used for entry of sales amounts and the variable dFee for fee calculation. The variable i is used as loop counter and the variable nr to entry of salesman numbers.

Then the sales array is initialized, where we set all items to 0.

Entry of salesman numbers and amounts is done in a while statement. If entry is successful, the loop continues. If you however you press Ctrl-Z, the while condition is false and the entry loop is interrupted, enabling the program to continue with the next statement.

Inside the loop the program checks if the salesman number is less than 1 or greater than 100. This is for safety reason to guarantee that we don't go outside the index interval of the array, since the salesman number gives the index value of the array. In addition, the program also checks if the sales amount is less than 0. If any of these conditions are true, the text "Input error" is printed and the user can enter new values. If everything is OK, the sales item is increased by the entered amount. Note that we decrease the salesman number by 1 to get the correct item in the array.

At loop completion (Ctrl-Z), the program goes on with printing the summary heading.

Then the last loop will calculate the fee and print one line per salesman. First in the loop, we check the sales total to be greater than 0, otherwise no line for that salesman is printed. Then we check if the sales total is less than the limit 50000. If so, the fee is calculated as the lower percent multiplied by the sales total. Otherwise the fee is calculated as the lower percent times the limit 50000 plus the greater percent multiplied by the difference between the sales total and the limit 50000.

When the inner if statement has completed the fee calculation is complete and the program writes a line with salesman number (i+1), sales total (sales[i]) and fee. Note that, when we print the salesman number, we must use the index value increased by 1, since the index value all the time is 1 less than the salesman number.

We have also used the formatting functions from the header file iomanip.h to get a nice layout with straight columns.

4.9 Product File, Search

We will now examine a situation where we use several arrays in parallel. We will build a simple product file, where we use an array for the product id:s and another array for the product prices. We will organize it so that a product in the product id array with for instance index 73 has its price in the price array at the same index position, i.e. 73:

```
Prodid     Price
2304      152,50
2415       75,40
3126       26,80
...
```

The array with product id:s is called iProdid and the price array dPrice.

Suppose we want to be able to enter a product id and get the corresponding price. Then we must search the iProdid array. Look at the following code section:

```
while (cin >> iProd)
{
   for (i=1; i<=100; i++)
   {
     if (iProd == iProdid[i])
       cout << "The price is: " << dPrice[i] <<
         endl;
   }
}
```

In the while condition we read a product id to the variable iProd. This allows for repeated entry of product id:s until you interrupt with Ctrl-Z.

The inner loop goes from 1 to 100 and checks one item at a time in the product id array to equal the entered product id. The loop counter i going from 1 to 100 is used as index in the product id array and represents the different product id:s in the array. Note that we use the same i-value in the price array as in the product id array. If for instance we encounter equality for the 23rd product, also the 23rd price should be printed, since the variable i then has the value 23.

4.10 Two-Dimensional Array

In many business systems on the market customer discounts are based on the customer group that the customer belongs to, and the product group for the bought product. Different customer segments will then get different discount profiles.

Example:

		Product group		
		1	**2**	**3**
Customer	**1**	10	12	13
group	**2**	13	14	15
	3	14	16	17

If for example a customer of customer group 3 orders a product from product group 2, he will get 16% discount.

To store a discount matrix in this way in a program, you will need a two-dimensional array. Such an array has two indeces, where the first index could be thought of as representing the lines in the matrix, and the second index the columns:

```
double dDiscount[5][8];
```

Here we have declared a two-dimensional array named dDiscount with 5 lines (index 0-4) and 8 columns (index 0-7).

To assign values to the different array items, we can write:

```
dDiscount[1][1] = 10;
dDiscount[1][2] = 12;
...
```

Note that we all the time must use two indeces for dDiscount.

Suppose that we want a program section where the user can enter customer group and product group and the program should respond with the corresponding discount percent:

```
cout << "Enter customer group ";
cin >> cgrp;
cout << "Enter product group ";
cin >> pgrp;
cout << "Discount: " << dDiscount[cgrp][pgrp];
```

Suppose that, when this program section is run, the user enters 3 and 2. The variable cgrp will get the value 3 and pgrp the value 2. These two values are used as indeces in the two-dimensional array. If we use the values from the discount matrix above, we will get the printout:

```
Discount: 16
```

Let us now turn the problem the other way so that the user enters a discount percent and that the program responds with corresponding customer group and product group. A prerequisite to this is that each percent only occurs once in the matrix, which is not very likely, but it shows how to search a two-dimensional array. The code will be:

```
cout << "Enter percent: ";
cin >> dPerc;
for (i=1; i<=5; i++)
{
```

```
   for (j=1; j<=8; j++)
   {
      if (dDiscount[i][j] == dPerc)
         cout << "Product group " << i <<
                 " and customer group " << j;
   }
}
```

First the user is prompted for a percent which is stored in the variable dPerc. A double loop then performs the search for the entered percent, where the outer loop goes through the lines of the matrix and the inner loop through the columns of the matrx. The loop counter i thus corresponds to line index and j to column index. The inner loop goes through all its values before the outer loop changes its value, which means that the matrix is searched one line at a time, where all items in the line are checked. The if statement checks if the matrix item equals the entered percent (the variable dPerc). If equal, the corresponding loop counters i and j are printed, which correspond to customer group and product group.

4.11 Sorting

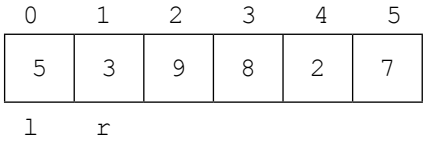
Many times it is easier to work with an array if the items are sorted, especially when searching for a specific value. For instance, in the product id array in an earlier section, if the product id:s are sorted by size, the process of finding a certain product, and consequently also its price, is much faster than for an unsorted array. We will therefore discuss array sorting.

We will as example use an array with 6 items named iNos:

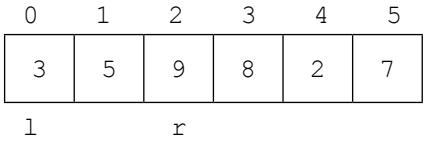
```
int iNos[6] = {5,3,9,8,2,7};
```

The items of the array are not yet sorted. We want to write a program that sorts them by size. The problem is that a computer program is not capable of, like the human eye, scan the values and instantly sort them. We have to write code that systematically compares two values in turn and interchange their positions in the array.

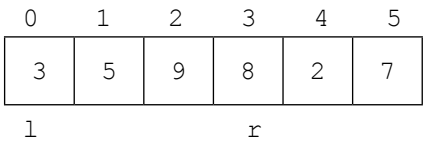
We will use two variables, l and r, which are indeces in the array and points to two items. l means "left" and r "right". These items are compared in pairs, and if the right item is less than the left, they will interchange their positions in the array:



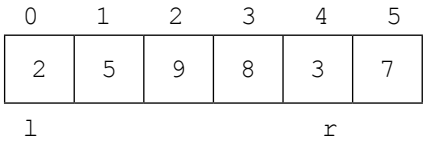
The indeces of the array have the interval 0-5. The variable l has from the beginning the value 0 and r the value 1, i.e. they point on the two first items of the array. Since the right item is less than the left (3 is less than 5), they are interchanged:



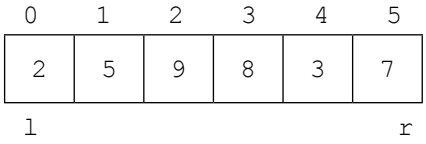
We then increase r by 1, so that it points to the value 9, while l remains. 9 is not less than 3, so no interchange is made. r is again increased by 1:



Neither this time there is no interchange, since 8 is greater than 3. We increase r by 1 again. Then r points to the value 2, which is less than 3, so the items are interchanged:

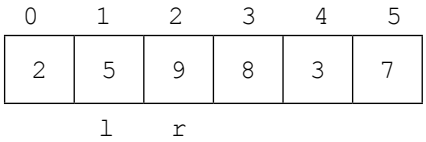


We increase r by 1 again. 7 is greater than 2, so the items remain:



Now r has gone through all values, and as result we have got the least item on index position 0 in the array. We have performed *a series of comparisons*.

Now we increase l by 1 and perform a new series of comparisons, where r goes from index position 2 to 5:



Here 9 is not less than 5, so we increase r by 1 and so forth. When r arrives at index position 4, the right item (3) is less than the lef t (5), so we interchange them.

When two series of comparisons have been completed, we have got the two least items on the two first positions:

0	1	2	3	4	5
2	3	9	8	5	7

Once again we increase l by 1 and let r go from the item immediately to the right of l to the last item of the array. This is repeated until we compare the two last items of the array:

0	1	2	3	4	5
2	3	5	7	8	9
				l	r

After completion of the last comparison, the entire array is sorted.

To summarize, l goes from 0 to the next last position of the array, while r goes from the position to the right of l to the last position of the array. We use an outer loop for the l-values and an inner for the r-values:

```
for (l=0; l<=4; l++)
{
   for (r=l+1; r<=5; r++)
   {
      //Check if right is less than left
      //and in that case interchange
   }
}
```

l goes from 0 (first position of the array) to 4 (next last position), while r goes from l+1 (the position to the right of l) to 5 (last position).

The check whether the right is less than the left is made by an if statement:

```
if (iNos[r] < iNos[l])
```

The interchange is a little tricky. We cannot directly let two variables change values. We must use an intermediary storage, a temporary variable that temporary stores one of the values:

```
temp = iNos[l];
iNos[l] = iNos[r];
iNos[r] = temp;
```

Here we let the variable temp get the value of the left item, then we let the left item get the value of the right item, and finally we let the right item get the value of the temporary variable, i.e. the old left value. This triangular exchange has the effect that the two array items interchange their values. After the triangular exchange the value of temp is of no concern.

Here is the complete code:

```
for (l=0; l<=4; l++)
{
   for (r=l+1; r<=5; r++)
   {

      if (iNos[h] < iNos[v])

      {
         temp = iNos[l];
         iNos[l] = iNos[r];
         iNos[r] = temp;

      }
   }
}
```

After completion of the double loop the array items are sorted.

4.12 Searching a Sorted Array

For a sorted array, when searching for a particular item, we don't need to scan the entire array from the first to the last position and check each single value. For a small array with only 6 items like in the previous example, there is no big deal. But what if we have a product array with thousands of product id:s. Then the search time would be considerable and our program would be regarded as having bad performance.

We will use a more refined method, namely to halve the index interval repeatedly. We go in to the middle item of the array and check if the searched value is to left or right. When having selected which half to continue with, we halve that part again. This is repeated until we find the searched value. The execution time will be reduced considerably.

Suppose we have a product array with 31 items (index 0-30)

```
   0     1     2   ...              15   ...                          30
 ┌──────────────────────────────────────────────────────────────────────┐
 │ 2314  2345  3123                 4526                          6745 │
 └──────────────────────────────────────────────────────────────────────┘
```

The index values are shown above the product id:s.

Suppose we are searching for product id 5321. We begin with checking whether 5231 is less than the middle item with index position 15, namely 4526. If so we go on with the left interval, otherwise the right. In our case we use the right interval, which we halve and get index position 22 (index must always be an integer). We check whether the searched product id 5231 is greater or less than the product id on position 22, etc.

When having divided the interval enough number of times, we will have found the searched item, or otherwise it does not exist in the array.

We will now discuss the code for this. First we declare some variables:

```
int l=0, r=30, iFound=0, iPos, iSrch;
```

The variables l and r are index positions of the array. l is the left end point of the interval, which from the beginning is 0. r is the right which from the beginning is 30.

The variable iFound is used to indicate whether or not the searched product id has been found. The value 0 means not yet found, and the value 1 means that it has been found.

The variable iPos is the index for the found product id. The variable iSrch is the searched product id, which is read from the keyboard:

```
cout << "Enter the searched product id: ";
cin >> iSrch;
```

We then perform some introductory checks to figure out if the searched product id is first or last in the array:

```
if (iSrch == iProdid[0])
{
  iPos = 0;
  iFound = 1;
}
if (iSrch == iProdid[30])
{
  iPos = 30;
  iFound = 1;
}
```

As long as the product id has not been found, we will divide the interval:

```
while (!iFound)
{
```

First we calculate the middle of the interval:

```
int iMid = l + (int)((r-l)/2);
```

From the beginning r is =30 och l=0. (r-l)/2 then makes 15. Since this division might give a decimal number, we perform a type cast with (int) within parenthesis in front of the division. In that way we ensure that the index always is an integer. This half interval is added to the value of l. Since l by the time not necessarily equals 0 all the time, this means that we take the left endpoint of the interval and add half the interval, i.e. we calculate the middle point of the current interval, which is stored in the variable iMid.

Then we check if there is a match to the middle item of the interval:

```
if (iSrch == iProdid[iMid])
{
   iFound = 1;
   iPos = iMid;
}
```

If the searched product id equals the product id at the position given by the variable iMid in the product array iProdid, there is a match, and the variable iFound is set =1 and the found position is stored in the variable iPos.

In case of no match, we check if the searched product id is to the left or to the right of the middle point:

```
if (iSrch > iProdid[iMid])
   l=iMid;
   else
   r=iMid;
}
```

If the searched product id is greater than the product id at position iMid, we set the left endpoint (the variable l) to the value of iMid, which means that we move the left endpoint to the new middle value, and we have a new interval which is the right half of the previous interval. Otherwise we focus on the left half of the interval and we let the right endpoint (the variable r) get the value of iMid. In both cases the loop performs another turn.

By the time the loop has divided the interval so many times that we certainly get a match in the statement:

```
if (iSrch == iProdid[iMid])
```

provided that the user has entered a product id that is present in the array.

Here is the entire program:

```
#include <iostream.h>
void main()
{
```

```
int l=0, r=30, iFound=0, iPos, iSrch;
int iProdid[31] = {2314, 2345, 3123, ... 6745};
cout << "Enter the searched product id: ";
cin >> iSrch;
if (iSrch == iProdid[0])
{
  iPos = 0;
  iFound = 1;
}
if (iSrch == iProdid[30])
{
  iPos = 30;
  iFound = 1;
}
while (!iFound)
{
  int iMid = l + (int)((r-l)/2);
  if (iSrch == iProdid[iMid])
  {
    iFound = 1;
    iPos = iMid;
  }
```

```
    if (iSrch > iProdid[iMid])
      l=iMid;
      else
      r=iMid;
  }
}
```

4.13 Summary

In this chapter we have learnt about arrays. We have learnt to declare arrays and assign values to them. We have also seen the advantage with using loops in connection with arrays.

We have also studied an algorithm for sorting the items of an array. You should try to really understand the algorithm. You should also remember how to write the sorting code in C++.

A sorted array is very efficient when searching for a particular value. We have studied how to do a smart search in an array. The search method presented here is often called binary search.

4.14 Exercises

1. Write a program where you declare an array with 10 integers and then read both positive and negative values to the array. The program should then:
   a) print the first, the fifth and the tenth item.
   b) print the sum of all items.
   c) print the numbers in reversed order.
   d) change sign of all negative numbers to positive, and then print them.
   e) ask for a number and then print all numbers less than that number.
   f) ask for an index and print the corresponding item.
   g) ask for a number and print the index of that number. We assume that the user enters a number that exists in the array.
   h) move the first item to the last position of the array.
2. Write a program that prompts the user for a month number and prints the number of days of that month. Use an array like this:
   `const int iDaysInM[] = {31, 28, 31, 30, …};`
3. Write a program that creates random temperatures between 15 and 25 degrees, one temperature per day of the month July. The temperatures are stored in an array. The program should then create a new array for August and copy the July values (see the section 'Copying an Array'). Finally the August values should be printed.
4. Expand the previous program to compare the values of July and August and check if the arrays are equal.
5. Complete the previous program so that one of the August temperatures is changed before the comparison.
6. Complete the previous program with calculation of the average temperature during August. The program should also print all temperatures greater than the average.
7. Declare an array that contains the following nine densities for metal alloys:
   1.5 2.8 4.6

5.7 7.9 8.3
8.6 8.8 8.9

Write a program that prompts the user for a density and prints the one closest below the entered density.

8. Write a program that fills an array with 25 integers between 0-9. The program should then ask the user for a number and print the number of occurrences of that number in the array.
9. Start with the Sales Statistics program earlier in this chapter and add logic so that if a salesman has sold for more than 100000, he will get a fee also including 20% of the portion exceeding 100000.
10. Expand the previous program to also print the number of sales per salesman.
11. Expand the previous program to also print the average sales amount per salesman.
12. Declare an array containing some product id:s, and a price array with unit prices per product. Write a program that prompts the user for a product id and prints the corresponding price. If the product id is not found, a suitable message should be printed.
13. Complete the previous program so that the user also can enter a quantity of the product and get the total price for the purchase.
14. Complete the previous program with a discount matrix according to the section 'Two-Dimensional Array'. The user should be able to enter a product group and a customer group, and the corresponding discount should be deducted.
15. Write a program that creates 10 random rolls of a dice and stores them in an array. The array should then be sorted and printed.
16. Start with the program that searches a sorted array for a product id. Complete the program with initialization of the array with as many product id:s as required, and run it.
17. Go on with the previous program and make it print the index position of the found product id.
18. Improve the previous program so that if you enter a product id not existing in the array, a suitable message should be printed.
19. Expand the previous program with a price array that contains the prices of each product, and with a printout of the price of the found product id.

5 Strings

5.1 Introduction

A string is a text, i.e. a sequence of characters (letters, digits and other special characters). String handling is a little tricky in C++, so we have dedicated an separate chapter to this subject.

Actually, a string is an array that consists of a number of items, where each item is a character in the string.

There are a number of string handling functions. We will in this chapter get acquainted with the most common and usable string functions, like calculating the length of a string, copying a string, concatenating strings and picking out parts of a string.

In programming, string handling is important, since a user often enters information in text form which is taken care of by the program. We will go through several programming examples, where we will have use of our string knowledge.

5.2 Data Type char

We will first get to know the most simple of all strings, namely the one containing just one character. To store one character in a variable we use the data type char. Below we declare a string variable of the char type:

```
char cLetter;
```

The variable cLetter can now contain any one character. We can assign a value to the variable:

```
cLetter = 'A';
```

Note that we use single quotes for the character.

We can also, like for all kinds of variables, assign a value directly in the declaration:

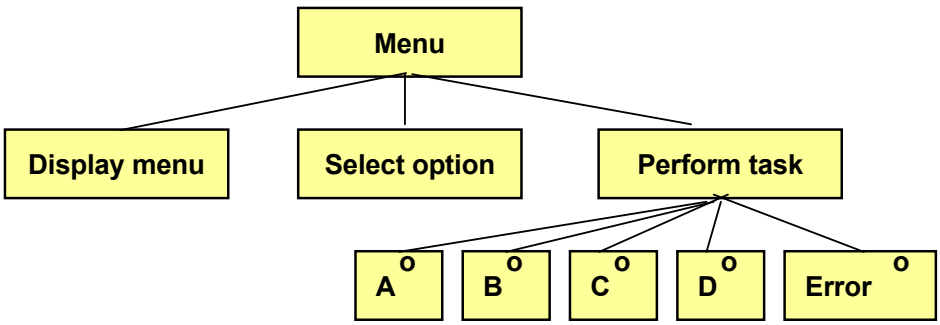
```
char cLetter = 'A';
```

5.3 Menu Program

Our first program shows how to handle entry of characters by the user to select a menu option. The program will first display a menu:

```
   Menu
   ====
A. Order
B. Invoice
C. Warehouse
D. Finance
Select:
```

Here the user can enter one of the letters A, B, C or D to choose an item. We will not build a full-featured order/invoice/ warehouse/ finance system, but we will only let the program print a text about the selected choice. We start with a JSP graph:



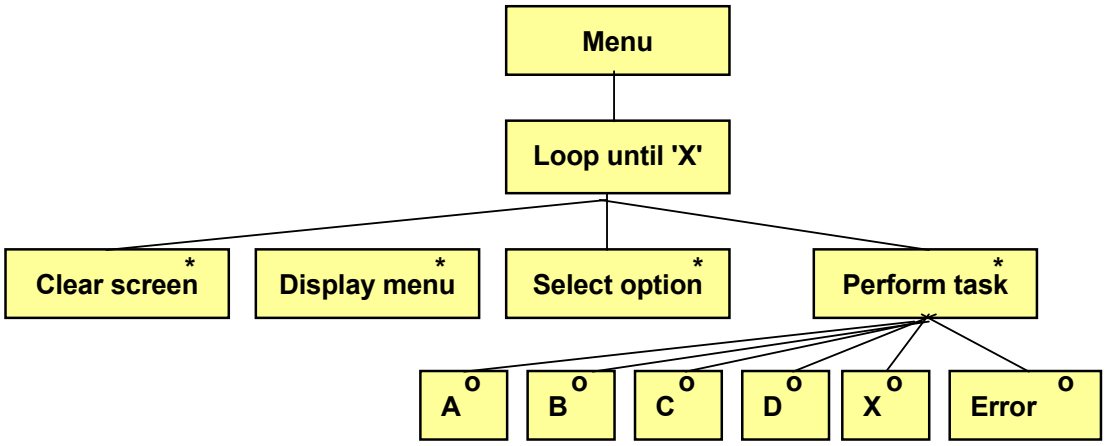
The first action is that the menu is displayed. Then the user is prompted for a choice by means of the letters A-D. Finally the requested option is executed, i.e. a simple text message will be printed. If the user enters another character than A-D, an error message is printed. Here is the code:

```
#include <iostream.h>
void main()
{
   char cSel;
       cout << "    Menu" << endl <<"    ===" << endl;
       cout << "A. Order" << endl;
       cout << "B. Invoice" << endl;
       cout << "C. Warehouse" << endl;
       cout << "D. Finance" << endl;
       cout << "Select: ";
       cin >> cSel;
       switch (cSel)
       {
     case 'A':
        cout << "You selected Order";
        break;
     case 'B':
        cout << "You selected Invoice";
        break;
     case 'C':
        cout << "You selected Warehouse";
        break;
     case 'D':
        cout << "You selected Finance";
        break;
     default:
        cout << "Erroneous choice";
        break;
   }
}
```

First a char variable is declared named cSel, and then the menu is printed with a number of cout statements. The subsequent cin statement reads a character from the user to the variable cSel, which then is checked in the switch statement. The switch statement contains one case section for each option. Note that each case line has the character A-D within single quotes, which is necessary since it is a char variable that is checked. The default section takes care of all characters other than A, B, C or D.

5.4 Menu Program with Loop

We will now improve our menu program so that the user repeatedly can enter different options without terminating the program. We then put the entire menu printing and switch statement in a loop. The JSP graph will then be:



We complete the menu with still another option, X – Exit. As long as the user does not enter X, the loop proceeds. Furthermore we also clean the screen before the menu is displayed, which is the first operation of the loop. Here is the code:

```
#include <iostream.h>
#include <stdlib.h>
void main()
{
  char cSel, temp;
  do
  {
    system("cls");
    cout << "    Menu" << endl <<"    ===" << endl;
    cout << "A. Order" << endl;
    cout << "B. Invoice" << endl;
    cout << "C. Warehouse" << endl;
    cout << "D. Finance" << endl;
    cout << "X. Exit" << endl;
    cout << "Select: ";
    cin >> cSel;
    switch (cSel)
    {
      case 'A':
        cout << "You selected Order";
        break;
      case 'B':
        cout << "You selected Invoice";
        break;
      case 'C':
        cout << "You selected Warehouse";
        break;
      case 'D':
        cout << "You selected Finance";
        break;
      case 'X':
        cout << "You selected to exit";
        break;
      default:
        cout << "Erroneous choice";
        break;
    }
    cout << endl << "Press a key to continue";
```

```
    cin >> temp;
  }while (cSel != 'X');
}
```

To be able to clear the screen we need the header file stdlib.h.

W use a do loop, where the condition is checked *after* the loop to ensure that the loop is run at least once.

The first action in the loop is to clear the screen with system("cls"). Then the menu is printed and the user is prompted for an option, i.e. a character to be stored in the variable cSel. That variable is checked in the switch statement, where a text is printed corresponding the selected option. If the user enters 'X', the text 'You selected to exit' will be printed, and the loop is terminated since the while condition specifies that cSel must not equal 'X'.

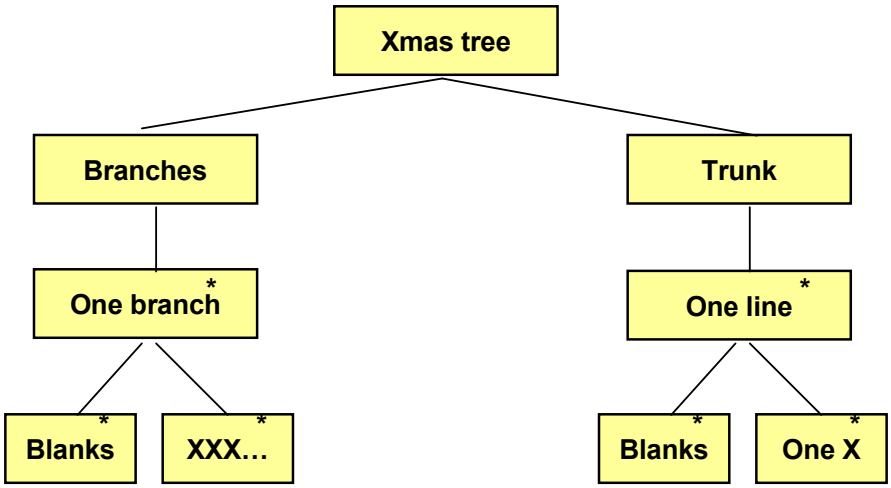
The method of letting the user enter an extra character to the variable temp at the end of the loop is a relatively unconvenient solution, but it has the advantage of avoiding to struggle with special C++ features regarding input, which we don't go into here.

5.5 Christmas Tree

We will now create a logically rather complex program that uses char variables to print a number of 'X' on the screen with the shape of a Christmas tree:

```
              X
             XXX
            XXXXX
           XXXXXXX
          XXXXXXXXX
         XXXXXXXXXXX
        XXXXXXXXXXXXX
       XXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXX
 XXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXX
              X
              X
```

As you can see the tree has eleven branches and two 'X':s to the trunk. We therefore need an outer loop that runs eleven times, where each turn prints a branch. Each branch consists of a number of 'X':s, different depending on which branch being printed. Furthermore we will have to print a suitable number of blanks before the 'X':s, so that the branches are centered symmetrically around the middle trunk. We therefore need two inner loops, one that prints the leading blanks and one that prints the 'X':s for each branch. After completion of the branches we need a loop that runs two turns and that prints the 'X':s for the trunk. We start with a JSP graph:



The JSP graph tells us that there is one outer loop for the branches, where each turn of the loop prints a branch. We then have one inner loop that prints the correct number of blanks and one inner loop that prints the 'X':s. The same is true for the trunk where we have an outer loop where each turn of the loop prints one line of the trunk, and one inner loop that prints the blanks before one single 'X' is printed. The code is as follows:

```
#include <iostream.h>
void main()
{
   int i, j;
   char x = 'X', blank = ' ';
   for (i=10; i>=0; i--)  //first outer for-loop
   {
      for (j=0; j<i; j++)  //first inner for-loop
         cout << blank;
      cout << x;
      for (j=0; j<10-i; j++)  //second inner for-loop
         cout << x << x;
      cout << endl;
   }
   for (i=0; i<2; i++)  //second outer for-loop
   {
      for (j=0; j<10; j++)  //inner for-loop
         cout << blank;
      cout << x << endl;
   }
}
```

We declare a variable named x that correspond to the 'X', and a variable named blank corresponding to the blank character.

The first outer for-loop has an i as loop counter and goes from 10 to 0 (11 turns). We have selected to let the values run backwards to make the subsequent math easier. i is consequently a line counter for the branches of the tree.

The first inner loop has a j as loop counter and goes from 0 to i. It prints the correct number of blanks for each branch. Since i is counted reversed, the number of blanks will decrease for each branch. Each turn of the loop prints one blank.

After completion of the blanks for a bransch, first an 'X' is printed. Since the number of 'X':s at each bransch is odd, the remaining number of 'X':s to be printed is even.

The second inner for-loop prints the 'X':s with two 'X':s for each turn of the loop. We again use j as loop counter, which goes from 0 to 10-i. This means that according to the decrease of i, the number of printed 'X':s increases. And since two 'X':s are printed for each turn of the loop, there will always be an even number of 'X':s.

The second outer for-loop, which builds the trunk of the tree, has i as loop counter and goes from 0 to 1 (two turns). For each of the two turns, there is an inner for-loop that goes from 0 to 9 and that prints 10 blanks. After the inner loop, an 'X' is printed.

This complex algorithm probably requires some thinking. You could preferably write a scheme of how i and j changes their values, and in that way follow the progress of the program.

5.6 int - char

Each character has an internal code of the integer type. E.g. the character A has the code 65, B has 66 tec. Therefore the data types integer and char can cooperate. Here is an example:

```
int iNo = 65;  //The code for A
char cChr;
cChr = iNo;
cout << cChr;
```

First we declare an integer variable, iNo, and give it the value 65. On the second line we declare a char bariable named cChr. On the third line cChr gets the same value as iNo, i.e. 65. When we then print cChr on the fourth line, the program understands that it is a char variable being printed, so the character corresponding to the code 65 will be printed, namely A.

5.7 Å, Ä, Ö

As you probably has guessed, the character codes follows the alphabetic order, i.e. A=65, B=66, C=67 etc. The Swedish characters Å, Ä and Ö don't follow that pattern. Therefore you can use the hexadecimal codes for these characters:

```
'\x86'   å
'\x84'   ä
'\x94'   ö
'\x8F'   Å
'\x8E'   Ä
'\x99'   Ö
```

The characters \x indicate that a hexadecimal value will follow. If you for instance want to print the word 'från', write as follows:

```
cout << "fr\x86n";
```

If you want to assign the char variable cChr the value Ö, write this:

```
cChr = '\x99';
```

5.8 String Array, char[]

A limitation with a char variable is that it can only store one single character. Many times you want to store a longer text in a variable like a customer name or a product description. Then you will need a string array. A string array works in the same way as any other array, i.e. it has a variable name and an index value that indicates an item of the array, i.e. a specific character of the string.

Below we declare a string array called cName, which can hold up to 30 characters:

```
char cName[30];
```

The indeces can be any of 0-29.

Suppose we want the user to enter a name to the array cName:

```
cout << "Enter your name: ";
cin >> cName;
```

Let's say that the user enters 'John'. These four characters are then stored in the cName array, where the first item cName[0] contains the character 'J', the second item cName[1] the character 'o', cName[2] the character 'h' and cName[3] the character 'n'.

In addition, an extra character is always stored as the last character of a string. It is the so-called null character '\0', which is stored in the fifth position in the item cName[4].

The program should now print what the user entered:

```
cout << "Your name is " << cName;
```

In our case the text 'Your name is John' will be printed. Note that when an entire string array is printed, you don't need to specify any indeces in the cout statement. The procedure by cout is to print one character after the other in the string array from the first position until the null character is found.

Suppose that the user in the program section above enters his entire name 'John Smith'. When the cout statement is executed, still only the text 'Your name is John' is printed. The reason to this is that when text is entered from the keyboard, only characters up to the first blank will be stored in the array. This is a limitation of the cin statement.

A solution to this problem is to use another input function instead of cin:

```
char cName[30];
cout << "Enter your name: ";
cin.getline(cName,29);
cout << "Your name is " << cName;
```

Here we use the function cin.getline() instead of only cin. This has the effect that, if the user enters 'John Smith', the printout from the cout statement will be 'Your name is John Smith'. Blanks consequently work in the correct way. The input is *not* interrupted at the first blank, but the entire text line entered by the user will be stored in the variable cName.

The function getline() must have two parameters in the parenthesis; the string array to receive the entered text, and the maximum number of characters allowed to receive. The reason for using 29 is to allocate space for the null character at the end of the string as the 30th character. If the user enters less than 30 characters, say 14, the null character will be stored in position 15. If the user enters more than 29 characters, only the first 29 are accepted.

5.9 Length of a String

We will now create a little program that calculates the length of a string, i.e. the number of characters contained in the string. For that purpose we use the function strlen():

```
#include <iostream.h>
#include <string.h>
void main()
{
   char cName[30];
   cout << "Enter your name: ";
   cin.getline(cName,29);
   cout << "Your name is " << cName << endl;
   cout << "Your name is " << strlen(cName) <<
            " characters long" << endl;
   cout << "The entire string is " << sizeof cName <<
            " characters long" << endl;
}
```

To be able to use string functions like strlen() we must include the header file string.h.

In the program we first declare a string array called cName with 29 positions plus the null character, totally 30 characters. Then the user is prompted for his name, which is stored in the variable cName, and a confirmation is printed 'Your name is John Smith'.

In the next last cout statement the text 'Your name is 10 characters long' is printed, provided that we have used a name of 10 characters like 'John Smith'.

In the last cout statement the text 'The entire string is 30 characters long' is printed. Here we use the operator sizeof, which gives the declared length of the string array cName, irrespective of how many characters actually have been stored. Thus, note the difference between strlen() and sizeof.

5.10 Upper/Lower Case

We have previously mentioned that the character codes for the letters A-Z start with 65 and go upwards. This applies to the upper case characters. Lower case letters a-z have the same pattern starting with 97, i.e. 32 greater than the upper case letters. An upper case is consequently achieved by subtracting 32 from the character code of the lower case character.

We will now create a program that reads a text in lower case and converts it to upper case and prints it:

```
#include <iostream.h>
#include <string.h>
void main()
{
   int i;
   char cName[30];
   cout << "Enter your name in lower case: ";
   cin >> cName;
   int iLen = strlen(cName);
   for (i=0; i<iLen; i++)
      cout << (char)(cName[i] - 32);
   cout << endl;
}
```

As usual in string handling we must include the header file string.h.

We declare the variable i to be used as loop counter, and a string array named cName with space for 29 characters. The user is then prompted for his name in lower case, which is stored in the array cName.

We then calculate the length of the input string, which is stored in the variable iLen. This value is used as loop limit in the subsequent loop, which performs as many turns as there are characters in the input string.

Inside the for-loop we take the i^{th} character from the string cName (cName[i]) and decrease it by 32. Since this is a math operation, the program understands that it is the character code for the i^{th} character that is decreased by 32. This value is then type cast to char by placing char within parenthesis in front of the subtraction. Then the program understands that it is the corresponding character that is to be printed.

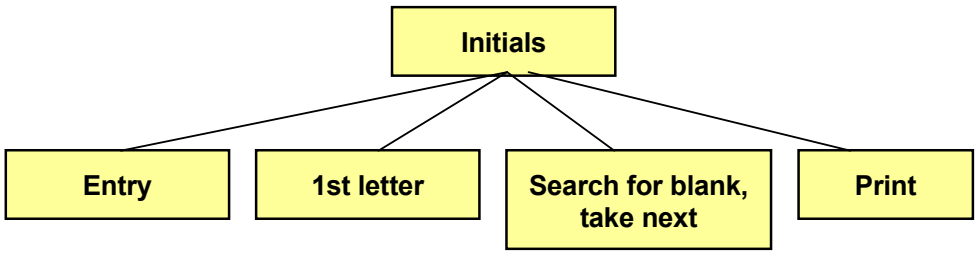
Thus, we have taken character by character from the input string, all in lower case, decreased the character code by 32, which gives the corresponding upper case character, and printed it.

5.11 Initials

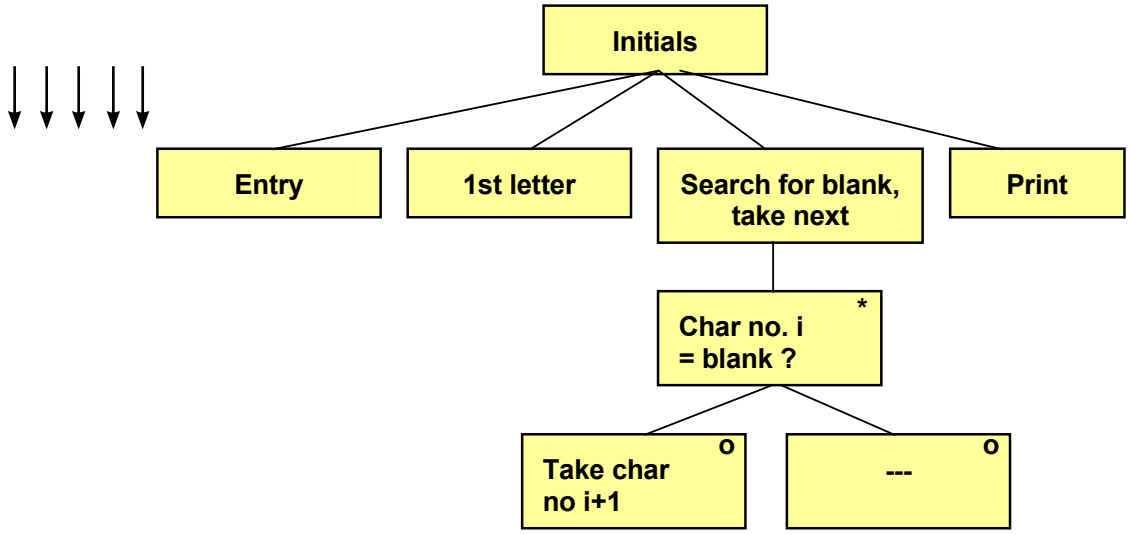
In most computer systems the programming work is to a large extent focused on checking and processing texts entered by users. Therefore, we will study some text processing examples..

In our next program example we will extract the initials from a name entered by the user. The process is to extract the first letter from the name. Then we search for the blank between the first name and the surname. In the next position the first letter of the surname is found.

We start with a JSP graph:



To search for the blank implies taking one character at a time from the beginning of the input name and check if it is a blank. We detail the 'Search for blank, take next' box:



Searching for the blank is made in a loop. For each turn of the loop we check if character no. i is blank. If so, we take character no. i+1 as the second initial. Here is the code:

```
#include <iostream.h>
#include <string.h>
void main()
{
   int i;
   char cName[30], cInit[3];
   cout << "Enter your name: ";
   cin.getline(cName,29);
   int iLen = strlen(cName);
   cInit[0] = cName[0];
   for (i=1; i<iLen; i++)
     if (cName[i] == ' ')
       cInit[1] = cName[i+1];
```

```
   cInit[2] = '\0';
   cout << "Your initials are " << cInit << endl;
}
```

We declare the string array cName, which is to contain the input name, and cInit to contain the initials. The reason for declaring 3 positions for cInit is to make space for the 2 initials plus the null character.

The user enters his name to the array cName, and its length is calculated and stored in the variable iLen.

The first character of cName (cName[0]) is stored in the first position of cInit.

Then we have the loop with the loop counter i, which represents the index in the input string, and goes from position 2 (index=1) to the end of the string. For each turn of the loop we check if the character is a blank. If so, we copy the next character (cName[i+1]) to the second position of cInit.

At completion of the loop we put the null character in the third position of cInit and print cInit.

5.12 Comparing Two Strings

In the Arrays chapter you learnt that, when comparing two arrays, you must compare each item of the arrays. For strings there is a special function, strcmp(), that can compare two string arrays. The function

```
strcmp(str1, str2)
```

- gives a negative result if str1 < str2, i.e. if str1 comes before str2 in alphabetic order

- gives 0 if str1 = str2
- gives a positive result if str1 > str2, i.e. if str1 comes after str2 in alphabetic order

We will now create a little program that prompts the user for two names and prints them in alphabetic order:

```
#include <iostream.h>
#include <string.h>
void main()
{
   char cName1[30], cName2[30];
   cout << "Enter a name: ";
   cin >> cName1;
   cout << "Enter another name: ";
   cin >> cName2;
   if (strcmp(cName1,cName2) < 0)
      cout << cName1 << endl << cName2;
   else
      cout << cName2 << endl << cName1;
   cout << endl;
}
```

First we declare two string arrays, cName1 and cName2, which can contain max 29 characters each (pos 30 is for the null character). The user enters two names. A name must not contain blanks. How would you do to allow blanks?

The if statement checks if cName1 comes before cName2 in alphabetic order. If so, cName1 is printed first and then cName2. Otherwise the names are printed in reversed order.

5.13 Copying Strings

In the Arrays chapter we copied an array's values to another array by copying each item singly. For strings there is a special function, strcpy(), that copies the entire string to another string. If for instance the string array str2 contains the string 'John Smith' and we execute the statement:

```
strcpy(str1, str2);
```

then str1 will also contain the string 'John Smith'.

5.14 Array with String Arrays

Suppose that we want to store a number of names in an array, where each name is itself a string array. Then we will need a two-dimensional array. Below we declare a two-dimensional array with space for 5 names with 30 positions each (29 characters plus the null character):

```
char cNames[5][30];
```

This two-dimensional string array could be represented like this:

	0	1	2	3	4	5	…
0	J	o	h	n	\0		
1	E	d	w	a	r	d	\0
2	B	o	b	\0			
3	E	v	E	\0			
4	A	d	a	m	\0		

To print one of the names, e.g. the third name (indes=2), we code:

```
cout << cNames[2];
```

Note that when we print or read a sequence of characters to a string array, we don't need to specify the index. The program prints the characters from the beginning of the string until the null character is found. However, for a two-dimensional array, we must specify which of the names to be printed, i.e. we must specify the first index. In the example above we used indes=2, which gives the printout 'Bob'.

To print a single character from the matrix we must use both indeces. The statement:

```
cout << cNames[1][4];
```

prints the character with index 4 from the name with index 1, i.e. 'r' in 'Edward'.

5.15 Sorting Strings

We want to sort the string matrix above, i.e. the names should be reorganized into alphabetic order. Do you remember the sorting algorithm from the Arrays chapter? If no, check it out. There we compared the items in pairs and interchanged their positions if the right item was less than the left. The logic is the same for string arrays, but we will have to use our special string functions to be able to compare and copy strings.

We first prompt the user for five names to be stored in the string matrix. Then we sort the strings in a double loop, and finally we print the sorted list of names:

```
#include <iostream.h>
#include <string.h>
void main()
{
  char cNames[5][30], temp[30];
  cout << "Enter 5 names. Press Enter after each" << endl;
  for (int i=0; i<5; i++)
    cin >> cNames[i];
  for (i=0;i<4;i++)
    for (int j=i+1; j<5; j++)
```

```
        if (strcmp(cNames[i],cNames[j]) > 0)
        {
          strcpy(temp,cNames[i]);
          strcpy(cNames[i],cNames[j]);
          strcpy(cNames[j],temp);
        }
   cout << endl << "Sorted names:" << endl;
   for (i=0; i<5; i++)
     cout << cNames[i] << endl;
}
```

First we declare a two-dimensional string array named cNames, and a string array named temp, which we will use for the triangular exchange inside the double loop.

The first for-loop reads five names to the string matrix cNames.

Then we have the double loop. The outer for-loop goes from 0 to 3 (next last name) with the loop counter i. The inner for-loop has the loop counter j which goes from 1 greater than i to 4 (last name).

The if statement compares the two names indicated by the indeces i and j. If the “left” name (i) is greater than the “right” (j), then cNames[i] comes after cNames[j] in alphabetic order, and they should interchange their positions.

The interchange is made by means of strcpy() by copying cNames[i] to the temporary string array temp. Then we copy the name in position j to position i. Finally we copy the temp name (the old name in position i) to position j. The names now have been interchanged in the string matrix.

At completion of the double loop, the names have been sorted in alphabetic order and we can with the last for-loop print one name at a time from position 0 to 4.

5.16 Substring

Sometimes you will need to extract a number of characters from a string. Then we will get use of the strncpy() function, which copies a given number of characters from one string to another. Example:

```
strncpy(cStr1, cStr2, iNo);
```

This statement copies the number of characters given by the variable iNo counted from the beginning of the string cStr2 and stores this substring in cStr1.

If you want to copy a number of characters from a given position and not from the beginning of cStr2, use this variant:

```
strncpy(cStr1, cStr2 + i, iNo);
```

This statement copies the number of characters given by the variable iNo counted from position i of cStr2 and stores this substring in cStr1.

Here we for the first time get in touch with the similarity between strings and **pointers**. A pointer is a variable that points to a certain memory address. When we write cStr2, it is interpreted as a pointer that points to the string starting with the first character of cStr2, i.e. with index=0. If we write cStr2+1, it is interpreted as a pointer that points to the character with index=1 in cStr2. If we write cStr2+i, it is interpreted as a pointer that points to the character with index=i of cStr2. Enough with pointers for this time.

We will get use of this function in subsequent program examples.

5.17 Concatenating Strings

It is possible to add strings together (concatenate) with the strcat() function. The statement

```
strcat(cStr1, cStr2);
```

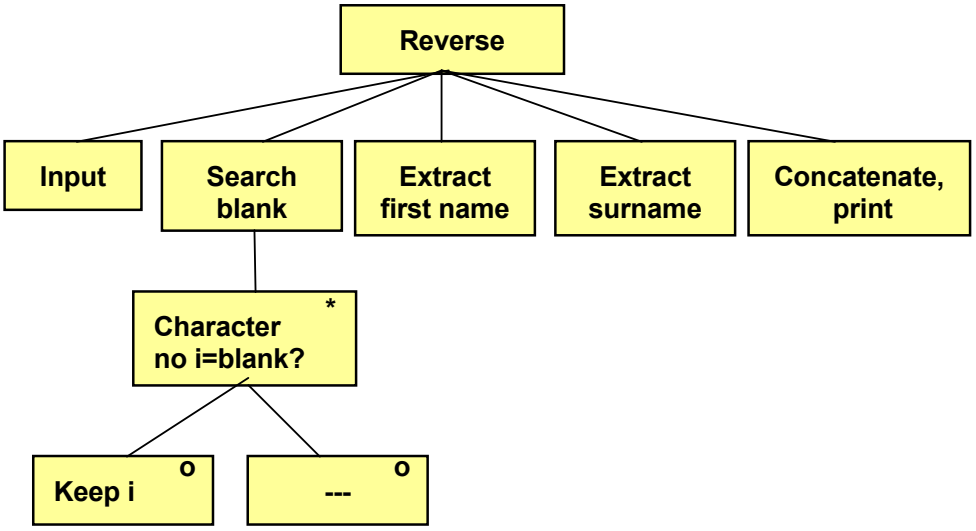
concatenates cStr1 and cStr2 and puts the result in cStr1, i.e. cStr2 is added at the end of cStr1.

We will get use of this function in subsequent program examples.

5.18 Interchanging First Name and Surname

We will now create a program that prompts the user for his first name and surname separated by a blank. The program will then interchange them so that the surname is printed first.

The way of doing this is to search for the position of the blank. With the strncpy() function we can then separate the first name and surname, and then with the strcat() function put the substrings together with the surname first. We start with a JSP graph:



First we read the name from the user input. Then we search for the position of the blank by looping through the characters and checking if any of them is blank. When a blank is found we save its position.

By means of the position of the blank we can now extract the two substrings consisting of first name and surname. Finally we put them together in reversed order and print them.

Here is the code:

```
#include <iostream.h>
#include <string.h>
void main()
{
   char cName[30], cFirst[15], cSur[30];
   cout << "Enter your name: " << endl;
   cin.getline(cName,29);
   for (int i=0; i<29; i++)
     if (cName[i] == ' ')
       break;
   strncpy(cFirst,cName,i);
   cFirst[i] = '\0';
   strcpy(cSur, cName + i + 1);
   strcat(cSur, " ");
   strcat(cSur, cFirst);
   cout << cSur << endl;
   cout << strrev(cSur) << endl; //reversed name
}
```

First we declare a string array called cName which will contain the name entered by the user (both first and surname). The string array cFirst will be used for the first name and the string array cSur to the surname. The reason for using 30 positions instead of 15 for the surname is that we will concatenate the final result in that string array.

The user is prompted for his name to be stored in the array cName.

The for-loop searchees for the blank. It goes from 0 to 28, i.e. it takes character by character from the name. If a character equals blank, we interrupt the loop with break, which has the effect that the loop counter i is not incremented any more. If the user enters 'John Smith', i will have the value 4 when the loop is interrupted, and that is the position of the blank.

The function strncpy() then copies the first i characters from cName to cFirst. In the case of John Smith the first 4 characters will be copied, i.e. 'John'.

The string cFirst is completed with a null character as the fifth character (index=4).

Then we use the function strcpy() to copy the characters from cName to cSur starting on position i+1. cName+i+1 is interpreted as a pointer that points to i+2nd character of cName. In the case of John Smith i is =4. cName+i+1 consequently points to the string starting in position with index 5, i.e. 'Smith'. Note that the copy is performed character by character until the null character is found.

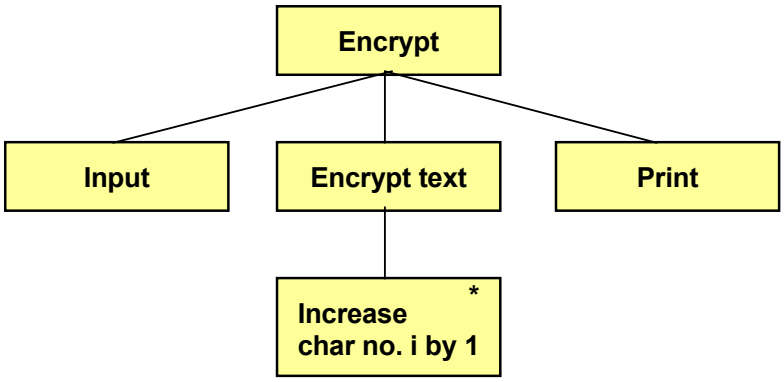
Then we use the strcat() function to concatenate the surname with the string ' ', i.e. we add a blank to the surname. Then we concatenate surname + blank with the first name. The string cSur now contains surname + blank + first name, which is printed.

The last statement has actually nothing to do with our task. We have only included it for curiosity. It prints the entire name in reversed order. The function strrev() reverses a string.

5.19 Encryption

We will now write a program that encrypts a text, i.e. distorts it to unreadability. The way of doing this is to take character by character of the text entered by the user and increase the character code by 1. A becomes B, B becomes C etc. This is a very simple encryption algorithm, but gives a hint about how different encryption algorithms work when encrypting mail and other information.

We start with a JSP graph:



The user is first prompted for a text. The encryption takes character by character from the string array and increases the character code by 1. Finally we print the encrypted message. Here is the code:

```
#include <iostream.h>
#include <string.h>
void main()
{
   int iLen;
   char cName[30], cEncrypt[30];
   cout << "Enter your name: " << endl;
   cin.getline(cName,29);
   iLen = strlen(cName);
   for (int i=0; i<iLen; i++)
     cEncrypt[i] = cName[i] + 1;
   cEncrypt[iLen] = '\0';
   cout << "Encrypted: " << cEncrypt << endl;
}
```

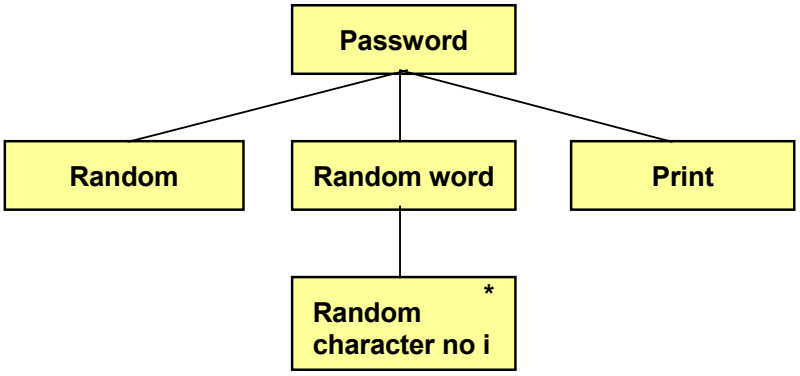
We declare the string array cName to be used for the entered name, and cEncrypt to be used for the encrypted text. The user is prompted for his name, which is stored in the string array cName. The length of the text is calculated and stored in the variable iLen.

The for-loop goes from the first to the last character of the entered string and increases the character code for each character with 1. The character is stored in the string array cEncrypt.

As the last character we insert the null character at the correct position in cEncrypt, which then is printed.

5.20 Random Password

Passwords should, as you probably know, be difficult to guess to prevent unauthorized access to a computer system. A good way of creating passwords is to let the computer randomly create them. We will write a program that makes the computer able to create passwords only consisting of upper case letters (A-Z). This is of course a limitation, but indicates the logic to be followed. You can then improve it to include lower case letters and digits. We start with a JSP graph:



We begin with creating a random length of the password, which we in this example will limit to between 5-7 characters. Then we create as many characters in the interval A-Z (character codes 65-90). The password is then printed. Here is the code:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
void main()
{
   int iLen, i;
   char cPw[30];
   srand(time(0));
   iLen = rand() % 3 + 5;
   for (i=0; i<iLen; i++)
      cPw[i] = rand() % 26 + 65;
   cPw[iLen] = '\0';
   cout << "Password: " << cPw << endl;
}
```

We declare the loop variable i, the variable iLen to contain the length of the password, and the string array cPw to contain the random password. We have declared 30 characters, but 9 would suffice in this example.

The random number generator is initiated with the function srand(). Then we create the random length of the password with the modulus operator %, which gives a length in the interval 5-7. rand()%3 gives a random number in the interval 0-2. By adding 5 we transform the interval to 5-7. The number is stored in the variable iLen.

The for-loop then goes from 0 to iLen-1, i.e. as many turns as the number of letters in the password. For each turn of the loop a random number in the interval 65 to 65+25 is created, which corresponds to the character codes for A-Z. Each character code is stored in the string array cPw at position i, i.e. cPw is built up with one more character for each turn of the loop. At the end of cPw we add the null character. cPw is then printed.

5.21 Translation Table

We will now show another method of encrypting a message, namely by means of a translation table. It consists of a set of the characters A-Z, very well mixed up. Thus, the characters are not in alphabetic order. We use the random number generator to mix the characters.

Suppose we have got the following translation table:

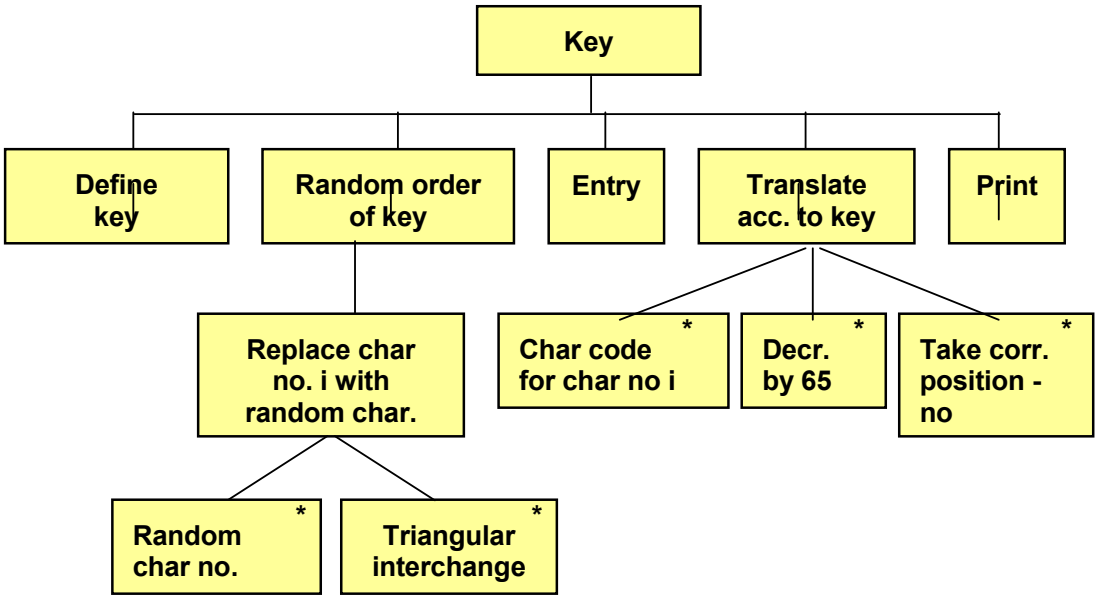
```
DGZCW...
```

where D is in position 0, G in position 1 etc. We use the table to translate a A to D, B to G etc. each character in the original message will consequently be translated to another character:

```
ABCDE...
```

DGZCW...

The JSP graph below shows the logic:



'Define key' means that we define a string containing the characters A-Z.

'Random order of key' implies that we mix the characters of the key, which is done by looping through the characters and create a random position for another character of the key. These two characters will then interchange their positions. In that way all characters will interchange its position with some other character.

The user is then prompted for the message in upper case.

'Translate acc. to key' is made by taking the character code for one character at a time in the entered message and decreasing the code by 65. A will then get the value 0, B the value 1 etc. This value gives the index in the key for the character to substitute the original character.

If we use the key:

```
DGZCW...
```

like in the introductory example, D has index 0 in the key, G index 1 etc. If the original message contains a B (character code 66), it is decreased by 65 to the value 1. This value is used as index in the key, which gives the character G.

Here is the code:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
void main()
{
```

```
   int i, j, iLen;
   char cTemp, cText[50], cEncrypt[50];
   char cKey[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   srand(time(0));
   for (i=0; i<26; i++)
   {
      j = rand() % 26;
      cTemp = cKey[i];
      cKey[i] = cKey[j];
      cKey[j] = cTemp;
   }
   cout << "Write a text in upper case: " << endl;
   cin.getline(cText,49);
   iLen = strlen(cText);
   for (i=0; i<iLen; i++)
      cEncrypt[i] = cKey[cText[i] - 65];
   cEncrypt[iLen] = '\0';
   cout << "Encrypted: " << cEncrypt << endl;
}
```

First we declare some supporting variables; i, j and iLen, the latter being used for the length of the entered message. The string array cText will store the entered text, and cEncrypt for the encrypted message. The string array cKey is initated with the characters A-Z. Then the random number generator is initiated.

The first for-loop performs the mixing of the characters of the array cKey. The loop counter i goes from 0 to 25 and points to character by character in cKey. A j value is randomly created to be used as index to the character to be interchanged with the i-character. Then the triangular exchange of these two characters is performed. At completion of the loop all characters in cKey has been interchanged with some other character, which makes cKey now be well-mixed.

The user is prompted for a message that is stored in the string array cText. The length of the message is calculated and stored in the variable iLen.

The second for-loop performs the translation to an encrypted message. The loop counter i goes from 0 to iLen-1, i.e. as many turns as there are characters in the input string. In the statement

```
cEncrypt[i] = cKey[cText[i] - 65];
```

a character is taken from cText (cText[i]).. This value is decreased by 65, and since it is a math operation, the result is of integer type. If the character is B (character code 66), the calculation 66-65 is performed which gives 1. This value is now used as index in cKey and gives the character from cKey that corresponds to the original position of B, for instance G. This character is stored in the string array cEncrypt at position i. The array cEncrypt is thus built up by one character at a time from the array cText, translated via cKey. At completion of the loop all characters in cText have been translated via cKey and been stored in cEncrypt.

After the last character in cEncrypt we add the null character. The array cEncrypt containing the encrypted message is then printed.

5.22 Summary

In this chapter we have introduced the char data type. A char variable can contain only one character. We have also learnt how to use string arrays or string variables to store texts consisting of more than one character.

We have also shown how to use character codes to manipulate texts, for instance to convert between upper and lower case.

Blanks is a problem at entry of texts, which is solved by using the function getline().

The header file string.h contains a number of nice functions for text management, for instance to calculated the length of a text, copy texts, concatenate texts and extract substrings of texts. We have also created a primary relation to pointers, which we will return to more in detail in a later chapter.

We have also worked through a number of programming examples to extract initials from a name, sort a list of names, where we got use of two-dimensional arrays (matrices), interchange first name and surname, encrypt messages and create random passwords.

5.23 Exercises

1. Start from the menu program earlier in this chapter and add an extra option to the menu:
   `E. Statistics`
   Complete the switch statement with a suitable text.

2. Write a menu program where you by means of entry of a character can select to get either your name, address, postal address or telephone number printed. It should be performed with a loop so that you repeatedly can select from the menu until you enter X to exit the program.
3. Write a program that asks for a character and then prints the character 10 times at the same line.
4. Change the previous program so that the program prints the character once at the first line, twice at the second line etc. to 10 times at the tenth line.
5. Write a program that prompts the user for a character and an integer. The program should then print the character as many times as given by the entered integer.
6. Change the Christmas Tree program in this chapter so that it prints 15 branches instead of 11.
7. Write a program that prompts the user for a character and prints the corresponding character code.
8. Complete the previous program with entry of a character code and a printout of the corresponding character.
9. Write a program that prints the text 'Här går vi över ån efter vatten' with the Swedish characters å, ä and ö in a correct way.
10. Write a program that reads a text from the keyboard and prints the length of the text as well as the entire declared size of the string variable.
11. Start with the Upper/Lower Case program and change it so that it reads a text in upper case and prints it in lower case. What happens if you enter other characters. Explain why the printout is the way it is.
12. Write a program that prompts the user for a word and prints each character doubled (twice).
13. Write a program that prompts the user for a word and prints it reversed, first by using the strrev() function, and then without strrev().
14. Start from the Initials program in this chapter and complete it so that the user can enter first name, middle name and surname and get all three initials printed.
15. Complete the previous program so that, if you enter the names with leading lower case characters, the printout will still be with the initials in upper case.
16. Write a program that prompts the user for two characters and prints all characters in between. For instance, if you enter F and K, the printout should be FGHIJK.
17. Write a program that prompts the user for two words and prints the one first in alphabetic order.
18. Write a program that prompts the user for a word in lower case. The program should respond with the number of vowels in the word, Don't include å, ä or ö.
19. The Robbery language is spoken by doubling each consonant and putting an 'o' in between. For instance the word 'bug' becomes 'bobugog'. Write a program that reads a word and prints it in the Robbery language.
20. Write a program where the user can enter 6 product names. Sort the list and print it.
21. Complete the previous program so that the user also can enter the prices of the 6 products. The printout should contain the correct price for each product, right-aligned with two decimals.
22. Write a program where the user can enter his first name and surname. The program should then print only the surname.
23. Write a program where the user can enter his e-mail address. The program should then check whether there is an @ in the address.
24. Complete the previous program so that, if there is no @, the text '@htu.se' is added and the address is printed.
25. Suppose that a person enters his e-mail address with a period between the first name and the surname, e.g.: john.smith@htu.se

Write a program that reads an e-mail address and prints the name in the correct way, e.g.:
John Smith
with upper case first character in both first and surname and with a blank in between.

26. Start with the Encryption program in this chapter. Change so that B becomes A, C becomes B etc.
27. Change the previous program so that a character is replaced by the one 3 positions earlier in the alphabet.
28. Write a program that decrypts instead of encrypts according to the method in the previous exercise.
29. Write a program that encrypts a text by reversing the alphabet, i.e. A becomes Z, B becomes Y etc.
30. Start with the Random Password program in this chapter. Complete it with also taking lower case characters and digits into account.
31. Change the previous program so that the password will be 6-10 characters in size.
32. Originate from the Translation Table program in this chapter. Complete it with also taking lower case characters into account.
33. Use your imagination to construct an own encryption algorithm and write a program for it.

6 Functions

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.

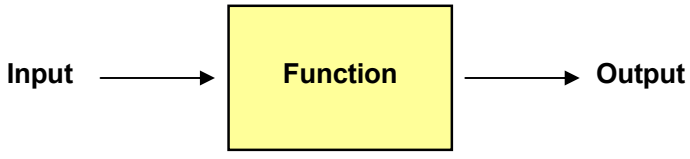
To summarize, the advantages with functions are:

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

In this chapter we will learn to write functions, supply input values (parameters) and receive the result from a function. We will learn how to declare and define functions and we will study how to use header files in connection with functions. We will also learn about reference parameters – an efficient tool to save memory and improve program performance. We will also get in touch with recursive functions, i.e. functions calling themselves.

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.

6.3 Average

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.

6.4 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.

We must *call* the function.

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 dNo1 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.

6.5 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)
{
```

```
   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.

6.6 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:

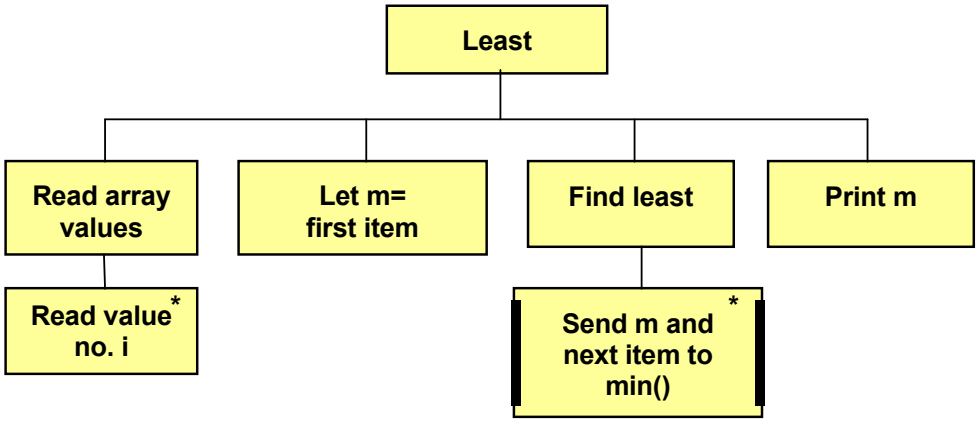
```
cout << min(min(a,b),c) << " is the least";
```

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.

6.7 Least Item of an Array

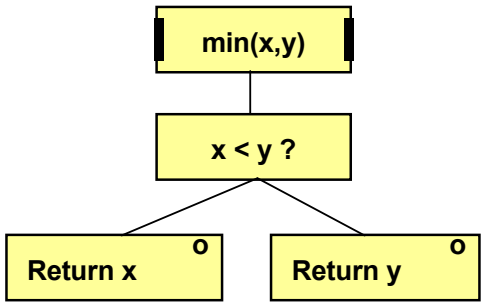
We will now use the min() function in a program that finds the least of the items of an integer array. The logic is similar to that of the previous program. The first two items of the array are sent to min(), which returns the least of these two integers. This integer and the next item of the array is again sent to min() which returns the least of them. This process is repeated until all items of the array have been processed. As a result we will get the least of the array items.

First we give a JSP graph:



Entry of values is made in a loop. Then we set the variable m equal to the first item of the array. In the loop 'Find least' we send m and the next item to the function min(). The return value is stored in the variable m. Since we use m to store the returned value, m will all the time contain the least of the items compared so far. At completion of the loop we print the m value, which now is the least of all array items.

We have used thicker border lines to the box with the function call to indicate that it is a function. We create a separate JSP graph for the function:



As shown by the JSP the function takes two parameters, x and y, and checks whether x is less than y. If so, x is returned, otherwise y.

Here is the program code:

```
#include <iostream.h>
int min(int x, int y)
{
```

```
   if (x<y)
      return x;
   else
      return y;
}
void main()
{
   int iNos[5], i, m;
   for (i=0; i<=4; i++)
   {
      cout << "Enter integer no. " << i+1 << ": ";
      cin >> iNos[i];
   }
   m=iNos[0];
   for (i=1; i<=4; i++)
   {
      m=min(iNos[i],m);
   }
   cout << m << " is the least integer" << endl;
}
```

In main() we declare the array iNos with 5 items. The first for-loop reads the integers from the user to the array. Then we assign the first array item to the variable m.

The second for-loop calls the function min() repeatedly with m and next array item as actual parameters. The returned value is stored in m. Finally we print the value of m.

6.8 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;
}
```

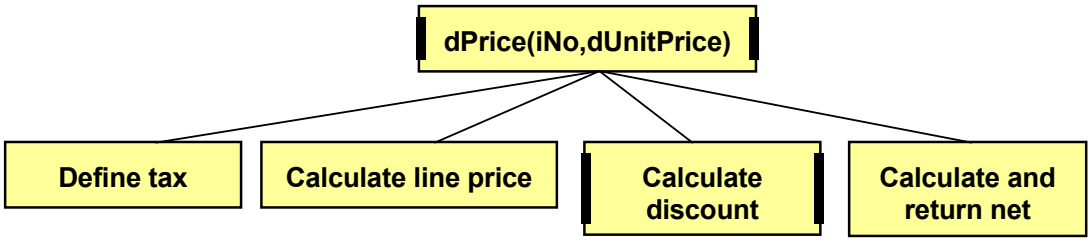
In main() we declare the string array cWord with max 30 characters, and the integer variable iLen, which later will contain the string length. A text is read from the keyboard. Then the function iLgth() is called with the entered string as actual parameter. The function returns the length of the string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

6.9 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.

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():

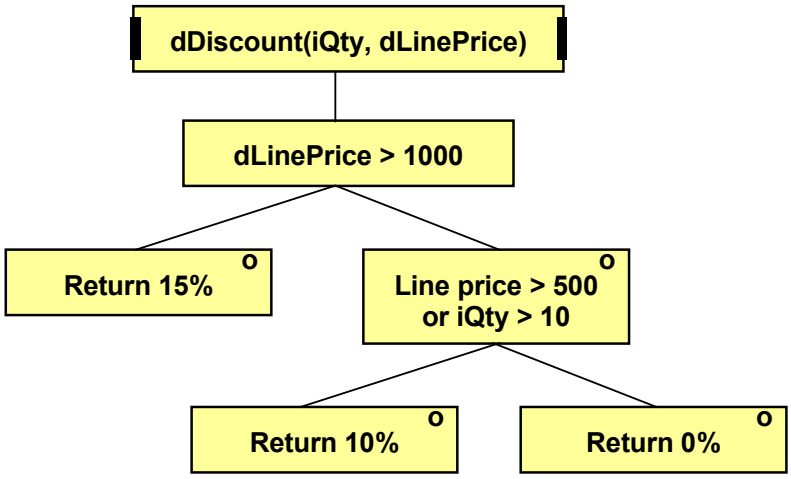


The function takes two parameters, number of units and the unit price.

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.

6.10 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.

6.11 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 iw replaced by cny. 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
```

6.12 Declaration Space

A variable declared inside a function is only valid within the function. The same applies to the formal parameters. In the previous program you can for instance not use the variable s outside the replace() function, like printing s from main().

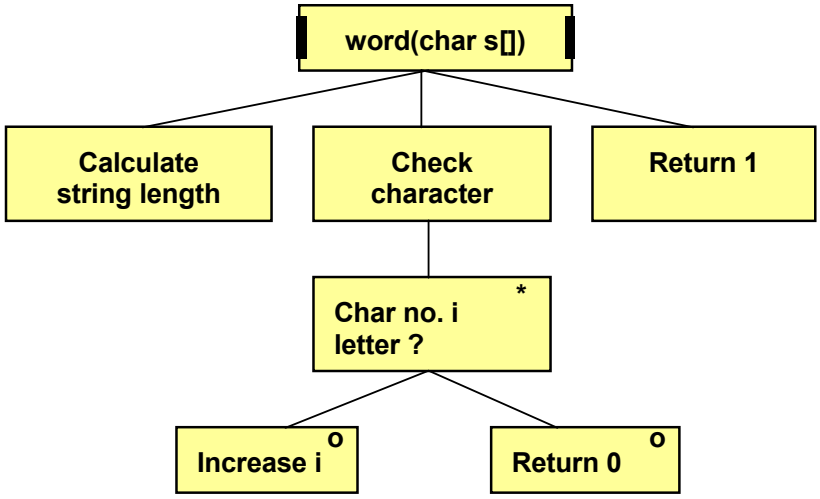
You can neither use a variable in replace() that is declared in main(). For instance, you can't print the variable a from inside of replace().

A variable is valid only in the function where it is declared.

There are however workarounds, but that is beyond the scope of this course.

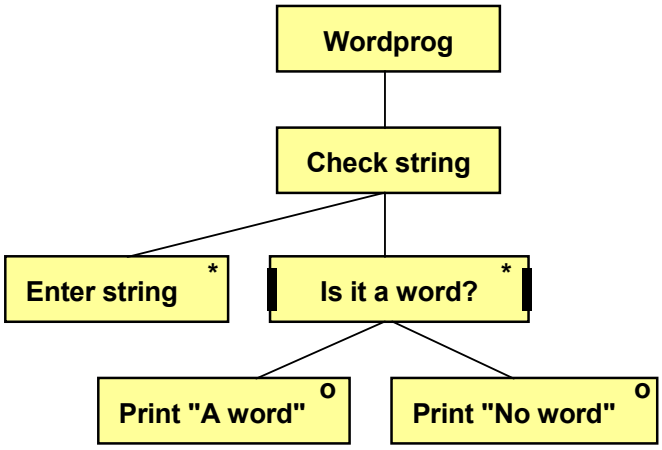
6.13 The Word Program

We will now create a function which checks if a string is a word, i.e. only contains the characters a-z or A-Z. The function should be used in a program where the usere repeatedly can enter a word and get it checked. We start with a JSP graph for the function:



The function begins by calculating the string length. The loop 'Check character' goes through character by character and checks if it is in the interval a-z or A-Z. If so, we increase the loop counter i by 1. Otherwise we return the value 0 and the function is terminated. If all characters are letters, the loop will complete and we return 1.

We now give a JSP graph for the program calling the word() function:



The program has a loop 'Check string'. For each turn of the loop we read a string from the keyboard. The string is sent to the function word(). If 1 is returned by the function, we print 'A word'. If 0 is returned, we print 'No word'.

Here is the code:

```
#include <iostream.h>
#include <string.h>
int word (char s[])
{
   int i=0, j;
   j = strlen(s);
   while (i<j)
   {
      char c = s[i];
      if ((c>='a' && c<='z') || (c>='A' && c<='Z'))
         i++;
      else
         return 0;
   }
   return 1;
}
void main()
{
   char str[10];
   while (cin >> str)
      if (word(str))
         cout << "A word" << endl;
      else
         cout << "No word" << endl;
}
```

The function word() begins by calculating the length of the string s and storing it in the variable j.

The while loop has the condition that the loop counter i should be less than the string length j. Character number i is copied to the char variable c.

The if statement checks if c is greater than 'a' and less than 'z' or greater than 'A' and less than 'Z'. If this condition is satisfied, the character is a letter and i is increased by 1. If it is another character, 0 is returned and the function is terminated.

If the while loop is allowed to complete, all characters are letters and the value 1 is returned.

In main() the string array str is declared. The while loop has the condition of a successful string entry, i.e. the user has not pressed Ctrl-Z.

The if statement in main() has a call to word() and sends the entered string as actual parameter. If the function returns 1 (it is a word), the condition is true and the text 'A word' is printed. If 0 is returned, the condition is false and the else statement provides the output 'No word'.

6.14 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.

6.15 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 recodgnized 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.

6.16 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++.

6.16.1 Project

When working with several files in this way, you must **create a project** in Visual C++ and add all files to the project. Do as follows:

- Select File - New and indicate the Projects tab.
- Mark the option Win32 Console Application and enter a name of the project in the box Project name. Click OK.
- A window is displayed. Click Finish.
- A confirmation window is displayed. Click OK.

When **creating a new cpp file**, do as follows:

- Select File - New and indicate the Files tab.
- Mark the option C++ Source File and enter a name of the cpp file. Click OK.
- The code window is displayed. Enter your code and click the Save button.
- Add the file to the project by selecting Project - Add To Project - Files. Mark the file and click OK.

When **creating a new header file**, do as follows:

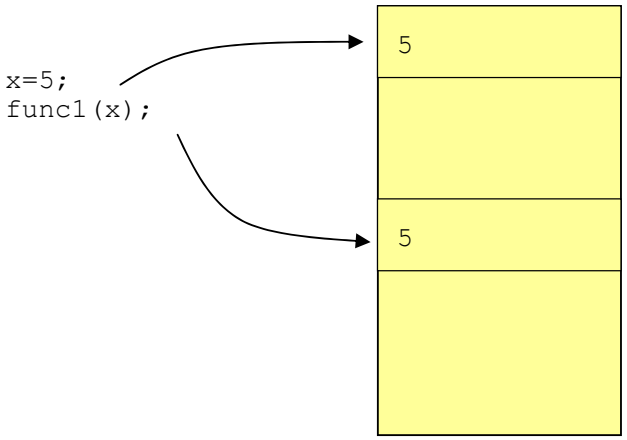
- Select File - New and indicate the Files folder.
- Mark the option C/C++ Header File and enter a name of the header file (the same name as the corresponding cpp file). Click OK.
- The code window is displayed. Enter your code and click the Save button.
- Add the file to the project by selecting Project - Add To Project - Files. Mark the file and click OK.

Compile the entire project by clicking the Build button (the same button as used for compilation so far).

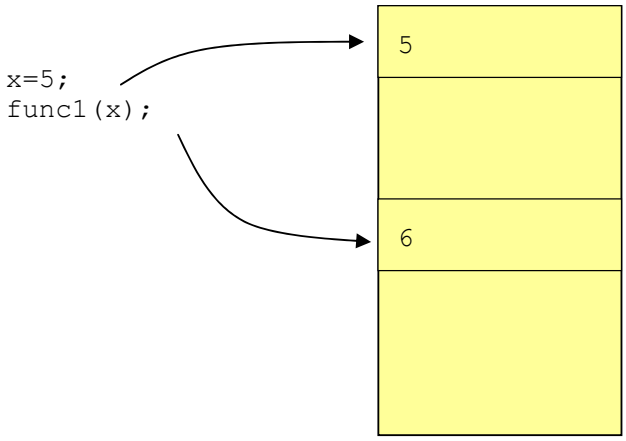
Run the program by clicking the Execute Program button (the same button as used for running a program so far).

6.17 Reference Parameters

When calling a function with actual parameters, the value for the acutal parameter is copied from its memory area to another memory area used by the function's formal parameter:



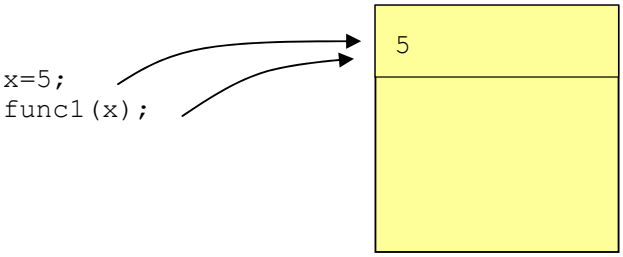
The variable x has the value 5 stored in a memory location. When func1() is called, that value is copied to another memory location used by the function. If a statement in the function would change the value to 6, the memory location of the function is affected, but not the original value of x:



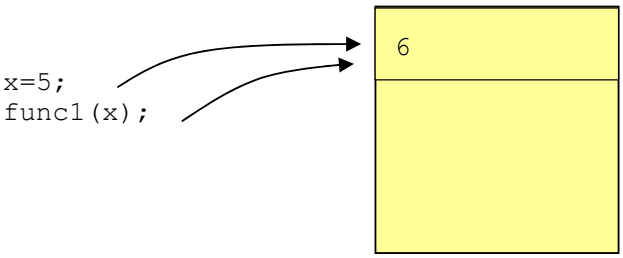
Sometimes this is good, but sometimes you also want the original value to be changed.

Another disadvantage is when a lot of data is to be transferred to the function, e.g. at object oriented programming when an object consisting of many Mbyte of data is to be transferred to a function. A lot of memory is then consumed and it takes time to copy huge amounts of data.

The solution is to use a reference parameter. No copying of data is then made, but the original actual parameter and the function's formal parameter point to the same memory location:



If the function func1() changes the value, it also affects the value of the original variable x:



Defining a function parameter as reference parameter is made by placing an & character after the data type:

```
void underline(int& n)
{
   //…
}
```

The parameter n is here a reference parameter.

You call the function in the usual way:

```
underline(iNo);
```

6.18 Parameters with Default Values

Sometimes it would be convenient to exclude an actual parameter when calling a function. The function must then itself be capable of assigning a value to the formal parameter. This is accomplished by defining the formal parameter with a default value, i.e. if no value is supplied by the function call, the formal parameter gets a standard value. Here is an example:

```
void print_many (char c, int iNo=1)
{
   for (int i=1; i<=iNo; i++)
      cout << c;
}
```

The function print_many() prints a character a specified number of times. It takes a character c and an integer iNo as parameters. The character c is printed iNo times.

The for-loop goes from 1 to iNo and prints c for each turn of the loop.

The parameter iNo has the default value 1, which means that if no integer value is sent to the function, iNo will get the value 1. Thus, you specify the default value in the function header:

```
int iNo=1
```

The call to the function can be in two different ways. Here is one:

```
print_many ('x', 4);
```

This call sends the character 'x' and the number 4 to the function. Since we specify a value in the call, the default value for the parameter iNo will be ignored and the value 4 will be used. The output will be:

```
xxxx
```

Here is the other way of calling the function:

```
print_many ('y');
```

Here, we don't send any integer value, so the default value 1 will be used. The character 'y' will be printed once:

```
y
```

One thing you should remember when using a function declaration first and a function definition later, is that the default value for a parameter should only be specified in the declaration of the function, and not be repeated in the function header of the function definition. This is the way it should be written:

```
void print_many (char c, int iNo=1);
void main()
{
   //…
}
void print_many (char c, int iNo)
{
   //…
}
```

Furthermore, the parameter with the default value must be the last one in the parameter list. You can't interchange the parameters c and iNo.

6.19 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.

6.20 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.

6.21 Exercises

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. Save this program, we will use it in later exercises.
11. Write a program for calculation of car rent. The program should contain a function that calculates the daily charge (500:-) plus kilometer charge (1:40 per km) plus the fuel price. The calculation of the kilometerage is made by a subfunction which prompts the user for start and end odometer value and returns the number of kilometers driven. The fuel price is calculated by another subfunction which reads the fuel consumption and returns the fuel charge (9.27 per litre). Save this program, we will use it in later exercises.
12. Suppose you want to create a function that prints the letters å, ä and ö correctly. The function takes a string as parameter, searches for the letter combinations aa, ae and oe, end replaces them by å, ä and ö respectively. In main() you read a string from the user and call the function. To make it work, the user must enter the word 'båda' as 'baada'.
13. Start from the 'Word program' earlier in this chapter. Change it to a digit program, i.e. the program should check an entered string to only contain digits and decimal point.
14. Write a "playing card" function which takes a card value (1-13) and prints the correct value (2-10, 'jack', 'queen', 'king', 'Ace'). Use the function in a main() program.
15. Improve the previous function to also take a colour parameter (1-4) and prints 'hearts', 'clubs', 'diamonds' or 'spades'.

16. Write the following boolean functions:
    - bool odd(int n) which gives the value true if n is odd.
    - bool divable(int a, int b) which returns true if a is evenly dividable by b.
    - bool digit(char c[]) which returns true if the first character of c is a digit.
    - bool letter(char c[]) which returns true if the first character of c is a letter.

    Use the functions in if statements which print the results.
17. Write a program which works as a calculator. There should be one main() function which reads the calculation, for instance 5 * 3, and four functions, one for each type of calculation + - * /.
18. Write a program that calculates the average score for a student. The program should prompt the user for course scores (MVG=20, VG=15, G=10, IG=0) and number of hours that the course comprises. These two values are multiplied. The entry goes on until all courses are complete. The course scores of all courses are added. The sum is divided by the total number of hours for all courses, which gives the average score. The transfer between score (for instance VG) to value (15) is made by a function.
19. Change the price calculation program in exercise 10 above, so that you get two override functions for discount calculation with the same function names. One of them takes the total price and quantity as parameters and the other takes a customer group as parameter. The entry of customer group from the user must then be made in the dPrice() function.
20. Modify the previous program so that you place the function declarations first and the function definitions after main().
21. Start from exercise 11 and put all code in a project with the function definitions in a separate cpp file, the function declarations in corresponding header file and main() in a separate cpp file.
22. Modify the previous exercise so that the parameters of the functions are used as reference parameters.
23. Change the function for calculation of the fuel price in the previous exercise so that it takes the litre price as parameter. It should have the default value 9.32.
24. Write a function which creates a number of random rolls of a dice. The number of rolls should be taken as a parameter with the default value 5. The function should return the average of the rolls. Use this function in a program where the user can enter a number of rolls and get the average printed. Store the program in a project with separate cpp files for the function and main() and a header file with the function declaration.
25. Expand the previous exercise so that the program runs the function using the default value, i.e. without sending the number of rolls to the function.
26. Improve the previous exercise so that you play against the computer and the program informs you about who won. The comparison between your and the program's score should be made by a function.
27. Change the previous exercise so that the function uses reference parameters.
28. Start from the function nfac() in the 'Recursive Functions' section and place it in a program where the user can enter the integer and get the faculty of it printed.
29. Write a recursive function which sums the integers n, n-1, n-2,… 1. Use it in a main() program.
30. Change the previous function so that it every other time adds and subtracts, for instance 6-5+4-3+2-1.

7 Files

7.1 Introduction

You have probably noticed when running our programs during the course so far, each run has started from the same origin as previous run. The data entered to the program is gone when running the program the next time. This is of course unacceptable. We must have the possibility to save data entered or calculated during one run, so we can continue where we stopped last time. The solution to this is to save the data on disk in files.

In this chapter we will go through the basic concepts about file management and how to read from and write to files in C++.

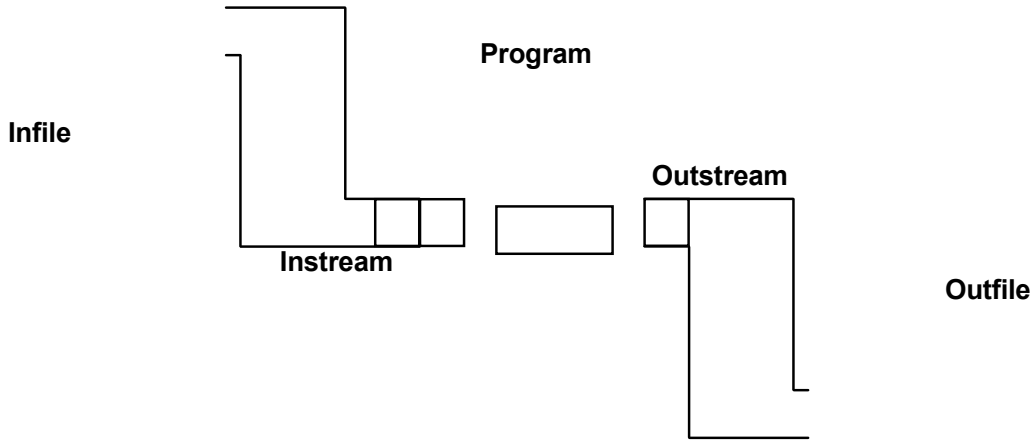
In professional programming relational databases of some kind are mostly used for data storage. A lot of special code is however required in C++ to do that, which is outside the scope of this course. Here, we will only store data in the simpliest format, namely text files which can be read and updated with a simple text editor like the Notepad program.

To be able to handle files in C++ we need some knowledge about streams, which we will first go through in this chapter. We will then show how to declare a file, open it, save data in it, read from it and close it.

You should normally open the file as late as possible in the program and close it as early as possible, since you want to minimize the risk for loss of data by keeping the files open as short time as possible. If the system would break down while a file is open, the result might be a corrupt file not possible to read from. You will then have to return to the last backup of the file.

7.2 Streams

When files are processed in C++ the communication goes between hard disk file and program via a stream:



When data is to be read from a disk file (infile) to a program, it goes via an intermediary store (instream) which works as a buffer between the hard disk and the program, where data is queued to be read to the program.

Similarly, when data is to be written from the program to a disk file (outfile), it is first stored in an intermediary store (outstream) before it is finally written to the file.

As programmer you only have to bother about reading from the instream and writing to the outstream. The operating system takes care of the physical reading and writing on the disk file.

7.3 Reading from a Stream

So far, you have read data from the keyboard with statements like:

```
cin >> cName;
```

We have said that cin stands for 'console in', i.e. reading from the keyboard. But actually, the data has been transferred via an instream called cin.

The same applies to reading from an instream. Suppose that we have an instream called is and that it is connected to a particular disk file. Then we read data from the instream with statements like:

```
is >> cName;
is >> dAmount;
```

Reading data from an instream with the >> operator is called **'formatted input'**, because the data from the instream is automatically accommodated to the data type of the receiving variable.

In some situations it is not possible to accommodate the data to a specific data type, for instance if you try to read letters to an integer variable. A run-time error will then occur.

You can also use **'unformatted input'**, which means that characters are read from the file exactly as they are stored, without any accommodation. Here is an example:

```
char cName[30];
is.getline(cName, 29);
```

The last statement reads up to 29 characters from the instream, and the null character is put after the last read character. The read operation continues until the end line character is reached. Suppose the data in the file is stored linewise (for instance if data has been entered using Notepad and Enter has been pressed after each line). Then one line at a time is read.

If there happens to be fewer characters than 29 at the current line in the file, for instance 17 characters, the null character is stored in the 18th position.

If there are more than 29 characters at the line in question in the file, the input is interrupted after 29 characters.

Thus, the programmer must carefully check how data is stored in the file, so as not to loose important information.

7.4 Writing to a Stream

Writing data to an outstream can also be done in two ways:

Formatted output is done with statements like:

```
os << cName;
```

Here we presume that an outstream named os has been created and been connected to a specific disk file. The statement implies that the characters in the variable cName are written to the outstream.

Formatted output also implies that you have the opportunity to control the layout of the data, for instance with the function width(). Compare the 'Variables' chapter, where we discussed formatted output.

Unformatted output means that the characters in the variable are written to the outstream exactly in the format they are stored in the variable, for instance:

```
os.put(c);
os.write(cName, 30);
```

The put() function prints a character, namely the character represented by the variable c, to the outstream os. The write() function writes 30 characters from the variable cName to the outstream os.

7.5 Attaching a File to a Stream

Before being able to use an instream or outstream, it must be declared and attached to a disk file. The statement:

```
ifstream infile("address.txt");
```

declares the instream infile and attaches it to the disk file named address.txt.

ifstream is the short for 'input file stream'. You can regard ifstream as a data type similarly to integer or double. But actually, ifstream is a class from which we derive an object of the ifstream type with the object name infile.

When this statement has been executed the stream is ready for read operations.

Below we declare an outstream:

```
ofstream outfile("newadr.txt");
```

The stream is called outfile and is connected to a disk file named newadr.txt. ofstream is short for 'output file stream'. ofstream is also a class from which we create the object outfile.

At completion of this statement the outstream is ready for write operations.

If the file newadr.txt exists, it will be deleted and a new file with the same name is created. Many times you want to add data to the end of an existing file without destroying existing information. The outstream is then declared as follows:

```
ofstream outfile("newadr.txt", ios::app);
```

app is short for 'append'.

To make the stream declarations above work you must *include* the file fstream.h:

```
#include <fstream.h>
```

To allow for reading from and writing to streams, you must as usual include the file iostream.h:

```
#include <iostream.h>
```

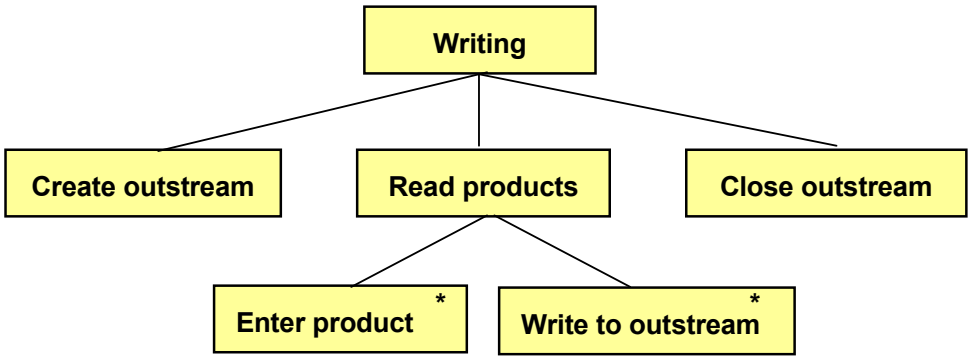
At completion of reading from or writing to the streams, you *close the streams*::

```
infile.close();
outfile.close();
```

7.6 A Complete Write Program

To summarize our experiences we will now create a simple program for writing of data to file. The program will read product names from the user (keyboard) and store them in a file named prodfile.txt

We begin with a JSP graph:



First we declare an outstream and attach it the the file prodfile.txt. Entry of product names from the keyboard is made in a loop. As soon as a product name has been entered by the user, it is written to the outstream. At entry completion, we close the outstream.

Here is the code:

```
#include<iostream.h>
#include<fstream.h>
void main()
{
   char cProd[30] = "";
   ofstream outfile("prodfile.txt");
   cout << endl << "Enter product, (only Enter to exit): ";
   cin.getline(cProd,29);
   while(cProd[0]!='\0')
   {
      outfile << cProd << endl;
      cout << endl << "Enter product: ";
      cin.getline(cProd,29);
   }
   outfile.close();
}
```

First we include the two header files iostream.h (to allow for input and output) and fstream.h (to allow for stream management).

In main() we declare the string variable cProd used for storage of product names in the program. Then we declare the outstream outfile and attach it to the disk file prodfile.txt.

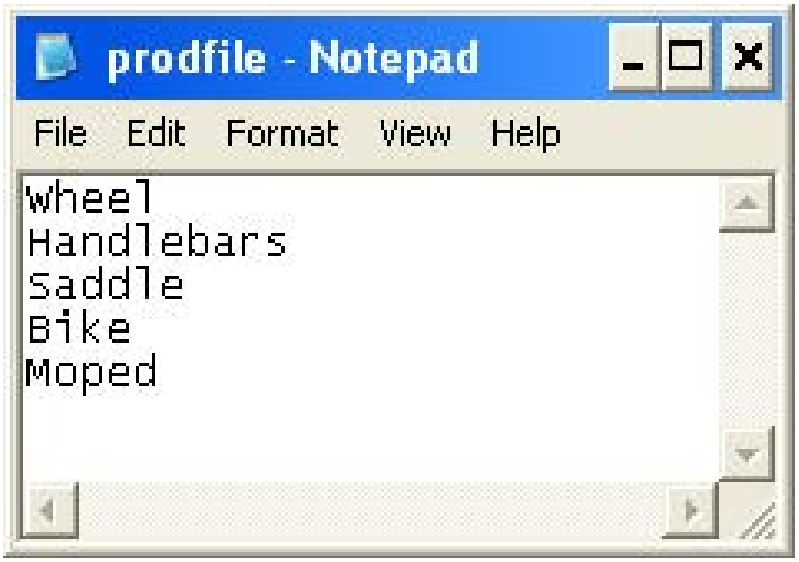
Entry and output is made by first reading the first product from the keyboard with the cin.getline() function before the while loop starts. Since the while condition checkst the variable cProd to actually hold a string, the string variable cProd must contain a string.

The while condition checks that the first character of the string variable cProd (cProd[0]) is not the null character. If it were, the user would have pressed Enter without having entered any product name, and the loop is then terminated.

The first statement in the loop prints the product name to the outstream outfile. The two subsequent statements read a new product name from the user.

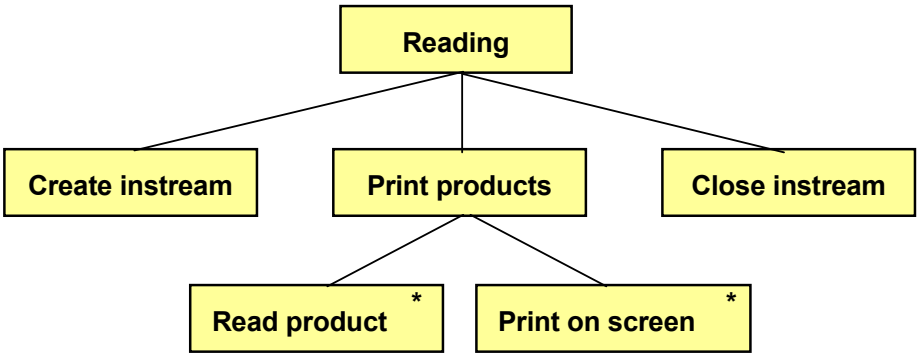
The loop is terminated when the user presses Enter without entering any product name. The outstream is then closed and the file operation is complete. The file prodfile.txt now contains a number of product names.

When having run the program you would probably like to examine the result. Start 'Explore' and find the file prodfile.txt, which is in the project folder where the cpp file is saved, or maybe in the 'Debug' subfolder, depending on your Visual Studio settings. Double-click the file to make the Notepad program be started and the file content be shown:



7.7 A Complete Reading Program

We will now create a new program that reads data from prodfile.txt and prints the information on the screen. We start with a JSP graph:



First we declare the instream and attach it the the prodfile.txt file. Reading and printing on the screen is made in a loop where we read one product at a time from the instream and print it on the screen. At completion, we close the instream.

Here is the program:

```
#include<iostream.h>
#include<fstream.h>
void main()
{
   char cProd[30] = "";
   ifstream infile("prodfile.txt");
   while(infile.getline(cProd,29))
      cout << cProd << endl;
   infile.close();
}
```

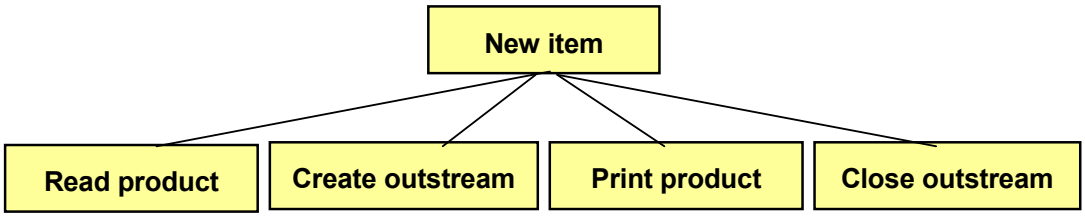
We include the same header files as in the previous progarm.

In main() we declare the string variable cProd used for holding product names in the program. Then we declare the instream infile and attach it to the disk file prodfile.txt.

The while loop has the condition of a successful reading from the instream. If so, the read product is printed on the screen. When there is no more data in the file, the read operation is unsuccessful and the loop is terminated. The instream is closed.

7.8 New Item at the End of the File

We will now show how to add one more product at the end of the file prodfile.txt. The solution is given by the following JSP graph:



This program reads only one more product. But, by placing the input from the keyboard and printing to the outstream in a loop, you could make the program more flexible to allow for entry of any number of products.

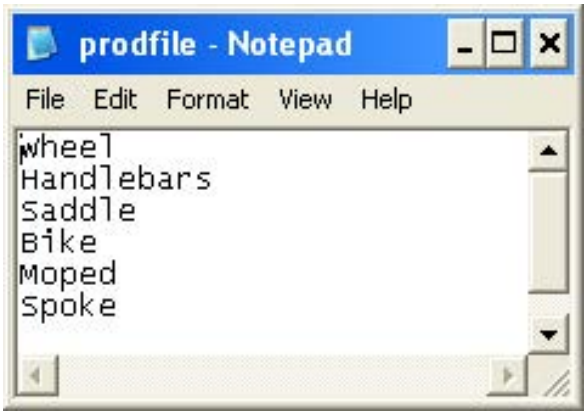
First we read the product name from the user, then we create the outstream and attach it to the file prodfile.txt, print the product to the outstream, and close the outstream.

Here is the program code:

```
#include<iostream.h>
#include<fstream.h>
void main()
{
   char cProd[30] = "";
   cout << "Enter new product: ";
   cin.getline(cProd,29);
   ofstream outfile("prodfile.txt",ios::app);
   outfile << cProd << endl;
   outfile.close();
}
```

We use the same include files as previously. In main() we prompt the user for a new product. Then we declare the outstream outfile and attach it to the disk file prodfile.txt. Note that we use ios::app to make existing data be kept and new data be added at the end of the file. Then we print the entered product to the outstream and close the outstream.

If you check the file prodfile.txt in Notepad, you will see one more product having been added at the end:



7.9 Products and Prices

We will now recreate the prodfile.txt to store both product id:s and prices for a number of products. The structure of the file will be:

Product id
Price
Product id
Price
etc.

The price of each product comes after the product id.

The program will be similar to the one used for writing to file:

```
#include<iostream.h>
#include<fstream.h>
void main()
{
   int iProdId = 1;
   double dPrice;
   ofstream outfile("prodfile.txt");
   while(iProdId !=0)
   {
      cout << endl << "Enter product id: ";
      cin >> iProdId;
      cout << " ...and price: ";
      cin >> dPrice;
      if(iProdId > 0)
         outfile << iProdId << endl << dPrice << endl;
   }
   outfile.close();
}
```

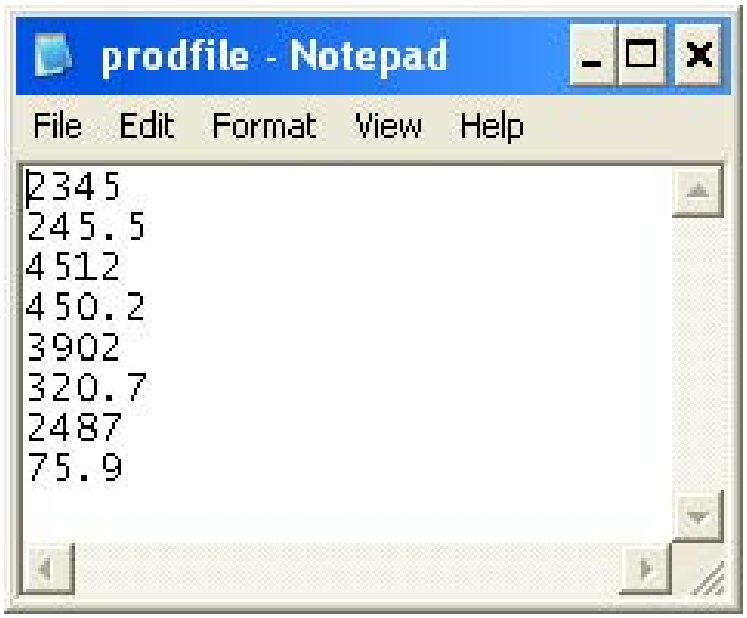
We use the same include files as previously.

In main() we declare the variable iProdId used for storage of the product id:s in the program. It is initialized to 1 for the sake of making the while loop start with a valid value of iProdId. The variable dPrice will hold the product prices. We also declare the outstream outfile and attach it to the disk file prodfile.txt.

The while loop reads product id:s and prices from the user and prints them to the outstream. The while condition checks that there is a valid product id different from zero. If so, the user is prompted for a product id and a price. If the product id is greater than zero, the information is written to the outstream outfile. Note that we also print endl after each value, which makes each information item be printed on a separate line. This way of storing data in a file facilitates printing and reading from a programming point of view. Avoid several values per line!

At completion of the loop the outstream is closed.

By looking at the file in Notepad, you can figure out the structure:

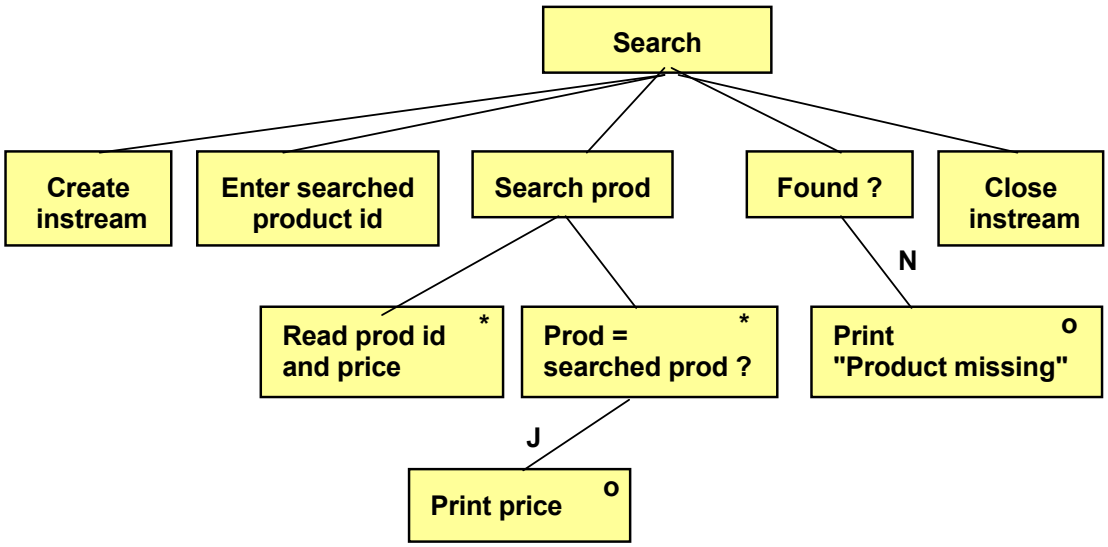


First there is the product id 2345 and then the price of that product 245.5. Then comes the next product etc.

7.10 Search for a Product Price

We will now use the new prodfile.txt to find the price of a product specified by the user. The laboursome thing about this kind of files is that we always must start reading from the beginning of the file until we find the correct product. Then we also easily can find the corresponding price.

First we give a JSP graph:



First we create an instream, and then the user is prompted for the searched product id.

The loop 'Search prod' reads one product id and price at a time from the instream. If it is the searched product id, the price is printed.

After the loop we check if the correct product id was found. If not, an error text is printed. Finally we close the instream.

Here is the program code:

```
#include <iostream.h>
#include <fstream.h>
void main()
{
   int iProdId, iSrch, iFound=0;
   double dPrice;
   ifstream infile("prodfile.txt");
   cout << "Enter product id: ";
   cin >> iSrch;
   while(infile >> iProdId >> dPrice)
   {
      if (iProdId == iSrch)
      {
         cout << "The price is " << dPrice;
```

```
        iFound=1;
        break;
      }
   }
   if (!iFound)
      cout << "Product missing";
   infile.close();
}
```

First in the program we declare the variables iProdId used for storage of the product id:s read from the instream, iSrch for the searched product id, iFound which is an indicator to remember whether or not the product id was found. The value 0 means that we have not found the correct product, and 1 means that we have found it. The variable dPrice is used for the price read from the instream.

Then the instream is created and attached to the disk file prodfile.txt.

The searched product id is read from the user and stored in the variable iSrch.

The while loop reads products and prices from the instream. The while condition reads one product id and the corresponding price from the instream. As long as there is data to be read, the loop continues.

When a product and a price has been read, the if statement checks if it equals the searched product id. If so, the price is printed, the variable iFound is set to 1 and the loop is terminated.

If the loop is allowed to complete, i.e. if all products have been read without finding the correct id, the variable iFound will still have the value 0.

After the loop the if statement checks if iFound still is 0. iFound=1 means 'true', iFound=0 means 'false', !iFound=1 (not found) means 'true'. Thus, if 'not found' is true, the error message about missing product is printed.

Finally the instream is closed.

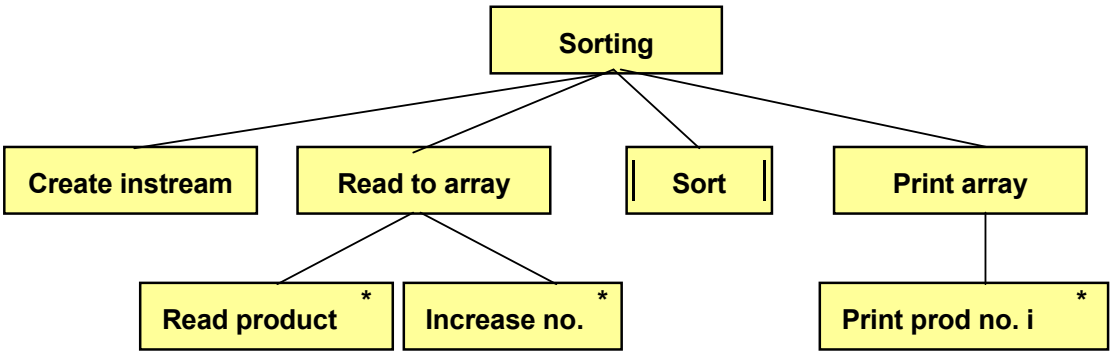
7.11 Sorting a File in Memory

We can't presume the products to be sorted in the file. But in a printout on the screen we want a sorted list of products. We will create a program which reads all products in the file to an array, sorts the array, and then prints the sorted array.

We now return to the first product file, namely the one only containing product names. You will probably with smaller amendments achieve the same result with the later file version.

The program will read all product names to an array (two-dimensional char array). The sorting is performed by a function. After the sorting by the function, the main() function will print the sorted list.

Here is first a JSP graph for the main() function:

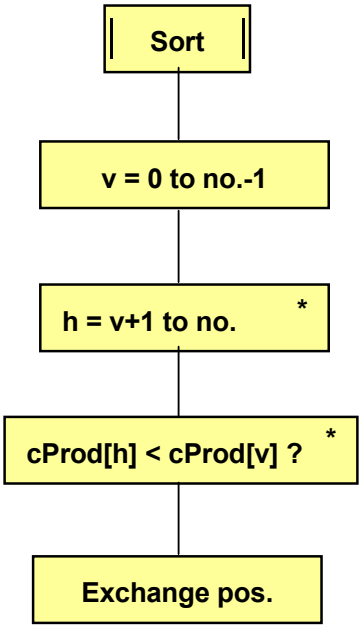


First we create the instream.

Reading of products to the array is made in a loop, where we increase the number of items for each single read. By doing so we keep track of the number of products read. This number is needed by the function Sort to be able to sort.

The printing of the sorted array is also made in a loop.

The JSP graph for the function Sort looks like this:



You probably recognize the sort algorithm from the Arrays chapter.

Here is the program code.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
void sort(char cList[][30], int n);
void main()
{
   int i=0, j, iNo;
   char cProd[50][30];
   ifstream infile("prodfile.txt");
   while(infile.getline(cProd[i],29))
      i++;
   iNo=i;
   infile.close();
   sort(cProd,iNo);
   for(j=0; j<iNo; j++)
      cout << cProd[j] << endl;
}
void sort(char cList[][30], int n)
{
   int v,h;
```

```
   char temp[30];
   for(v=0; v<n-1; v++)
   {
      for(h=v+1; h<n; h++)
         if(strcmp(cList[h],cList[v])<0)
         {
            strcpy(temp, cList[v]);
            strcpy(cList[v], cList[h]);
            strcpy(cList[h],temp);
         }
   }
}
```

The include files are iostream.h for input and output, fstream.h for streams and string.h for the string functions. Furthermore, we also declare the function sort().

In main() we declare the variable i, which is initialized to 0 and which will accumulate the number of products read, the variable j used as loop counter, and the variable iNo which finally stores the number of products. In addition, we declare the two-dimensional array cProd, which will be used for storage of the product names. We also create the instream infile, which is attached to the disk file prodfile.txt.

The while loop manages reading of product names to the array. The while condition is true as long as there is data to read from the instream. In the first turn of the loop i is = 0 according to the initiation in the beginning of the program. Therefore, the first product is stored in the item cProd[0]. The loop increases the value of i to 1, and the next product is stored in cProd[1] etc.

After the loop we save the value of i, i.e. the number of products read, in the variable iNo, and then we close the instream.

Then we call the function sort() and send the array and iNo. After the sort operation we print the array in the last for-loop.

The function sort() takes the array and number of products as parameters. Note that n is the index of the last item of the array.

In the function we declare v and h to be used as indeces in the array when two items are compared. We also declare the string array temp, which is used in the triangular exchange of array items.

The outer for-loop with v as loop counter goes from 0 to n-1, i.e. from the first to the next last position of the array. The inner for-loop goes from the position after v to the last position of the array.

Inside the inner for-loop we compare item h to item v by means of the function strcmp(), which gives a negative result if item h is less than item v. In that case the items will exchange positions, which is made in the triangular exchange by means of the string array temp.

At completion of the loop, the array has been sorted.

Remember that, when an array is sent as parameter to a function, it is always done as reference parameter, so the function operates on the same memory area as used by the array in main(). As a consequence, the array does not need to be returned from the function.

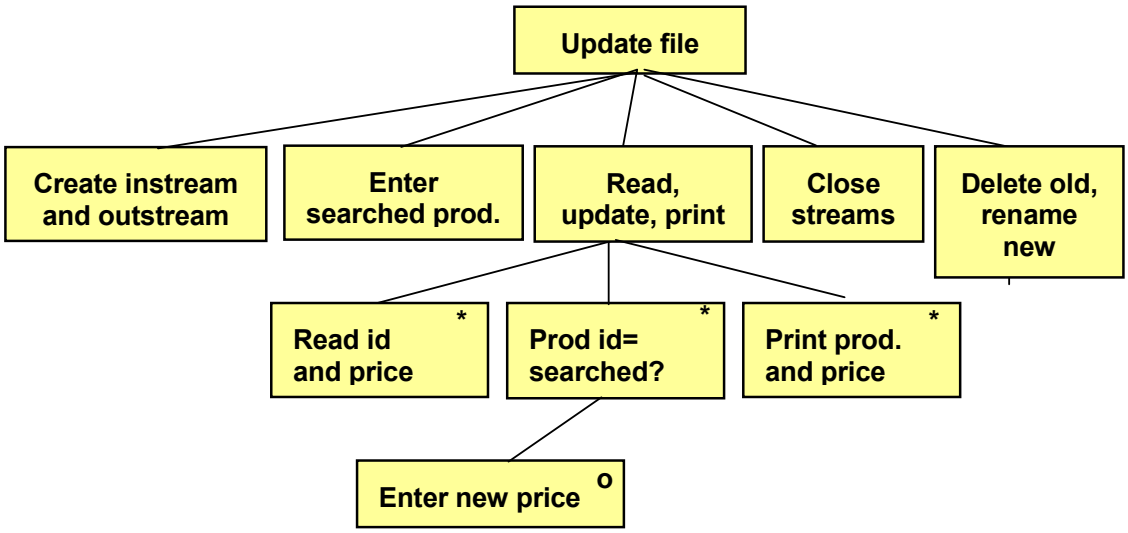
7.12 Updating File Content

Changing the content of a file of the type used in our programs is rather troublesome. The reason is that you can only read a file from start to end. You cannot jump into the requested position in the file and change information.

As a consequence you will have the original file as input file and a new file as output file. You read data from the infile and prints to the outfile. When arriving at the position in the file to be changed, after having read the input information, you change the value and print to the outfile. Then you will have to continue item by item from the infile and print to the outfile until all information has been transferred. Finally you delete the original file and change the name of the new file to equal the name of the original file. The information has then been updated.

We now presume that our product file contains product id:s and prices for each product. The user is prompted for a product id and a new price for that product.

We draw a JSP graph for this:



First we create the instream for the original file and the outstream for the new, which by now is empty. The user is then prompted for the product id to be updated.

Then we use a loop to read product id:s and prices from the original file. The product id is compared to the one entered by the user. If equal, the user is prompted for a new price, otherwise the old price will be used. The product id and price are then printed to the outstream.

The streams are closed and at the end of the program we delete the old file and rename the new file to the old name.

Here is the program code:

```
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
void main()
{
  int iSrch, iProdId;
  double dPrice;
  ifstream infile("prodfile.txt");
  ofstream outfile("temp.txt");
  cout << "Specify product id: ";
  cin >> iSrch;
  while(infile >> iProdId >> dPrice)
  {
    if (iProdId == iSrch)
    {
      cout << "Specify the new price: ";
      cin >> dPrice;
```

```
      }
      outfile << iProdId << endl << dPrice << endl;
   }
   infile.close();
   outfile.close();
   remove("prodfile.txt");
   rename("temp.txt", "prodfile.txt");
}
```

The include files are the usual ones, except that we also need stdio.h to be able to delete and rename files.

In main() we declare the variable iSrch to be used for the product id entered by the user, iProdId for product id:s read from the file, and dPrice for prices from the file.

Then we create the instream, which is attached to the original file prodfile.txt, and the outstream, which is attached to a new file, temp.txt. Then the user is prompted for the searched product id.

The while loop reads product id and price from the instream as long as there is data. Each product id is checked in the if statement against the product id specified by the user. If equal, the user is prompted for a new price, which is stored in the variable dPrice, i.e. the old price is replaced by the new one. Then the product id and price are written to the outstream. At completion of the while loop all products have been transferred to the new file and the requested price has been updated.

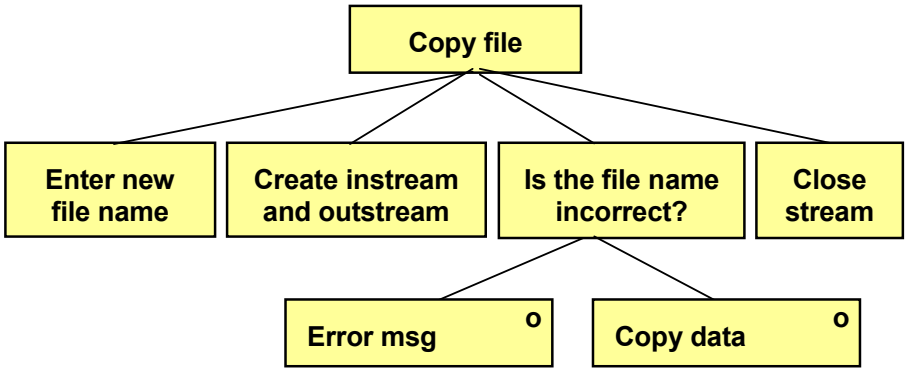
After the while loop the streams are closed, the old file is deleted by the remove() function and the new file temp.txt is renamed to prodfile.txt by the rename() function.

7.13 Copying Files

Copying a file could be done according to the same method as used by the previous program with the exception that no price is updated and that the original file is not deleted. We will however show a shortcut of copying a file with the file name specified by the user.

The copy of data is made by the function rdbuf(), which in one single operation reads all data from the original file without the need of picking item by item in a loop.

First we give a JSP graph:



First the user is prompted for the new file name. Then we create the instream for the original file and the outstream for the new file.

We then check if the outstream creation was successful. It might happen that the user enters characters not allowed in file names. If so, we would get a run time error. In case of an error, we print an error message. If, however, everything is OK, we copy all data. Finally we close the streams.

Here is the program code:

```
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
void main()
{
   char cNewName[12];
   cout << "Specify new file name: ";
   cin >> cNewName;
   ifstream infile("prodfile.txt");
   ofstream outfile(cNewName);
   if (!outfile)
   {
      cout << "The file could not be created";
   }
```

```
   else
   {
      outfile << infile.rdbuf();
   }
   outfile.close();
   infile.close();
}
```

The include files are iostream.h for input and output, fstream.h for stream management, and stdio.h to allow for using the function rdbuf().

In main() we prompt the user for the new file name, which is stored in the variable cNewName. Then we create the instream, which is attached to the disk file prodfile.txt, and the outstream, which is attached to a disk file with the user supplied name. Note that cNewName is not enclosed in quotes, since it is a variable and not a specific string.

The if statement checks if the outstream creation succeeded. If so, the variable outfile contains an address to the outfile object. If it didn't succeed, the address is = 0. That means that !outfile is true if the address is 0. In that case we print an error message to the user. Otherwise, i.e.if the outstream could be created, we use the function rdbuf() to copy all data in one single operation from the infile to the outfile.

Finally the streams are closed.

Having run the program you can by means of 'Explore' check the new file.

7.14 Summary

In this chapter we have learnt the basics of file management. You have learnt how to use streams and attach them to physical disk files. You have also learnt that you communicate with the streams, and not directly with the disk files.

We have discussed the meaning of formatted and unformatted input and output. You are now able to write programs where the user can enter information to be stored in a file, and read information from a file and present it on the screen.

We have also studied examples of how to search for information in a file, read and sort file information before presentation on the screen, update file information and copy files.

7.15 Exercises

1. Start with the program in the section 'A Complete Write Program'. Expand the program so that it is also possible to specify warehouse location (for instance EH23) for each product. Check with the Notepad program that the file contains the expected information.
2. Start with the program in the section 'A Complete Reading Program' and modify it so it also will be capable of reading the warehouse locations entered in the previous exercise.
3. Start with the program in the section 'New Item at the End of the File' and modify it so that you also can enter the warehouse location of the new product. Use the same file as in the two previous exercises. Then run the program in exercise and check the existence of the new product in the output.

4. Start with the program in the section 'Products and Prices' and modify it so that you also can enter quantity in stock for each product.
5. Start with the program in the section 'Search for a Product Price' and modify it so that also quantity in stock is printed on the screen. Use the same file as created in the previous exercise.
6. Start with the program in the section 'Sorting a File in Memory' and accommodate it to also be able to manage the file with product names and warehouse locations created in the first exercise.
7. Start with the program in the section 'Updating File Content' and modify it so that the user will be able to update the quantity in stock. Use the file with product id, price and quantity in stock created in a previous exercise.
8. Create a program where you can enter
   - first name
   - surname
   - city

   for some of your course mates. These should be saved in a file. The program should be possible to run several times while keeping existing file information.
9. Create a program which reads the course mate information from the file created in the previous exercise and prints it on the screen.
10. Create a program which can update the city of a person. The first and surname must then be entered from the keyboard. The new city should also be possible to specify.
11. Create a program which can remove a person from the file. The first and surname of the person must then be entered.
12. Create a program which sorts the names of the file by surname. Then use the program from exercise 9 to check the result of the sorting.
13. Create a program which copies the file to a new file. The new file name should be entered by the user.
14. Write a menu program where you gather the tasks from the latest exercises. The menu could look like this:

    1. Enter information
    2. Print
    3. Update city
    4. Remove
    5. Sort
    6. Copy
    0. Exit

    Select 0-6:

8 Pointers

8.1 Introduction

A pointer is a special kind of variable which contains a memory address to for instance a number instead of directly refer to the number. That implies a detour to the value via the memory address.

This might sound unnecessarily complicated, but implies a number of advantages like for instance more efficient program code, faster execution and memory saving. Especially in object oriented programming you get these advantages when copying objects or sending objects to functions. Object oriented programming is however beyond the scope of this course.

The pointer concept is unique for C++. It is for instance not present in the programming languages Visual Basic or Java. As a consequence C++ might be felt more complicated than other languages.

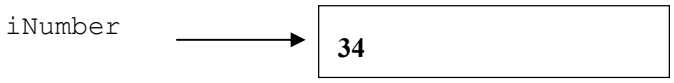
In this chapter we will aqcuire basic knowledge about pointers. We will learn how to use pointers to different data types, how to declare pointers and assign values. We will examine the anology between pointers and arrays and how to use pointers as parameters to functions.

Finally we will touch the subject dynamic memory allocation, which actually does not closely relate to pointers, but still often is used in connection with pointers.

8.2 What Is a Pointer

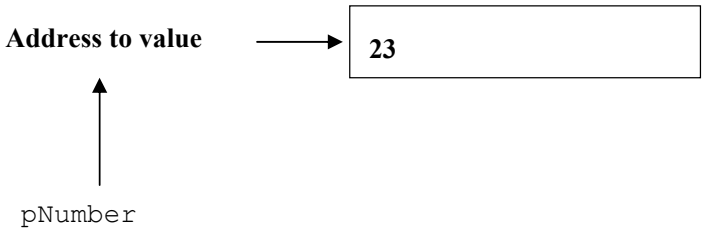
A pointer is a variable of a special kind which only can contain a memory address of the primary memory. This memory location in turn contains a value of some kind.

Le tus first study the situation for a common variable:



In the figure above we have the variable iNumber which contains the actual value of the variable, in our example 34.

Let us now focus on the corresponding pointer:



In the figure above we have a pointer named pNumber. It contains an address in the primary memory. If we go to that address, there is a number, 23 in our example.

8.3 Declaring a Pointer

Below we declare the pointer variable pNumber:

```
int* pNumber;
```

The asterisc (*) indicates that it is a pointer. `int*` means that it is a pointer to an integer value. You must always specify the data type pointed to by the pointer variable. Below we declare a pointer to a double value:

```
double* pPrice;
```

Below we declare a pointer to a char value:

```
char* pChr;
```

You can as well place the space in front of the asterisc. The declarations above could be written:

```
int *pNumber;
double *pPrice;
char *pChr;
```

You can use both variants.

8.4 Assigning Values to Pointers

An ordinary variable, say iNo, is assigned a value in the usual way:

```
int iNo = 23;
```

To get the address to the variable iNo, we use the & operator. The expression &iNo gives the address to the variable iNo.

In the declaration you can specify the memory location to be pointed at by a pointer variable:

```
int* pNumber = &iNo;
```

Here we create a pointer variable named pNumber and assign the address of the variable iNo to it. The variable iNo and the pointer variable pNumber now points to the same memory location, which means the value 23.

Note that in a pointer declaration you can't directly assign a fixed value:

```
int* pNumber = 23;  //wrong
```

since currently there is no specific memory location pointed to by pNumber. However, when pNumber has got its memory address, we can change the value in the location indicated by pNumber:

```
*pNumber = 25;
```

Here we must remember to use the asterisc together with the name of the pointer variable. The program then understands that it is the value that is to be changed.

Compare this to this erroneous statement:

```
pNumber = 25;  //wrong
```

This would mean that we updated the address pointed to by pNumber. The address 25 would be pointed to, which of course is erroneous.

We have introduced two operators in connection with pointers:

*	means 'the content of'
&	means 'the address to'

In the same way we can write:

```
//ordinary variable:
double dPrice = 34.75;
//pointer variable with the same address as
//dPrice:
double* pPrice = &dPrice;
//change the price:
*pPrice = 45.25;

// ordinary variable:
char cChr = 'x';
//pointer with the same address as cChr:
char* pChr = &cChr;
//change the character:
*pChr = 'y';
```

When printing a value pointed to by a pointer variable, you use:

```
cout << *pNumber;  //prints 25
```

This means 'print the content of pNumber'.

To print the address pointed to by a pointer variable you write:

```
cout << pNumber;
//prints the address, e.g. 0x0066FDF0
```

The printed address is in hexadecimal format. Normally we don't have to bother about the exact address. The only thing to remember is whether we mean 'the address to' or 'the content of'.

8.5 Addresses and char Pointers

We will now take a look at how pointers work in connection with string variables, i.e. arrays of char type. We declare a string array named cName:

```
char cName[] = "John Smith";
```

We then declare a char pointer named pName which points to the same text as the content of cName.

```
char* pName = cName;
```

Why didn't we use the & operator in front of cName like in the previous example? The explanation is that an array actually is a pointer. When using the name of the array, cName, it is interpreted as a pointer to the first item of the array. So when writing the statement:

```
pName = cName;
```

it means that we let the pointer pName get the same address as the pointer (array) cName.

8.6 cout and char Pointers

The print function cout has some peculiarities you ought to know when printing strings. The statement:

```
cout << pName;
```

should actually print the address in hexadecimal format of pName. But cout performs a reinterpretation. It takes the content in the memory location pointed to by pName, i.e. the character 'J', and prints character by character until the null character is found. This means that the entire name 'John Smith' is printed. Compare the statement:

```
cout << cNamn;
```

which gives the same result, which we discussed in the Strings chapter.

The statement:

```
cout << *pName;
```

correctly prints the content of the memory location pointed to by pName, but it only takes that character. This means that only 'J' is printed.

The statement:

```
cout << &pName;
```

prints the address of the memory location in which pName is stored.

The statement

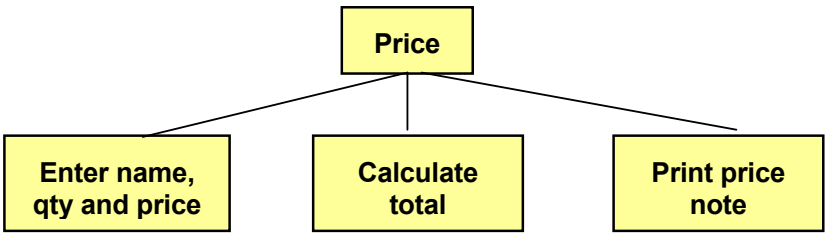
```
cout << &cName;
```

prints the address of the memory location where the name 'John Smith' is stored.

8.7 Price Program with Pointers

We will now create a program which reads quantity and unit price of a product from the user, and the name of the user. The program will then calculate the total price of the product and print a personal price note on the screen. We will use pointer variables.

The logical process is given by the following JSP graph:



Here is the code:

```
#include <iostream.h>
void main()
{
  //Declare variables and corresponding pointers
  //Set the pointers to point to the address of the
  //corresponding variable
  int iNo;
  int* pNo = &iNo;
  double dPrice, dTotal;
  double* pPrice = &dPrice;
  double* pTotal = &dTotal;
  char cName[20];
  char* pName = cName;

  //Read data and store in the pointer variables
  cout << "Enter your name: ";
  cin.getline(pName, 19);
  cout << "Enter quantity and unit price: ";
  cin >> *pNo >> *pPrice;

  //Calculate total
  *pTotal = *pNo * *pPrice;

  //Printout of personal price note
  cout << "Dear " << pName << ", your price is " <<
    *pTotal << " kr." << endl;
}
```

Let us say that we enter 'John Smith', quantity 5 and unit price 12. Then the printout will be:

```
Dear John Smith, your price is 60 kr.
```

8.8 Pointer Arithmetics

By pointer arithmetics we mean how to increment and decrement a pointer, i.e. how to make a pointer to an array move stepwise from item to item.

Let's say that we have an array of integers:

```
int iNos[] = {5, 12, 3, 24, 125, 8};
```

Here we have declared the array iNos to contain six items. Suppose an integer takes 4 bytes in the working memory. If the first integer (5) is stored in memory address 2000, then the next integer (12) will be stored in memory address 2004, the third in 2008 etc.

Now we declare an integer pointer to point to the first item of the array:

```
int* pNos = iNos;
```

pNos now has the value 2000 (the address of the first item in the array).

We can now perform the following pointer arithmetic:

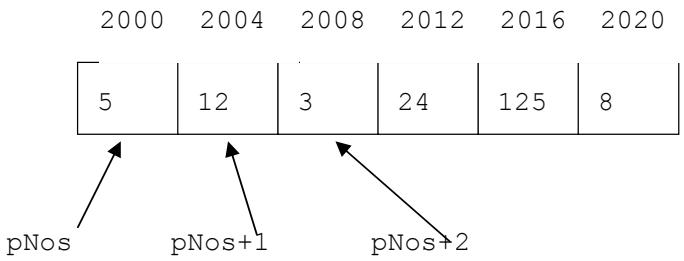
```
pNos++;
```

which means that pNos is increased by 1. You might then be fooled to beleive that pNos now has the value 2001. But that is not the case. Since we have declared pNos as a pointer to integer, the system knows that each integer requires four bytes, and pNos is consequently increased by 4 making the new value of pNos be 2004. This implies that pNos now points to the second item of the array.

If the array would have been declared as double and pNos declared as a pointer to double, the system knows that each double number takes 16 bytes, and a stepwise increase would add 16 to pNos.

As a consequence the data type pointed to by a pointer is of ultimate importance when using pointer arithmetics.

The situation with integers is summarized by the following figure:



To print the numbers you can use the statement:

cout << *pNos << *(pNos+1) << *(pNos+2) …

Note that, when printing the second, third item etc. we must enclose pNos+1, pNos+2, etc. with a parenthesis, so the address is calculated first before taking the asterisk ('the content of') into account, otherwise the content of pNos, i.e. the first item of the array, would be increased by 1, 2 etc.

We may as well use a loop for the printout:

```
for (int i=0; i<6; i++)
   cout << *pNos++ << endl;
```

Here we use the loop counter i, which goes from 0 to 5, i.e. as many turns as there are items in the array. For each turn of the loop the content of the memory address pNos is printed, and the pointer is increased by 1, i.e. it moves to the next item of the array.

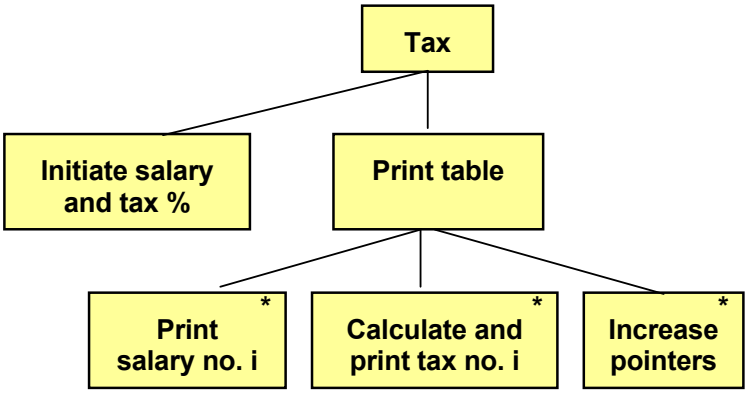
8.9 Tax Program

We will not create a full-featured tax calculation program, but only calculate the tax deduction for a number of monthly salaries based on fix tax percentages.

We store a number of monthly salaries in an array and corresponding tax percentages in another. We multiply each salary by corresponding tax percentage, which gives the tax deduction amount. Each salary and tax percent are then printed in table format.

We will use pointer arithmetics to move from item to item in the arrays.

The program logic is explained by the following JSP graph:



First we initiate the arrays with salaries and tax percentages. When printing the table we go through one item at a time of the arrays and print salary and corresponding tax, which is calculated by multiplying salary by tax percentage. Then we increase both pointers to proceed to the next salary and tax percentage.

Here is the code

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
      int iSal[] = {14000, 15000, 16000, 17000, 18000, 19000};
      int* pSal = iSal;
      double dTax[] = {0.32, 0.34, 0.35, 0.36, 0.365, 0.37};
      double* pTax = dTax;
      cout << setiosflags(ios::fixed) << setprecision(2);
      cout << "  Salary        Tax" << endl;
      for (int i=0; i<6; i++)
      {
            cout << setw(8) << *pSal << setw(10) << (*pSal *
                  *pTax) << endl;
            pTax++;
            pSal++;
      }
}
```

The include files are iostream.h for input and output, and iomanip.h to be able to format the printout into a nice table.

In main() we initiate the array iSal with a number of salaries and the array dTax with a number of tax percentages. We also declare a pointer pSal which is set to point to the first item of the array iSal, and a pointer pTax for the dTax array.

The first cout statement fixes the decimal point and states two decimals. The second cout statement prints the heading of the table.

The for-loop goes from 0 to 5, i.e. as many turns as there are items in the arrays. The cout statement sets the width of the printed numbers with the setw() function, prints the salary by means of the pointer, and multiplies the salary by the tax percentage and prints the result. Note that we use asterixes in front of the pointers to get 'the content of'.

After the cout statement we increase the two pointers by 1, i.e. we move the pointers to the next item of the arrays. The program automatically remembers that one of the pointers is an integer pointer and the other a double, and moves them the corresponding number of bytes in the primary memory.

8.10 Functions and Pointers

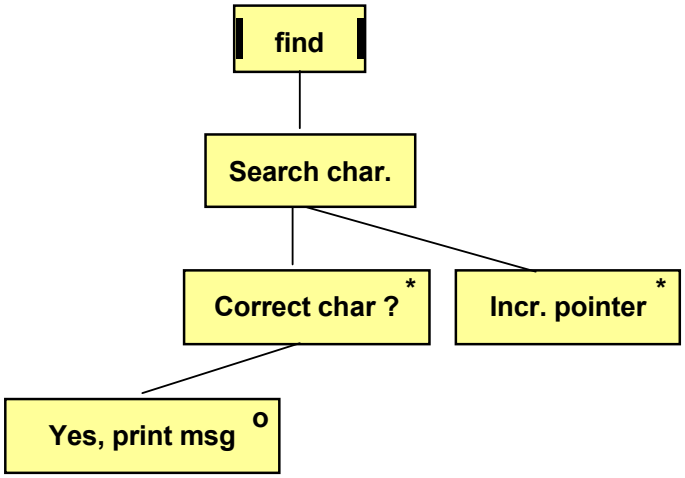
Many times you use pointers as parameters to functions. That means that you send the memory address to the function instead of sending all data. Especially in object oriented programming when you want to send an object to a function which is several Mbytes big, it is a great advantage to only send a memory address instead. It saves both memory and time.

You may remember from the Functions chapter that it was possible to send reference parameters to a function, which means that you send the address to the value instead of the value itself. It is very similar to sending pointers. An advantage with pointers is however that it is possible to use pointer arithmetics, which gives more compact code.

You should however keep in mind that if you send a pointer to a function, and the function updates the value, the value will be updated also after completion of the function. Unintentional update of a value can however be prevented by means of the keyword const. More about this later.

We will create a function which searches for a particular character, for instance @, in a string to check if it is an email address. The string is sent as pointer to the function find(). A for-loop in the function goes through the string, character by character, and checks if it is an @. If @ is found, a suitable text is printed on the screen.

We begin with a JSP graph:



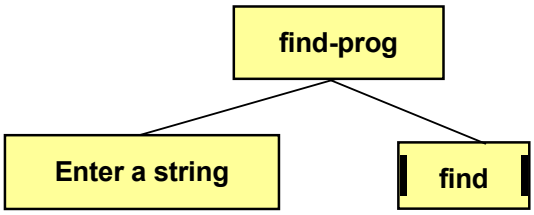
Here is the code:

```
void find(char* str)
{
   for (int p=0; p<8; p++)
   {
      if (*str == '@')
         cout << "It is an email address";
      str++;
   }
}
```

The function takes a parameter str which is a pointer to a char variable. That means that we don't send the entire string but only the address to the first character of the string. The for-loop goes from 0 to 7, which is a limitation, but we have done it as simple as possible to illustrate the use of pointers.

The if statement checks if the content of the address str is the @ character. If so, the suitable text is printed. At the end of the loop the pointer is increased by 1, i.e. it is set to point to the next character.

We will now write an entire program, where the user is prompted for a text that is sent to the function. First a JSP graph:



Here is the code:

```
#include <iostream.h>
void find(char* str);
void main()
{
  char cString[9];
  char* pString=cString;
  cout << "Enter a text: ";
  cin.getline(cString, 8);
  find(pString);
}
void find(char* str)
{
  for (int p=0; p<8; p++)
  {
    if (*str == '@')
      cout << "It is an email address";
    str++;
  }
}
```

The program first declares a string array named cString. The pointer pString is set to point to the first character of the string. When the user has entered a text (maximum 8 characters), the function find() is run.

We will now illustrate the risk of having the value outside the function be changed. We will modify the function find() so that it replaces @ by a blank:

```
void find(char* str)
{
  for (int p=0; p<18; p++)
  {
    if (*str == '@')
      *str = ' ';
    str++;
  }
}
```

Thus, the function chages the value of the string. If we would print the email address in main() after completion of the function find() with the purpose of having the original email address printed, then we would get a wrong result. The statement

```
cout << "The email address is " << pString << endl;
```

will give a printout with a blank instead of @.

One way of preventing modification of a value in the function is to use the keyword const in front of the parameter in the function header:

```
void find(const char* str)
```

If we still would write a statement in the function that tries to modify the value, we will get a compilation error. Many programmers use const in this way to clearly indicate that the value is not changed in the function.

8.11 Dynamic Memory

When declaring an array in Visual C++ we have until now been forced to specify the number of items of the array to allocate the correct memory space. That is called static memory.

Many times we cannot in advance predict the number of items needed for the array, for instance when reading a number of product id:s to an array from a file, where the number of products is unknown.

The solution to this problem is to use dynamic memory. The dynamic memory area is capable of assigning space during the execution of the program and not at the compilation. Different amounts of memory might be needed at different execution occasions. A disadvantage is however that the program cannot guarantee that the requested amount of memory is available. Therefore, you must in the code insert a check that the requested memory could be allocated.

To allocate dynamic memory you use the keyword new. Look at the following statements:

```
int* pNo;
int iNumber;
```

```
cout << "How many products will be entered? ";
cin >> iNumber;
pNo = new int[iNumber];
```

Here we declare a pointer to int, pNo, and a common int variable iNumber. The user enters an integer which is stored in iNumber. This value is used as the number of items of the array declared in the last statement.

This is however a dangerous way of coding since there might be a lack of memory, and the program will crash. Therefore, we insert a check by means of the following:

```
if ( (pNo = new int[iNumber]) == 0)
{
   cerr << "Not sufficient memory. The program
              will exit!";
   exit(1);
}
//program continues
```

Here we put the declaration of the array as a condition in an if statement. If there is not enough memory, the result of the declaration will be equal to 0. Then the warning text is printed and the program is terminated with exit(1).

If there is enough memory the value of the condition is not 0, and the program continues with subsequent statements.

You can run the code above and enter different numbers to figure out how the code behaves in different situations. Don't forget to include the header file stdlib.h, which is needed for the exit(1) function.

After the if statement we can now continue with product entry to the array:

```
for (int i=0; i<iNumber; i++)
{
   cin >> *pNo ;
   pNo++;
}
```

The loop performs as many turns as the number of items of the array. For each turn a product id is read from the user, which is stored in the memory address given by pNo. pNo is increased by 1, i.e. the pointer moves on to the next item of the array.

If we want to print the products, we must reset the pointer to the original position, i.e. to the first item of the array, and then perform a new loop which prints the products:

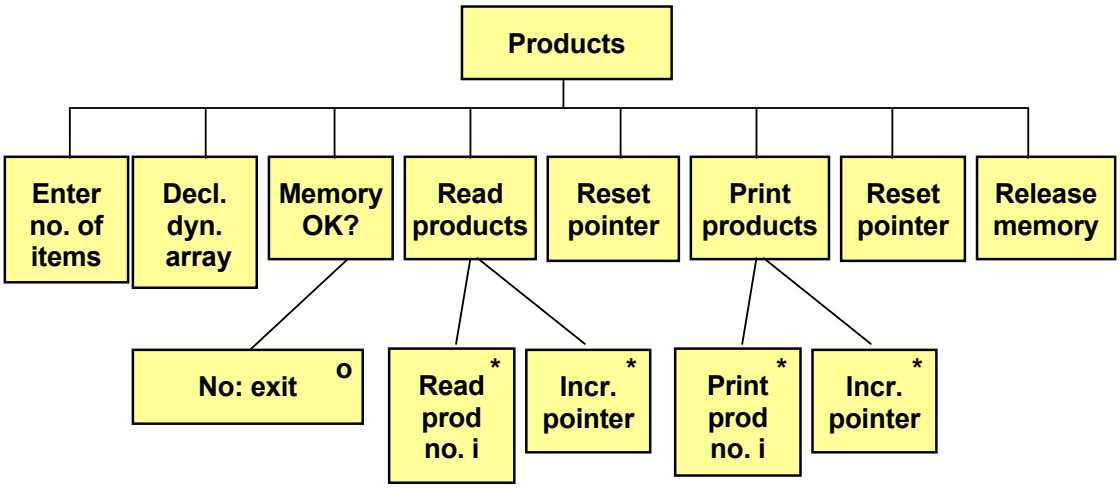
```
pNo = pNo - iNumber;
for (i=0; i<iNumber; i++)
{
   cout << *pNo << endl;
   pNo++;
}
```

When having used dynamic memory it is common by programmers to release the memory when it is not needed any more, thus freeing memory for other tasks of the program. We release the memory for the array with the statements:

```
pNo=pNo-iNumber;
delete[] pNo;
```

First we reset the pointer to its original position and then we use the delete statement to release the dynamic memory.

We now give a JSP graph that shows the process:



Here is the entire program:

```
#include <iostream.h>
#include <stdlib.h>
void main()
{
   int* pNo;
   int iNumber;
   cout << "How many products will be entered? ";
   cin >> iNumber;
   if ( (pNo = new int[iNumber]) == 0)
   {
      cerr << "Not sufficient memory. The program will exit!";
      exit(1);
   }
   for (int i=0; i<iNumber; i++)
   {
      cin >> *pNo ;
      pNo++;
   }
   pNo = pNo - iNumber;
   for (i=0; i<iNumber; i++)
   {
```

```
    cout << *pNo << endl;
    pNo++;
  }
  pNo = pNo - iNumber;
  delete[] pNo;
}
```

We will now create still another program where we instead of product id:s enter a number of names of char type and store the names in arrays in the dynamic memory. You should then remember that each name is itself an array of char type. We will then create a new char array with the new keyword for each name. Each char array will be exactly of the length required by the entered name, with the purpose of saving memory space. Here is the code:

```
#include <iostream.h>
#include <string.h>
void main()
{
  const int iNo = 5;  // Number of strings to be stored
  char temp[30];  // Tempory storage of entered name
  char *cNames[iNo]; // Space for 5 string pointers with
                     // arbitrary number of characters
  cout << "Enter the names of 5 course mates" << endl;
```

```
  for ( int i = 0; i < iNo; i++)
  {
    cout << "Mate no. " << i + 1 << " ";
    cin.getline(temp, 30);  // Temporary storage in temp
    // Don't waiste memory! strlen() gives exactly the
    // number of characters required plus 1 for null:
    cNames[i] = new char[strlen(temp) + 1];
    // Copy the name to the pointer array:
    strcpy(cNames[i], temp);
  }
  // Print the names
  for ( int j = 0; j < iNo; j++)
    //cNames is an array of pointers:
    cout << cNames[j] << endl;
  // Release memory for each separate string array:
  for ( int k = 0; k < iNo; k++)
    delete [] cNames[k];
}
```

8.12 Summary

In this chapter we have learnt the basics of one of the areas that makes C++ unique compared to other programming languages. Pointer management is complicated but offers opportunities to efficient coding at professional level.

We have learnt to declare and assign values to pointers of different data types. One of the great advantages with pointers is pointer arithmetics, where you can step through the items of an array in an efficient way.

We have also learnt to send pointers as parameters to functions and we have learnt some about dynamic memory allocation. C++ programmers must often pay attention to memory allocation at a detailed level not required by other language programmers. This might seem to be unnecessarily complicated from the beginner's point of view, but provides rich opportunities to effective programming aiming at memory minimizing programs with high performance.

We will experience further advantages of pointers in the next chapter about structures.

8.13 Exercises

1. Write a little program which declares an integer variable and initiates it to the value 25. Then declare a pointer to that value. Print the value by means of the pointer.
2. Write a program similar to the previous applying it to a string with your own name instead of an integer.
3. Start from the program in the section 'Price Program with Pointers'. Complete it with a facility to enter a discount percentage to be deducted from the total price. Use a pointer for the discount.
4. Write a program which prompts the user for a driven number of miles and the fuel consumption for the trip. The program should then calculate and print the fuel consumption per mile. Use pointers like in the previous programs.

5. Complete the previous program to also allow entry of the car brand, which should be included in the printout in a suitable way.
6. Write a program that reads 5 integers to an array. The integers should then be printed. Use pointer arithmetics.
7. Complete the program to also calculate the sum of the integers. Use a pointer variable for the sum.
8. Start from the program in the section 'Tax Program' and modify it so that the user enters the salaries and tax percentages.
9. Modify the previous program to instead read the information from a file instead of from the keyboard.
10. Write a program which reads product id:s and prices from a file and stores them in arrays, one array for the product id:s and one for the prices. Use pointers. The program should then print a nice table of products and prices.
11. Modify the program in the previous exercise so that the user can enter a product id and get the corresponding price printed. Use a pointer also for the product id entered by the user.
12. Start from the last version of the program with the find() function in the section 'Functions and Pointers', where the @ character is replaced by a blank. Modify the function so that it prints the updated string from inside of the function. Use the main() program to test the function.
13. Change the previous program so that both the original and the updated email adress is printed.
14. Write a function which replaces lower case characters in a string to upper case. The string should be sent to the function as a pointer. Test the function in a program which prompts the user for his name and then sends the name to the function. The program should print the updated string.
15. Write a function which takes a pointer to an integer array and the number of items of the array as parameters, finds the greatest item and returns it. In the main() program the user should enter 8 numbers to be stored in the array. The function is called and the returned greatest item is printed.
16. Modify the previous exercise so that the user first enters the number of integers being entered. The array should be stored in the dynamic memory.
17. Start from the last program of the section 'Dynamic Memory' where you entered the names of 5 course mates. Since telephon numbers can contain blanks and hyphen (e.g. 0522-23 23 23) they are of char type. Use a second two-dimensional array with pointers to the dynamic memory exactly as for the names.
18. Expand the previous program with checks about enough memory available.

9 Structures

9.1 Introduction

When working with data from files and databases it is often convenient to process big portions of data in one lump, for instance an entire customer record in a customer file. A good tool for this is the structure concept. A structure is a set of data that in some way has an intermediary relation.

In connection with structures we will be using pointers and pointer arithmetics that we learnt in the previous chapter.

Structures are a pre-state to classes within object oriented programming. Therefore, this chapter is a bridge to the next step of your programmer education, object oriented programming.

In this chapter we will learn how to define structures, handle information stored in structures, work with arrays of structures and files in connection with structures. We will also learn how to use pointers to structures, how to sent structures to a function and store structures in the dynamic memory.

9.2 What Is a Structure

Think of a customer record in a customer file that contains name, address, telephone, email, discount profile, terms of delivery, terms of payment and so forth. All this information is stored for each customer in the customer file.

When reading, processing and saving this information to a file or database it is convenient to be able to handle all data for a customer in a uniform way. It is then gathered into a structure, which provides better organization of the program code.

A structure is like a template for all information per customer. A structure behaves in the code like a data type such as int, double or char. You declare a variable of the structure type defined. In the structure variable you can then store all information for a particular customer.

You can also create an array of structure items, where each item of the array is a structure with all information per customer. The array will thus contain all information for all customers.

9.3 Defining a Structure

First we will learn to define a structure template, i.e. specify the shape of the structure, the structure members and the data type of each member of the structure. Suppose we want to work with a product file with:

- Product name
- Product id
- Price
- Quantity in stock
- Supplier

This means that each product in the file will contain these five members.

Here is the code for definition of the structure:

```
struct Prod
{
  char cName[20];
  int iId;
  double dPrice;
  int iNo;
  char cSupp[25];
};
```

First there is the keyword struct, and then the name of the structure or data type (Prod). Within curly brackets you then enumerate the members of the structure, where each member is declared in the usual way of declaring variables. Each member is ended with a semicolon. After the last right curly bracket there must also be a semicolon.

The structure above shows that the different members can be of different data types (char, int, double) and also arrays like cName. You can also have other structures as members of the structure, if applicable.

The names of the structure and members are of course arbitrarily selected, but they should in some way correspond to their usage.

9.4 Declaring and Initiating Structure Variables

To declare a structure variable, i.e. a variable of the data type Prod, you write:

```
Prod prodOne;
```

Here we declare a variable prodOne which is of the Prod type. You can also initiate it with values already in the declaration:

```
Prod prodOne = {"Olive Oil", 1001, 120.50, 250, "Frescati Oil S/A"};
```

Within curly brackets we enumerate values for the structure members in the correct sequence, separated by commas. The data types of the values must correspond to the definition of the members.

9.5 Assigning Values to Structure Members

When updating, copying or in other ways processing the value of a structure member, you use the following way of coding:

```
prodOne.iNo = 251;
```

You write the name of the structure variable, followed by a period and the name of the member in question. Here the quantity in stock will be set to 251 for the 'Oliv Oil' product. Or:

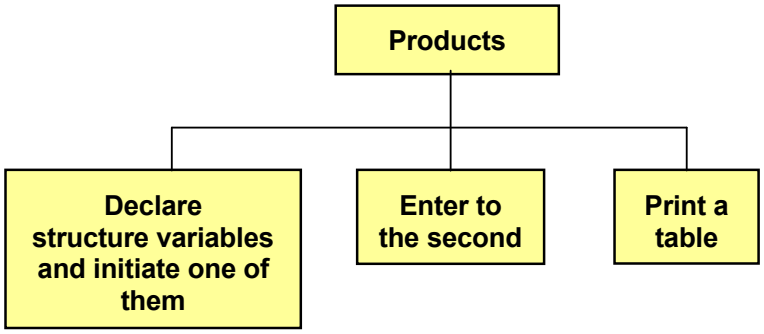
```
strcpy(prodOne.cSupp, cString);
```

This requires that cString is a string array whose content is copied to the cSupp member.

9.6 A Structure Program

We will now create an entire program using structures. We will create a product structure according to the previous example and two structure variables with product information. One of them should be initiated directly in the declaration and the other is supposed to be supplied with information from the user. Finally the program should print a table of the products.

We start with a JSP graph:



The logic is simple. The most difficult task is to handle the structure in the correct way. Here is the code:

```
#include <iostream.h>
struct Prod
{
   char cName[20];
```

```
   int iId;
   double dPrice;
   int iNo;
   char cSupp[25];
};
void main()
{
   // Declare and initiate a variable of type Prod
   Prod prodOne = {"Olive Oil", 1001, 120.50, 250,
       "Frescati Oil S/A"};
   // Declare a new Prod variable
   Prod prodTwo;
   // Prompt the user for product information
   cout << "Enter information for a product:" << endl;
   cout << "Start with the product name: ";
   cin.getline(prodTwo.cName, 20);
   cout << "The product id: ";
   cin >> prodTwo.iId;
   cout << "The price: ";
   cin >> prodTwo.dPrice;
   cout << "How many items are there in stock? ";
   cin >> prodTwo.iNo;
   cin.get();  // clear in-buffer from new line char
   cout << "Who supplies the product: ";
   cin.getline(prodTwo.cSupp, 25);
   cout <<"Prodname  \tProduct id \tPrice \tQuantity
     \tSupplier" << endl;  // tab with \t
      cout << prodOne.cName << '\t' << prodOne.iId
     << "\t\t" <<  prodOne.dPrice << '\t' <<
     prodOne.iNo << '\t' <<  prodOne.cSupp <<
     endl << endl;
      cout << prodTwo.cName << '\t' << prodTwo.iId <<
     "\t\t" << prodTwo.dPrice << '\t' << prodTwo.iNo <<
     '\t' << prodTwo.cSupp << endl << endl;
}
```

The definition of the structure is before main(), which makes it valid for the entire program, also inside functions. You can also define the structure inside main(), but then it is only valid in main() and not in other functions.

The first structure variable prodOne is initiated with values directly in the declaration. Then there are a number of heading texts and entry of values to the structure members of the second structure variable. Note that we use a period between the structure variable and member.

The output is done by means of tabs \t. You might need to accommodate the length of the entered texts to make the information be printed in the correct column. We could have done that more flexible by means of the text formatting functions from chapter 1, but we used a rough method for simplicity's sake.

9.7 Array with Structure Variables

A disadvantage with the previous program is that we needed a separate structure variable (prodOne, prodTwo) for each product. A more convenient solution is to use an array with structure variables allowing the use of a loop to process the structure variables in a uniform way.

Below we declare a structure array sProds of the type Prod with three items:

```
Prod sProds[3];
```

We have allocated memory space for three products, but we have not yet assigned values to the structure members. That could be made directly at the declaration:

```
Prod sProds[3] = {
   {"Food Oil", 101, 12.50, 100, "Felix Ltd"},
   {"Baby Oil", 102, 23.75, 25, "Baby Prod"},
   {"Boiler Oil", 103, 6100, 123000, "Shell"},
};
```

Note that the values for each structure variable are surrounded by curly brackets, and that the values are enumerated in the usual way within each pair of curly brackets. All three pair of brackets are surrounded by an extra pair of curly brackets delimiting the initiation list of values. After the last bracket there must be a semicolon.

If you want to let the user enter values, this is preferably done in a loop:

```
for (int i=0; i<3; i++)
{
   cout << "Enter values for product no. "
     << i+1 << endl;
   cout << "Start with the product name: ";
   cin.getline(sProds[i].cName, 20);
   cout << "The product id: ";
   cin >> sProds[i].iId;
   cout << "The price: ";
   cin >> sProds[i].dPrice;
   cout << "How many in stock? ";
   cin >> sProds[i].iNo;
   cin.get(); //clear in-buffer newline char
   cout << "Who supplies the product: ";
   cin.getline(sProds[i].cSupp, 25);
}
```

9.8 Pointer to Structure

You can declare pointers to structures in the same way as declaring pointers to other data types:

```
Prod *pProd;
```

To instead make the pointer point to the structure variable prodOne we write:

```
Prod* pProd = &prodOne;
```

This means that pProd equals the address to prodOne.

The advantage is to be able to use pointer arithmetics to step through the different structure variables:

```
pProd++;
```

This statement moves to the next structure variable.

With a pointer to a structure you are not allowed to use the period (.) as delimiter between the variable and member. You must use the -> characters:

```
pProd->iNo = 25;
```

For instance, to print the information for the structure pointed to by pProd, we can use the following statement:

```
cout << pProd->cName << endl <<
  pProd->iId << endl <<
  pProd->dPrice << endl <<
  pProd->iNo << endl <<
  pProd->cSupp;
```

Suppose we have declared and initiated an array sProds with space for three structure variables and with values like in the previous section:

```
Prod sProds[3] = {
   {"Food Oil", 101, 12.50, 100, "Felix Ltd"},
   {"Baby Oil", 102, 23.75, 25, "Baby Prod"},
   {"Boiler Oil", 103, 6100, 123000, "Shell"},
};
```

Then we can declare a pointer of Prod type which points to the first item of the array:

```
Prod* pProd = &sProds[0];
```

Then we can use a loop to print for example the product id:s for the three products:

```
for (int i=0; i<3; i++)
{
   cout << pProd->iId << endl;
   pProd++;
}
```

Note that we have used pointer arithmetics to step from product to product.

9.9 Structures in the Dynamic Memory

When we at the compilation cannot predict the number of products to be stored in the array, it is convenient like for other arrays (see the 'Pointer' chapter) put the structure array in the dynamic memory area. As usual, we accomplish this with the new keyword.

In the following code we prompt the user for the number of products to be entered, and then declare a pointer to an array in the dynamic memory space:

```
int iQty;
cout << "How many products should be
          entered? ";
cin >> iQty;
Prod *pProd = new Prod[iQty];
```

In the declaration of the array we must specify the number of structures to allocate memory for. This is done by means of the variable iQty.

Then, with a loop, we can let the user enter information to the requested number of products:

```
for (int i=0; i<iQty; i++)
{
   cin >> pProd->iId;
   // … and the remaining structure members
```

```
   pProd++;
}
```

The loop runs the number of turns as stated by the variable iQty. For each turn of the loop we increase the pointer by 1, thus making it point to the next structure variable of the array.

To print the product information we create a new loop. But first we must reset the pointer to the position of the first item of the array:

```
pProd = pProd - iQty;
for (i=0; i<iQty; i++)
{
   cout << pProd->iId << endl;
   // … and the remaining structure members
   pProd++;
}
```

Also here, we use pointer arithmetics to proceed from item to item of the array.

Finally we have to free the dynamic memory area when it will not be used any more to not block other parts of the program:

```
pProd = pProd - iQty;
delete[] pProd;
```

Innan vi använder delete, ställer vi tillbaka pekaren till sin ursprungsplats så att rätt minnesområde frigörs.

9.10 Structure As Function Parameter

9.10.1 Reference Parameter

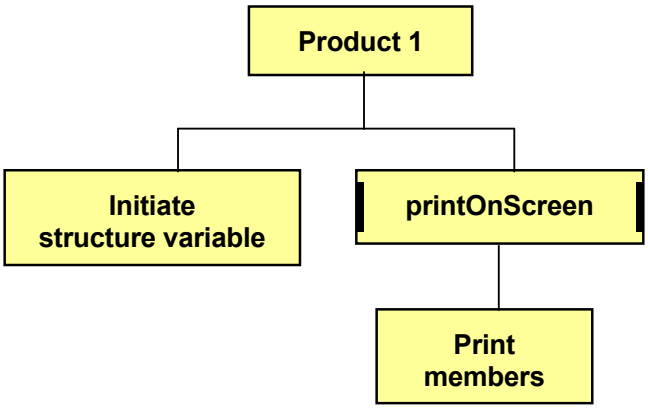
We will now show some examples of how to send structures as parameters to functions. Suppose we want to create a function which prints the content of the structure sent to the function. We give the function the name printOnScreen().

The first example of printOnScreen() takes a parameter that is a reference parameter (see the 'Functions' chapter) of a structure:

```
void printOnScreen(Prod & rProd)
{
   cout << rProd.cName << '\t' <<
      rProd.iId << "\t\t\t" <<
      rProd.dPrice << '\t' <<
      rProd.iNo << '\t' <<
      rProd.cSupp << endl << endl;
}
```

The function does not return any value (void) and takes a parameter of Prod type, which is a reference parameter (& character). It is no pointer, so we use a period as delimiter between the structure variable and member. We have used \t to tab.

We will create a simple program to test the function. Here is the JSP graph:



We initiate a structure variable, which is sent to the function, where the members are printed.

Here is the code:

```
#include <iostream.h>
struct Prod
{
   char cName[20];
   int iId;
   double dPrice;
```

```
   int iNo;
   char cSupp[25];
};
void printOnScreen(Prod & rProd)
{
   cout << rProd.cName << '\t' <<
      rProd.iId << "\t\t\t" <<
      rProd.dPrice << '\t' <<
      rProd.iNo << '\t' <<
      rProd.cSupp << endl << endl;
}
void main()
{
   Prod prodOne = {"Olive Oil", 1001, 120.50, 250, "Frescati
      Oil S/A"};
   printOnScreen(prodOne);
}
```

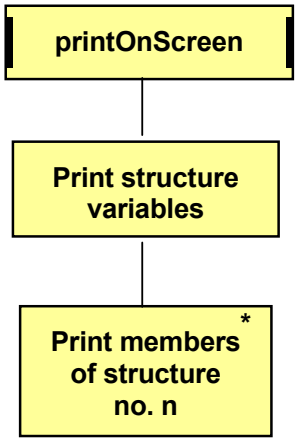
First we define the structure Prod. Then there is the function printOnScreen(). In main() we declare a structure variable prodOne of the Prod type and initiate it with values. The the function printOnScreen() is called, to which we send prodOne as actual parameter.

9.10.2 Array Parameter

Next variant of the function printOnScreen() takes a parameter which is a structure array and a parameter to the number of items in the array.

Remember that, when sending an array as a parameter to a function, it is always made on reference basis. Therefore, you should not explicitly specify that it is a reference parameter by using the & character.

Here is the JSP graph for the function:



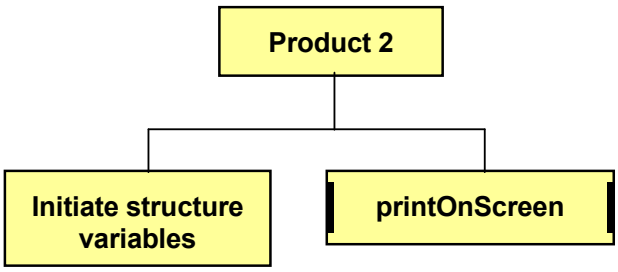
Here is the code:

```
void printOnScreen(const Prod p[],
   const int n)
{
   for (int i = 0; i < n; i++)
       cout << p[i].cName << '\t' <<
       p[i].iId << '\t' <<
       p[i].dPrice << '\t' <<
       p[i].iNo << '\t' <<
       p[i].cSupp << endl << endl;
}
```

The first parameter is of Prod type and has the name p with a subsequent square bracket to indicate array. The second parameter is called n and corresponds to the number of items of the array. Both have been given the const keyword to ensure that the information is not updated in the function.

The loop has the counter i and performs as many turns as there are items in the array. The variable i is used as index in the array to indicate specific items.

We create a simple program to test the function. The JSP graph is:



Here is the entire code:

```
#include <iostream.h>
struct Prod
{
   char cName[20];
   int iId;
   double dPrice;
   int iNo;
   char cSupp[25];
};
void printOnScreen(const Prod p[],
   const int n)
{
```

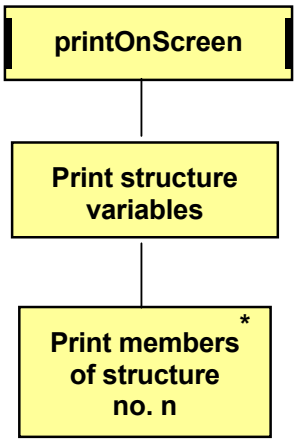
```
   for (int i = 0; i < n; i++)
       cout << p[i].cName << '\t' <<
       p[i].iId << '\t' <<
       p[i].dPrice << '\t' <<
       p[i].iNo << '\t' <<
       p[i].cSupp << endl << endl;
}
void main()
{
   Prod sProds[3] = {
     {"Food Oil", 101, 12.50, 100, "Felix Ltd"},
     {"Baby Oil", 102, 23.75, 25, "Baby Prod"},
     {"Boiler Oil", 103, 6100, 123000, "Shell"},
   };
   printOnScreen(sProds, 3);
}
```

In main() we declare an array of Prod type with three items, which are initiated with values. The function printOnScreen() is called, to which we send the structure array and the number 3 as actual parameters.

9.10.3 Pointer Parameter

The last example of the function printOnScreen() takes a pointer to the Prod structure and the number of products as parameters.

Here is the JSP graph and the code:

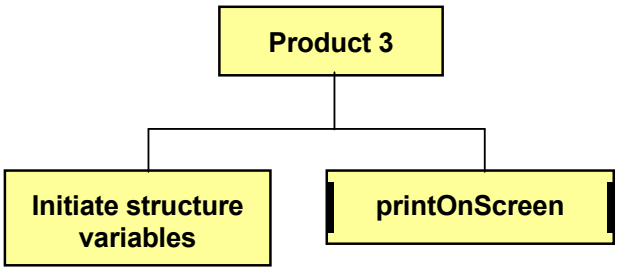


Here is the code of the function:

```
void printOnScreen(Prod *p , const int n)
{
   for (int j=0; j<n; j++)
   {
      cout << p->cName << '\t' <<
         p->iId << '\t' <<
         p->dPrice << '\t' <<
         p->iNo << '\t' <<
         p->cSupp << endl << endl;
      p++;
   }
}
```

The pointer parameter has the name p and the number of products n. The loop performs as many turns as given by the number of products. In the loop we print the members of the structure. Note that, since p is a pointer, we use -> between pointer and member. At the end of the loop we use pointer arithmetics and increase p by 1, i.e. moves p to the next structure.

We create a simple program to test the function. Here is the JSP graph:



Here is the entire code:

```
#include <iostream.h>
struct Prod
{
   char cName[20];
   int iId;
   double dPrice;
   int iNo;
   char cSupp[25];
};
void printOnScreen(Prod *p , const int n)
{
   for (int j=0; j<n; j++)
   {
      cout << p->cName << '\t' <<
         p->iId << '\t' <<
         p->dPrice << '\t' <<
         p->iNo << '\t' <<
         p->cSupp << endl << endl;
      p++;
   }
}
void main()
{
   Prod sProds[3] = {
      {"Food Oil", 101, 12.50, 100, "Felix Ltd"},
      {"Baby Oil", 102, 23.75, 25, "Baby Prod"},
      {"Boiler Oil", 103, 6100, 123000, "Shell"},
   };
   Prod *pProd = &sProds[0];
   printOnScreen(pProd, 3);
}
```

In main() we declare an array of Prod type with three items and initiate it with values. Then we declare a Prod pointer to point to the first item of the array (the & character means 'the address to'). The pointer and the number 3 is sent as parameters to the function.

9.11 Summary

In this chapter we have learnt what a structure is, namely a tool to handle items of different information that in some way belong together, for instance all data for a customer, all data for a product etc. We have learnt to define structures and declare structure variables and fill the structure with values.

We have also seen how to use arrays with structures and pointers to structures. When unable to predict the number of items to be contained by the array, we have stored it in the dynamic memory area with the new keyword.

Finally you have learnt how to send structures as parameters to functions, either as reference parameters, array parameters or pointer parameters.

You will now try your new knowledge in a number of exercises.

9.12 Exercises

1. Define a structure with customer information: name, address, customer category (one letter), discount percent, total invoice amount year-to-date. Then write a program that declares a structure variable and initiates it with some values according to your preference. Then print the information on the screen.
2. Change the previous program so that the user can enter the customer information.
3. Change the previous program to declare a structure array with 3 customers and initiates the structure members directly at the declaration. All three customers should then be printed.
4. Change the previous program so that the user can enter an order total to be added to the member 'total invoice amount year-to-date'. Then print all information.
5. Change the previous program to let the user enter information for the three customers.
6. Write a program that uses the structure definition for customers according to the previous exercises and declares a pointer to a customer. It should then be possible to enter information for the customer. The program should finally print the customer information by using the pointer.
7. Change the previous program to let the pointer point to an array with three customers. Entry and printing as before.
8. Change the previous program to let the user first specify the number of customers to be entered. The array should reside in the dynamic memory area.
9. Write a function that stores a Prod structure in a file. Use the Prod definition given previously in the chapter. Each time the function is called a new structure is written to the file without destroying previous information. Use reference parameters. Also write a main() program to test the function. Examine the file content afterwards by means of the Notepad program.
10. Change the function in the previous example to print an array of structures to the file. The array and number of items in the array should be sent as parameters.
11. Change the function in the previous example to take a pointer to the structure array and the number of products as parameters.
12. Complete the previous program with a function that reads from the file and presents the information on the screen.

10 Answers

10.1 Variables

Exc. 1

```
#include <iostream>
using namespace std;
void main()
{
  int iNo1, iNo2;
  cout << "Specify 2 numbers: ";
  cin >> iNo1 >> iNo2;
  cout << "You entered: " << iNo1 << " and " << iNo2 << endl;
}
```

Exc. 2

```
#include <iostream>
using namespace std;
void main()
{
  int iNo1, iNo2;
```

```
  cout << "Specify 2 numbers: ";
  cin >> iNo1 >> iNo2;
  cout << "Total = " << iNo1 + iNo2 << endl;
}
```

Exc. 3

```
#include <iostream>
using namespace std;
void main()
{
  int iNo1, iNo2;
  cout << "Specify 2 numbers: ";
  cin >> iNo1 >> iNo2;
  cout << "Total = " << iNo1 + iNo2 << endl;
  cout << "Difference = " << iNo1 - iNo2 << endl;
  cout << "Product = " << iNo1 * iNo2 << endl;
  cout << "Quotient = " << (double)iNo11/iNo2 << endl;
}
```

Exc. 4

```
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
  double iNo1, iNo2, iNo3;
  cout << "Enter 3 decimal numbers: ";
  cin >> iNo1 >> iNo2 >> iNo3;
  cout << "Below are the entered numbers: " << endl;
  cout << setiosflags(ios::fixed) << setprecision(2);
  cout << setw(10) << iNo1 << endl;
  cout << setw(10) << iNo2 << endl;
  cout << setw(10) << iNo3 << endl;
}
```

Exc. 5

```
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
  //Declarations
```

```
int iNo;
double dUnitPr, dPriceExTax, dCustPrice, dTax, dTaxPerc,
  dDisc;

//Entry of quantity and unit price
cout<< "Specify quantity and unit price: ";
cin >> iNo >> dUnitPr;
//Entry of tax
cout << "Specify tax percent: ";
cin >> dTaxPerc;
//Calculations. First the price without tax
dPriceExTax = dUnitPr * iNo;
//Discount:
dDisc = dPriceExTax * 0.1;
dPriceExTax -= dDisc;
//then the tax amount
dTax = dPriceExTax * dTaxPerc / 100;
//and finally the customer price
dCustPrice = dPriceExTax + dTax;

//Output.
cout << endl << "INVOICE" << endl << "=======" << endl;
```

```
  cout << "Quantity:" << setw(11) << iNo << endl;
  cout << setprecision(2) << setiosflags(ios::fixed);
  cout << "Price per unit:" << setw(8) << dUnitPr << endl;
  cout << "Excl. tax: " << setw(12) << dPriceExTax << endl;
  cout << "Discount:  " << setw(12) << dDisc << endl;
  cout << "Total price:" << setw(11) << dCustPrice << endl;
  cout << "Tax:" << setw(19) << dTax << endl;
}
```

Exc. 6

```
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
  //Declarations
  double dNoOfLit, dLitPrice, dTotal;

  //Entry of quantity and unit price
  cout<< "Enter no. of litres and price per litre: ";
  cin >> dNoOfLit >> dLitPrice;

  //Calculations. First the price excl tax
  dTotal = dNoOfLit * dLitPrice;

  //Printout
  cout << endl << "          RECEIPT" << endl;
  cout << setprecision(2) << setiosflags(ios::fixed);
  cout << "Volume:     " << setw(9) << dNoOfLit << " l" <<
          endl;
  cout << "Lit.price: " << setw(9) << dLitPrice << " kr/l"
          <<endl;
  cout << "To be paid:" << setw(9) << dTotal << " kr" <<
          endl;
}
```

Exc. 7

```
#include <iostream>
using namespace std;
void main()
{
  double dPrev, dCur, dPricekwh, dTotal;
```

```
  cout << "Enter current meter value: ";
  cin >> dCur;
  cout << "Enter previous meter value: ";
  cin >> dPrev;
  cout << "Enter price per kWh: ";
  cin >> dPricekwh;
  dTotal = (dCur-dPrev)*dPricekwh;
  cout << "Electricity charge: " << dTotal << endl;
}
```

Exc. 8

```
#include <iostream>
using namespace std;
void main()
{
  int t1, t2, t3, t4, t5;
  cout << "Specify 5 numbers: ";
  cin >> t1 >> t2 >> t3 >> t4 >> t5;
  cout << "Sum = " << t1 + t2 + t3 + t4 + t5 << endl;
  cout << "Average = " << (double)(t1 + t2 + t3 + t4 +
          t5)/5 << endl;
  cout << "Sum of squares ="<<t1*t1+t2*t2+t3*t3
          +t4*t4+t5*t5<<endl;
  cout << "Sum of cubes = " << t1*t1*t1 + t2*t2*t2 +
          t3*t3*t3 + t4*t4*t4 + t5*t5*t5 << endl;
}
```

Exc. 9

```
#include <iostream>
using namespace std;
void main()
{
  int iNo;
  cout << "Enter a number: ";
  cin >> iNo;
  cout << iNo/3 << " and remainder " << iNo%3 << endl;
}
```

Exc. 10

```
#include <iostream>
using namespace std;
void main()
```

```
{
   double dTempC, dTempF;
   cout << "Enter temperature in Celsius: ";
   cin >> dTempC;
   dTempF = 1.8 * dTempC + 32;
   cout << "Corresponding temperature in Fahrenheit is " <<
         dTempF << endl;
}
```

Exc. 11

```
#include <iostream>
using namespace std;
void main()
{
   //Declarations
   int iNoOfMin, iMinLeft, iNoOfHours;

   //Entry of minutes to be converted
   cout << "Enter number of minutes: ";
   cin >> iNoOfMin;

   //Calculate whole hours:
```

```
  iNoOfHours = iNoOfMin / 60;
  //and number of minutes left:
  iMinLeft = iNoOfMin % 60;

  //Printout
  cout << "No. of hours = " << iNoOfHours << endl;
  cout << "No. of minutes = " << iMinLeft << endl;
}
```

Exc. 12

```
#include <iostream>
using namespace std;
void main()
{
  //Declarations
  int iNoOfDays, iDaysLeft, iNoOfMon, iMonLeft, iNoOfYears;

  //Entry of number of days to be converted
  cout << "Enter number of days: ";
  cin >> iNoOfDays;

  //Calculate whole months:
  iNoOfMon = iNoOfDays / 30;
  //and number of days left:
  iDaysLeft = iNoOfDays % 30;
  //Similarly with years:
  iNoOfYears = iNoOfMon / 12;
  iMonLeft = iNoOfMon % 12;

  //Printout
  cout << "No. of years  = " << iNoOfYears << endl;
  cout << "No. of months = " << iMonLeft << endl;
  cout << "No. of days   = " << iDaysLeft << endl;
}
```

Exc. 13

```
#include <iostream>
using namespace std;
void main()
{
  double s, v, t;   //distance, velocity, time
  cout << "Enter distance in km and velocity in km/h: ";
```

```
  cin >> s >> v;
  t = s/v;
  cout << "The trip takes " << t << " hours" << endl;
}
```

Exc. 14

```
#include <iostream>
using namespace std;
void main()
{
  double s, v, t;   // distance, velocity, time
  cout << "Enter distance in km and time in hours: ";
  cin >> s >> t;
  v = s/t;
  cout << "You must drive with " << v << " km/h" << endl;
}
```

Exc. 15

```
#include <iostream>
using namespace std;
void main()
{
  double s, v, t;   // distance, velocity, time
  cout << "Enter average speed in km/h and time in hours: ";
  cin >> v >> t;
  s = t*v;
  cout << "The distance is " << s << " km" << endl;
}
```

Exc. 16

```
#include <iostream>
using namespace std;
void main()
{
  int iOre,iOreLeft,iOre50,iOre50Left,iKr,iKrLeft,iKr5,
      iKr5Left,iKr10,iKr10Left,iKr20,iKr50,iKr50Left,iKr100;
  cout << "Enter number of Swedish ore: ";
  cin >> iOre;

  iOre50 = iOre/50;
  iOreLeft = iOre%50;
  iKr = iOre50/2;
```

```
  iOre50Left = iOre50%2;
  iKr5 = iKr/5;
  iKrLeft = iKr%5;
  iKr10 = iKr5/2;
  iKr5Left = iKr5%2;
  //Take 50kr values before 20, since 20 is not possible
  //to evenly divide in 50
  iKr50 = iKr10/5;
  iKr10Left = iKr10%5;
  iKr20 = iKr10Left / 2;
  iKr10Left -= (iKr20 * 2);
  iKr100 = iKr50/2;
  iKr50Left = iKr50%2;

  cout << "No. of 100 kr notes = " << iKr100 << endl;
  cout << "No. of 50 kr notes  = " << iKr50Left << endl;
  cout << "No. of 20 kr notes  = " << iKr20 << endl;
  cout << "No. of 10 kr coins  = " << iKr10Left << endl;
  cout << "No. of 5 kr coins   = " << iKr5Left << endl;
  cout << "No. of 1 kr coins   = " << iKrLeft << endl;
  cout << "No. of 50 ore coins = " << iOre50Left << endl;
  cout << "No. of ore          = " << iOreLeft << endl;
}
```

Exc. 17

```
#include <iostream>
using namespace std;
void main()
{
  double dLen, dWidth, dHeight, dSpace, dFenceLen,
         dNoOfSticks, dTotMeters;
  cout << "Enter length and width of the field: ";
  cin >> dLen >> dWidth;
  cout << "Enter face height and space between boards
          in m: ";
  cin  >> dHeight >> dSpace;
  dFenceLen = dLen * 2 + dWidth * 2;
  //Each board takes its width 0.10 + one space
  dNoOfSticks = dFenceLen / (0.10 + dSpace);
  dTotMeters = dNoOfSticks * dHeight;
  cout << "Total board length = " << dTotMeters << endl;
}
```

Exc. 18

```
  //Before the last cout statement:
  dTotMeters /= 0.9;
```

Exc. 19

```
  //In the beginning of the program:
  double dMeterPrice;
  cout << "Enter price per meter: ";
  cin >> dMeterPrice;
  //After the last cout statement:
  cout << "Price = " << dTotMeters * dMeterPrice <<
          " kr"<<endl;
```

Exc. 20

```
#include <iostream>
#include <iomanip>    //for formatting of printouts
#include <stdlib.h>     //for random numbers
#include <time.h>       //for system clock
using namespace std;
void main()
{
```

```
  //Declarations
  int iRoll1, iRoll2, iRoll3, iRoll4, iRoll5;
  double dAvg;
  const int iNo = 5;
  //Initiate random number generator
  srand(time(0));
  //Roll 5 times
  iRoll1 = rand()%6+1;
  iRoll2 = rand()%6+1;
  iRoll3 = rand()%6+1;
  iRoll4 = rand()%6+1;
  iRoll5 = rand()%6+1;
  //Calculate average
  dAvg = (double)( iRoll1+ iRoll2+ iRoll3+ iRoll4+ iRoll5) /
         iNo;
  //Printout
  cout << "No. of rolls: " << iNo << endl;
  cout << setprecision(1) <<  setiosflags(ios::fixed);
  cout << "Average score: " << dAvg << endl;
  cout << "Roll scores: "  << iRoll1 << iRoll2 << iRoll3 <<
          iRoll4 << iRoll5;
}
```

Exc. 21

```
#include <iostream>
#include <stdlib.h>     //for random numbers
#include <time.h>       //for system clock
using namespace std;
void main()
{
  //Declarations
  int iRoll11, iRoll21, iRoll31, iRoll41, iRoll51;
  int iRoll12, iRoll22, iRoll32, iRoll42, iRoll52;
  //Initiate radom number generator
  srand(time(0));
  //Roll 5 times
  iRoll11 = rand()%6+1;
  iRoll21 = rand()%6+1;
  iRoll31 = rand()%6+1;
  iRoll41 = rand()%6+1;
  iRoll51 = rand()%6+1;
  iRoll12 = rand()%6+1;
```

```
   iRoll22 = rand()%6+1;
   iRoll32 = rand()%6+1;
   iRoll42 = rand()%6+1;
   iRoll52 = rand()%6+1;
   //Printout
   cout << "The double rolls are: " <<endl<<
             iRoll11+iRoll12<<endl<<iRoll21+iRoll22<<endl<<
             iRoll31+iRoll32<<endl<<iRoll41+iRoll42<<endl<<
             iRoll51+iRoll52<<endl;
}
```

Exc. 22

```
#include <iostream>
#include <stdlib.h>     //for random numbers
#include <time.h>       //for system clock
using namespace std;
void main()
{
   int L1, L2, L3, L4, L5, L6, L7;
   srand(time(0));
   L1 = rand()%35+1;
   L2 = rand()%35+1;
```

```
  L3 = rand()%35+1;
  L4 = rand()%35+1;
  L5 = rand()%35+1;
  L6 = rand()%35+1;
  L7 = rand()%35+1;
  cout << "The Lotto scores are: " <<endl
          <<L1<<endl<<L2<<endl<<L3<<endl<<L4<<endl
          <<L5<<endl<<L6<<endl<<L7<<endl;
}
```

Exc. 23

```
#include <iostream>
#include <stdlib.h>     //for random numbers
#include <time.h>       //for system clock
using namespace std;
void main()
{
  double t1, t2, t3, t4, t5;
  srand(time(0));
  t1 = (double)(rand()%56+180)/10;
  t2 = (double)(rand()%56+180)/10;
  t3 = (double)(rand()%56+180)/10;
  t4 = (double)(rand()%56+180)/10;
  t5 = (double)(rand()%56+180)/10;
  cout << "The temperatures are: "<<endl<<
          t1<<endl<<t2<<endl<<
          t3<<endl<<t4<<endl<<t5<<endl;
}
```

10.2 Selections and Loops

Exc. 1

See the outline of the program in the introductory section about if statements.

Exc. 2

Declare an integer variable to be used for storage of the user entered number. The if statement should then check if the number is less than 15. If so, one of the texts should be printed, otherwise the other text.

Exc. 3

```
//Complete with the followin code
```

```
if (iNo<15)
  cout << "You'll got to stick to the bike some more time ";
else if (tal<18)
  cout << "You are allowed to drive moped";
else
  cout << "You may drive the car";
```

Exc. 4

```
if (iNo1>iNo2 && iNo1>iNo3)
  cout << "The greatest is " << iNo1;
else if (iNo2>iNo1 && iNo2>iNo3)
  cout << "The greatest is " << iNo2;
else
  cout << "The greatest is " << iNo3;
```

Exc. 5

```
if (dGross > 500)
  dDisc = 10;
  else if (dGross > 250)
  dDisc = 5;
  else
  dDisc = 0;
```

Exc. 6

```
cout << "Enter product type (1=food, 2=misc): ";
cin >> iProdType;
if (iProdType == 1)
  dTaxAmount = dGross * 0.12;
  else if (iProdType == 2)
  dTaxAmount = dGross * 0.25;
  else
  cout << "Wrong product type";
```

Exc. 7

Use else if to locate different intervals. For 0-10000 the tax is 0. For 10000-50000 the tax is 0.5*income-5000. For 50000-100000 the tax is 0.5*income. For >100000 the tax is income*0.5 +(income-100000)*0.2.

Exc. 8

See the code proposal in the 'Even or Odd' section.

Exc. 9

Repeat the code so that you divide by 3 instead of 2 and print a suitable text.

Exc. 10

Declare a variable for each type of value and read the quantity of each. The sum of the crown types is:

```
iNoOf1kr + iNoOf5kr * 5 + iNoOf10kr * 10
```

For the 50-ore coins the number of whole crowns will be:

```
iNoOf50ore / 2
```

provided that iNoOf50Ore is declared as integer. If iNoOf50ore is odd, there will be an extra 50-ore coin:

```
if (iNoOf50ore%2 == 1)
  cout << " and 50 ore";
```

Exc. 11

Use else if **repeatedly** to locate the various intervals and print corresponding discount.

Exc. 12

```
if (iNo>20 && iTotal>1000)
  dDisc = 0.2;
else if (iNo>20 || iTotal>1000)
```

```
  dDisc = 0.1;
else
  dDisc = 0;
```

Exc. 13

```
case 9:
  cout << "You selected to exit";
  break;
```

Exc. 14

```
case 4:
  cout << "The product is " << iNo1*iNo2;
  break;
```

Exc. 15

Combine the method for the previous menu program and the method to compare three numbers in a previous exercise.

Exc. 16

See the first program in the section about loops.

Exc. 17

Also print iNo*iNo*iNo inside the loop.

Exc. 18

```
do
{
  cout << "Enter a number: ";
  cin >> iNo;
}while (iNo!=0);
```

Exc. 19

Initiate a variable to 0 to keep track of the sum. Inside the loop you increase the sum variable by the entered value. After the loop you print the sum variable.

Exc. 20

Use the program structure from the previous program. Inside the loop you write an if statement which checks if iNo is less than 0 and print a suitable text.

Exc. 21

```
do
{
```

```
   cout << "Enter a number: ";
   cin >> iNo;
}while (iNo%3 != 0);
```

Exc. 22

Use input of the score in the while condition:
`while (cin>>iScore)`
Use a sum variable to store the accumulated sum of all scores entered so far. The accumulation is made inside the loop.
Also declare two variables, iGreatest and iLeast, which store the greatest and the least score respectively. Inside the loop you will have to check if the recently entered score is greater than iGreatest. If so, assign this new value to iGreatest. Do the same with iLeast.
After the loop you subtract iGreatest and iLeast from the sum before printing the sum.

Exc. 23

Insert a statement which reads the requested product from the keyboard.

Exc. 24

Start from the program in the previous exercise. Let the loops go to 100 instead. Inside the first loop you check if the division numerator/denominator is 5 and if
numerator%denominator is 0. Is so, you print the numerator and the denominator.

Exc. 25

Print iRoll inside the loop.

Exc. 26

```
while (iRoll!=5 && iRoll!=6)
```

Exc. 27

Print iRoll1 and iRoll2 from inside of the loop.

Exc. 28

```
while ((iRoll1 + iRoll2) != 12)
```

Exc. 29

Write an outer loop which goes from 1-13 and an inner which goes from 1-5. In the inner loop you create the random scores for the first match in the 5 different pools. You will have to accommodate the printout by means of the proper number of blanks to get the 5 pools separated from each other in 5 nice columns.

Exc. 30

rand()%2 gives a random number between 0-1.
Inside the loop you check if the number is 0. If so, you print "heads", otherwise "tails".

Exc. 31

You can solve this in different ways. One way is to first create a random number in the interval 0-3, where the 4 different numbers correspond to colour (hearts, spades etc.). Then you can create a new random number between 1-13, where 2-10 is the corresponding value, 11 is jack, 12 is queen, 13 is king and 1 is ace.

Exc. 32

Change the calculation of LP to:

```
LP = iRoot * iRoot - 8 * iRoot + 15;
```

Exc. 33

Change the calculation of LP to:

```
LP = iRoot * iRoot * iRoot - 9 * iRoot * iRoot + 23 * iRoot - 15;
```

The while condition should check that iNo<3.

10.3 Arrays

Exc. 1

```
int iNos[10];
for (int i=0; i<10; i++)
{
  cout << "Enter integer: ";
  cin >> iNos[i];
}


a)
cout << iNos[0] << " " << iNos[4] << " " << iNos[9];


b)
In the for-loop you increase a sum variable by the recently entered number:
iSum += iNos[i];
After the for-loop you print iSum.


c)
for (i=9; i>=0; i--)
{
  cout << iNos[i] << " ";
}


d)
Create a loop which checks:
if (iNos[i] < 0)
  iNos[i] = -iNos[i];


e)
Read a number from the user to a variable iUserNo. In a loop you check:
if (iNos[i] < iUserNo)
  cout << iNos[i] << " ";


f)
Read a number from the user between 0-9 to the variable k. Then print
iNos[k].
```

g)
Read a number from the user. Then write a loop which checks:

```
if (iNos[i] == iUserNo)
   cout << i;
```

h)
Copy iNos[0] to a variable temp. Then write a loop which goes from 0-8 and shifts location to the left:

```
iNos[i] = iNos[i+1];
```

Then you copy:

```
iNos[9] = temp;
```

Exc. 2

Complete the declaration of iDaysInM. Then let the user enter a month number to be used as index in the array (decreased by 1).

Exc. 3

Create a loop from 0-30 which creates random temperatures for the array:

```
dblTempJuly[i] = rand()%11 + 15;
```

Copy item by item to the array dblTempAug in a loop.
Then print dblTempAug in a loop.

Exc. 4

Code example is given in the "Comparing Arrays" section.

Exc. 5

Assign another value to one of the items of dblTempAug, e.g.:

```
dblTempAug[12] += 1;
```

Exc. 6

See code example in the "Average" section.

Exc. 7

Initiate the array dDens with the densities. Read one density from the user. Then write a loop which goes through all densities and checks if dDens[i] is greater than the entered value. If so, print dDens[i-1] and break the loop, so that only one value is printed.

Exc. 8

Create a loop from 0-24 which assigns values to the items:

```
iNos[i] = rand()%10;
```

Read a value from the user and create a loop which increments a variable each time an item equals the entered value:

```
int iNo = 0;
for (i=0; i<25; i++)
  if (iEntryValue == iNos[i])
    iNo++;
```

Then print iNo.

Exc. 9

```
if (sales[i] <= dLimit5)
  dFee = perc1 * sales[i];
  else if (sales[i] <= dLimit10)
  dFee = perc1*dLimit5 + perc2*(sales[i]-dLimit5);
  else
  dFee = perc1*dLimit5 + perc2*(dLimit10-dLimit5) +
    perc3*(sales[i]-dLimit10);
```

Exc. 10

Declare a new array iNoOfSales[100] which should hold the number of sales per salesman. Initiate the array to 0 values. When a salesman enters a value, you increase:

```
iNoOfSales[nr-1]++;
```

The printout should contain an additional column where you print iNoOfSales[i].

Exc. 11

Add one more column to the output where you print sales[i]/iNoOfSales[i].

Exc. 12

See code example in the "Product File, Search" section.

Exc. 13

Read a quantity from the user and multiply it by the achieved price.

Exc. 14

Declare and initiate a two-dimensional array according to the code example in the "Two-Dimensional Array" section. Let the user enter product group and customer group and use these values as indeces in the discount matrix to get a discount percent. Calculate the discount amount by multiplying the discount percent by the total price, and subract the discount amount from the total price.

Exc. 15

Code example for random dice rolls is given in the chapter about Selections and Loops. Code example for sorting is given in the "Sorting" section.

Exc. 16

Code example is given in the "Searching a Sorted Array" section.

Exc. 17

```
if (iSrch == iProdid[iMid])
{
  iFound = 1;
  iPos = iMid;
  cout << iMid;
}
```

Exc. 18

```
if (r == l+1 && iFound == 0)
  cout << "The product id was not found";
```

In this situation you have enclosed an interval with the distance 1 between l and r. If then not found, the product id is not stored in the array.

Exc. 19

Declare and initiate a price array with prices for the products. Use the value of the variable iMid to get the corresponding price of the price array.

10.4 Strings

Exc. 1

Add the following statements:

```
cout << "E. Statistics" << endl;
and:
case 'E':
  cout << "You selected Statistics";
  break;
```

Exc. 2

Start from the program 'Menu Program with Loop'.

Exc. 3

Read a character from the keyboard to a char variable. Then create a loop which goes from 1 to 10 and prints:
cout << cChar;

Exc. 4

Create an outer loop which goes from 1 to 10 with the loop variable i. Then create an inner loop which goes from 1 to i. Print the character i in the inner loop. After the inner loop, but inside the outer you print endl for line feed.

Exc. 5

Read the character to a char variable for instance named cChar. Read the number to an integer variable for instance named i. Then create a loop which goes from 1 to i. Inside the loop you print cChar.

Exc. 6

The first outer loop goes from 14 to 0. The second inner loop goes from 0 to 14-i. The last inner loop goes from 0 to 13.

Exc. 7

Read the character to a char variable. Then assign this value to an integer variable and print it.

Exc. 8

Read the character code to an integer variable. Assign this value to a char variable and print it.

Exc. 9

Use the character codes in the section about å, ä and ö in this chapter.

Exc. 10

Read the text to a string array with cin.getline(). Then use the functions strlen() and sizeof.

Exc. 11

Increase by 32 instead of decrease.
If you enter other characters, an increase by 32 gives a character that does not correspond to the relation between upper and lower case.

Exc. 12

Read the word to a char array. Then create a loop which goes from 0 to the length of the string minus 1. In each turn of the loop you print a character twice:

```
cout << cWord[i] << cWord[i];
```

Exc. 13

Read the word to a char array. Then create a loop which goes from 0 to the length of the string minus 1. In each turn of the loop you print a character:

```
cout << cWord[i];
```

The other alternative is solved with:

```
cout << strrev(cWord);
```

Exc. 14

Change to the following code:

```
index = 1;
for (i=1; i<iLen; i++)
  if (cName[i] == ' ')
  {
    cInit[index] = cName[i+1];
    index = 2;
  }
cInit[3] = '\0';
```

Also adjust the declarations of the variables.

Exc. 15

Write a new if statement where you check whether a character is greater than 90. If so, decrease the character code by 32 before inserting the character into the initials string.

Exc. 16

Create a loop which goes from character1 to character2 and prints the character corresponding to each character code.

Exc. 17

Use the strcmp() function which compares strings.

Exc. 18

Create a loop which goes from 0 to number of characters minus 1. Inside the loop you check whether cWord[i] equals a, o, u, e, i or y. If so, increase a counter (integer variable) by 1.

Exc. 19

Do like in the previous exercise when checking if the character is a consonant. It it is, you print:

```
cout << cWord[i] << "o" << cWord[i];
```

Exc. 20

See the code example in the 'Sorting Strings' section.

Exc. 21

Declare a new array with prices:

double dPrices[5];
and read values to it.
Use the formatting functions setiosflags(), setprecision() and setw() for the printout.

Exc. 22

See the example code in the "Interchanging First Name and Surname" section.

Exc. 23

Create a loop which goes through all characters of the entered string and checks if cEmailadr[i] == '@'. Print a suitable text in both cases.

Exc. 24

Use the strcat() function to concatenate two strings.

Exc. 25

Create a loop which searches for the period in the email address and a loop which searches for @. Save the positions of these characters. Then use the functions strcpy() and strncpy() to extract first name and surname. Then decrease the character codes of the first characters by 32 in the first and surnames.

Exc. 26

Change to:
cEncrypt[i] = cName[i] - 1;

Exc. 27

Change to:
cEncrypt[i] = cName[i] - 3;

Exc. 28

Increase by 3 instead of decrease.

Exc. 29

Find out the code of a character and decrease by 65, so that A corresponds to 0, B corresponds to 1 etc. (the variable iCode). The encrypted character should then have the code 90-iCope, i.e. the code for 'Z' decreased by the value of iCode.

Exc. 30

Insert all characters into one bit string array:
char cKey[63] = "ABCD…Zabcd…z0123…9";
Then create a random position in this array:
iPos = rand() % 63;

The character cKey[iPos] is then used for the password:

```
cPw[i] = cKey[iPos];
```

Exc. 31

Change to:

```
iLen = rand() % 5 + 6;
```

Exc. 32

Insert also lower cases in the string cKey. Change the limit of the first for-loop to 52. Let j get the value:

```
j = rand() % 52;
```

In the last for-loop you will have to check whether to subtract 65 or 71 from the character code, since there is a gap of 6 characters in the interval between upper case Z and lower case a.

```
if(cText[i]<97)
       cEncrypt[i] = cKey[cText[i] - 65];
else
  cEncrypt[i] = cKey[cText[i] - 71];
```

10.5 Functions

Exc. 1

```
double dAvg(double x1, double x2, double x3)
{
  double mv;
  mv = (x1 + x2 + x3)/3;
  return mv;
}
```

Exc. 2

Start from the min() function at the beginning of this chapter and modify the if condition.

Exc. 3

See the code proposal in the "Least of Three Numbers" section.

Exc. 4

Add this statement:

```
n=max(iNos[i],n);
```

and

```
cout << n << " is the greatest number" << endl;
```

Exc. 5

```
double dCirc(double dLen, double cWid)
{
  return 2 * (dLen + dWid);
}
double dArea(double dLen, double dWid)
{
  return dLen * dWid;
}
double dPrice(double l, double b)
{
  return dCirc(l, b) * 145 + 650;
}
```

Exc. 6

Write a for-loop in the function which goes from 1 to n and prints i, i*i and i*i*i.

Exc. 7

Write a for-loop in the function which goes from 1 to n and prints c.

Exc. 8

Write a for-loop in the function which goes from 0 strlen(cWord)-1 and prints:
cout << cWord[i] << " ";

Exc. 9

In the chapter about Strings you will find code that solves these tasks.

Exc. 10

The new function header without parameters will be:
double dCustDisc()
Inside the function you read a character from the user. A switch statement can then return the correct factor (0.05, 0.07 or 0.09). Also use a default section providing a en error message about "wrong customer category" and returns 0.
In the dPrice() function you change to:
return dLinePr * (1-dDiscPerc)*(1-dCustPerc)*(1+dTax);

Exc. 11

In main() you read the number of days, which is sent to the function dayCost(). That function calls kmInput() which reads start and end value. The difference between these values is returned to dayCost(). The function litrePrice() is called, which reads the fuel consumption in litres and returns that value * 9.27. The function dayCost() then returns the number of days * 500 plus number of km * 1.4 plus the fuel cost.

Exc. 12

Create a for-loop in the function which goes from 0 to strlen(cWord)-1 and checks if strcat(cWord[i], cWord[i+1]) equals "aa". If so, the character 'å' is printed, otherwise cWord[i] is printed. In the same way you check the characters "ae" and "oe". The character codes for å, ä and ö are given in the "Strings" chapter.
Try to solve this problem without looking at the code proposal below:

```
void check (char s[])
{
      int len=strlen(s);
      char p[3];
      for (int i=0; i<len; i++)
      {
            p[0]=s[i];
```

```
            p[1]=s[i+1];
            p[2]='\0';
        if (strcmp(p, "aa") == 0)
        {
              i++;
              cout << "\x86";
        }
    else if (strcmp(p, "ae") == 0)
    {
              i++;
              cout << "\x84";
        }
        else if (strcmp(p, "oe") == 0)
        {
              i++;
              cout << "\x94";
        }
        else
              cout << s[i];
     }
}
```

Exc. 13

Change to:

```
if ((c>='0' && c<='9') || (c=='.'))
```

Exc. 14

```
if ((val>=2) && (val<=10))
  cout << val;
else
  switch val
  {
    case 1:
      cout "Ace";
      break;
    case 11:
      cout << "Jack";
      break;
    etc.
```

Exc. 15

Create another switch statement which tests the other parameter and prints the correct colour of the card.

Exc. 16

In the functions odd() and divable() you use the modulus operator %. In the other functions you check if the character code of c[0] is in the correct interval.

Exc. 17

In main() you read values to three different variables, which means that the user has to enter a blank between each character. The three variables must be of type int, char and int. In a switch statement you then check if the char variable is +, -, * or / and send the two integer variables to the respective function. The functions return int (+ - *) or double (/). The returned values are a+b, a-b, a*b and (double)a/b.

Exc. 18

The function takes a char[] parameter and returns 0, 10, 15 or 20. Use a switch statement that compares the first character of the parameter against the first character of "G", "VG", "MVG" and "IG". In main() you use a loop for entry of score and number of hours. In the loop you accumulate the score * number of hours for each course. Also accumulate the total number of hours. After the loop you divide the total score by the total number of hours.

Exc. 19

```
double dDiscount(int iNo, double dTotPrice)
double dDiscount(char cCustGroup)
```

Exc. 20

See the section 'Declaration - Definition'.

Exc. 21

See the section 'Project'.

Exc. 22

Use & after the data type, for instance:

```
double dayCost(int& iNo)
```

Exc. 23

```
double fuelPrice(double dLitrePrice=9.32)
```

Exc. 24

```
double dRoll(int iNo=5)
```

In the function you write a loop which goes from 1 to iNo and create random dice scores with rand()%6+1. These are accumulated in the loop. After the loop you divide the sum by iNo.

Exc. 25

In main() you need one variable which stores the user score and one that stores the computer's score.

Exc. 26

The user and computer scores are sent as parameters to the function, where you compare them and print a suitable text.

Exc. 27

Use & for the parameters.

Exc. 28

See the section 'Recursive Functions'.

Exc. 29

```
int nsum(int n)
{
  if (n<=1)
    return 1;
```

```
   else
      return n + nsum(n-1);
}
```

Exc. 30

```
int nsum(int n)

{
   if (n<=1)
      return -1;
   else
   {
      int k;
      if ((n%2)==0) //If the number is even, k is set =1
         k=1;
      else
         k=-1;
      k *= n;   //Here odd numbers will be negative
      return (k + nsum(n-1));
}
```

10.6 Files

Exc. 1

Add the statement:

```
cin.getline(cWhloc,8);
```

at two different locations in the program

Also change the file output statement:

```
outfile << cProd << endl << cWhloc << endl;
```

Exc. 2

Change to:

```
while(infile.getline(cProd,29) &&
  infile.getline(cWhloc,8))
     cout << cProd << "   " << cWhloc << endl;
```

Exc. 3

Read the warehouse location from the user in the same way as the product name. Also write the warehouse location to the file with endl in between.

Exc. 4

Add:

```
cout << " ...and stock quantity: ";
cin >> iQty;
```

Change to:

```
outfile << iProdId << endl << dPrice << endl << iQty << endl;
```

Exc. 5

Change to:

```
while(infile >> iProdId >> dPrice >> iQty)
```

Add:

```
cout << "Stock quantity is " << iQty;
```

Exc. 6

You will need another two-dimensional array, where you store the warehouse locations. The sort function must have an additional parameter for the warehouse location array. When interchanging position of two product names, you must also interchange corresponding warehouse location items.

Exc. 7

You will need additional statements to read the new stock quantity value. In the file output statement you must also print the stock quantity to the file.

Exc. 8

Start from the program where you print product name and warehouse location. You should modify it to print first name, surname and city. Use ios::app to prevent cancellation of previous information in the file.

Exc. 9

Start from the program reading products and warehouse locations from a previous exercise, and modify it to conform to the file created in the previous exercise.

Exc. 10

Compare to the program where you were able to change price and stock quantity (exc. 7). Remember to read with getline since it is strings and not numeric values in this program.

Exc. 11

The program has a similar structure as the previous program. The only difference is that, when the name from the file equals the one entered by the user, you don't print that name to the file.

Exc. 12

Start from the program in Exc. 6 which sorted products and warehouse locations. You will however need an additional two-dimensional array, so that you have one for first names, one for surnames and one for cities. When interchanging positions of two surnames, you must also interchange corresponding positions in the two other arrays.

Exc. 13

You can use the code example in the 'Copying Files' section.

Exc. 14

You need one additional while loop where the menu is printed and where the user can enter a menu choice. By means of a switch statement you can perform the requested task. Utilize the code already created in the previous exercises.

10.7 Pointers

Exc. 1

See the code example in the "Assigning Values to Pointers" section.

Exc. 2

See the code example in the sections "Addresses and char Pointers" and "cout and char Pointers".

Exc. 3

Add this code:

```
  double dDisc;
  double* pDisc = &dDisc;
```

and

```
  cout << "Enter discount in percent: ";
  cin >> *pDisc;
```

Change the calculation statement to:

```
  *pTotal = *pNo * *pPrice * (1 - *pDisc/100);
```

Exc. 4

It should be possible to use the program in the section "Price Program with Pointers" as the starting point. The number of litres should be divided by the number of km to get the fuel consumption.

Exc. 5

See the code example in the section "Price Program with Pointers".

Exc. 6

Declare the array as shown in the section "Pointer Arithmetics". Then write a for-loop which reads the values from the user. Increment the pointer variable

with the ++ operator. Write another for-loop which prints the values. Don't forget to reset the pointer to the first item of the array before the last for-loop.

Exc. 7

Declare a sum variable and initiate it to 0. Declare a pointer (pSum) which points to the sum variable. In the first for-loop you accumulate the read values with a statement like this:
*pSum += *pNumber;

Exc. 8

Delete the initiation lists for iSal and dTax. Write a for loop which reads one salary and one tax percent at a time by means of the pointers. Don't forget to increment the pointers with the ++ operator inside the loop. Before printing the table you must reset both pointers to point to the first item of each array by decreasing them by 6:

```
pSal = pSal - 6;
pTax = pTax - 6;
```

Exc. 9

First, prepare the file by writing in Notepad each second salary and each second tax percent. Press Enter after each item. You might also need to take a look in the Files chapter to be able to declare an instream correctly.

Exc. 10

First prepare the file like in the previous exercise. Then it should be possible to accommodate the code from the previous exercise.

Exc. 11

Delete the printout code of the table. First read the product id from the user. Then write a loop which goes through the items of the product array and checks if the entered id equals the one in the array. If so, the corresponding price is printed:

```
for (int i=0; i<iNoOfProducts; i++)
{
  if (*pEnteredProdId == *pProdId)
    cout << "The price is: " << *pPrice;
  pProdId++;
  pPrice++;
}
```

Exc. 12

First reset the pointer to the beginning of the string and then print it in the cout statement.

Exc. 13

Before sending the pointer to the function you can create a copy of the string, so that the copy is not modified by the function. The copy can be printed, which gives the original email address.

Exc. 14

The function should go from character to character in the string by means of pointer arithmetics and check if the character code exceeds 96. If so, decrease it by 32. The new character should replace the original in the string.

Exc. 15

The function should use pointer arithmetics when stepping through the items of the array. Create a local variable, iMax, in the function to which you assign the value of the first item. Then a loop will step through the remaining items and compare them to iMax. If any item exceeds iMax, you set iMax equal to that item. Return iMax.
In the main() program you declare an array of 8 items and a pointer which points to the first item of the array. In a for-loop you let the user enter values to be stored in the location pointed at by the pointer. Don't forget to increment the pointer for each turn of the loop. Before calling the function you must decrease the pointer by 8 to make it point to the beginning of the array. Send the pointer and the number 8. Print the returned value.

Exc. 16

In the main() program you first let the user enter the number quantity. Then declare the array dynamicly. Also check that the requested memory space could be allocated as described in the section "Dynamic Memory". Don't forget to release the dynamic memory with delete at the end of the program.

Exc. 17

You should in principle be able to copy some of the lines in the program and accommodate them to the telephone numbers:

```
  char cTeltemp[15];  //Temporary storage of entered tel no
  char *cTel[iNo];  // Pointer array with telno:s

    cout << "Telephone number " << i + 1 << " ";
```

```
cin.getline(cTeltemp, 15);   // Temporary storage

cTel[i] = new char[strlen(cTeltemp) + 1];
// Copy the telno to the pointer array:
strcpy(cTel[i], cTeltemp);

cout << cNames[j] << "   " << cTel[j] << endl;

delete [] cTel[k];
```

Exc. 18

A checking example is given in the first program of the section "Dynamic Memory". A similar check is inserted at each usage of the new keyword.

10.8 Structures

Exc. 1

Start from the Prod structure given in the chapter text and modify data types and names of the members.

Exc. 2

See the code example of the section "A Structure Program", where entry to a structure variable is given.

Exc. 3

See the section "Array with Structure Variables" where code examples are given.

Exc. 4

Create a loop which goes through each customer and prompts the user for the invoice total. It is added to the structure member which contains invoice total year-to-date.

Exc. 5

See the section "Array with Structure Variables" where code examples are given.

Exc. 6

See the section "Pointer to Structure" where code examples are given.

Exc. 7

See the section "Pointer to Structure" where code examples are given.

Exc. 8

Use the code example in the section "Structures in the Dynamic Memory".

Exc. 9

See the section "Structure as Function Parameter – Reference Parameter". Declare an outstream with ios::app. You might need to take a look in the Files chapter.

Exc. 10

See the section "Structure as Function Parameter – Array Parameter".

Exc. 11

See the section "Structure as Function Parameter – Pointer Parameter".

Exc. 12

Declare an instream from which you read. The input goes to a structure which is printed on the screen. The function is of void type and takes no parameters.