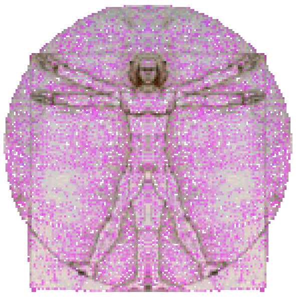
PROBIQ

Borland C++ Builder 5.0 Manual

bfi

bytework

ANS



B·F·W

RNIB

idec

TRANSFER Slovensko®
spol. s r.o.
Centrum vzdelávania manažérov

Education and Culture

Leonardo da Vinci

This project was co-financed by the European Union

Introduction to the ProBiQ BCB5 manual

This manual and learning guide is designed for use by students and tutors wishing to program using Borland C++ Builder version 5. The materials used are varied and interesting and provide sufficient knowledge to enable an individual to use their skills in the workplace.

The manual begins with an introduction to the fundamentals of programming and, using a variety of teaching and learning tools, guides the student through basic modules, culminating in advanced programming techniques.

The theoretical materials included in the early chapters will help develop good background knowledge and have been designed to be used as a reference guide as well as a teaching aid. Later in the manual, practical tasks and exercises allow the students to practice and demonstrate what they have learned.

We wish you a successful course with this manual!

Content

Chapter 1: Introduction to programming

Aim of this chapter:
The student will learn the basics of programming including the fundamentals of developing algorithms

1.1 Number Systems

Aim of this subchapter:
The student should know about the different number systems and be able to convert numbers from one system to another.

This chapter will give an overview to the number systems, especially those used in computer technology: the binary system, the hexadecimal system, and the octal system. We will learn how these are connected and how these can be converted one to another. First we will start with our conventional number system, the decimal system.

1.1.1 The Decimal System

The decimal system is used in everyday calculation. It consists of ten symbols, the Arabic numbers. Each symbol is interpreted on its place in a number and has to be multiplied with a power of ten. Therefore the decimal system is often called place-value-system. The value of a numeral in a number is called its place value. Depending on the position of the numeral it is multiplied with a power of the basis of the numbering system. In the decimal-system the basis is the number 10.

Example:

Let’s take the number „240568“:
If we take into account the place value of the numerals we have to calculate:

$$2 * 10^5 + 4 * 10^4 + 0 * 10^3 + 5 * 10^2 + 6 * 10^1 + 8 * 10^0 = 240{,}568$$

In computer systems another numbering system is used, the binary-system. Let us see what happens there.

1.1.2 The Binary System

The binary system is like the decimal system a place-value-system. The base of the binary system is 2. That means, the place-value of a numeral increases from place to place by a power of 2.

How many numerals has the binary system?

If we divide any desired integer by 2 the carry will be 0 or 1. So the binary system is build by the digits 0 and 1.
Now we see why this numbering system is so important for computer systems. A computer works with magnetism and electric currents. An electric current can circulate or not, magnetic materials can be magnetised or not. All of these can be described by two numbers, 0 and 1.

Examples:
The binary number 11001011 has to be displayed in the decimal system:
$(11001011)_2 = 1 * 2^7+1 * 2^6+0 * 2^5+0 * 2^4+1 * 2^3+0 * 2^2+1 * 2^1+1 * 2^0 = 203$
The other way round:
$(203)_{10}$ =
203/2 = 101 and remainder = 1; (Least Significant Bit:LSB)
101/2 = 50 and remainder = 1;
50/2 = 25 and remainder = 0;
25/2 = 12 and remainder = 1;
12/2 = 6 and remainder = 0;
6/2 = 3 and remainder = 0;
3/2 = 1 and remainder = 1;
1/2 = 0 and remainder = 1. (Most Significant Bit:MSB)
If we read the remainders from bottom to top we get the binary number 11001011.

In this example we see that we can derive a binary number from a decimal number if we divide the decimal number by the base 2 and take the remainder.
Now we switch to another important numbering system for computers, the hexadecimal system.

1.1.3 The Hexadecimal System

Because of its name we easily understand that this is a place number system on the base 16. But how do we arrive at 16 numerals, as until now we know only 10. The mathematicians have solved this problem by agreeing to take the capital letters A, B, C, D, E, and F for the numbers 10, 11, 12, 13, 14, and 15.
Example:

$(0A6E)_{16} = 10 * 16^2+6 * 16^1+14 * 16^0 = 2{,}670$

We derive the hexadecimal presentation of decimal numbers easily by computing first the binary number and than splitting the binary number into groups of four:

Example:
$(2670)_{10}$ = $(0000\ 1010\ 0110\ 1110)_{2}$ = $(0A6E)_{16}$

Finally, we will have a look at another numbering system, that has not as much relevance in computer systems as the binary or hexadecimal system: the octal system.

1.1.4 The Octal System

The base of this place-value-system is 8 and we use only the numerals from 0 to 7.

Example:
$(127)_{8} = 1 * 8^{2}+2 * 8^{1}+7 * 8^{0} = (87)_{10} = (0101\ 0111)_{2} = (57)_{16}$
In this system we can take a binary number and build couples of three, as you see in the following example:
$(001\ 010\ 111)_{2}= (127)_{8}$

1.1.5 Further exercises:

In this exercises you should get used to calculating with binary numbers and to transform the different numbering systems one in another:

a. Transform following decimal numbers to binary numbers:
361
255
129

b. Transform following binary numbers to decimal numbers:

1111100101
10000101
10101010

c. Transform following decimal numbers to hexadecimal numbers:

894
777
1010

d. Transform following hexadecimal numbers to decimal numbers:

3AB
23F
ABC

e. Try to transform the numbers in exercise d. to binary numbers. Remember at that point the method of building pads of 4.

1.2 Binary Numbers

Aim of this subchapter:
As the most important number system in computer technology the student will learn about the rules in calculating with binary numbers.

In this chapter you learn how to add, subtract, multiply and divide two binary numbers and that all calculation with binary numbers can be reduced to simple addition.

1.2.1 The Theory about Computing with Binary Numbers

In computing with binary numbers there are the following algebraic rules:
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0 and 1 remainder for the next place.

With this rules it is easy to manage the addition of 2 binary numbers.
Example:

```
1 1 1 0 1
+              1 0 1 0
remainder         1 1
             1 0 0 1 1 1
```

Computers are not able to distinguish between the different length of these binary numbers. But this is not necessary because the computer puts each data to a special cell in the computer memory. Depending on the computer used, this cell can hold 8, 16, or 32 bits of data.

If the computer has 16 bits of memory in a cell, it would interpret the example used above in the following way:

```
        0000 0000 0001 1101
+       0000 0000 0000 1010
        0000 0000 0010 0111
```

The so called Most Significant Bit (on the left side) defines the algebraic sign, 0 for minus and 1 for plus. The last number on the right side is called “Least Significant Bit”.
But how would we write negative numbers? We write it in applying a rule called Ones Complement:
0 leads to 1, 1 leads to 0.

Example:

+23 is 0001 0111
-23 is 1110 1000

Now we are able to subtract one binary number form another:

Example:

We will compute 13+(-10):
13 is 1101 in binary
10 is 1010 in binary

```
        0000 1101
+       1111 0101 Ones Complement
Remainder 1       0000 0010
```

Now we have a problem: What should be done with the last remainder? Because the memory cell only can receive 8 bit, that is one byte, the remainder is added to the Least Significant Bit. So 0000 0010 plus the remainder leads to 0000 0011.
This binary number is equal to the decimal number 3 and is the correct result of our calculation.
Remark: The last remainder is often added to the Least Significant Bit at the beginning. This is called the Twos Complement:
Subtraction in binary numbers is done by the addition of Twos Complement.

Example:

We will compute 13+(-10) again:

```
                1101 13 as decimal
                0101 Ones Complement of 10
+               1    last remainder
        1       0011
```

Because we already added the last remainder we have to withdraw the leading 1. The result is 3 in decimal.
We remember as a basic rule:
The Ones Complement + 1 at the least significant bit = the Twos Complement
Following this considerations we see, that all arithmetic can be reduced to simple addition. Multiplication represents only a multiple addition, division only a multiple subtraction and subtraction itself is, as shown, reduced to simple addition.

1.2.2 Further exercises:

a. Add the following binary numbers:
11101 and 10101
100101 and 111001

b. Subtract the following binary numbers:
1011 from 111001
10010 from 10000101
1100 from 1011
1111 from 1101

Remember the remainder

c. Do the following multiplications:
$(1011)_2 \times 8_{10}$
$(1010)_2 \times 4_{10}$

Do you determine any rules?

What do you think the result of the following would look like?

„1101 multiplied with 16“

1.3 Logics

Aim of this subchapter:
The student will learn, how a computer does all the calculation with three logical functions.

In this chapter the logical gate AND, NOT and OR are discussed. At last we will do some addition, as a computer will perform it.
In the last chapter we indicated, that all computer calculation can be reduced to addition. But how does the computer perform this task. Surely it cannot calculate as humans do, but does all calculation by using simple logical elements.

These elements are also called logical gates.
There are three kinds of gates:
The NOT-element (Negation)
The AND-element (Conjunction)
The OR-element (Disjunction)

We will illustrate each gate by a table of values also called truth-table. It shows all possible combinations of two input values A and B and displays the output value C.

1.3.1 Negation

The NOT-element is the basic gate. It simply twists the input value to its complement. Because the computer knows only 0 and 1 there is no problem.
The Ones Complement is done by:

A	C = NOT A
t	f
f	t

t stands for true, f for false.
T and f is often show as 1 and 0:

A	C = NOT A
1	0
0	1

1.3.2 Conjunction

The AND-element is carried by two input values. Both input values have to be true to lead to an output value also true.

A	B	C=A AND B
1	1	1
1	0	0
0	1	0
0	0	0

1.3.3 Disjunction

The OR-element also has two input and one output value. In this case only one input value has to be true to become an output value also true.

A	B	C=A OR B
1	1	1
1	0	1
0	1	1
0	0	0

1.3.4 Performing Addition

With this logic gates the computer does all calculations. We will show in an example how addition works. In addition to the result C there may be a remainder

The Half-Adder*:*

Remark to the trainer:

This graphic has to be built by LEGO or drawn on the back of the blind students.

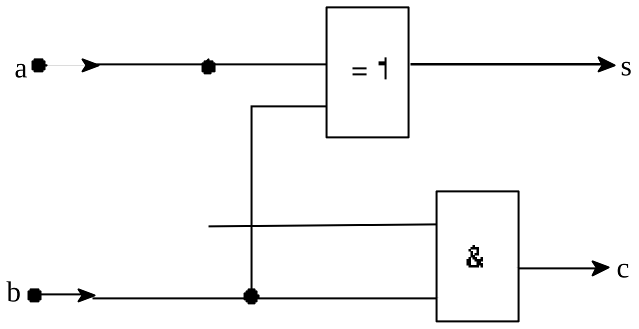


Abbildung 4

A and b are logical variables that have to be added, s is the sum, c the carry. The sum is established by the so called XOR-element (exclusive OR). This gate proves that precisely one input value has to be true leading to an output value that is also true (see truth-table).

Here is the proper truth-table:

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

1.4 Computer and Algorithms

Aim of this subchapter:
The student will learn, what is the principle computing and what influences the power of a computer.

Content:
In this chapter we will talk about algorithms, processes, processing units, programmes and programming languages.

1.4.1 Introduction

History has led to two revolutions. The first was the industrial revolution, the second the computer revolution. In the first case it led to a better utilisation of physical power, in the second case to a better utilisation of mental skills.

But how did the computer have such an revolutionary impact? First the computer is a machine doing simple mental routine jobs by performing simple operations in a high speed. The simplicity of the operations (typical examples are addition and comparing two numbers) is disproportionate to the calculation-speed of the computer (hundred millions or billions operations per second). The result is a high output of operations so that major tasks can be done in a short time.

But this leads to a problem. To direct a computer to do a job is to tell it which operations should be carried out. We have to describe in words how the job has to be done. This description is called algorithm. An algorithm describes a method to solve a problem. The algorithm consists of a sequence of steps, which have to be done without errors to lead to a desired result. The attention handling is called the Process.
Our everyday life is full of algorithms, for example:

Process	Algorithm	Typical steps to algorithm
Baking a cake	recipe	weigh flour and sugar, take three eggs
Play music	sheet of music	interpret the notes
Knit a pullover	pattern for knitting	reverse stitch, full stitch

With these examples, the term algorithm is clear. But what is a process?
Generally, a unit carrying out processes is called a processing unit. A processing unit may be a person or a computer. A computer is a very specialised processing unit, therefore computers have this great influence on our everyday life.

A general computer has three main components:
These are
The Central Processing Unit (CPU)
The memory that stores the operations of the algorithm, the information, and the data on which the operations are executed.
Input- and output-devices, which transports the algorithm and the data to the main memory and over them the computer displays the results of its work.

1.4.2 What Makes Computer Powerful?

There are three main characteristics of a computer:

1.4.2.1 Speed

The CPU of a typical computer can do more than 2 billion operations per second (called MIPS). Thus, very complex tasks can by computed by undertaking thousands of simple operations. However, we should be aware, that there are problems that are not adequately solved by a computer perfectly, e.g. unstructured work like management decisions.

1.4.2.2 Reliability

Contrary to popular belief, computers errors are uncommon. Statements about high telephone bills are mostly attributed to wrong input values or errors in algorithms. That is both the power and the weakness of computers. It executes a default algorithm, independently whether this algorithm is correct or not.

1.4.2.3 Memory

One of the main characteristics of a computer is its power to store huge amounts of information that can be accessed very quickly. Memory capacity and fast access depend heavily on the storage medium.

1.4.2.4 Programmes and Programming languages

It was mentioned before, that a processing unit needs an algorithm to perform a process. The algorithm must be expressed in a way the computer understands and is able to execute it. The processing unit must be able to interpret the algorithm That means it has

to know, what each step means
to be able to execute the operation
If the processing unit is a computer, the algorithm has to be expressed in the form of a programme. A programme is written down in a programming language. The work that has to be done to write an algorithm as a programme is called programming. Each step in an algorithm is expressed in an instruction or command in the programme. A programme therefore consists of a series of instructions, each defining an operation that has to be executed by the computer.

The type of instruction in a programme depends on the programming language used. There are plenty of programming languages. Each has its own instruction set. The elementary languages are so called machine languages. They are built in such a way, that each instruction can be understood by the computer directly. That means, the CPU is capable of understanding each instruction and is able to execute it immediately. Because these instructions are so simple (e.g. adding two numbers), each can express only a small part of the algorithm. Therefore, a large number of instructions is needed to express most algorithms. Programming in machine language is tedious and time consuming.

It has become easier because of the development of so-called higher programming languages, e.g. Cobol, Fortran, C or BASIC. These are much easier to learn and more user-friendly than machine languages. A single instruction includes a bigger part of the algorithm. Of course, here the computer has to interpret each single instruction too. The connexion between higher programming languages and machine languages is still very close. Almost all programmes written in higher programming languages must be translated to machine code before they can be executed. The translation expresses each instruction of the higher programming language in a series of equivalent instructions in machine code.

This process is shown in the next example:
Algorithm

Programming
Leads to
Programme in a higher programming language

Translation
Leads to
Programme in machine language
Leads to
Interpretation by the CPU

(The preferred process is executed)

1.5 Design of Algorithms

Aim of this subchapter:
The student will learn how to avoid mistakes in the design of algorithms. He also will see, how he develops his programme further by reducing the problem to single steps, which have do be carried out to solve the problem. Moreover, he will learn about some tools, which help to solve problems.

Early in this chapter, we will discuss the three mistakes we can make in programming, in syntax, semantic, and logic. Then we will talk about the "Top-Down"-principle to solve a problem. Finally, we will learn how to use tools to solve problems: sequence, selection, and iteration.

1.5.1 Syntax, Semantics, Logic: three kinds of error

After clarifying the concept of programming in the former chapter, we now take a closer look at the design of algorithms. As mentioned before the computer must be able to interpret an algorithm so it can execute the described process. That means, the processing unit must be able

to understand the representation of the algorithm
to execute the specific operations

Let us focus on point a):

To understand the algorithm requires two steps. First, the Processing unit must be able to identify the symbols in which the algorithm is presented and to assign a meaning. To fulfil these tasks the processing unit must have knowledge of the vocabulary and the grammar of the language used expressing the algorithm.

The quantum of grammatical rules assigning how the symbols of the language are correctly used is called the syntax of the language. A programme that is written in the correct syntax is called syntactical correct, which means, each symbol used in the programme is used in the right way. An error in the syntax is called a syntax error. Syntactical correctness is normally required to interpret a computer programme. Exceptions to this rule are only allowed if the processing unit is smart enough, after recognizing a syntax error, to guess the right syntax intended. In practice, this is very rare.

The second step in understanding an algorithm is to assign a meaning to each step in the algorithm. This is done in form of operations that the processing unit has to perform. E.g. the equation revenue = price * quantity means, that two numbers, called price and quantity must be multiplied and a third number, called revenue, is the result. The meaning of special forms of expression in a language is called semantics. Programming languages are very simple designed regarding their syntax and semantics. So a programme can be easily analysed syntactically regardless its semantics. In other words in almost each programming language it is very easy to detect syntax errors, but it is very difficult to detect semantic errors.

To detect semantic differences you have to know about the specified objects. Particularly you have to know about the properties of these objects and the connexion between them. That means, a processing unit can only detect semantic differences if it knows enough about the objects that the algorithm deals with. A simple example will explain that more exactly. Assuming a processing unit detects the instruction
“Write the name of the thirteen’s month in the year”
If the processing unit knows, that there are only 12 months in a year it will detect this semantic inconsistency before it tries to executes this instruction. If another processing unit knows nothing about months and years, it will accept it as a legal instruction and try to execute it, e.g. looking for the month in a calendar, unsuccessfully. Not until the moment the instruction is executed does the processing unit notice that there is no thirteenth month and only then does the discrepancy becomes obvious.

Additionally to the syntactical and semantic errors there is a third kind of errors worth mentioning. These are logical errors. A programme may be syntactically clean and with no semantic inconsistencies present, but it may not describe the desired procedure in a correct way. Look at the following algorithm to calculate the circumference:
“Compute the circumference by multiplying the radius with PI”
This algorithm is correct in a syntactical and semantic manner. But it leads to a wrong result because the factor 2 is missing, indicating a logical error. At this point, we should state clearly, that a computer is unable to detect a logical error. To find a logical error a human has to spend time in the observation of the sequence of an algorithm and the use of testing data.

1.5.2 Top-Down Strategy in Solving Problems

The design of an algorithm normally is astonishingly difficult, especially when the executing process is complicated. The difficulties in designing an algorithm accelerate if the processing unit is a computer. The computer lacks in intuition and flair and it is not able to detect if the algorithm describes the aspired procedure. A general error is, that a procedure is almost but not fully reached.

Therefore, when designing an algorithm you must ensure that the algorithm exactly describes the process and all circumstances are noted. If the process is very complex, the design is very difficult. In fact, the success of a developer depends on her or his rigorous methodical approach in designing algorithms. This approach is called Top-Down-Design.
The Top-Down-Design is the latest version of the old "divide et conquera" (lat. "divide and rule"). The principle is, to sample the executing process in many small steps, each described by an algorithm less extensive and simpler than the original whole process. Because each of these partial algorithms is simpler than the whole algorithm, the developer normally has a better understanding of how to build it. Therefore, he is able to design in more detail than if she tries to handle the whole algorithm.

The partial algorithms themselves can be divided into smaller pieces, also because the simpler they are the more detailed and more precisely they can be expressed. In this manner the improvement in the design of the algorithm may proceed until the steps are detailed and precise enough for execution by the processing unit.

A simple example will demonstrate the operation. Suppose you have a little robot working as a butler. This robot owns an algorithm to make a cup of cocoa. The first version of this algorithm may look like:

```
Boil milk.
Put cocoa in the cup.
Fill the cup with milk.
```

Probably these steps are not detailed enough, so the robot will not understand them. Therefore we must improve each step in a series of simpler steps, specifying each step more detailed than the original.
For instance the step
Boil milk
May be improved to:

```
Fill the pot with milk.
Turn on the cooker
Wait, till the milk is boiling.
Turn off the cooker.
```

In the same manner
Put cocoa in the cup
May be improved to

```
Open the cocoa box
Take a spoonful cocoa
```

Tip the cocoa in the cup
Close the cocoa box
And the third step:
Fill the cup with milk
May be improved to
Pour milk from the pot into the cup until the cup is full.
Hereby the first improvement of our original algorithm is complete. If our robot is able to interpret all of these steps in the right manner, the improvement may be attained and the design is completed. But it is also possible, that the improvement can be taken one step further. The step
Fill pot with milk.

May be improved to:

Fetch the milk from the refrigerator
Tip the milk in the pot
Put the rest of the milk back into the refrigerator

After some further improvements, each step can be interpreted by the robot. At that point the design of the algorithm is finished.

1.5.3 Tools for Solving Problems

In addition, for the development of algorithms, there are tools available. In this case, we use Sequence, Selection and Iteration.

1.5.4 Sequence

In the previous section the algorithm is obvious. It includes simple steps that have to be executed one after another. Such an algorithm is a sequence of steps, meaning
A step is executed at a certain point of time
Each step is executed only once, none is repeated, none is omitted
The order is of importance. Each step is done one after another.
With the execution of the last step the algorithm is completed.
An algorithm, consisting only of a sequence of steps, is very inflexible, because the process of its execution is very rigid and may not be altered by any circumstances. You have to take into account, that the combination of steps in forming a sequence is a very primitive algorithm. In the next sections we will have a look at more difficult patterns.

1.5.5 Selection

We use the selection to change the sequence of execution under certain circumstances. Remember the robot. Our robot would not be able to prepare a cup of cocoa if the cocoa box was empty. In this case we should

get an instruction like that:
“Take a new, full cocoa box from the shelf”
The distinction between the two cases is improved by a new formulation:

```
Take the cocoa box from the shelf.
IF the cocoa box is empty
THEN take a new box from the shelf.
Remove the top of the cocoa box.
```

The general formulation of this instruction is:

```
IF condition
    THEN step
```

Whereby condition indicates the circumstances, when the step has to be executed. If the condition is fulfilled then the step is executed, otherwise not.
In the example above, the selection is used to specify whether a certain step has to be executed or not. An enhancement is possible by offering two alternative steps. The general Formulation of such a choice is:

```
IF condition
    THEN step 1
    ELSE step 2
```

whereby condition obviously specifies, whether step 1 or step 2 has to be executed. Take into account, that it is possible to put another IF condition at the place of step 2. This is called a multiple selection.
The power of the selection is that it allows a process to go different ways through an algorithm, depending on the circumstances. The interleaved selection allows multiple ways. Without the selection, it would not be possible to write algorithm for practical use.

1.5.6 Iteration

With the iteration, we try to find possibilities to repeat certain parts of the algorithm. For instance, we search in an endless list of names for a certain name. You may be able to find an algorithm that works exclusively with interleaved selections but this solution would be in no circumstances satisfactory

Here the iteration comes in, which is generally formulated as

```
REPEAT
    Algorithm
UNTIL condition
```

This means, the part of the algorithm between repeat and UNTIL must be repeated unless the condition, placed after UNTIL is satisfied.
The occurrence of a repetition is called loop and the part, that has to be

repeated, is called the loop body. The condition after until is called termination condition. Another name for this kind of loop is post-tested loop.

The power of the iteration is the possibility, that a process of undefined length can be described by an algorithm of finite length. But one of the gravest mistakes of many developers is to omit to define an accurate termination condition. This case is called a closed loop, which means, that this loop will never end and the algorithm will never stop working.
The repeat … until … iteration is not the only possible iteration. This loop has its truncation condition at the end, which means, the loop body is executed at least once. In some cases, it is not necessary to execute the loop body, because the condition is not fulfilled at the beginning of the loop. For this case exists the WHILE … repeat … loop.
The general form is:

```
WHILE condition REPEAT
        loop body
```

This means, the loop body has to be executed as long as the condition is fulfilled. But in this case the loop body must not be executed. Therefore, this kind of loop is called pre-tested loop.
The last possible iteration is a very simple repetition. In this case the number of repetitions is known before the loop is executed.
The general form is:

```
REPEAT N-TIMES
        loop body
```

Whereby n is any integer. This kind of loop is called counting loop. How long this loop lasts is known from the beginning and therefore the end of the loop is clear. This kind of loop is a safe way to developing algorithms, but is not as powerful as both the other loops.

1.5.7 Further exercises:

Here, some short algorithms for the different kind of loops are presented. Try to understand these examples.

1. The first example for the selection is an algorithm used when you approach a traffic-light

```
IF the signal is red or yellow/
        then stop
        else go on
```

2. If the traffic light fails, we could improve the algorithm.

```
If no signal
        then drive with extreme caution
        else if the signal is red or yellow
                then stop
                else go on
```

(This is an example for multiple selections)

3. You search for the largest of three numbers. In this case you are only able to compare two numbers at the same time. If these numbers are named x, y, and z then the algorithm is:

```
If x > y
        then if x > z
                then choose x
                else choose z
else
if y > z
                then choose y
                else choose z
```

4. You search in a list of names for a certain name and want to get the address. Would this algorithm be a solution?

```
Read the first name of the list.
Repeat
        if the name is the searched
                then note the associated address
                else read the next name in the list
until the name is found or the list has ended.
```

What would happen, if the appendix “or the list has ended” in the until part is missing?

5. You want to get the product of the first n numbers. In mathematics this is called factorial of n or n!. The algorithm may be:

```
Take the value of n
put product to 1
repeat for each integer from 1 to n
        multiply product with integer
```

write down the product

For which kind of loop is this an example?

More examples for the development of algorithms
Develop an algorithm that finds all primes. How could we calculate, if a number is a prime? What could an algorithm look like, that finds all primes nearest to any number?
An old example in mathematics is to identify the greatest common divisor of two integers. If you don’t use a post-test loop then argue why.

1.6 Translation Programmes

Aim of this subchapter:
The student should learn which tools a computer has available to translate the written code (words) of a programming language to machine code.

In this Chapter, we will learn about the compiler, linker, assembler, and interpreter and how these tools are used.

As mentioned in former chapters most programmes, which are not written in a machine language, but in a higher programming language like C or Pascal must be translated into machine code. For this, tools like compiler, linker, assembler and interpreter exist. These tools will be identified now. At the beginning, we will repeat the definition of a translation programme.
A translation programme is a tool, which reads instructions from a higher programming language or assembler language, analyses it and transforms it into machine commands of the same significance. Beside the actual translation (= conversion of terms into other terms) the source programme is tested syntactically and if necessary, error warnings are displayed.

The Assembler is a translation programme, which converts source instructions written in an assembler language to target instructions in the appropriate machine language. Here we use the term Assembler language for the first time and we shall explain it immediately. An Assembler language is a tool to represent machine codes more memorable and understandable. Thereby the instruction are displayed with mnemonic abbreviations, which makes the handling much easier.
A command to add two numbers may look like:

```
ADD 3 , 4
```

A compiler is a translation programme that converts source code written in a higher programming language to target instructions in a machine-oriented language. Generally, the target language of the compiling is the machine language (object code) itself, but sometimes the translation is done to a assembler language specific to the computer. During the translation, normally the source instruction is split up in many instructions in machine code (so called 1:n translation).

Assembler and compiler translate the entire source programmes in object programmes before they are executed. During this translation, it is not possible to interfere.

The object programmes built by assembler and compiler are not executable, because some parts of the programme, needed by the source programme, like input/output procedures, have to be added. This job is done by the linker, which subsequently justifies this satisfied external addresses in relation to the beginning of the programme. Now the programme is loadable and executable.

An interpreter is a programme that translates and executes the existing source code immediately. This means, no object code is produced. The disadvantage is, that this approach produces a slowly execute programme. A procedure such as linking, mentioned before, is not used. The programme runs dynamically without interaction by the user. The advantage of the interpreter lies in interactive programming.

Examples of Interpreter are programming languages like BASIC, APL, VBA, Perl. These mostly work interpretatively. Other higher programming languages suchas Pascal, Cobol, Fortran, C, Java or Visual Basic use various compilers and linkers.

1.7 Appendix: Control Structures in Practice

Aim of this subchapter:
The student should learn about the implementation of the different control-structures in different languages.

In this chapter the will discuss the operation of the post-tested loop, the pre-tested loop, the counting loop, and selection in Basic, C/C++, and Pascal
The following shows the individual control structures in the common programming languages, C/C++, Pascal and BASIC.

1.7.1 The Post-Tested loop

BASIC:
```
DO
        Loop body
LOOP UNTIL condition
```

C/C++:
```
do
        {
                Loop body
        }
while (condition)
```

Pascal:
```
repeat
        Loop body
until condition
```

In BASIC and Pascal you find the word Until with the termination condition. In these two cases one can understand the post-test loop in the sense that the loop is valid when the termination condition is reached. Unlike C/C++, where you discover that the termination condition is While. Here the loop is repeated as long as the condition is valid i.e. true.

1.7.2 The Pre-Tested Loop:

BASIC:

```
DO WHILE condition
        Loop body
LOOP
```

C/C++:

```
while (condition)
        {
                Loop body
        }
```

Pascal:

```
while condition
        begin
                Loop body
        end
```

1.7.3 The Counting Loop

BASIC:

```
FOR counter=1 TO 20
        Loop body
NEXT counter
```

C/C++:

```
FOR (counter =0; counter <20; counter ++)
        {
                Loop body
        }
```

Pascal:

```
FOR counter:=1 to 20 do
        begin
                Loop body
        end
```

In this section indentation of blocks of code was used to improve legibility. In the programming languages C and Pascal , such blocks are delineated with special symbols,{ } (curly brackets) in C, and the statements begin and end in Pascal.

1.7.4 The Selection

BASIC:
```
IF condition
        THEN selection 1
        ELSE selection 2
END IF
```

C/C++:
```
if (condition) selection 1;
        else selection 2
```

Pascal:
```
if condition
        then selection 1
        else selection 2
```

Chapter 2: Basics about C++

Aim of this chapter:
The Student should get an overview to the history of C, C++ and Borland C++ Builder, also in delimitation to other programming tools for C++.

The student should also get a short overview about C/C++ and how it is related to other programming languages.

2.1 History of C++, Borland C++ and definition of this Programming Language

In this section we will talk about the development of C in the 1970's and it's further development C++ in the 1980's. Then we will show, how Windows led to graphical development environments.

C was developed in 1971/72 in the A&T Bell Laboratories for the Operating System UNIX. It's inventors were Denis Ritchie and Brian Kernighan. The predecessor of C was a programming language called B. In 1978 C got it's first exact definition and in 1983 the American National Standard Institute (ANSI) standardised C to ANSI C.

At that time C was still a procedural programming language: That means, programming-code once programmed could only be used for one project and could not be developed further or used (easily) in another project.
Therefore C was developed further to C++, an object oriented programming language (OOP), much more powerful than C. Object Oriented code is able to be used more than once, because each object oriented project is self-contained (called "Encapsulated") and a new OO-project may be derived from an existing OO-project (called "Overloading" and "Inheritance"). So by using OOP you do not need to reinvent the wheel for each new programming project. The basic of C++ is still C, so often both are mentioned together as C/C++.

When Microsoft Windows came into use the developers had to programme windows, menus, mouse-cursors, buttons, input-boxes, drop-down-fields etc. That was very tiresome and boring. So some manufacturer s(like Microsoft or Borland) brought tools onto the market, that had included all these components and only these had to be used by the programmer. These tools are called Integrated Development Environments (IDE). The most famous are Microsoft's Visual Basic and Visual C++ and Borland's

Delphi and C++ Builder.

Borland was already famous its fast C/C++ Compilers (Compilers are tools, that create an application from an programming code). So Borland C++ Builder is a programming tool based on an IDE, that uses C and C++ to programme applications.

2.2 C++ and its categorization and advantages

C++ is, as mentioned before, an object oriented programming (OOP) language. It's not the only one. There are others like SmallTalk, Java, Delphi. Some think, it's not the best OOP language, but it is widely used so Microsoft Windows itself is programmed mostly in C++, and the so called WinAPI-functions, functions made available by Microsoft for programmers, to directly interact with Windows, are written in C++.

Many other programming languages are derived from C/C++ e.g. Java, Perl, PHP. So if you know programming in C/C++ you can easily switch to this other programming languages. C++ uses the same syntax (the kind you write the programming code) as C. But more than that you can write C-programmes with C++, because C is C++ without object orientation. Therefore it is called C/C++. The object oriented concept is introduced and described with examples later on in this manual.

C++ is easily extended with new functions. One of the major concepts of C/C++ is to use functions for a programme that are stored in files called "libraries". So if you want to make available special functions you programmed, you may publish them with a library and every other programmer is able to use them. Many other programming languages followed this concept, but it is still one of the concepts that makes C/C++ so powerful.

Another advantage of C/C++ is that you can programme at a low level, very close to the hardware you use. Most so called "Function-Calls" of hardware devices are C Calls. So if you want to develop programmes that use special features of devices you use C/C++.

The Borland-Builder comes in three versions, different in equipment and costs:

The Borland C++ Builder Standard: Smallest equipment with no Database-Connectivity-tools

The Borland C++ Builder Professional: More equipment and with Database-Connectivity-tools

The Borland C++ Builder Enterprise: Most equipment for large programming groups and for developing network applications.

Chapter 3: Introduction in the IDE of BCB 5

Aim of this chapter:
The student should get a introduction to the Integrated Development Environment (IDE) of Borland C++ Builder 5.

3.1 IDE of BCB5

The IDE (Integrated Development Environment) is powerful editor that helps you to create Windows and DOS programmes. The later is less important.
The IDE helps you
to build windows-like surfaces of programmes with window-behaviour, menus, buttons, dropdown-lists, mouse-cursors etc. by not stressing you with details.
to write easily function for your buttons, menus, dropdown-lists, etc.
to manage easily large applications with many different parts
to find easily errors (called “bugs”) in your programming code
to write easily powerful applications with database connectivity
to write easily internet-applications
to do many other things that has a programmer to do very easily.

Why do programmers, especially blind programmers need instruction in BCB? The IDE of the Borland Builder is not centred in one application window, but has many windows floating on the desktop and behind them the MS Windows-desktop peeps out (or any other programme you kept open in the background). So if you read the screen line by line with your screen-reader-software you will get information, that belongs to the Builder and information, that belongs to other programmes.

If you start the Builder the first time you get four of these floating windows:
The “Builder C++ - Project1”-window on the top of the screen holding the main menu and the toolbars.
The “Object Inspector”-window on the left side of the screen.
The “Form1” window, that belongs to your first programme, beyond the first and right of the second window.
The “Unit1.cpp”-window not visible but behind the “Form1”-window, that holds your programming code.
You may close each of these windows except the first “Builder C++”-window, because that will close the Borland Builder C++ 5.

You activate the “Builder C++”-window with the ALT-Key. You activate the “Object Inspector” with the function key F11. And you toggle between the “Forms1”- and the “Unit1.cpp” with the function key F12. If you have problems with the toggling, press ESC first.

As you may have guessed, are the windows-names “Forms1” and “Unit1.cpp” only temporary names for our first programming projects with the temporary name “Project1”. If you save your job the first time those windows will get the names you gave them.

You can also access all Builder-windows with the menu “View”, ALT+V. There are many of them but I will not mention them all yet, because you would not see any use of them until you do your first programming job. If you opened one of them by accident close them with ALT+F4. This command will not close the Borland Builder but close the active window.

You can end your Borland Builder session by activating the menu “File” and “Exit, or short ALT+F, X.

The “Builder C++” main-window will be described in the next chapter. But some of the windows I have to describe for you now.

3.1.1 The “Form”-Window

The “Form1”-window is the surface of your first application. It is on this window that you will place your buttons, write your text, place your dropdown-lists etc. These are called “components”.
If you activated the “Form”-window with function key F12 you will toggle between all the components you placed on your form.
If you activated one of the components in your form (or the form itself) and press F11 you will activate the “Object Inspector”-window.

3.1.1.1 The “Unit”-Window

The “Unit”-window holds all the programming code, some generated automatically by the Builder (very good!) and that you wrote by yourself (even better!).
The unit-window is the “Editor” of the Borland Builder C++ 5.
Because the Builder generates code automatically the window isn’t empty if even when you opened it the first time.
The unit-window holds all the unit-files you saved your programming code in and some additional text-files too. You can activate those files by pressing CTRL-TAB. If you cannot reach a unit-file by pressing CTRL-TAB you must open it first.
You may close a unit- or additional-file by pressing CTRL + F4.

3.1.2 The “Object Inspector”

The “Object Inspector”-windows holds information about the component of your form you activated previously.
This information is
a list of the currently used objects in this form
a list of properties of the selected object
and on another tab the list of events of the selected object
In detail:

the name of the current component, followed by a colon and the kind of component, the current component belongs to, e.g. “Form1:TForm1", that means an component named “Form1” of the kind of “TForm1” (if you are an experienced programmer you would say “an object Form1 of the class TForm1").This information is at the top of the object-inspector beyond the window-title. You can reached this area by pressing CTRL + CURSOR DOWN. This will open a list with all the components of your form and you can easily make one of them current.
A list with “Properties” of the current component like the name, the width, the height and many others. This information is presented in the form of a table with two columns. The left column holds the name of the property, the right column the value of the property, e.g. left “Name”, right “Form1”. Each row holds the information about one property. You can change from property to property by pressing CURSOR UP or CURSOR DOWN. You can toggle between the left and right column by pressing the TAB-key. In the left column the property-names are sorted alphabetically. You can also reach a special property if you know its name. If you are typing the name of the property in the left column you will come closer to your desired one the more letters you type. For instance if you press “n” you will go to the row with the first property starting with an “n”, in this case the property “Name”. Press the TAB-key and you will switch to the right column holding the actual property of “Name”, that is “Form1”, the name of the current form. Type a new name so you will rename your form (it is supposed you name your form “frm_MyFirstForm”, where the leading “frm” stresses, that the named component is of the kind “Form” and the part after the underscore is any name of your choice). With the “Properties” you can determine many properties of the current component. E.g. you can position it on your form without using the mouse with the properties “Top” and “Left”, where “Top” is the distance between the upper border of the form and the upper border of the current component and “Left” is the distance between the left border of the form and the left border of the current component. If the current component is the form itself the distances are between the screen and the form. Later you will see that you can change the properties of a component by your programming code.
If your cursor is in the row of a property and you press F1 you will get some

help for this property.
Some properties must have certain predefined values. Than the right value-column is formed as a dropdown-list, that you can open by pressing ALT + CURSOR DOWN. For instance the property “Enabled” may be “true” or “false”. These values are presented in a dropdown-list.
Some properties may be even more complicated so you get a dialog-window. The existence of a dialog-window for a certain property is shown by a button with three dots, positioned at the right border of the value-column. You can open the dialog-window by pressing CTRL + ENTER. For instance the “Font”-property has a font-dialog-window.
And last but not least some properties are a collection of other properties, e.g. the property “Font” once more. There are two information for the existence of a property-collection: first on the left side of the property-name is a plus-sign (+) and second the value of the property is in brackets. Press the plus-key (+) in the left column with the property-name to open the collection-branch and press CURSOR-DOWN down to visit them. Press the minus-key (-) to close the branch (your cursor must be on the name of the property-collection to close the branch).

Beside the “Properties” there is another Tab “Events” and you can toggle between “Properties” and “Events” by pressing CTRL + TAB. ”Events” are triggered by a user, e.g. by a click, a double-click, by pressing a key and many more or by an Windows-action like opening or closing a form. The events are named appropriate like “OnClick”, “OnDblClick”, “OnKeyPress”, “OnShow”, “OnClose”. You can reach an event just as a property. But at the beginning the values of all events are empty. The value of an event is the name of a function, a certain part of programming code which tells e.g. a button what to do if it reaches the event “OnClick”. If you position your cursor in the value-column of a certain event end press CTRL + ENTER you will switch to the “Unit1.cpp”-window (if you didn’t save and name it till now). If the value was empty before it is now filled with a function name and a function of this name is created automatically in the unit-window (and all the stuff you need to define a function). Now you only have to write your programming code. Easy, isn’t it. If the event was assigned a function name already, the unit-window is also activated and you jump to the function.

3.1.3 The “Window-List”

If you have many windows opened (sometimes this will happen) you can open another window, the “Window-List” in the menu “View” or by pressing ALT + 0. In this list you get all opened windows. Use the curser-keys and the ENTER-key to activate one.

3.1.4 The “Component-List”

If you want to place one or more components on the form open the component list in the menu “View”, short ALT + V, C.
This list holds all available kinds of components.
You can use the CUSROR DOWN and CURSOR UP keys to scroll between the components. But beware: depending on the Builder-version you use you may have more then 100 components in this list. So it is easier if you know or even guess the name of a component. Most of he components start with the letter “T”. So type T plus the name of a component and you will activate it, e.g. “TBu” to reach “TButton”. Then press ENTER to insert it in the form. Each time you press ENTER you will insert another component of the selected kind in the form (they are named by default “Button1”, “Button2”, “Button3”, and so on if you insert TButtons).
When you have finished, press ESC, that will close the “Component-List”

3.1.5 The “Messages”-Window

If the “Messages”-window appears something went wrong with your programming code, that means it contains an error. The message-window is shown automatically when you run your application by using the menu “Run” and “Run” or pressing F9. If the Builder finds an error in your programming code it stops starting the application and shows the line with the error and the message-window, where it describes the error so you can easily correct it.]

But by default the message-window is hard to reach for the non sighted users, or generally for those who do not use a pc-mouse. The message-window is docked by default at the bottom of the unit-window. But you can click in the message window the first time, open the context-menu by pressing SHIFT + F10 and deactivate the menu-entry “Dockable”. Then you can reach the message window by using the window-list (ALT + 0) or from the unit-window over its context menu and menu-entry “Message-Window”, short SHIFT + F10, M.
The message list will stay undocked for the current programming-session.
If you are in the message-list you see all error-messages. Go to the message, read it and press CTRL + ENTER or CTRL + S and you will jump to the error in the unit-window. After you corrected the error in your programming code go back to the message-window and do the same with the next error.
After correcting all errors run your application again by pressing F9.

3.1.6 Other windows

The other windows will be described at the point you use them the first time.

3.2 Menus, component- and Toolbars of BCB5 IDE

Aim of this subchapter:
The student should get an overview to the menus an toolbars of the Builder. They will also learn about the special toolbar, the component-bar.

3.2.1 Menus

The menus are described here only in general, because it’s no use knowing exactly each command in each menu if you cannot associate it with a function in your programming.
There are nine menus in the main menu.
The menu “File”, shortcut ALT + F:
As in other programmes here are the commands for New, Open, and Save. But because the Builder knows more than one kind of file there are more than one New, one Open and one Save.
The menu “Edit”, shortcut ALT + E:
The menu Edit holds the commands for Cut, Copy, Paste, Delete, and some other similar commands but also commands to position and size components in the form in a proper way.
The menu “Search”, shortcut ALT + S:
Here are the commands for searching and replacing text and those for go to.
The menu “View”, shortcut ALT + V:
Here you can find all the windows of the Builder.
The menu “Project”, shortcut ALT + P:
This menu contains many commands to manage your current project.
The menu “Run”, shortcut ALT + R:
Here are the commands to run and test your application and commands for “debugging”(= finding errors).
The menu “Component”, shortcut ALT + C:
You need this menu if you want to use components that are not with the Builder by default but another programmer has developed to make life easier for fellow programmers.
The menu “Tools”, shortcut ALT + T:
Here you can adjust some defaults for your editor, your IDE in general.
The menu “Help”, shortcut ALT + H:
Help is self explanatory

3.2.2 The Toolbars

There are five toolbars arranged by default on the right side of the main menu and in two rows beyond the main menu. All toolbars can be removed

and displayed by the context menu, shortcut SHIFT + F10.
The “Standard”-toolbar is the first menu beyond the main menu and holds the buttons
New
Open (with a list of most recently open files)
Save
Save All
Open Project
Add File to Project
Remove form Project
The “View”-toolbar is beyond the standard-toolbar and holds the buttons
View Form
View Unit
Toggle Form/Unit
New Form
The “Debug”-toolbar is left of the view-toolbar and holds the buttons
Run (with a list of most recently run programmes)
Pause
Trace Into
Step Over
The “Custom”-toolbar is left of the standard-toolbar and holds only one button, the Help Contents.
The “Desktop”-toolbar is left of the main-menu and holds
A list of “Desktops”, the default value is “None”
The button “Save Desktop”
The button “Set Debug Desktop”

3.2.3 The Component-Bar

The Component-Bar or Component Palette is the sixth toolbar. It contains all the components you can use in a project or on a form. Because there are so many components, this palette is arranged in form of registers or tabs. The first tab is titled “Standard”.
Each tab holds special components.
This bar is positioned beyond the main menu, but right of all other toolbars and its height is the height of two standard toolbars.
This toolbar can not be reached with a shortcut, you have to click.
If you activated the component-bar you can use the context-menu, SHIFT + F10 to activate the right tab.
But if you activated the right tab you have to use the mouse-cursor and click again to select a component.
It is advisable to use the component window to add components to a form and not the component-bar.

Chapter 4: Creating A Simple Windows Application

Aim of this chapter:
In this chapter you'll learn how to develop a simple C++ programme with Borland Builder, to get an overview about the important and necessary components of Borland Builder

4.1 Creating a simple Windows application

4.4.8 RadioButton object. ***Exercise***

4.5 Designing Blind Accessible Forms

4.5.1 Sticking to Windows standards
4.5.2 Font type and size
4.5.3 Text description of each component
4.5.4 Correct positioning of the components on the form

4.6 Moving from the Form to the Application *(Exercise)*

4.6.1 An example of Application “Hello”
4.6.2 Adding the components to “Hello”
4.6.3 Setting the components properties
4.6.4 Defining the event handlers
4.6.5 The project
4.6.6 Compiling, running the program

4.7 The Menus

4.7.1 Adding the component TMainMenu
4.7.2 Designing the Menu Bar
4.7.3 Adding, deleting, editing menus
4.7.4 Implementation of the event OnClick

4.1 Creating a simple windows application

Aim of this subchapter: The student will learn how to create a simple windows application.

In this unit we will introduce an application, its Form, its Components with their properties and events.

General: A windows application is an executable program in the Microsoft operating system family: Windows 95/98, Windows ME, Windows 2000.

A windows application is generally made by a group of graphic components (windows, edit boxes, buttons, menus etc) which enable the user to perform a given task (i.e. for a simple text editing we will use the application Notepad).

Shortcuts.

IO	= window Object Inspector.
BCB	= Borland C++ 5 Builder.
Up	= up arrow.
Down	= down arrow.
Right	= right arrow.
Left	= left arrow.

Suggested Naming conventions for BCB Components

btn_	Button
chk_	Check box
cmb_	Combo box
edi_	Edit box
frm_	form
lbl_	Label
lst_	List box
mem_	Memo
rad_	Radio button
rdg_	Radio group

4.1.1 Designing Windows Forms.

Aim of this subchapter: The student will learn how to create a Form

In this unit we will show how to design the graphical structure of a Form

When creating a Windows application the first step is to design its main Form. Forms are the equivalent of the Operating System Windows

A windows Application is made of at least one Form. When BCB5 is opened the default active element is an empty Form. An empty form can be compared to the ground floor of a building under construction. The student will first set the objective of the application he wants to create; then he will design the graphical structure of the form, coherently to the selected objective. The tasks to be performed are:

a)Set the Form attributes (dimension, colour, etc)
b)Arrange all the necessary components on the Form
c)Set the components attributes in a harmonic and functional way.

4.1.2 Object Inspector : properties and events

Aim of this subchapter: The student will learn the concept of Property and Event and how to use Object Inspector

In this unit we will show how to use Object Inspector to modify properties and events

The Form, just like the components, has properties and events. For the definition of the properties and events of an object the student must interact with a module called Object Inspector.

4.1.3 Properties and Events of an Object

Properties are the object's attributes: for instance name, width, height, colour etc
Events are the user's actions on a certain object, for example Key pressing, mouse clicking etc
A Windows application is driven by the events triggered by the user's actions. Writing Event Handlers is the main task of a Windows

programmer.
An Event Handler will be activated when the event occurs. (for instance the event handler **OnClick** will be activated when the event **Click** occurs).

4.1.4 The Object Inspector (OI)

General: Object Inspector is the IDE window which allows to set the properties and to reach the Event Handler of a Form and its components.

The Object Inspector Window is constituted by 3 parts:
a combo-box which enables the choice of the component to be modified; the combo-box is placed immediately under the **Object Inspector** title bar
The **Properties** list related to the selected component
The **Events** list related to the selected component.
The tab labels of the **Properties and Events** lists are the only reference useful for display braille navigation.
Both lists are ranged in two columns; on the left you can find the properties in alphabetical order or the events related to the selected component; on the right the student will select or digit the property or the event name.

4.1.5 The names of properties and events

The names of properties and events on the left column are common words in the programming language; no spaces or other separations are foreseen, the first letter of each word is a capital letter, for instance **TabOrder** or **OnKeyPress.** The names of events are marked by an "**On"** prefix.

4.1.6 Reaching the Object Inspector

The following commands are used to move inside the OI:
(F11) to reach the Object Inspector
(Ctrl+ Down): **(Scroll up on Braille Display)** to reach the components combo-box
(Up or Down): to scroll the list of the components on the form, including the form
(Enter) to select a component
(Ctrl+Tab): to switch from components list to events list (and viceversa)
(Up or Down): To scroll up properties and events; some properties have sub-properties lists which can be reached with the **Tab** key to deselect the text of the property on the right column; (Left) to close the list, (Right) to

open it.
(Alt+Down): to open the list of the properties parameters
(Enter) to confirm or (Ctrl+Enter) to move to the parameter you wish to select.

4.1.7 Avoiding errors while navigating in OI

Usually when properties or events are selected the content of the right column (containing the property value or the event name) is displayed; at this point any key you press on the keyboard will modify the displayed information.

To avoid such inconvenience the student will follow this procedure:

A) Press **Tab** to avoid the selection of the values on the right column
B) scroll the list of properties and events to reach the property or event you are looking for.
C) Press Tab again to view the right column
D) Modify the value

4.1.8 Updating of the graphic characteristics of an object

The values of the properties, especially if graphical, are updated while being modified or by pressing enter and reaching another property. For example, the name displayed on the title bar of the form will be updated while you are typing it; once the value of the width property of a form is modified, width will reset after you press Enter or when you move to another property.

4.1.9 Defining an Event Handler

In the right column of the events list it will be possible to personalize the name of the selected event.
(Ctrl+Enter) to reach the text window and operate on the selected event. The default name of the event will appear on the right column, in the box next to the selected event; to the right, above the Form, a text editor window will be displayed; within this window, it will be possible to digit the code related to the handling of the selected event.
(F11) to switch from the Form to Object Inspector and/or the text editor window and viceversa.

4.2 Designing a Form: properties and events.

Aim of this subchapter: The student will learn how to design a Form

In this unit we will show how to add a new Form to the application under construction

A Windows Application can be formed by one or more forms ; in BCB5 the student will find an empty default Form

Open the menu **File** and select **New Form** to add a new Form to the Application under construction.

4.2.1 Main Properties of a Form

Name: sets the name of the form within the application. This name identifies the component within the program.
Caption: sets the name of the Form to be shown on the title bar
Width: sets the width of the form in pixel
Height: sets the height of the form in pixel
Colour: sets the background colour of the form
Obviously the Form includes many other properties; to obtain their descriptions, select a property and press F1

4.2.2 Main Events of a Form

OnShow: Defines what happens when opening an Application and viewing a Form
OnActivate: Defines what happens when a form is reactivated, for instance by pressing (Alt+Tab)
OnClick: Defines what happens when clicking the mouse on the form
OnKeyPress: Defines what happens when a key is pressed on the keyboard
OnClose: Defines what happens when the form is closed.

4.3 Adding Components to a Form

Aim of this subchapter: The student will learn how to add

components to a Form

In this unit we will present the main graphic components of a Form

In addition to Forms an Application is made by other graphic components which allow the student to interact with the Application.

4.3.1 The main components of the Form (adding properties and events)

TEdit: Single-line edit box used to add or modify the text
TMemo: Multi-line edit box used to add or modify the text
TButton: This component starts the execution of one or more actions
TLabel: Read-only text label
TListBox: Box for the selection of an item from a list of items
TCombBox: It is a combined control which relates the list of items displayed in a list box to the text of an edit box. It is possible to select an item of the list by pressing the initial letter. When the selected item of the listbox is confirmed pressing (Enter) it will go in the edit box.
TCheckBox: Check box can be used to select one or more options within a range of alternatives.
TRadioButton: Option component; these components are used to choose among several options. Each option automatically cancels the previous one.

After this long theoretical introduction we are now going to try a practical exercise.

Exercise: Design a Form to collect the personal data of the students in your school.

For this Form you will need:

An Edit component for the name
An Edit component for the surname
Two RadioButtons for the sex
A ComboBox for the age
A checkBox component for disability
An Edit component for the address
A listBox for the city of residence
A Memo component for notes
A Button to confirm the entered data

The four Edit Boxes (Three Edit and 1 Memo) will be preceded by a descriptive label

4.3.2 Adding a component to a Form

To see the complete list of the default basic components go to menu **View** and select **Component list**; there are no shortcuts for this list.
You will now view the window **Components**; this window is formed by an edit box and a list box (separated from each other but working together as a combo-box) and by the button **Add to form**; all these components can be reached using **Tab.**

On opening the window the last component selected before closing will appear; its name is viewed and highlighted on the edit box
For all the components starting with "T" the student will type "T" followed by the first letter of the component's name; on the display the first available component in alphabetic order will be selected and the complete name will be displayed and highlighted in the edit box; (the only component not starting with "T" is "Frame", in this case we will type "F").
Press Up or Down to scroll the component list
Press Enter to add the selected component to the form.
Press Esc to leave the window .

To add the components to your Form (collect the personal data of the students in your school) you will
Move to the menu bar (Alt)
Move to menu View (Right)
Move to the item Component List (Down), Enter
When the window Component List opens Move to TButton (Down), Enter

Warning: After pressing Enter you will still view the window Component; it is not important at this stage to verify that the components have been added correctly to the Form.

Move to **TcheckBox** (Down**),** Enter
Move to **TcomboBox** (Down)**,** Enter
Move to **Tedit** (Down**),** three times Enter
Move to Tlabel (Down) , four times Enter
Move to TlistBox (Down), Enter
Move to Tmemo (Down) , Enter
Move to TradioButton (Down) , Enter twice
Esc to leave the window Component List

You should now have 15 items in the list box of the components OI.

Move to OI (F11)
(Ctrl+Down) to open the combo box of the components
(Up) or (Down) to scroll the list

4.3.3 Positioning the components

When a component is added it automatically positions itself in the middle of the Form; if more components are added at the same time, they will overlap. It will be necessary to set the properties of each component in order to give them the right position.

4.3.4 The tab order

A Form can contain different components; to optimise navigation among the components Dialog Box it is necessary to set the **Tab Order** Property. This property sets the components tabulating order; for instance in a Window used for collecting addresses it would be illogical to press **Tab** and move from Name to City leaving out Address.

4.4 The main components of the Form (adding properties and events)

Aim of this chapter: The student will learn how to arrange components on the Form

In this unit we will learn about the components Properties and the most common Events of the Form

4.4.1 Edit Object

To add the component **Edit** to a Form, the student will select **TEdit** on the Window **Components,** press **Enter,** then **Esc** to close the window.
Then the student will place a single-line textbox named **Edit#** in the middle of the Form; the name of the component will appear in the box itself as a text. The component looks like a long narrow rectangle, the background and text colour are those used by default in Windows; the outline of the component has a hollow aspect.

Main Properties of Edit Object

Name Property: It represents the component's identification name within the application
Text Property: It is used to add the caption of the component
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the top of the Form
Left Property It sets the distance in pixel from the left of the Form
Color Property: It sets the background colour of the component
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.
TabOrder Property: It sets the tabulation order

Main Events of Edit Object

OnChange Event: it defines the behaviour in the application when the text of the edit box is modified.
OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnDblClick Event: : it defines the behaviour of the application when the mouse is double-clicked on the component
OnKeyDown Event: it defines the behaviour of the application when a key is pressed
OnKeyUp Event: it defines the behaviour of the application when a key is released
OnKeyPress Event: it defines the behaviour of the application when a character is pressed on the keyboard
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when the mouse is placed on the component
OnEnter Event: it defines the behaviour of the application when the component becomes active (receive the focus)
OnExit Event: it defines the behaviour of the application when the component stops being active

4.4.2 Memo object

To add the component **Memo** to the Form, select **TMemo** on the

Window **Components,** press **Enter,** then **Esc** to close the window.
A multiple-line textbox named **Memo#** will be displayed in the middle of the Form; the name of the component will be shown in the box as a text. The component looks like a rectangle, the background and text colour are those used by default in Windows; the component has a 3D appearace.

Main properties of Memo object

Name Property: it represents the component's identification name within the application
Text Property: It is used to add the text of the component
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the top of the Form
Left Property It sets the distance in pixel from the left of the Form
Color Property: It sets the background colour of the component
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.
TabOrder Property: It sets the tabulation order
ScrollBars: It sets the visualisation of the ScrollBar

Main Events of Memo Object:

OnChange Event: it defines the behaviour in the application when the content of the text is modified.
OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnDblClick Event: : it defines the behaviour of the application when the mouse is double-clicked on the component
OnKeyDown Event: it defines the behaviour of the application when a key is pressed
OnKeyUp Event: it defines the behaviour of the application when a key is released
OnKeyPress Event: it defines the behaviour of the application when a character is pressed on the keyboard
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when the mouse is placed on the component
OnEnter Event: it defines the behaviour of the application when the component becomes active (receive the focus)

OnExit Event: it defines the behaviour of the application when the component stops being active

4.4.3 Button object

To add the component **Button** to the Form, select **TButton** on the Window **Components,** press **Enter,** then **Esc** to close the window.
A button named **Button#** will be displayed in the middle of the Form; the name of the component will be viewed as a label. The component looks like a long, narrow rectangle, the background and text colour are those used by default in Windows; the outline of the component has a hollow aspect .

Main Properties of Button object

Caption Property: sets the text content of the component
Default Property : when this property is set to **True** the button is connected to the **Enter** key. By pressing Enter from each point of the Form it will be possible to enter the **Event handler** of the component. When this property is set to **True** the edge of the button will appear with a black border.
Cancel Property: When setting this property to **True** the button is connected to the **Esc** key. By Pressing Esc from each point of the Form it will be possible to enter the **Event Handler** of the component.
Name Property: it represents the component's identification name within the application
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the upper edge of the Form
Left Property: It sets the distance in pixel from the left edge of the Form
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.
TabOrder Property: It sets the tabulation order

Main Events of Button object

OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnKeyDown Event: it defines the behaviour of the application when a key is pressed
OnKeyUp Event: it defines the behaviour of the application when a key is released
OnKeyPress Event: it defines the behaviour of the application when a

character is pressed on the keyboard
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when mouse is placed on the component
OnEnter Event: it defines the behaviour of the application when the component becomes active (receive the focus)
OnExit Event: it defines the behaviour of the application when the component stops being active

4.4.4 Label object

To add the component **Label** into the Form, select **TLabel** on the Window **Components,** press **Enter,** then **Esc** to close the window.
A label named **Label#** will be displayed in the middle of the Form; the name of the component will appear on the label The component looks like a text label, the background is transparent and the text colour is the one used by default in Windows; the border of the component is visible only if selected while being designed.

Main Properties of Label object

Caption Property : sets the text content of the component;
Name Property: it represents the component's identification name within the application
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the upper edge of the Form
Left Property: It sets the distance in pixel from the left edge of the Form
Color Property: It sets the background colour of the component
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.

Main Events of Label object

OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key

of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when you move the mouse on the component

4.4.5 ListBox object

To add the component **ListBox** into the Form, select **TListBox** on the Window **Components,** press **Enter,** then **Esc** to close the window.
A listbox named **ListBox#** will position itself in the middle of the Form. The component looks like a rectangle, the background and the text colour are those used by default in Windows; the outline of the component has a 3D appearance

Main Properties of ListBox object

Items Property: It sets the items on the list. By pressing **Ctr+Enter** a **StringList Editor** will open on which it will be possible to type the items, one on each line, to be displayed (we suggest to limit to the minimum the number of characters per item of the list)
IntegralHeight Property: It set the height in pixel to obtain the desired number of lines.
Name Property: it represents the component's identification name within the application
Text Property: It is used to add the text of the component that will be viewed
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the upper edge of the Form
Left Property It sets the distance in pixel from the left edge of the Form
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.
TabOrder Property: It sets the tabulation order

Main Events of ListBox object

OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnDblClick Event: : it defines the behaviour of the application when the mouse is double-clicked on the component
OnKeyDown Event: it defines the behaviour of the application when a key is pressed
OnKeyUp Event: it defines the behaviour of the application when a key is

released
OnKeyPress Event: it defines the behaviour of the application when a character is pressed on the keyboard
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when mouse is placed on the component
OnEnter Event: it defines the behaviour of the application when the component becomes active
OnExit Event: it defines the behaviour of the application when the component stops being active.

4.4.6 ComboBox object

To add the component **ComboBox** into the Form, select **TComboBox** on the Window **Components,** press **Enter,** then **Esc** to close the window. A checklist/edit-box named ComboBox will be displayed in the middle of the Form; the name of the component will be viewed in the edit-line of the box. It is a composite component; it looks like a single-line text-box (only the edit-line is visible, not the check-list); on the right border there is a small tri-dimensional button containing a small arrow pointing downwards.

Main Properties of ComboBox object

Items Property: It sets the items on the list. By pressing **Ctr+Enter** a **StringList Editor** will open on which it will be possible to type the items, one on each line, to be displayed (we suggest limiting to the minimum the number of characters per item of the list)
Name Property: it represents the component's identification name within the application
Text Property: It is used to add the text of the component that will be displayed
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the upper edge of the Form
Left Property It sets the distance in pixel from the left edge of the Form
Color Property: It sets the background colour of the component
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.
TabOrder Property: It sets the tabulation order

Main Events of ComboBox object

OnChange Event: it defines the behaviour in the application when the content of the text is modified.
OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnDblClick Event: : it defines the behaviour of the application when the mouse is double-clicked on the component
OnKeyDown Event: it defines the behaviour of the application when a key is pressed
OnKeyUp Event: it defines the behaviour of the application when a key is released
OnKeyPress Event: it defines the behaviour of the application when a character is pressed on the keyboard
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when the mouse is placed on the component
OnEnter Event: it defines the behaviour of the application when the component becomes active
OnExit Event: it defines the behaviour of the application when the component stops being active

4.4.7 CheckBox object: adding properties and events

To add the component **CheckBox** into the Form, select **TCheckBox** on the Window **Components,** press **Enter,** then **Esc** to close the window. A checkbox named **CheckBox#** will be positioned in the middle of the Form; the name of the component will appear in the label of the box. It is a composite component; it looks like a small square with the graphical characteristics of a single-line textbox on which it is possible to add or remove the checkmark; next to it on the right side there is always a label.

Main Properties of CheckBox object

Checked Property: it defines the status of the box: checked or unchecked
Caption Property: it defines the text of the box
Alignment Property : it defines the position of the text on the right or left of the checkbox
Name Property: it represents the component’s identification name within

the application
Text Property: It is used to add the text of the component that will be visualised
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the upper edge of the Form
Left Property It sets the distance in pixel from the left edge of the Form
Color Property: It sets the background colour of the component
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.

Main Events of CheckBox object

OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnKeyDown Event: it defines the behaviour of the application when a key is pressed
OnKeyUp Event: it defines the behaviour of the application when a key is released
OnKeyPress Event: it defines the behaviour of the application when a character is pressed on the keyboard
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when you place the mouse on the component
OnEnter Event: it defines the behaviour of the application when the component **Event** becomes active
OnExit Event: it defines the behaviour of the application when the component stops being active

4.4.8 RadioButton object

To add the component **RadioButton** into the Form, select **TRadioButton** on the Window **Components,** press **Enter,** then **Esc** to close the window. An option button named **RadioButton#** will be displayed in the middle of the Form; the name of the component will be shown as a the label of the button. It is a composite component; it looks like a small circle with the graphical characteristics of an edit box; on the right side there is always a label.

Main Properties of RadioButton object

Caption Property: it defines the text of the circle
Checked Property: True/False whether it is selected or not
Alignment Property: it defines the position of the text near the circle (right or left)
Name Property: it represents the component's identification name within the application
Text Property: It is used to add the text of the component that will be visualised
Width Property: It sets the width of the component in pixel
Height Property : It sets the height of the component in pixel
Top Property: It sets the distance in pixel from the upper edge of the Form
Left Property It sets the distance in pixel from the left edge of the Form
Colour Property: It sets the background colour of the component
Font Property: (with submenu): It sets the properties of the used font
Hint Property: It sets the text-line to be viewed when you position the mouse on the component.

Main Events of RadioButton object

OnClick Event: it defines the behaviour of the application when the mouse is clicked on the component
OnDblClick Event: : it defines the behaviour of the application when the mouse is double-clicked on the component
OnKeyDown Event: it defines the behaviour of the application when a key is pressed
OnKeyUp Event: it defines the behaviour of the application when a key is released
OnKeyPress Event: it defines the behaviour of the application when a character is pressed on the keyboard
OnMouseDown Event: it defines the behaviour of the application when a key is pressed on the mouse
OnMouseUp Event: it defines the behaviour of the application when a key of the mouse is released
OnMouseMove Event: it defines the behaviour of the application when the mouse is placed on the component
OnEnter Event: it defines the behaviour of the application when the component becomes active (receive the focus)
OnExit Event: it defines the behaviour of the application when the component stops being active

Exercise.

The next step of the exercise is to arrange the components you have added to the form in a correct and logical way. You will set the property of each component, including the Form.

A. You will start from the Form
Move to Object Inspector (F11)
Move to the selection box of the components(Ctrl+Down)
Move to Form 1 (Up) or (Down)
Enter
Move to Caption (Up) or (Down) and type “Students’ data”
Move to Height (Down) and type “600”
Move to Name (Down) and type “data“
Move to ShowHint (Down),open the selection box(Alt+Down) and select True (Up) or (Down) or press (Ctrl+Down) until you reach it
Move to Width (Down) and type “800”
Move to **WindowsState (**Down) open the selection boxt(Alt+Down) and select **wsMaximized**(Up) or (Down) or press (Ctrl+Down) until you reach it

Now you have to set the properties of the first component, the label preceding the editbox student’s name; it will be displayed top left in the Form. This is the procedure to follow:

check that you are still in OI; if not move there (F11)
Move to the selection box of the component (Ctrl+Down)on which you are going to operate
Move to Label1 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Caption (Up) and type “Name”
Move to Hint (Down) and type ””
Move to **Left** (Down) and type **“**10”
Move to Name (Down), and type ”LabelName”
Move to TabOrder (Down) and type “”
Move to Top (Down) and type “10”
Move to **Width** (Down) and check that it is set on value 60

B. The next job is to set the properties of the second component, the editbox student’s name; its position will be to the right of the first component (the label)
check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are

going to operate
Move to Edit1 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Hint (Down) and type ””
Move to Left (Down) and type “70”
Move to Name (Down), and type ”EditName”
Move to TabOrder (Down) and digit “0”
Move to Text (Down) and press Del to cancel the text you find by default
Move to Top (Down) and digit “10”
Move to **Width** (Down) and digit “250”

Now you have to set the properties of the third component, the label preceding the editbox student’s surname; it will be displayed to the right and slightly further away from the preceding editbox, as if to form a new group of components. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to Label2 (Up) or (Down)
Enter
Move to **Font** (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
Enter
Move to **Caption** (Up) and type “Surname”
Move to **Hint** (Down) and type ””
Move to **Left** (Down) and type **“**410”
Move to **Name** (Down), and type ”LabelSurname”
Move to **Top** (Down) and type “10”
Move to **Width** (Down) and check that it is set on value 76

Now you have to set the properties of the fourth component, editbox of the student’s surname; it will be positioned to the right of its label. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to Edit2 (Up) or (Down)

(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Hint (Down) and type ””
Move to Left (Down) and type “496”
Move to Name (Down), and type ”EditSurname”
Move to TabOrder (Down) and type “1”
Move to Text (Down) and press Del to cancel the text you find by default
Move to Top (Down) and type “10”
Move to Width (Down) and type “250”
Now you have to set the properties of the fifth component, the first of the two RadioButtons for gender; it will be displayed below the label of the student’s name, further away vertically, as if to form a new group of components. This is the procedure to follow:
check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to RadioButton1 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Caption (Up) and type “Male”
Move to Hint (Down) and type ””
Move to Left (Down) and type “10”
Move to Name (Down), and type ”RadioButtonMale”
Move to TabOrder (Down) and type “2”
Move to Top (Down) and type “70”
Move to Width (Down) and type “65”

Now you have to set the properties of the sixth component, the second of the two RadioButtons for sex; it will be displayed to the right of the preceding one (the label of the student’s name).This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to RadioButton2 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)

(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Caption (Up) and type “Female”
Move to Hint (Down) and type ””
Move to Left (Down) and type “85”
Move to Name (Down), and type ”RadioButtonFemale”
Move to TabOrder (Down) and type “3”
Move to Top (Down) and type “70”
Move to **Width** (Down) and type “90”

Now you have to set the properties of the seventh component, the Combobox for age; it will be displayed to the right of the preceding RadioButton, slightly further away; it is not part of any group of components and stands graphically isolated from the others .This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to ComboBox1 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Caption (Up) and type “Female”
Move to Hint (Down) and type ””
Move to Items (Up) or (Down
(Ctrl+Enter)
When the window StringListEditor opens type “0 - 10”
(Enter)
Type ”20 – 30”
(Enter)
With Tab move to OK and press (Enter) to confirm and leave the window.
So far you have created three items to select at a later stage.
Move to Left (Down) and type “205”
Move to Name (Down), and type ”ComboBoxAge”
Move to Text (Down) and type “Select age…”
Move to TabOrder (Down) and type “4”
Move to Top (Down) and type “70”
Move to Width (Down) and type “150”

Now you have to set the properties of the eighth component, the Checkbox to identify disability (if any); it will be displayed to the right of the preceding ComboBox, slightly further away; it is not part of any group of components and stands graphically isolated from the others. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to CheckBox1 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Caption (Up) and type “Disability”
Move to Hint (Down) and type ””
Move to Left (Down) and type “385”
Move to Name (Down), and type ”CheckBoxHandicap”
Move to TabOrder (Down) and type “5”
Move to Top (Down) and type “70”
Move to Width (Down) and type “105”

Now you have to set the properties of the nineth component, the label preceding the editbox address; it will be displayed below the first RadioButton, further away vertically, as if to form a new group of components. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to Label3 (Up) or (Down)
Enter
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Caption (Up) and type “Address”
Move to Hint (Down) to and type ””
Move to Left (Down) and type “10”
Move to Name (Down), and type ”LabelAddress”
Move to Top (Down) and type “130”
Move to Width (Down) and check that it is set on value “72”

Now you have to set the properties of the tenth component, the editbox address; it will be displayed to the right of its label. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to Edit3 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Hint (Down) and type ””
Move to Left (Down) and type “92”
Move to Name (Down), and type”EditAddress”
Move to TabOrder (Down) and type “6”
Move to Text (Down) and press Del to cancel the text you find by default
Move to Top (Down) and type “130”
Move to Width (Down) and type “500”

Now you have to set the properties of the eleventh component, the ListBox for the selection of the city of residence; it will be displayed to the right of the editbox address, being part of the same group of data. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to ListBox1 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Height(Up) and type “30”
Move to Hint (Down) and type ””
Move to Items (Down)
(Ctrl+Enter)
When the window StringListEditor opens type “Select a city…”
(Enter)
Type ”Milan”
(Enter)
Type ”Rome”
(Enter)

Type ”Naples”
(Enter)
With Tab move to OK and press (Enter)to confirm and leave the window. So far you have created three items to select at a later stage, plus the introductory text.
Move to Left (Down) and type “602”
Move to Name (Down), and type ”ListBoxCity”
Move to TabOrder (Down) and type “7”
Move to Top (Down) and type “130”
Move to Width (Down) and type “160”

Now you have to set the properties of the twelveth component, the label preceding the multiline editbox (memo) for the notes; it will be displayed below the label of the address, further away vertically, as if to form a new group of components. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to **Label4** (Up) or (Down)
Enter
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to **Caption** (Up) and type “Note”
Move to **Hint** (Down) to and type ””
Move to **Left** (Down) and type **“**10”
Move to **Name** (Down), and type ”LabelNote”
Move to **Top** (Down) and type “190”
Move to **Width** (Down) and check that it is set on value “40”

Now you have to set the properties of the thirteenth component, the “Memo” box; it will be displayed to the right of its label. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to Memo1 (Up) or (Down)
(Enter)
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14

(Enter)
Move to Hint (Down) and type ””
Move to Left (Down) and type “60”
Move to Lines (Down),
(Ctrl+Enter)
When the window StringListEditor opens, press Back to delete the text added by default “Memo1”
With Tab move to OK and press (Enter)to confirm and leave the window.
Move to Name (Down), and type “MemoNote”
Move to TabOrder (Down) and type “8”
Move to Text (Down) and press Del to cancel the text you find by default
Move to Top (Down) and type “190”
Move to Width (Down) and type “700”

Now you have to set the properties of the fourteenth and last component, the component to confirm and save the data that you have added so far; it will be displayed to the bottom right of the window. This is the procedure to follow:

check that you are still inside Object Inspector; if not move there (F11)
Move to the selection box of the component (Ctrl+Down) on which you are going to operate
Move to Button1 (Up) or (Down)
Enter
Move to Font (Up) or (Down)
(Ctrl+Enter)
Select font Arial 14
(Enter)
Move to Caption (Up) and type “Add”
Move to Height (Down) to and type “30”
Move to Hint (Down) to and type ””
Move to Left (Down) and type “650”
Move to Name (Down), and type ”ButtonAdd”
Move to TabOrder (Down) and type “9”
Move to Top (Down) and type “500”
Move to Width (Down) and type “100”

At the end of this long exercise you can see some results. If you press (F9) and wait a few seconds you will see on the monitor the window you have designed in “executable” version; you can now move around, write in the editboxes, select the desired item, add a tick at the item “sex”; however if you press Add nothing will happen: you have more to learn for this.

4.5 Designing Forms accessible to blind users

Aim of this subchapter: The student will learn how to design forms accessible to blind users

In this unit we will give some simple rules in order to make standard Windows applications accessible to blind users.

What graphic indications shall we give to the visually impaired programmers in order to design applications that can be used by visually impaired users?
The aim is not the construction of special “dedicated” applications, but to use standard Windows applications applying the following simple rules.
The target of such rules are both the programmers and all the users of the applications.

Each component has a Top property and a Left property, which determine the distance in pixels of the component from the top and left edges of the from. The height and width properties of the component determine the size of the component. you will need to establish your own conventions for the sizing and positioning of components, but as a general rule you should work in multiples of 10, and leave at least 10 pixels between components. You may find it easiest to work with square components 50 pixels high and 50 pixels wide, with a 50 pixel space between components. This is what is suggested for simplicity in the exercises in the next chapter. However, most programmers use components that are rectangular with the width at least twice the height. If you are working with a complicated from you will need to design it first; try laying it out using lego or a peg board before you start the coding.

4.5.1 Sticking to Windows standards

Generally speaking the graphic aspect of an application should be as similar as possible to the Windows standards. The standard settings of the components properties are those offered by the computer: (Display, Appearance, Scheme)
The user’s options are obviously those most comfortable to him; therefore the programmer should avoid overwriting such parameters; in particular;
The components colour and the caption colour, including the form caption.
If for any reason it should become necessary to change the colour of the background or the text colour, we recommend that you change the text colour as well; otherwise you may display an editbox with black text on

black background.
The components edges. A text box without edges, for example, could be mistaken for a label and vice versa.
Mouse pointers. Personalised pointers could have a negative effect on the display or the readability of the operating system.

4.5.2 Font type and size.

The font should not be smaller than 14 points or at least it should be possible to for the user to change it to their preference. We suggest an easy font (Arial, Times New Roman, Courier New); alternatively it should be possible to personalise the font, too. The default Font is MS Sans Serif, 8 point.

4.5.3 Text description of each component

All the components should include a text description, the description that will be displayed when you move the mouse on the component; such description is often used by screen readers to give the user information about the component. The text should be included in the property "**Hint**" of the component, then the "**ShowHint**" property must be set on "True". For more information select the property "Hint" and press "F1",

4.5.4 Correct positioning of the Components on the Form

Special attention should be given to the positioning of the components on the Form in order to make the application work easily.
We know that the properties "Width" and "Height" are responsible for the components dimensions and "Top" and "Left" are responsible for the positioning of the form. It is therefore necessary to set correctly the properties "Top" and "Left" also when we have to position a component on the bottom right part of the Form.
Generally once the Form dimensions are selected it is possible to position the components paying particular attention to the following:

Leave a space of at least 10 pixels between the components and the edge of the form
Leave a space of at least 10 pixels between the components of the same set, e.g. label and edit box
Leave a space of at least 10 pixels between two sets of components
Check the dimension and the length of the text to be displayed in the

component that is being positioned, before going on to the next component (to the right or below)

WARNING: The setting of some components takes place automatically or semi-automatically; in particular:

“Edit” and “ComboBox”. The height of the component is set automatically on the dimension of the text set by the property Font
“Label”. The Height and the Length of the component are set automatically on the dimension of the text set on the Property Font.
“Memo”. Being a multiline text box, adjusting of the component does not take place automatically.
“ListBox” The setting of the component does not take place automatically. By setting the property “IntegralHeight” on “True” the height of the component will be adjusted so as to contain exactly one or more lines of text. Select the dimension of the text, then type in the property “Height” a value equal or superior to the height in pixel of the lines to be displayed; the height will then automatically set on the correct value.

Example: A text with font “Arial”, 14 points

Height,	Line,	AutoHeight;
0 to 2	0	0
26 to 4	1	26
48 to 69	2	48
70 to 91	3	70

5. “Button, CheckBox and RadioButton” the dimensions of the component do not automatically adjust to the label caption. To adjust correctly the dimensions of the component we advise setting “Courier New”, (a fixed space font), with the bold style in the Font property.
In the following chart the suggested height and width in pixel for specific caption dimensions (in points).

FontSize	Height	Width (1 character)
8	17	28+7 per character
10	20	30+8 per character
12	23	33+10 per character
14	26	36+11 per character
16	29	39+13 per character
18	32	42+14 per character

4.6 Moving from Form to Application

Aim of this subchapter: The student will learn the correct way to create an Application by means of an example

In this unit we will show how to create an Application, step by step.

4.6.1 An example of Application: "Hello!"

Exercise.
We are going to design a simple application showing the word "Hello!" followed each time by the name of a European country, by simply pressing a key repeatedly.

To design such application you will:
Add the necessary components to the default form
Set the components properties
Define the event handlers

4.6.2 Adding the components to Hello

After adding the components **Tedit, Tlabel and TButton** (plus the Form that, as you know now in BCB is automatically added by default) you will find four components in the components selection box of Object Inspector: Button 1, Edit1, Label 1 and Form1

4.6.3 Setting the components properties

Setting the property of **Button1**:
Caption = "" cancel leaving a void
Font = Arial, Bold, 14 (select from the standard dialog window);
Hint = "Change..." (type);
Left = 150 (type);
ShowHint = True (select from the list);
TabOrder = 1 (type);
Top = 70 (type);
Width = 380 (type).

Setting the property of **Edit1**:
Font = Arial, Bold, 14 (select from the standard dialog window);
Left = 150 (type);
Top = 50 (type);
Hint = "View" (type);
ShowHint = True (select from the list);
TabOrder = 0 (type);
Text = "Press a button..." (type);
Width = 380 (type).

Setting the property of **Label1**:
Alignment = taRightJustify (select from the list);
Caption = "Description" (type);
Font = Arial, 14 (select from the standard dialog window);
Hint = "Description" (type);
Left = 10 (type);
Top = 10 (type);
ShowHint = True (select from the list);
Width = 130 (type).

Setting the properties of **Form1**:
Name = "Hello!" (type);
Caption = "Hello!" (type).

4.6.4 Defining the Event Handlers

To write the code for the event handler you will move to the window **Unit1.cpp** by pressing (F11**)**
Leave an empty line under the text THello *Hello.

You will type this code:

```
const int MAX_HELLO = 6;
String Edit[MAX_HELLO];
String Button[MAX_HELLO];
String Label[MAX_HELLO];
int counter_Edit=0;
int counter_Button=0;
int counter_Label=0;
```

Back to OI you will select the Form **Hello,** then move to **Events,** select **OnCreate** and press (Ctrl+Enter**)**

You will type this code between the braces:

```
Edit[0]="Hello Austria!";
Edit[1]="Hello England!";
Edit[2]="Hello Germany!";
Edit[3]="Hello Greece!";
Edit[4]="Hello Italy!";
Edit[5]="Hello Slovakia!";

Button[0]="Austria";
Button[1]="England";
Button[2]="Germany";
Button[3]="Greece";
Button[4]="Italy";
Button[5]="Slovakia";

Label[0]="Austria";
Label[1]="England";
Label[2]="Germany";
Label[3]="Greece";
Label[4]="Italy";
Label[5]="Slovakia";

Button1->Caption=Button[0];
counter_Button++;
```

Back to OI select the component **Button1,** then move to **Events,** select **OnClick** and press **Ctrl+Enter.**

Now, type this code between the braces:

```
Edit1->Text=Edit[counter_Edit];
Button1->Caption=Button[counter_Button];
Label1->Caption=Label[counter_Label];

if (counter_Edit==MAX_HELLO-1)
        counter_Edit=0;
else
        counter_Edit++;

if (counter_Label==MAX_HELLO-1)
        counter_Label=0;
else
        counter_Label++;
```

```
if (counter_Button==MAX_HELLO-1)
        counter_Button=0;
else
        counter_Button++;
```

Now, after saving the work done, you can run the program.

4.6.5 The project

To build an application you start with a project. The project contains the ingredients that will be compiled to produce the application, called: file "**.exe”**
To display the components of the designed application you will open the menu **View/Project/Manager (**Press Ctrl+Alt+F11) and navigate through the items of the tree menu.
Each component will be saved with a different extension according to the nature of the components. In our application the following files will be saved:

Unit1.dfm includes the graphical aspects of the form and the components added to it.
Unit1.cpp includes the C++ code of the event handlers or any additional functions
Unit1.h includes the implemented functions of C++ files
Project1. bpr project file is used to open to open the project in C++ development environment
Project1. cpp includes the function “winmain” to enter the program
Project1. res (we will come back to that later)
To save our project we will open the menu (FileSave All) by pressing Ctrl+Shift+S
You now open a new folder, to save the form *Unit 1*, use the normal way of saving a file. To save the project, *project 1*, follow the normal way of saving. The computer asks you to save the project in the same folder in which the form has been saved; the next job is to replace “project1” with “hello”.
If you minimize the C++ application and open the folder containing the project you will see the results. Now let’s compile and run the program.

4.6.6 Compiling, running the program

Compiling means transforming the development and implementation of the

project in machine language; i.e. a sequence of binary instructions to be interpreted by the operating system.
Now, Move to the menu:
(Run/Run) or F9, after a few seconds Hello.exe will run
Using Tab key move to the button of “Hello”, by pressing (Enter) each time the message on the Edit Box changes.
(Alt+F4) to leave the program.
In the folder where the project has been saved you will find now four new files
Hello.exe = execute file
Hello.obj e **Unit1.obj** = binary files obtained by compiling a .cpp.
Hello.tds = for debugging

4.7 The Menus

Aim of this subchapter: The student will learn how to add menus to an Application

Windows standard applications have Menu Bars. A menu bar is a horizontal line of primary text links usually positioned under the window title bar and above the tools bar (in the case the application has a toolbar). As you know, when clicking on each item a window appears with multiple sub-items.

4.7.1 Adding the component TMainMenu

In BCB the Menu Bar is really a component (just like Edit, Button, Label etc) which must be added to the Form.

- (F11) to move to Form **Hello!**
- (Alt) to move to the **Menu Bar**
- (Right) to move to the menu **View**
- (Down) to move to the item **Component List**,
- (Enter)
- (Down) to move to **(TMainMenu)**
- (Enter)
- (Esc) to leave the window **Component List**

The component is now displayed as an icon at the centre of the Form; its default name is **MainMenu1**.
(F11) to move to window **OI**

Scrolling the properties of **MainMenu1**, you will notice there are no properties related to its positioning, like Top property, Left property etc This means that the component **MainMenu** is a graphic container to be personalised with specific functions similar to those of the navigation bar in an Internet browser.
The next job is to give the Menu Bar a graphical structure.

4.7.2 Designing the Menu Bar

The tasks to design the menu bar are:

- Object Inspector
- (Ctr+Down) to move to the list box of the component on which to operate
- (Up) or (Down) to move to **MainMenu1**
- (Enter)
- (Up) or (Down) to move to **Name**
- Digit “HelloMenuBar”

Each item of “HelloMenuBar” is treated as a sub-component, has its list of properties in window OI and is triggered by event (OnClick). We will see this later.
The properties of each item of MainMenu are:
Caption sets the label of the item in the list (File, Modify,etc)
Checked displays the tick on the left of the item in the list; the tick has a “switch” function
Enabled tells you that the item is active
Hint sets the text line to be viewed when you position the mouse on the component.
Name sets the under-component identification name within the application, the name used by the program to identify it.
Shortcut from a drop-down window it will be possible to select a scheme of shortcut keys from which it will be possible to move to event **OnClick of the item**

The next job is to build our menu bar.
The mail list of **HelloMenuBar** will be constituted by File, View, Help.
The menu **File** has a sub-item called **Close** to leave the application.
The menu **View** has a sub-item called **Change** to trigger the event OnClick which changes each time the language of “Hello”.
The menu **Help** has a sub-item called **Info** to view the Help window.

Check that you are still in Object Inspector

(Ctrl+Down to move to the selection box of the component on which you will operate
(Up) or (Down) to move to **MainMenu1**

(Enter)
(Up) or (Down) to move to **Items**,
(Ctrl+Enter)
You will view the window “Hello->HelloMenuBar”; top left there is an edit box: this is the first item of the main menu
Type “&File”
(Enter)
You will see two new edit boxes, the first to the right of File, the second (active) below
Type “&Close”
(Enter)
You will see now a new edit box below **Close**; press (Right) to move to the empty edit box to the right of **File**)
Type ”&View”
(Enter)
You will view two new edit boxes, the first to the right of **View**, the second (active) below
Type “&Change”
(Enter)
You will see now a new edit box below **Change**; move to the right of View by pressing (Right)
Type “&Help”
(Enter)
You will view two new edit boxes, one to the right of Help, the other one (active), below
Type “&Info”
(Enter)
You will view a new edit box under Info

What is this symbol “&” that you see before each item?

It is an automatic function. The alphabetical character following this symbol is underlined; this means that if you press (Alt+ underlined alphabetical character) you will trigger the event OnClick. If this character is already used for other items, you can put the symbol “&” before the character you want to underline.

Back to OI
(Ctrl+Down)
you will view all the items of the implemented menu, plus the items

previously added.
The properties **Name** of each item will be constituted by their labels (caption) plus “1”,”File1,”View1”,”Help1” etc

4.7.3 Adding, Deleting and Editing Menus

The tasks to add or delete items in **HelloMainMenu** are:

1. Move to Object Inspector
2. (Ctr+Down) to move to the component **HelloMenuBar**
3. (Enter)
4. (Up) or (Down) to move to the property **Items**

(Ctrl+Enter)
Arrow keys to move to the item to be deleted,
(Del)
Arrow keys to move where you want to add a new item.
(Ins)
Type the label of the new item

5. (Enter)

Arrow keys to move to the item to which you want to add a sub-menu,
(Ctrl+Right)
Type the label of the first item of the sub-menu
You will now view two new edit boxes, the first to the right of the item you have just edited**,** the second (active) below.
Now follow the usual procedure

Exercise: Modify the property of an item of the Menu
Move to OI
(Ctrl+Down) to select the component HelloMenuBar
(Enter)
(Up) or (Down) to move to move to the property to modify
Now follow the same procedure as the other components

4.7.4 Implementation of the event OnClick

To implement HelloMenuBar, you will have to trigger the event **OnClick** of **Close**, **Change** and **Info**

Event OnClick of Close.
The procedure is now quite familiar to the student.

From the window **Object Inspector**
(Ctrl+Down) to select the component **HelloMenuBar**,
(Enter)
(Up) or (Down) to move to the property **Items**
(Ctrl+Enter)
Arrow keys to move to **Close**
Back to **OI**
(Ctrl+Tab)to move to **Events**
(Down) to move to **OnClick**
(Ctrl+Enter)
In the window **Unit1.ccp,** type this code between the braces:

Close()

Exercises:

Implement the event **OnClick** of **Change;** add the following code: Button1Click(Sender)
Implement the event **OnClick** of **Info;** add the following code on one line: **Application->MessageBox** ("Press a button to change a message", `"Help",MB_OK);`

Saving your application.

(File/Save All)
F9 for the compilation
Start the program
(Alt) to move to the Menubar

Now you can see the result of the work done.

Chapter 5: Elementary C++ Programming

Aim of this chapter: In this chapter you will learn how to create C++ programs to solve very simple problems.

Solution of simple problems involving concepts of events and sequence, and manipulation of data.
Use of edit boxes, labels and command buttons.
C++ syntax:
variables and constants;
global and local scope;
character and numeric data types;
operators on numeric and character data types;
conversion between data types;
keywords and terminators;
comments;

5.1 Where the components are placed

Each component has a Top property and a Left property, which determine the distance in pixels of the component from the top and left edges of the from. The height and width properties of the component determine the size of the component. you will need to establish your own conventions for the sizing and positioning of components, but as a general rule you should work in multiples of 10, and leave at least 10 pixels between components. You may find it easiest to work with square components 50 pixels high and 50 pixels wide, with a 50 pixel space between components. This is what is suggested for simplicity in the exercises in the next chapter. However, most programmers use components that are rectangular with the width at least twice the height. If you are working with a complicated from you will need to design it first; try laying it out using lego or a peg board before you start the coding.

Problem 1 - Adding two numbers

In the first exercise, you will input two numbers, add them together and output the result.

Input is often performed by means of edit boxes, while output uses labels. The user may want to alter the input, but should not be able to alter the output as it is the result of the calculation.

For this exercise, you will need two edit boxes, for the two numbers to be added. You will need one label for the answer. Each of the edit boxes should also have a label near it, with a suitable caption that explains what it is. The answer label will incorporate a caption.
Note that we are using labels for two purposes. A label is being used to describe an adjacent box containing input or output data; it is also being used to contain output data.
Finally, you need a component to trigger the event of adding the numbers together. There are many ways of doing this, but we will use a command button and click on it when the two numbers have been entered.

Start a new project and add to the form:
2 edit boxes (Tedit)
3 labels (Tlabel)
1 command button (Tbutton).

Reminder:
You add these from the Components box (Alt-V,C). Type the first letter or two of the name of the component you need and press (Enter). When you have all the components you need, press (Escape) to close the Components box.

The components you need are now all piled in a heap in the middle of the form, so the next job is to arrange the components on the form and give them meaningful names and captions. A name like “Edit1” or “Label2” tells you what kind of component it is, but not what it is for. When you use several forms in a program, these should be given meaningful names as well. Our initial programs will only have one form each, so you can leave them with the default name “Form1” if you wish.

For this program, we are going to arrange all the components in a horizontal line across the top of the form, so that it will read almost like a normal addition sum. We will give every component a width of 50 pixels, with a 50 pixel space between it and the next component. So every component will have its Top property set to 0 and its Width property set to 50. The first component will have a Left property of 0, the next one will be 100, then 200 and so on. There are going to be 6 components across this form, so you need to check that the form is wide enough. Set its Width property to at least 600. We will try to make all the components square, so their Height property will also be 50. You will find this does not always happen, especially for label components, but do not worry about it now.

Start with “Label1”. You will need to change its name and caption properties and then move it.
(Reminder: (F11) to reach the Object Inspector, then (Ctrl+Down) to reach the component list.)
Change its Name property to lbl_firstnumber.
Change its Caption property to “First number”.
Change its WordWrap property to true. This will ensure that all of the caption is visible in the label.
Move it to the top left of the form by setting both its Top property and its Left property to 0.
Set its Width and Height properties to 50.
If you change the Height property and then another property of a label, the height may well return to its default value. It does not really matter at this stage, but you should be aware that it will happen.

The next component to place is “Edit1”. This is going to contain the first number for the sum. Again, you will need to change its name and caption properties and then move it.
Change its Name property to edi_firstnumber.
Change its Text property to a blank space, as you will want to type the number in here when you run the program.
Change its position properties to Top = 0, Left = 100, Height = 50, Width = 50.
This is going to be the first component used by the program, so check that its TabOrder property is 0.
Note that the TabOrder is 0 for the first component. You will soon get used to the C++ way of counting, which usually starts at 0 rather than 1.

Next we need to place the button. Change its Name property to btn_add and its caption to Add. Change its position properties to Top = 0, Left = 200, Height = 50, Width = 50.

Now try to place “Label2” and “Edit2” correctly on the form. They are going to be used for the second number. Do this before you check the answer below.

Answer:
Label2 should have properties Name = lbl_secondnumber, Caption = Second Number, Top = 0, Left = 300, Height = 50, Width = 50.
Edit2 should have properties Name = edi_secondnumber, Text is empty, Top = 0, Left = 400, Height = 50, Width = 50.

Label3 is initially going to contain the caption “Answer”. Later it will also contain the answer itself. Place it on the form and then check with the answer below.

Answer:
Label3 should have properties Name = lbl_answer, Caption = Answer, Top = 0, Left = 500, Height = 50, Width = 50.

Note that once the form has been laid out you can navigate round the components using the (Tab) key. You can also move left and right between components which are roughly in a horizontal line, and up and down between components which are roughly in a vertical line. Use the arrow keys for this.

You now have the form designed. If you try to run the program now it should compile and run but not actually do anything. Try it. When you enter two numbers and click the Add button, nothing appears to happen. Why not? You know what should happen when you press the Add button. It is obvious, it should add the numbers together. But a computer can only do what you tell it to do, and you have not actually told it to add the numbers when you press that button.

Double click on the Add button or (F11) to get to the Object Inspector, then (Ctrl+Down) to get the component list, choose btn_add, and then (Ctrl+Tab) to get to the Events tab. Find the OnClick event, tab to the right column, which will be blank, and (Ctrl+Enter). You will be taken to the code window and should find the stub of an event called btn_addClick has been generated for you. When you eventually return to the Object Inspector for btn_add, you will find that the OnClick event has also been named there for you.

The code window for Unit1 has opened, and the cursor should be within the function btn_addClick, which is the event triggered by clicking on the add button. It should look like this:

```
void __fastcall TForm1::btn_addClick(TObject
*Sender)
{

}
```

The braces have an important purpose. They act as a container for all the code that belongs to the function. Carefully type this code between the braces:

```
int num1, num2, answer;
num1 = edi_firstnumber->Text.ToInt();
num2 = edi_secondnumber->Text.ToInt();
answer = num1 + num2;
lbl_answer->Caption = "Answer = " +
String(answer);
```

Now try running the program again. This time, when you click on the Add button you should get an answer to the sum.

What is this magic spell you have just typed in, that causes the program to work as you want it to?

A C++ program contains a number of functions. The first functions you meet are all going to be a special type of function called an event handler. An event handler contains the code to handle the event that you trigger by, in this case, clicking on the add button. This is an “OnClick” event, and C++ names it for you by taking the name of the button and appending the word “Click”. The code for a function is always enclosed in braces { }. Functions also have parameters, which come immediately after the function name and are enclosed in parentheses ( ). We will come back to these later, but for now we can let C++ generate the parameters automatically while we concentrate on the code within the braces.

Our code for this event handler had five lines, which we need to analyse carefully.

```
int num1, num2, answer;
```

In most functions you will need to declare some variables. This statement tells the compiler that you will have three variables which are all integers (whole numbers). Note that the variables in the list are separated by commas. C++ abbreviates integer to int. These are local variables, which only have scope inside the function so they cease to exist as soon as the final } of the function is reached.

You can declare variables anywhere in the code, but it is good practice to declare them all together, at the start of the function. You must always spell a variable’s name correctly, and the correct version is the one you chose when you declared the variable. So when you get compiler error “Undefined symbol”, just check your spelling of the variable in that statement against the spelling when you declared it.
There also seem to be rather a lot of semi-colons in the code.

Punctuation is an important part of the code. In everyday writing, you will get away with missing out commas, or using commas where semi-colons would be more correct. But in programming the punctuation is as important as the spelling. The semi-colon is used by C++ as a terminator, to show that a statement (such as a declaration or a command to do something) has finished. In some languages, you put each statement on a new line. C++ is a free format language, which means that programmers have freedom to squash up or space out the code as they wish. In general, it is easier to read spaced out code, but that is a benefit for the human reader, not the computer. The compiler does not need the code to be separated onto different lines, but it does need statements to be separated with semi-colons, and blocks of code such as functions to be enclosed in braces.

```
num1 = edi_firstnumber->Text.ToInt();
```

This is an assignment statement. The value of the identifier (variable) on the left of the equals sign is assigned (set equal to) the value of the expression on the right. Here, the variable num1 is given the value that you typed into the edit box. C++ uses the symbol to link an object to a particular property. In this case, we want the Text property of the object `edi_firstnumber`. But we do not want it as text, just a string of characters, we want the text as a whole number. The function `ToInt()` converts a string of characters into an integer.

```
num2 = edi_secondnumber->Text.ToInt();
```

This is another assignment statement, very similar to the previous statement. It sets num2 equal to the number you typed into the second edit box.

```
answer = num1 + num2;
```

This assignment statement just adds the two numbers together and puts the answer in the variable “answer”. Remember that the assignment statement must have the result identifier on the left. If you wrote

```
num1 + num2 = answer;
```

you would get a compiler error.

```
lbl_answer->Caption = "Answer = " +
String(answer);
```

This is another assignment statement. The left side is the caption property of the label `lbl_answer`. The variable answer has been declared as an integer, so it has to be turned back into a string of text before it can be displayed in the label. This is achieved using the function `String()`. The + sign also has a new use here, to add two strings together. The result is that the second string is added to the end of the first one, or concatenated. This is called operator overloading, and is a powerful feature of languages such as C++. The + sign “knows” from the context whether it is to add two numbers or concatenate two strings. The inverted commas around `“Answer = ”` tell the computer to output exactly, or literally, what is inside the inverted commas. This is often called a string literal.

Note that we have used functions in two different ways here:

```
edi_secondnumber->Text.ToInt()
String(answer)
```

In the first case we have the syntax

```
objectname.functionname()
```

and in the second case we have the syntax
functionname(parameter)

Which should you use? Strictly, you need

```
objectname.functionname(parameter)
```

The function header for the button click event was generated automatically for you by C++. It is:

```
void __fastcall TForm1::btn_addClick(TObject
*Sender)
```

Here the object is `Tform1`, the functionname is `btn_addClick` and the parameter is `*Sender`. The double colon :: in the function declaration becomes a .or a -> when the function is called.
Sometimes you do not need an objectname and sometimes you do not need a parameter. Hopefully, this will become clearer as the course progresses. For now, program by following the examples you are given and do not worry too much about it.

5.2 Testing

Try running your program a few times. Does it always work correctly? What if you enter numbers with decimal points? What happens if you enter letters instead of numbers? Testing is an important part of program production. Only when you have thoroughly tested a program can you be reasonably sure that you have found the errors. How many commercial programs have you used which always work perfectly and never crash? A large program is rarely bug-free, despite great efforts to test it.

5.3 Comments

There is one further important aspect of programming we have not considered, and that is documentation. If you load the code for this program in six months time, will you remember exactly what it does? The best place to document your program, to explain what it does and how, is in the program itself. That way, you cannot lose the documentation. If you have the code, then you have the explanations.

There are two ways in which you should document code. One is "self documentation"; let the program explain itself by choosing sensible, meaningful variable names and writing clear code which is easy to read. There is often a choice of several ways to code a piece of a program – in general you should choose the one you think is easiest to understand, even if it is not quite the most efficient. You will be thanked for it later when you or a colleague have to make modifications to the program.

Secondly, but at least equally importantly, you should add comments to your code. Comments should not be an optional extra, or something you add just before you hand in your code to a teacher for marking. They should be there from the beginning in every program you write. Start now, and make comments an integral part of your programming life.

C++ offers comments in two forms. Anywhere the compiler encounters `//` it assumes the rest of the line is a comment. Alternatively, you can enclose a comment between `/*` and `*/`. This can be within a line or extend over several lines.

What do you need to comment? Every program should begin with a comment to say who wrote it and when, and possibly the version number and a revision date. There should then be a brief explanation of what the program does; this can save you hours of wading through code looking for a particular program. For example, the current program code might start with:

```
/*
      Enter and add two integers and display result.
      MPW 01/01/02
      Version 1.0  last amended 01/01/02
    */
```

You should also give a comment at the start of every function, to explain what it does:

```
void __fastcall TForm1::btn_AddClick(TObject
*Sender)
// add integers from text boxes, display result
```

Sometimes you will need comments in the code. In a simple program, the variable names themselves should tell the reader enough, but if it is not obvious then add a brief comment to explain their purpose. Similarly a tricky bit of code needs a note of explanation. Will you remember in a year's time why a counter had to start at 3? If not, add a comment. Use your judgement about a sensible level of comments. The following comment is quite unnecessary, and just clutters the program – it makes the program harder, rather than easier, to read.

```
answer = num1 + num2;   /* adds num1 and num2
together and puts the result in answer */
```

Add suitable comments to your code, and then save the code as unit_addnums.cpp and the project as proj_addnums.bpr.

5.4 More Arithmetic

We now want to add buttons to the program so that we can perform subtraction, multiplication and division.
The Add button is positioned at (Top = 0, Left = 200). We want a subtraction button under it, at (Top = 100, Left = 200). Put this button on the form now. It will need to be given a caption of "Subtract", and a name of "btn_Subtract".

The code for Subtract is going to be very similar to the code for Add. Copy and paste the code from the btn_addClick event to the btn_subtractClick event. Change the sign from + to – in the arithmetic. Run the program and test that it works for the new Subtract button and that it still works for the Add button.

Did you remember to include a comment for the new event, and to change the comment for the whole program?
When you alter a program, it is important to check by testing that “unaltered” parts of the program still work. If you code in an organised modular fashion, there should be no side effects on other parts of the program, but you should still check.

Now add multiply and divide buttons under the subtraction button, add code and comments and test again.

If you have coded in the order suggested, you should find that when you run the program you can use the tab key to visit the components in the order edi_firstnumber, edi_secondnumber, btn_add, btn_subtract, btn_multiply, btn_divide. If you examine the TabOrder property for these components, you will find it increases from 0 to 5 in that order. You can change the TabOrder of any component, provided that number is not already used by another number. If there are gaps in the sequence, the focus (the cursor) will just move to the component with the next lowest number. Experiment with this, and then restore the order.

Your testing should also have found a problem with division. If you try to divide 4 by 2 you will get the correct answer, 2. But what happens if you divide 5 by 2? You again get the answer 2. Surprisingly, this is correct! You have told the compiler that you are working using integers, whole numbers. So any fractional or decimal parts of numbers are simply ignored. The division symbol / is another overloaded operator. If it is dividing two whole numbers it gives a whole number answer; if it is dividing two real numbers, with decimal points, it gives a real number answer. What if the numbers are one of each type? In this case, it converts the integer to a real and then divides. With integer division, the result is actually an integer answer plus a remainder. So 5 divided by 2 gives the answer 2 remainder 1. We will come back to this later.

We have decided we want to display the decimal part of the answer if there is a remainder during division. 5 divided by 2 should give the answer 2.5. C++ has several data types for real numbers, one of which is called double. So try declaring answer as `double` in the btn_divideClick event:

```
int num1, num2;
double answer;
```

Test it. What happens? Why are the answers still wrong? Well, you have changed the type of the result but not the type of the operands. You are still dividing two integers, and this results in another integer. 5 divided by 2 is 2 remainder 1. You can change the type of the resulting 2 to a real number, but it is still 2, or 2.00000, it is not suddenly 2.5. If you want to perform "real" division, then the operands have to be reals. Change the type of num1 and num2 in DivideClick to `double`. You also need to change the conversion function on the text of the edit boxes. Instead of the function `ToInt()`, you now need to use the function `ToDouble()`. Compile the program and test again. It should work as you want now.

Look back at the code. In three of the click events we have declared num1 and num2 as integers, but in btn_divideClick they are doubles. Does it matter? No, because they are all independent and have different scope. The integer variable num1 declared in btn_addClick is identified by the compiler as num1 that belongs inside btn_addClick; it is quite different from the variable in btn_subtractClick which also happens to be an integer called num1. These two integers called num1 do not exist outside their function, they occupy different memory locations and the compiler will never muddle them. Similarly, if your name is John, and you are the John who is a member of the Smith household, then you are not going to get accidentally muddled and go home as the John in the same College who is a member of the Williams household.

Save the unit code and project with suitable names. It will be easiest to remember which go together if you give them the same name, but prefixed with unit_ or proj_ ; they will also have different extensions, .cpp for the code and .bpr for the project.

5.5 Converting Currency

The task is to write a program to convert currencies.

You can convert between UK pounds and euros, or US dollars and euros, or between other currencies of your choosing. At the time of writing, the exchange rates are approximately:

£1 = €1.60

$1 = €0.93

but you may wish to use the latest rates.

Before you start coding the form, think about how you are going to design it.

Do you want to convert pounds to euros, or euros to pounds, or both?

If the conversion is just one way, then you need an edit box for the input and a label for the output. But what about a program which converts in both directions? Are you going to have a form in two parts for the two conversions, each with an edit box, a label and a button?

Are you going to have a form with two edit boxes, each of which can be used for either input or output, and two buttons?

Are you going to have another button, which clears the previous conversion before you do a new one?

Are you going to have just one set of components, and check that one edit box contains a value and the other is empty, before performing the conversion?

You might not think so at the moment, but coding is the easy part of writing a program! It is the design which is the hard part, but good design at an early stage can result in a lot less frustration later on. It is very important to think through and design the program before you start coding. The temptation is always to get to the computer and start coding, but resist it, and think through the design first. It really does save time and effort in the long run.

From the list above, the best solution is probably the last one – but we need some more programming concepts before tackling that solution. The solution we will code now will have one set of edit boxes to hold the data and three buttons, for conversion in each direction and clearing previous data.

Start a new application, and add to the form

2 labels

2 edit boxes

3 buttons

Place the two labels and the two edit boxes in the top row (with Top = 0). Make them all suitable sizes and give them suitable names and captions.

Place the three buttons in a row below them.

Do this before you check the suggestion below.

Label1 could be

Name = lbl_pounds

Caption = UK pounds

Top = 0 Left = 0 Height = 50 Width = 50

Label2 could be
Name = lbl_euros
Caption = Euros
Top = 0 Left = 200 Height = 50 Width = 50

Edit1 could be
Name = edi_pounds
Text = (blank)
Top = 0 Left = 100 Height = 50 Width = 50

Edit2 could be
Name = edi_euros
Text = (blank)
Top = 0 Left = 300 Height = 50 Width = 50

Button1 could be
Name = btn_pound_to_euro
Caption = £ to €
Top = 100 Left = 0 Height = 50 Width = 50

Button2 could be
Name = btn_euro_to_pound
Caption = € to £
Top = 100 Left = 100 Height = 50 Width = 50

Button3 could be
Name = btn_clear
Caption = Clear
Top = 100 Left =200 Height = 50 Width = 50

(Hint – remember that it is also a good idea to set the WordWrap property to True.)

You now need the code for the conversion in each direction. Here we make the distinction between variables and constants. A constant is a quantity which does not change while the program is running, while a variable is a quantity which may change while the program is running. The exchange rate is going to be a constant for the duration of the program. You will write it into the code. The amount of money to convert may be changed several times during a run of the program, and will be input as data by the user. To make sure that the exchange rate cannot be changed accidentally elsewhere in the program, you declare it as a constant using the keyword `const`. Constants are usually used by the whole program, so they need to be declared at the start of the code, rather than inside a function. They are said to have global scope. Anything declared inside a function has local scope – it can only be seen and used inside the function.

In the code window (F11), constant declarations should be placed after all the statements starting # and before the first function. Type the following declaration on the line after the form declaration (`TForm1 *Form1;`):

```
        const double rate_ptoe = 1.6; /* conversion rate for pounds to euros */
```

We also need a conversion factor from euros to pounds. This is going to be another constant. You could calculate the value yourself and then code it. There are disadvantages to this. You might make a mistake; it seems silly to use a computer for the other calculations and then have to do a calculation yourself; and finally, if the exchange rate changes, you will have to do another calculation. A good program involves the minimum number of changes when it has to be updated. So declare the second constant in terms of the first:

```
const double rate_etop = 1/rate_ptoe; /*
conversion rate for euros to pounds */
```

Then, when the exchange rate changes, you only have to change the code in one place. Add this line of code under the first constant declaration. Another golden rule of programming is that you cannot use anything until you have declared it. If you put this declaration of rate_etop before the declaration of rate_ptoe, then you would be trying to use the value of rate_ptoe before you had declared it and set its value. The compiler will not let you do this, and will give you an error message. Try it now – it is a good idea to deliberately make small and controlled mistakes to see how the compiler responds. Then, when you get an unintentional compiler error later in your programming, you will have a better idea about what the error messages mean. What happens if you miss out the semi-colon at the end of the line? That is a very common error, but the compiler rarely tells you directly that the problem is a missing semi-colon.

Now that you have the exchange rate coded, you need to write the event code for the buttons. You will type a value into the edit box edi_pounds, click the button labelled “£ to €”, and read the result from the edit box edi_euros. The event code needs to:

Take the value from edi_pounds and turn it into a number.
Multiply the number of pounds by the conversion factor to get euros.
Turn the euros back to text and put it in edi_euros.

You will need to declare two local variables in the btn_pound_to_euroClick event, to hold the number of pounds and the number of euros. Declare them both as double:

```
  double pounds, euros;
```

Try to code the other three lines of the event. Compile the program and test that it works correctly. The other two buttons do not do anything yet. Your code should be similar to this:

```
double pounds, euros;
pounds = edi_pounds->Text.ToDouble();
euros = pounds * rate_ptoe;
edi_euros->Text = String(euros);
```

Now code the btn_euro_to_poundClick event.

Finally, code the btn_clearClick event. That just needs to set the two edit boxes to blank. Try it before checking with the code given below.

```
  edi_euros->Text = "";
edi_pounds->Text = "";
```

Test that the program works properly. There are some things you may want to note for later improvement. For example, what happens when you click a button before entering a value in the edit box?

(Reminder – have you included suitable comments in your program? Save the unit and project.)

5.6 Further Exercises

1. Write a program to convert temperatures between the Fahrenheit and Celsius scales. You will need the formulae:

degF = degC * 9/5 + 32
degC = (degF – 32) * 5/9

For your testing, remember that water freezes at 0°C (32°F) and boils at 100°C (212°F).
(Hint – are you going to work with int or double numbers? If your answers do not seem quite correct, remember the earlier exercise about integer division)

2. Write a program to convert between other units, such as grams and pounds, miles and kilometres, litres and pints.

3. A theme park sets its ticket prices as follows:

Infants under 3	FREE
Children aged 3 – 11	€ 10
Adults and children 12 or over	€ 15
Senior Citizens	€ 12

Write a program which allows the user to enter the number of people in each category and then calculates the bill. Code the ticket prices as constants.

4. Program 3 has to be recompiled every time the ticket prices change, and at some times of the year the prices change almost daily. Amend your program so that default ticket prices are given in edit boxes when the program loads, but can be altered by the user.
(Think about how you are going to do this. Should you change the text property of the edit box, or code it to execute when the form loads?)

5. The management would like to know the total number of people in each category who have bought tickets, and the total amount of money taken. Amend your program again so that it displays updated totals every time a new bill is calculated.

6. Reload your first program, that did simple arithmetic. Add buttons to perform extra functions. For example, the operator % is used to calculate the remainder in integer division, so that 5 / 2 = 2 and 5 % 2 = 1. Add another label to the form which only becomes visible during integer division, and displays the remainder.

7. Perform the same functions (add, subtract etc.) on three numbers rather than two.

8. Change all the numbers to double rather than integer.

9. Write a program which invites the user to enter their name in an edit box. On clicking a button, a personalised greeting is displayed.

5.7 Summary

In this chapter you have written your first programs.

You are familiar with the label, edit box and button components.

You can position and size components on a form and change their names and captions.

You can code OnClick events.
You can declare global and local variables and constants.
You can use terminators.
You can use assignment statements.
You can perform arithmetic on integers and real numbers.
You can convert string data in edit boxes into numbers for calculations, and convert numbers back to strings for display.
You can document your program with comments.
You can test your program.

Chapter 6: Solving Problems and Creating Solutions Using BCB5

Aim of this chapter
In this chapter you will learn how to create program solutions to more complicated problems. You will learn the programming constructs for selection and iteration.

In this chapter rhe following subjects will be eplained:
Solution of simple problems involving concepts of selection and iteration;
Coding in C++ of if, if … else and switch statements;
Coding in C++ of while, for and do … while loops;
Strings
The use of dialog boxes.

6.1 If statements

In the previous chapter, you wrote a program to convert amounts of currency between pounds and euros. You may remember that several ways of doing this were suggested, including:
“Are you going to have just one set of components, and check that one edit box contains a value and the other is empty, before performing the conversion? …
From the list above, the best solution is probably the last one – but we need some more programming concepts before tackling that solution. The solution we will code now will have one set of edit boxes to hold the data and three buttons, for conversion in each direction and clearing previous data.”

The time has now come to look at that solution. It would be a good idea to create a new folder for the examples in this chapter. In it, make a copy of all the files from the currency program. Load this version of the currency program and remind yourself how it works. You used three buttons, for converting £ to €, € to £, and clearing the input boxes. We now want to amalgamate the two conversion buttons into one, called “Convert”.

When you click the Convert button, there are four possible states for the two edit components. Each of them can be empty, or hold data. There is an added complication, in that the data may not be a number, but we will not worry about that now.

We shall start by assuming the user is behaving properly, and has entered data into one of the edit components and then clicked Convert.
The OnClick event for Convert has to do the following:

1. detect which edit component contains data
2. convert the data in the edit component from string to numeric
3. perform the conversion between currencies
4. write the answer into the empty edit component.

Steps 2, 3 and 4 should present no problems, as you have already done them. What about step 1? Which edit component has the number and which is empty? It is easier to check for the empty one. Check first whether edi_pounds is the empty component. The code for this is:

```
if (edi_pounds -> Text == “”)
```

Note that the condition to be tested is always placed in parentheses ( ).The condition usually includes a relational operator. There are six relational operators in C++. These are:

- equals	==
- not equals	!=
- greater than	>
- greater than or equals	>=
- less than	<
- less than or equals	<=

Most of them are obvious, but be especially careful to use a double equals for the relational operator equals. It is easy to forget and use a single equals, and this may not cause a compiler error. However, it will set the object on the left equal to the value on the right and then the condition will always be true – which is a semantic error as it is not what you intended to happen.

Steps 2, 3 and 4 of the algorithm are to be performed if the condition is true, and these statements are enclosed in braces { … }. So the whole thing is therefore:

```
if (edi_pounds -> Text == “”)
{
   euros = edi_euros -> Text.ToDouble();
   pounds = euros * rate_etop;
   edi_pounds -> Text = String(pounds);
}
```

Now, what if edi_euros is the empty component? You could write another block of similar code:

```
if (edi_euros -> Text == “”)
{
pounds = edi_pounds -> Text.ToDouble();
euros = pounds * rate_ptoe;
edi_euros -> Text = String(euros);
  }
```

This would work, but is not as efficient as it could be. If the user has done as instructed, and entered a value into one of the edit components before clicking the button, then either one is empty, or the other is empty. We only need to check one of the boxes, and do not have to check both separately. The C++ syntax corresponding to this is the if … else statement. In this case, it becomes:

```
if (edi_pounds -> Text == “”)
{
   euros = edi_euros -> Text.ToDouble();
   pounds = euros * rate_etop;
   edi_pounds -> Text = String(pounds);
}
  else
{
pounds = edi_pounds -> Text.ToDouble();
euros = pounds * rate_ptoe;
edi_euros -> Text = String(euros);
  }
```

Now edit your program. Add a button with the caption “Convert” and an OnClick event containing the above code. When you are sure it is working correctly, you can remove the two conversion buttons which are no longer needed. Note that you will also need to remove the code for the OnClick events from the unit (.cpp) and corresponding header (.h) files. You will need to open the header file for this.

What happens if both edit components contain data when you click on the convert button? What happens if they are both empty? If you cannot answer these questions, then you have not tested the program thoroughly! In the first case, where both components contain data, whatever is in the pounds box will be converted into euros and the value will appear in edi_euros. In the second case, the program will crash.

6.2 Dialog boxes

Programs often contain code so that if the user makes a mistake they get an error message, rather than the program simply crashing. The error message often uses a dialog box. The user reads the message, clicks an OK button, and tries again. In BCB5 the dialog box is a form.

We are now going to create a dialog box which will appear if the user makes a mistake and clicks the convert button before entering data to convert.

Add a new form to the project (File – New Form).
Change its Name property to frm_NoData, and its Caption property to Data Error.
Its size needs to be changed so that it will occupy the middle of the main form when it is activated. Suitable values are:

Height = 150
Left = 300
Top = 200
Width = 300

Change the BorderStyle property to bsDialog. (This has the effect of removing the minimise and restore buttons at the top right of the form, so that the user cannot resize it.)
Add a label to the form, with a caption which explains the error, e.g. “Error – no data to convert.” Place the label in a suitable position.
A dialog box like this usually also has an OK button to close it. BCB5 provides a suitable component for this, as a TbitBtn. Add one of these to the form. A suitable position is near the bottom and in the middle. Change its Kind property to bkOK (either type it in, or choose it from the dropdown list using Alt+DownArrow). This is a predefined button which has a caption of a green tick and the word “OK”, and already contains code to close the form (dialog box) when it is clicked.
Save the form as error.cpp (Alt+F, S)

Now go back to the original form you were working on. There are several ways of doing this, such as (Shift+F12). It is probably still called Form1. We now need to add code so that the error message appears if both edit boxes are empty. You can combine conditions using the logical operators AND and OR, which are coded in C++ as && and ||.
So the condition “if the pounds box is empty and the euros box is empty” would be coded as:

```
if ((edi_pounds -> Text == “”) &&
    (edi_euros -> Text == “”))
```

Note the two sets of parentheses. The inner sets, around each individual condition, are not essential but make the statement easier to read.

If that condition holds, we want the error form (dialog box) to appear:

```
frm_NoData->ShowModal();
```

Using ShowModal rather than just Show means that the user has to close the dialog box before continuing with the program.

You could try just adding this code, but it does not completely solve the problem. What you really need to do is two separate tests, where the second test depends on the result of the first test:

```
If the pounds box is empty then
  if the euro box is empty then
give the error message
  else
convert euros to pounds
else (the pounds box was not empty, so)
  convert pounds to euros.
```

This is called a nested if statement, and occurs quite frequently. The code for this is:

```
if (edi_pounds -> Text == “”)
{
  if (edi_euros -> Text == “”)
  {
      frm_NoData->ShowModal();
  }
  else
  {
   euros = edi_euros -> Text.ToDouble();
   pounds = euros * rate_etop;
   edi_pounds -> Text = String(pounds);
}
}
else
{
pounds = edi_pounds -> Text.ToDouble();
euros = pounds * rate_ptoe;
edi_euros -> Text = String(euros);
}
```

You might like to try compiling. If you do, you will get a compilation error. Why? Remember what we said earlier, that everything has to be declared before it can be used. The OnClick event for the Convert button in Form1 has no knowledge of the second form you have created. The information about the second form is all contained in the pair of files error.cpp which you saved earlier, and error.h. This is called a header file. It is created by the IDE and contains a list of the information which will be needed if its form needs to be accessed by another form. You need to tell the compiler to #include the error unit in the currency unit. Switch to the code for the currency form, and choose File, Include Unit Hdr from the main menu (Alt+F,I). Choose the error.h file from the list in the dialog box. Then compile, run and test the program again.

6.2.1 Exercise

We still have not solved the problem of the user clicking the Convert button when there is data in both boxes. Can you add another error dialog box (or even make the existing one multi-purpose) to deal with this situation?

Example - Checking Passwords
Passwords are frequently needed to access computers and similar systems such as cash dispensers. For this exercise, we want the user to enter a four digit code, and then check whether it is valid. The rule for a valid code is that the first three digits added together and divided by 7 give a remainder equal to the fourth digit.
So valid codes would include 1236, 9873 and 5551, while invalid codes would include 5678, 2222 and 6543.
The algorithm is fairly simple:

Enter a 4 digit code;
- Click a button to check the code;
- Check the code;
- Display a message about the code’s validity.

You need to create a form with:
- A label giving instructions
- An edit component to contain the code
- A button to activate the check
- A label or a dialog box to give the result of the check. Try both!

Create the form now. The following discussion assumes you have called the edit component edi_code, and that the result is going into a label called `lbl_message`.

We will begin by assuming that the user has entered a numeric code, and not included any letters or other characters. The steps needed to check the code are:

- Convert the code in the edit component to an integer;
- Separate the digits of the code;
- Calculate the sum of the first 3 digits and the remainder when the sum is divided by 7;
- Check whether the remainder equals digit4.

Consider the four digit number 1234.
You can obtain the first digit by dividing by 1000. 1234 / 1000 = 1.
You can obtain the fourth digit by taking the remainder after dividing by 10. 1234 % 10 = 4.
Obtaining the two middle digits is slightly more difficult.
1234 / 100 = 12 and 12 % 10 = 2. So (1234 / 100) % 10 = 2. We obtain the second digit by dividing the number by 100 and then taking the remainder when the answer is divided by 10.
The third digit may be obtained by dividing the number by 10 and then taking the remainder when the answer is divided by 10 again. 1234 / 10 = 123 and 123 % 10 = 3. So (1234 / 10) % 10 = 3.

The OnClick event can now be coded:

```
void __fastcall TForm1::btn_checkClick(TObject
*Sender)
{
int number, digit1, digit2, digit3, digit4,
remainder;
number = edi_code ->Text.ToInt();
digit1 = number /1000;
digit2 = (number / 100) % 10;
digit3 = (number / 10) % 10;
digit4 = number % 10;
remainder = (digit1 + digit2 + digit3) % 7;
if (digit4 == remainder)
   lbl_result -> Caption = "Valid code entered";
else
   lbl_result -> Caption = "Invalid code
entered";
}
```

Complete the coding and run the program. Test it thoroughly. Amend it so that a dialog box gives the result, rather than a label.

What happens if the user enters letters or other characters for the code? The program crashes. The way to prevent this, is to examine each character entered and make sure it is a digit before trying to do any arithmetic on it. To do this, we will use the BCB5 String type, and begin by copying the contents of the edit component into a String, which can be examined character by character. Every rule has its exceptions, and Strings are the one exception to the C++ rule that counting always starts at 0. The first character in a String is 1 and not 0. (There is, of course, a good reason for this – compatibility with Pascal and Delphi which is used to write many of the underlying classes for BCB). Characters in a String are accessed using square brackets […] with the number, or index, of the character being placed in the brackets.

The start of the OnClick event will look like this:

```
// declarations
String code;
int digit1, digit2, digit3, digit4, remainder;
//copy code from edit component into String
code = edi_code -> Text;
// check that first character is a digit
if (code[1]>='0'&& code[1] <= '9')
        digit1 = code[1] - '0';
```

The last two lines need some more explanation. We need to check that the first character is between 0 and 9, but we are checking it is between characters 0 and 9, not numbers 0 and 9. Single quotes are used in C++ to denote a character constant, or literal, to distinguish it from a number. digit1 is then obtained by taking the character (which is still an ASCII code) and subtracting the ASCII code for 0 from it to obtain the correct number.
This if statement needs to be repeated for each of the other three characters in the code.

What are we going to do if the character is not a digit? We need some means of flagging this up, so that we do not try to do arithmetic later on invalid data, and crash the program. The simplest solution is to declare a bool (true-false) variable. It can be set to true initially, and reset to false if a problem is found. The whole OnClick event then becomes:

```
void __fastcall TForm1::btn_checkClick(TObject
*Sender)
{
   String code;
   int digit1, digit2, digit3, digit4, remainder;
   bool valid = true;
   code = edi_code -> Text;
   if (code[1]>='0'&& code[1] <= '9')
        digit1 = code[1] - '0';
   else valid = false;
   if (code[2]>='0'&& code[2] <= '9')
        digit2 = code[2] - '0';
   else valid = false;
   if (code[3]>='0'&& code[3] <= '9')
        digit3 = code[3] - '0';
   else valid = false;
   if (code[4]>='0'&& code[4] <= '9')
        digit4 = code[4] - '0';
   else valid = false;
   if (valid)
   {
      remainder = (digit1 + digit2 + digit3) % 7;
      if (digit4 == remainder)
         lbl_result -> Caption = "Valid code
entered";
      else
         lbl_result -> Caption = "Invalid code
entered";
   }
   else
      lbl_result -> Caption = "Invalid code
entered";
}
```

Why have we written the code for setting the caption to invalid, twice? Try rewriting the code so that it is only needed once. It can be done, but it makes the program much less easy to read. It is set to invalid under two different circumstances – a numeric code which does not obey the rule, or a code which contains non-numeric characters.
Enter the code and test that the program works correctly. Again, you might like to use a dialog box instead of a label.

6.2.2 Further Exercises on Selection

1. Write a program which converts percentage marks into grades, e.g.
   75% and above is grade A
   66 – 74% is grade B
   below 35% is a grade F.

2. In exercise 3 at the end of the last chapter, you wrote a program for selling theme park tickets. Adapt the program so that when a group of people arrives they are offered family tickets if they are cheaper. A family ticket admits 2 adults and up to 4 children for the same price as 2 adults and 2 children. (Hint – you might want to make some components on the form visible only if family tickets are being offered.)

3. Amend the program again, so that senior citizens can be counted with the children for family tickets.

4. Adapt the password checking program so that a valid code is 4 digits and the first digit is a check digit such that it equals the sum of digits 2, 3 and 4 divided by 3. Thus, 3234 and 5674 are valid but 4321 and 3330 are invalid.

6.3 Switch Statements

The if statement allows a choice to be made between two alternatives. If there are several alternatives, nested if statements are needed and the program can look quite complicated. The switch statement offers an alternative way of dealing with multiple choices.

For example, if you had the days of the week as day numbers and wanted to convert them to day names, you could either use a number of if … else statements or a more readable switch statement:

```
int daynumber;
String dayname;
…
switch (daynumber)
{
  case 1: dayname = “Sunday”;
          break;
  case 2: dayname = “Monday”;
          break;
case 3: dayname = “Tuesday”;
          break;
case 4: dayname = “Wednesday”;
          break;
case 5: dayname = “Thursday”;
          break;
case 6: dayname = “Friday”;
          break;
case 7: dayname = “Saturday”;
          break;
    default: dayname = "not known";
}
```

Note that there is a `break;` between every case of the switch statement. This is necessary so that when the correct statement has been executed, control passes to the end of the switch statement. Without the break, the correct case statement will be executed, plus every other case until the end.

Take the grades program you wrote for exercise 1 above. For every grade, add an appropriate comment using a switch statement, e.g.

```
switch (grade)
{
  case ‘A’: lbl_grade -> Caption = “A – excellent
work!”;
  break;
```
…

The variable for the switch statement can be an integer, a character, or any other enumerated type. The default case at the end is not essential, but it is a good idea to include it, in order to trap any errors.

In the next exercise, we are going to determine whether an input date is valid. The user will enter a day of month number, a month number and a year, and the program will decide if the date is valid. So 1 1 2001 will be valid, but 31 11 2002 will not as there are only 30 days in November. 13 13 2013 will not be valid as there are only 12 months in the year, but what about 29 02 2016? 29 February is going to need special consideration, as it only occurs in a leap year. For simplicity, we will restrict the program to dates between 1901 and 2099.

Set up the form for this program, and then check that you agree with the suggestion below.

You will need three edit components, each with a label, for the input of day, month and year. You will need a label or dialog box for the result, and a button to click for the calculations.

Write the algorithm, the set of steps that the OnClick event will have to perform. It should do the same as the list below, but may not be in the same order.

1. Check that edi_year contains digits.
2. Convert the number in edi_year to an integer and check it is between 1901 and 2001.
3. Check that edi_month contains digits.
4. Convert the number in edi_month to an integer and check it is between 1 and 12.
5. Check that edi_day contain digits.
6. Check the number in edi_day is an integer between 1 and 31.
7. If the day is 31, check the month is 1,3,5,7,8,10 or 12.
8. If the day is 30, check the month is not 2.
9. If the day is 29 and the month is 2, check the year is a leap year.

Now you just need to code it. Steps 6, 7, 8 and 9 could be coded together using a switch statement that determines the maximum number of days allowed in the month. The following code assumes that you have already tested the year and month and that they are valid:

```
switch(month)
{
  case 1: case 3: case 5: case 7:
  case 8: case 10: case 12:
       maxdays = 31;
       break;
  case 2:
       if (year % 4 == 0)
           maxdays = 29;
       else
maxdays = 28;
       break;
  default:
       maxdays = 30;
}
if ((day < 1) ||(day > maxdays))
  valid = false;
```

If at any stage, the bool valid becomes false, there is no need to continue the testing as the date is invalid. How are you going to do this? You will probably use a series of nested if statements. Note that in the switch statement, if more than one case has the same code, then the cases can be listed together. They do not have to be in any particular order. By convention, the default case is usually last, but it does not have to be.
Finish coding your program, save, compile and test it.

6.4 Iteration

Iteration, or repetition, is the third of the main control structures used in programming. It allows you to write a block of code once and then repeat it a number of times. There are three main ways of constructing a loop:

- a definite loop is used when you know in advance how many times you need to repeat the code. In many languages this is called a “for” loop. C++ has a for loop, although it does not work in quite the same way as in most languages.
- An indefinite loop is used when the code is to be repeated while a condition is true or until a condition is true. There are two different indefinite loops:
- In a pre-tested loop the continuation condition is tested before the loop is executed. If the condition is initially false, the loop is not executed at all. This loop in C++ is usually called a “while” loop.
- In a post-tested loop the continuation condition is not tested until after the loop has been executed. Even if the condition is initially false, the loop is always executed at least once. This loop in C++ is usually called a “do … while” loop.

So which loop should you use? Often, it does not matter, and one kind of loop will just seem a more natural choice. The pre-tested while loop is the fundamental loop, in that it can be used in every case. Let us do a simple example, showing how the three loops can be used.

First, we want to write a program which calculates the average of 2 numbers. Try it, and then compare your solution with the suggested solution below.

You need a form with 2 edit components for the two numbers, 2 labels to describe the edit boxes, a label for the answer and a button to click for the calculation. The code for the OnClick event will be something like:

```
      double num1, num2, average;
```

```
num1 = edi_num1 -> Text.ToDouble();
num2 = edi_num2 -> Text.ToDouble();
average = (num1 + num2)/2.0;
lbl_answer -> Caption = String(average);
```

Now, suppose you want a program that calculates the average of 10 numbers. You could code 8 more edi_num components and have 10 num variables in the code, but it would be a little tedious. What if you wanted the average of 20 numbers, or 200, or 2000? At some stage this solution method is going to become impossible.

An alternative is to have just one edit box and type the numbers into it one at a time, clicking a subtotal button each time. The label caption can be amended to “Enter number n”, where n is updated to the current number each time. When all the numbers have been entered, the average can be calculated using a second button. This is closer to the idea of a loop, but does not actually use one, as the repetition is performed by the user clicking on the subtotal button and not by the code.
This example gives code for the solution :

```
TForm1 *Form1;
int total = 0;
int count = 0;
//-------------------------------------------------
------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//-------------------------------------------------
------
void __fastcall TForm1::btn_subtotalClick(TObject
*Sender)
{
int number;
number = edi_num -> Text.ToInt();
total += number;
count ++;
lbl_instructions -> Caption = "Enter number" +
String(count+1);
edi_num -> Text = "";
}
//-------------------------------------------------
------
```

```
void __fastcall TForm1::btn_averageClick(TObject
*Sender)
{
float average;
average = float(total)/count;
lbl_answer -> Caption = "Answer =  " +
String(average);
}
//-----------------------------------------------
------
void __fastcall TForm1::btn_clearClick(TObject
*Sender)
{
count = 0;
total = 0;
lbl_instructions -> Caption = "Enter number 1";
lbl_answer -> Caption = "Answer";
}
```

The variables total and count are going to be used by all three events. They need to be accessible by all the events, so they need to be declared globally rather than locally inside one of the event handlers. They are therefore declared near the beginning of the code, straight after the form declaration.

The variable number is only used to transfer a number from the edit box and add it into the total. It can therefore have local scope and be declared inside btn_subtotal's OnClick event handler. Two new operators are introduced in this function, both very common in C++ programs.
The operator += is a shorthand form and means add whatever is on the right to what is on the left, and store the answer back in the left variable.

```
total += number;
```

is just a shorthand form of

```
total = total + number;
```

Related shorthand operators include -= *= and /=.
The operator ++ is very common and means increment by 1.

```
count ++;
```

is just a shorthand form of:

```
count = count + 1;
```

Closely related to this is the decrement operator --.
As an aside, one explanation for the name of the language C++, is that it is an incremental improvement on C.

Finally, look at the calculation of the average in the OnClick event handler for btn_average.

```
float average;
average = float(total)/count;
```

average has been declared as float (it could also have been declared as double) because the average of the numbers may not be a whole number. But we know from the previous chapter that if we divide one integer by another, the answer is also an integer. To obtain a float answer, we need to make sure at least one of the operands is a float. This has been done by writing float(total), which is known as “casting” and has the effect of turning total from an integer into a float.

If you have not done so already, create this program and test it.

The disadvantage of this solution to the original problem is that we have typed in each number, and then it has been lost before we could type in the next number in the list. Is it possible to keep all the numbers on the screen as we enter them, and then just click one button to calculate the average? One solution to this is to use a different component to store the numbers – a memo component. A memo component is a multi-line edit component. The numbers can be added to the memo component, one per line, and then the button clicked to calculate the average.

Design a screen for this new program. It needs a memo component, a label for instructions, with a caption such as “Enter data, one number per line”, a button and a label to contain the answer.

The code for the OnClick event is:

```
void __fastcall TForm1::btn_averageClick(TObject
*Sender)
{
int count;
int total=0;
float average;
count = mem_numbers->Lines->Count;
for (int i = 0; i < count; i++)
   total+= mem_numbers -> Lines ->
Strings[i].ToInt();
average = float(total)/count;
lbl_answer->Caption = "Average:  " +
String(average);
}
```

The Lines->Count property of the memo component gives the number of lines of data in the memo box. The contents of line n is stored in Lines->Strings[n] with counting beginning at 0. The function has used a for loop, as the number of numbers to be added is known at the beginning of the loop; it is count. The for loop itself consists of two lines:

```
for (int i = 0; i < count; i++)
   total+= mem_numbers -> Lines ->
Strings[i].ToInt();
```

If there was more than one statement of code to be executed within the for loop, braces would be needed to enclose the code.
The for line consists of three parts:

an initial condition, setting i = 0 (note that it is common practice to declare the loop variable within the for loop; i has a very local scope of just two lines of code);
a continuation condition, the loop will be repeated as long as i is less than count;
an increment, i is incremented by 1 at the end of each loop.

In the following line, the string on line i is converted to an integer and added into the total.

A while loop could also be used here. In this case, the code would be:

```
void __fastcall TForm1::btn_averageClick(TObject
*Sender)
{
int count;
int total=0;
int i = 0;
float average;
count = mem_numbers->Lines->Count;
while (i < count)
{
   total+= mem_numbers -> Lines ->
Strings[i].ToInt();
   i++;
}
average = float(total)/count;
lbl_answer->Caption = "Average:  " +
String(average);
}
```

The code is very similar, but the loop consists of two statements as the counter I has to be explicitly incremented within the loop.

Assuming that there will be at least one number in the list, a do … while loop could also be used for this program:

```
void __fastcall TForm1::btn_averageClick(TObject
*Sender)
{
int count;
int total=0;
int i = 0;
float average;
count = mem_numbers->Lines->Count;
do
{
   total+= mem_numbers -> Lines ->
Strings[i].ToInt();
   i++;
}
while (i < count);
average = float(total)/count;
lbl_answer->Caption = "Average:  " +
String(average);
}
```

Copy the code for at least one of these loops, and compile and test the program.

Another common use of loops is to search a list for a particular item. The number of items to be looked at is not known in advance as the search can stop as soon as the item is found. A definite (for) loop is therefore not appropriate and so a while loop is usually used.

For this program, you will need a memo component to contain a list of names, an edit box to contain the name to be searched for, a button, and labels to explain the memo and edit boxes and to contain the search result. Try the following code for the OnClick event:

```
void __fastcall TForm1::btn_searchClick(TObject
*Sender)
{
String name;
int count, i=0;
bool found = false;
count = mem_numbers->Lines->Count;
name = edi_name->Text;
while (i<count && !found)
{
    if (name == mem_numbers->Lines->Strings[i])
       found = true;
    else
       i++;
}
if (found)
   lbl_answer -> Caption = "Name found in list";
else
   lbl_answer -> Caption = "Name not found";
}
```

Read through the code carefully, and ensure you understand it.

6.5 Exercises

Write programs which allow the user to enter a list of numbers and then determine:

1. how many times a given number occurs in the list;
2. how many numbers are greater than a given number;
3. the largest and smallest numbers in the list.

Chapter 7: More Solutions

Aim of this chapter: In this chapter you will create solutions to further problems. The solutions will build on the programming constructs you have already used, and introduce other useful and commonly used components.

In this chapter the following subjects will be explained:
Solutions to problems using the list and combo boxes;
Solutions to problems using check boxes and radio buttons.

7.1 Introduction with the exercise “Shopping list”

The basic task is that we want to create a shopping list of items to buy next time we visit the supermarket.
First, we need to decide what we shall be able to do to with the shopping list. Start by making your own list of the features you want from the program.
A suggested list follows. Does it agree with your requirements?

Add items to the list;
Delete items from the list;
Change items on the list;
Scrap the list and start again.

These are the basic requirements for many list applications. For simplicity, we can omit item 3, change items on the list. The same result can be achieved by deleting the incorrect item and then adding it again correctly.

A deluxe version of the list program would have extra features such as

sort the list alphabetically;
count the number of items in the list.

We will start with the basic features, and add the extra ones later.

7.1.1 Designing the form

The basic form will need 5 components:
3 buttons, for add, delete and clear;
1 edit box in which to type the new item to be added to the list;
1 list box to contain the list.
An item will be added to the list by typing its name in the edit box and then

clicking the Add button. An item will be deleted from the list by selecting (highlighting) it in the list box and clicking on the Delete button. Clicking on the Clear button will empty the list.

Add the components to the form, using the names suggested in the table below.

Component	Name	Caption/Text
Form	frm_Shopping	Shopping
Button	btn_Add	Add
Button	btn_Delete	Delete
Button	btn_Clear	Clear
Edit box	edi_item	
List box	lst_shopping	

Add labels to make the form clearer, e.g. add a label to explain what to type in the edit box.

If you examine the properties of the list box, you will find a property called Items. This is the property which we shall need to use most, as it contains the list of items. It has methods of its own, such as Add, Delete and Count, which we shall use in this program.

First, write the code for the Add button.
It will need to take the text from the edit box and add it to the list box:

```
lst_shopping -> Items -> Add(edi_item -> Text);
```

Now write the code for the Clear button. It will just use the Clear method of Items:

```
lst_shopping -> Clear();
```

Test the program and see if it works correctly so far. Nothing will happen when you click the Delete button, but the other two buttons should work correctly. Make a list of improvements that would make the program easier to use. You can come back to the list at the end and see if there are any improvements on your list that have still not been made.

Now program the Delete button. The delete method needs to be told which item in the list is to be deleted. It is “told” by passing the number, or index, of the item as a parameter. Passing parameters is one of the ways in which different functions and methods share information. The parameter is placed in the parentheses which follow the function or method name. If there are no parameters, the parentheses are still needed but are empty. Look back at the two buttons you have already programmed. The Clear method had no parameters, because everything had to be cleared, so it was written

```
Clear();
```

The Add method needed to be told what to add, so that was the parameter:

```
Add(edi_item -> Text);
```

So far, we have only looked at properties which are available at design time and are listed in the Object Inspector. Components may have other properties which are only available at run time, and so are not listed in the Object Inspector. The ItemIndex property of the list box is one of these. It contains the index of the currently selected (highlighted) item in a list box. If nothing is selected, it contains the value –1. Why is that? Well, remember that counting in C++ usually starts from 0, so 0 is the index of the first item in the list.
The code for this function is:

```
   int index;
   index = lst_shopping -> ItemIndex;
   lst_shopping -> Items -> Delete(index);
```

Compile and run the program, and test that it works correctly.

Improving the Solution

What was on your list of improvements?
One very easy feature to add, but which will look impressive, is to resort the list into alphabetical order every time an item is added.
In the Object Inspector, find the Sorted Property for the list box and set it to true.
Compile and run the program again. What happens if you add the same item twice? Does it matter? What could you do about it?

If you add an item to the list, what do you need to do before you add another item? Where is the cursor? Think about where you want the cursor to be after every action. The FocusControl method allows you to set the focus to whichever component you choose. In this program, the next action after pressing one of the buttons is likely to be to add something else to the list. So at the end of the code of each of the three Click vents, you could add:

```
            FocusControl(edi_item);
```

to send the cursor to the edit box.

In our original list of requirements, we also wanted to count the number of items in the list. This uses the Count property of Items.

```
numitems = lst_shopping -> Items -> Count;
```

You will need a label to display the number of items. Give it the name

```
                lbl_count
```

and the caption Number of items: 0

Add a line to btn_ClearClick to display “Number of items: 0” in the label. Add code to btn_AddClick and btn_DeleteClick to count the number of items and then display it in the label:

```
                   lbl_count -> Caption = "Number of
items: " +                   String(numitems);
```

(Hint – remember you will have to declare numitems as an int)

Compile, run and test your program again. What happens if you try to add an item before you have entered it in the edit box, or try to delete an item before you have highlighted it? The program will be more robust if you write code to trap these two events.
Use an if statement in btn_AddClick so that the item is only added if the edit box is not blank. Similarly, use an if statement in btn_DeleteClick to check that an item is highlighted before deleting. (Hint – what is the index if nothing is selected?)

You can check your final listing for the program with the code below:

```
void __fastcall
Tfrm_Shopping::btn_ClearClick(TObject *Sender)
{
   lst_shopping -> Clear();
   FocusControl(edi_item);
   lbl_count -> Caption = "Number of items: 0";
}
//-------------------------------------------------

void __fastcall
Tfrm_Shopping::btn_AddClick(TObject *Sender)
{
   int numitems;
   if (edi_item -> Text != "")
   {
      lst_shopping -> Items ->
 Add(edi_item -> Text);
      edi_item -> Text = "";
   }
   FocusControl(edi_item);
   numitems = lst_shopping -> Items -> Count;
   lbl_count -> Caption = "Number of items: "
+ String(numitems);
}
//-------------------------------------------------

void __fastcall
Tfrm_Shopping::btn_DeleteClick(TObject *Sender)
{
   int index, numitems;
   index = lst_shopping -> ItemIndex;
   if (index != -1)
   lst_shopping -> Items -> Delete(index);
   FocusControl(edi_item);
   numitems = lst_shopping -> Items -> Count;
   lbl_count -> Caption = "Number of items: "
 + String(numitems);
}
```

7.2 Further explanation with the exercise “The Restaurant Menu”

For this problem, we want to create a restaurant menu and invite the user to select a meal.
The first course, or starter, will always be a choice between soup of the day, fruit juice or melon.
The main course will be a choice between several dishes, and will vary from day to day.
Cheese and biscuits and/or coffee may also be selected.
Puddings will also be offered – but later!
How should this be programmed? We need to choose the most appropriate components.

The starter will be one of three options, and so radio buttons could be used here. Radio buttons are usually designed to work together, so that if one is checked the others are automatically unchecked. However, if you use three separate radio buttons, you will have to check each one until you find the one that has been checked. An even more useful component is the RadioGroup component, which is, as the name suggests, a component which groups radio buttons together. You can then check the whole group at once to find the index of the button that has been checked. Like the list box of the last example, the RadioGroup also has an ItemIndex property, and this has a value of –1 if nothing is selected.

The main course will be one of a number of options which will vary daily. We could use a list box for this, and populate it at design time. The program would need to be recompiled daily as the menu changed, as a new list would have to be typed in. An alternative would be to use a more specialised list box, the combo box.
Cheese and biscuits and coffee are optional extras which may be chosen independently of anything else. Check boxes are the appropriate component for these.
When the food has been selected, the user will press the “Order” button, and the choices will appear in a list box.

Start a new project, and place the components on the form, as described in the table below.

Component	Name	Caption/Text
Form	frm_Restaurant	Restaurant
RadioGroup	rgr_Starter	Starter
ComboBox	cmb_Main	

CheckBox	chb_Cheese	Cheese and Biscuits
CheckBox	chb_Coffee	Coffee
ListBox	lst_Order	
Button	btn_Order	Order

Start with the RadioGroup. Find its Items property in the Object Inspector, tab to the right column and press Ctrl+Enter. Alternatively, just click on the little box at the right. A String List Editor will open. Type in the three menu items (soup, fruit juice, melon) one per line. Click the OK button, and you will find that three labelled radio buttons have appeared on the form.

Now do the same with the ComboBox. This time, add five main menu items, one per line to Items. Do not make the descriptions too long, or they will only be partially visible in the box on the form. There are several different types of combo box. The simplest one allows the user to type in their own choice. We want to restrict the user to our menu only, so we need a DropDownList. You will find the various types of combo box as a drop down list in the Style property. Later you should experiment with them, but for now just choose the drop down list.

Nothing further needs to be done with the check boxes or the list box.

The code will be executed when the Order button is clicked. The choices made will be transferred to a list in the list box. To stop the user accidentally pressing the Order button twice, it can then be disabled.

The first item to copy to the list is the starter. Check that an item has been chosen (the ItemIndex is not –1), and then add that item to the list box. Remember that you originally typed the list of strings into the Items property. So now you have to retrieve the correct string from Items. This will be String[index] where index is the number or ItemIndex of the item selected.
So the code will be:

```
   int index;
   index = rgr_Starter -> ItemIndex;
   if (index >= 0)
   {
      lst_Order -> Items -> Add(rgr_Starter ->
  Items -> Strings[index]);
   }
```

Next, you need to copy the choice of main course. The code will look very similar. Note that you have finished using the variable index for the radio group, so you can recycle it and use it again for the combo box. There is no need to declare another variable to hold the index of the combo box item chosen.

```
   index = cmb_Main -> ItemIndex;
   if (index >= 0)
   {
      lst_Order -> Items -> Add(cmb_Main ->
  Items -> Strings[index]);
   }
```

If the customer has chosen cheese and biscuits, that needs to be added to the list:

```
    if (chk_Cheese -> Checked)
    {
      lst_Order -> Items -> Add("Biscuits and
  Cheese");
```

The Checked property of a check box can take two values – true or false. If the box has been checked, this property will be true and so the condition in parentheses above will be true and the item will be added to the list.

Add similar code for coffee.

Finally, we want to disable the button. To do that, you need to set its Enabled property to false:

```
           btn_Order -> Enabled = false;
```

Now compile, run and test the program. If you have any difficulty with debugging, the full code for the Click event is given below:

```
void __fastcall
Tfrm_Restaurant::btn_OrderClick(TObject *Sender)
{
   int index;
   index = rgr_Starter -> ItemIndex;
   if (index >= 0)
   {
      lst_Order -> Items -> Add(rgr_Starter ->
  Items -> Strings[index]);
```

```
   }
   index = cmb_Main -> ItemIndex;
   if (index >= 0)
   {
      lst_Order -> Items -> Add(cmb_Main ->
  Items -> Strings[index]);
    }
    if (chk_Cheese -> Checked)
    {
      lst_Order -> Items -> Add("Biscuits and
  Cheese");
    }
    if (chk_Coffee -> Checked)
    {
      lst_Order -> Items -> Add("Coffee");
    }
    btn_Order -> Enabled = false;
}
```

7.3 Exercises

What about some pudding? Decide what you are going to offer, and devise a suitable component for it. Add the pudding choice to the list box.

Experiment with the different types of combo box.

Offer a choice of drinks after the meal or some wine or a soft drink with the meal.

Extra vegetables? A side salad? Use your imagination, and experiment.

Chapter 8: Dissemination of a programme with the InstallShield.

Aim of this chapter: In this chapter you'll learn how to prepare with the "Install Shield- function" of Borland C++ Builder your developed programme for dissemination like any other usual commercial programme.

8.1 Introduction

The next step after planning the project, writing, debugging and compiling a BCB is the dissemination of the final product.
As you have experimented so far, through BCB you can create executable files (**.exe**), that will run on any computer; this happens mainly with simple products.
However a non professional programmer often encounters complex applications needing standardised setup procedures. In other words you may need to install configuration files (**.INI**), registry files **(.reg),** library files (**.dll**) etc; the final user will decide in which folder he prefers to install the programme, the kind of installation (typical, minimal or customised), whether to install the program in the Start menu , whether to accept the terms of the user's license etc.

This is why the professional version of BCB includes **InstallShield Express - Borland Limited Edition**. Installshield Express is not automatically installed with BCB, and you should check whether you need to install it now. It is a short version, specific for BCB programmers, of a software package dealing with the distribution of the applications; though this version of **InstallShield** it is possible to arrange the main phases of a project setup. Once more, you are dealing with a project: when you select **File\New**, you are immediately asked to specify the path and the name of the project to be created.
Tip: we suggest you create a new folder for this.

Press **Ok** and you will see a window similar to **My Computer**: on the left a tree menu presents the whole series of steps leading to the creation of a setup, on the right the elements of each step, further right the parameters related to the selected element and next a help window. The number and the position of the frames and the setting method (edit boxes , selection boxes etc) varies according to each element.
Navigation through the frames is similar to the navigation on **My Computer**. This means following step by step the options available and implementing the setup program according to the dissemination needs of the BCB application.

Some frames may be partially or totally unaccessible to screen readers.
Next we will examine the steps leading to the creation of an installation project.

8.2 Organize Your Setup.

Your first step in creating your setup is to outline your application and your company information. The views in this step let you give basic information about your project, define the logical groupings that will eventually hold all of your application data, and specify which options your user will have when installing your application.
The groundwork you define in these early steps will stick with you throughout the rest of your setup creation process. Therefore, it is important to take the time to properly lay a solid base for your setup that you can build upon as your setup matures.

8.2.1 General information view

The General Information view is where you set global project properties such as the application name and support information. Most of the properties in this view are optional. Your setup will work properly if you accept the defaults or do not provide values for the optional properties. However, the following properties are required: Subject, Product Name, Publisher, Product Code, Upgrade Code, and Destination Folder.

8.2.2 Features.

Features are the building blocks of your setup. They represent a distinct piece of your application—such as program files, help files, or clip art—to your end users. components You can create features and up to 15 levels of subfeatures in the Features view.

8.2.3 Setup Types.

Setup Types offer different configurations of your applications to your customers. These configurations may be useful if you distribute large features that are not required in order for the application to run.

8.2.4 Upgrade Paths.

NOTE: This feature is supported only in the full edition of InstallShield Express.

8.3 Specify Application Data.

Now that your application has been outlined, you'll need to fill in the blanks with your application data. InstallShield has several views dedicated to this purpose. You can add your files directly to predefined destination folders or subfolders of those destinations and then alter the features with which they are associated. To save time, you can add merge modules—prepackaged groups of commonly used application files—to your setup project.

8.3.1 Files.

In the Files view, you can select your application files from their location on your system and specify the location where you would like them installed on the end user's system. The interface is split into two sections: the top is a display of your system, the bottom of the display is several predefined target folders.
This display operates just like Windows Explorer, so you can drag and drop files from the top to the bottom. The files that you drop will be assigned to the feature listed in the drop-down list shown above this display. You can change the feature that you drop files into at any time by selecting a different feature from this drop-down box.

8.3.2 Files And Features.

The Files and Features view gives you a chance to modify your file and feature associations. For example, suppose you moved all of your application files into destination folders without changing the feature. In this view, you can drag files from one feature and drop them on another without affecting their destination folder. You can also copy files from one feature and paste them in another, if you need a certain file to be installed with both features. That file will retain all of its properties within InstallShield. See Associating Files with Features for more information.

8.3.3 Objects/Merge Modules.

In the Objects/Merge Modules view, you can add common pieces of functionality, known as merge modules, to your setup. This functionality is helpful if your application requires commonly used file libraries, such as an MFC library. This way, you won't need to collect and manually add all of these files every time you package an application that needs them; you can just select the right merge module, and the files will all be installed to the correct folder automatically.
In earlier versions of InstallShield, objects provided a similar function as merge modules. Many companies, Microsoft included, have created merge

modules for commonly needed functionality, such as the Visual Basic run-time files. These modules provide you a nearly fail safe method of incorporating another company's technology into your setup.

8.3.4 Dependencies.

NOTE: This feature is supported only in the full edition of InstallShield Express.

8.4 Configure The Target System.

Applications often have shortcuts, registry entries and INI files associated with them. In this step, you can configure these advanced application components simply and quickly. In earlier versions of InstallShield, configuring the target system in this way was not always directly supported. Now you can create shortcuts in multiple locations on the target machine, edit registry entries, and INI files, all within the IDE.

8.4.1 Shortcuts/Folders.

The Shortcuts/Folders view gives you the means to place shortcuts to your application's files on the target system. There are already several default program folders for your convenience. These include the Desktop, the Programs menu, and the Send To menu. You may add subfolders and shortcuts to these defaults.
In the Shortcuts/Folders view you can also specify an icon file for your shortcut, pass command-line parameters to the application your shortcut points to, and the hot key combination you would like to use. See Shortcuts for more information.

8.4.2 Registry.

The Registry view lets you arrange the changes you would like to make to the target system's registry. If you have all the registry entries organized on your machine as you would like them to end up on the target machine, you can drag-and-drop them much in the same way as you did with files in the Files view. If you prefer to author your registry entries in REG files, you can import those files in this view. You can always add your entries manually, as well as edit any entries you have added using one of the above methods.

8.4.3 ODBC Resources.

The ODBC Resources view allows you to include in your setup any ODBC

driver or translator present on your system. To add a driver or translator:
find the resource you want to include in the list shown;
select the resource and the feature with which you would like this ODBC file to be installed;
below the list of ODBC resources are the properties for this file, as configured on your system. You can change any value and add new properties to this list.
In addition to the ODBC Resources view, you can add ODBC resources to your setup in both the Setup Design and Components views. In these views, ODBC resources are handled under a component's advanced settings.

8.4.4 File Extension.

File extensions allow you to link a certain type of file to your application. When that file is double-clicked, your application launches and open that file. When you open a text (.txt) file, you are actually sending a message to the operating system, telling it to launch Notepad so you can view the contents of that text file. If you would like to provide similar functionality for your application and its files, you can create a file extension association.

8.4.5 Environment Variables.

NOTE: This feature is supported only in the full edition of InstallShield Express.

8.5 Customize The Setup Appearance.

Once you've finished organizing all of your application's data, you can focus on the delivery interface for your setup. What your customer sees during an installation has a large impact on how they regard your product and your company. Often, your setup is the first experience your users have with your product. If it doesn't look professional, or it is difficult to use, their first impression may be a bad one.
This step gives you the opportunity to customize all of the visual aspects of your setup in order to create an appealing installation experience.

8.5.1 Dialogs.

The Dialogs view provides you the chance to determine which of the InstallShield standard end-user dialogs your customer will encounter during an installation. Some of these dialog boxes are required for all setups and cannot be removed. These will have a checkbox with a grayed-out background. The others you can select or deselect.

For each dialog box you include in your setup, you can customize the attributes it has using the property sheet at the right. These properties include the banner that is displayed at the top of every dialog, allowing you to put your company's branding on all of your dialogs. Below the list of dialogs is a sample image of what that dialog will look like to your customer. Note that this image will not reflect the values you enter into that dialog's properties.

8.5.2 Billboards.

NOTE: This feature is supported only in the full edition of InstallShield Express.

8.5.3 Text And Messages.

NOTE: This feature is supported only in the full edition of InstallShield Express.

8.6 Define Setup Requirements And Action.

Your application and its setup may have special requirements. With that in mind, this step lets you set the requirements that the target system must meet and add additional functionality through custom actions if the Windows Installer service does not inherently meet your setup's needs.

8.6.1 Requirements.

The target system needs to be able to provide the processing power, video capabilities, and memory to ensure that your application runs smoothly and properly. So, in the Requirements view, you can specify which capabilities the target system must meet before your product will be installed. If the target system does not meet the requirements that you set here, the user receives an error message and the installation exits.

8.6.2 Custom Actions.

NOTE: This feature is supported only in the full edition of InstallShield Express.

8.6.3 Support Files.

NOTE: This feature is supported only in the full edition of InstallShield Express.

8.7 Prepare For Release.

Finally, you must prepare your setup for release. You'll need to build it into a working installation file, test that installation, and distribute your setup to your target media. This step lets you take care of all three of these tasks without ever leaving the IDE.

8.7.1 Build Your Release.

Building your setup is the culmination of all the work that you have put into your project. This includes all the coding and planning that was done before you ever thought about building a setup. The Build Your Release view, as the name implies, allows you to compile the information you've entered into your setup project into a functional Windows Installer setup. You can select one of a number of standard media formats, such as CD-ROM or various DVD-ROM types, compile your setup into a single disk image, or create a custom size for your disks.
Once you have selected and configured your media type, you can then build your setup. Afterwards, InstallShield provides you with feedback on your build in the form of build logs and reports. These are unique to each build and will not be overwritten by subsequent builds of that media type.

8.7.2 Test Your Release.

In the Test Your Release view, you can test your newly built setup to make sure everything goes as planned. You have two options here: run or test your setup. If you run your setup, it will perform exactly as if you had double-clicked the setup executable. All files will be installed, all dialogs will display, and all custom actions will execute. If you test your setup, no changes will be made to the target system; only the user interface and any custom actions you have included will execute. This second option is useful if you want to test how your setup looks without taking up further disk space on your system.

8.7.3 Distribute Your Release.

Once you've built your setup project and made sure everything works as you want it to, you can move your release to the target media in the Distribute view. You have the option of copying it directly to the specified media type or a staging directory or uploading it to an FTP site so it can be accessed from a remote location.

Chapter 9: Project Management

Aim of this chapter:The student should get a simple introduction to managing bigger programming projects.

In this chapter, we will break up a programming project into different tasks. These are formulating the problem, formulating the tasks, analysing the tasks, coding and testing. In addition, we will talk about the documentation of a programming project.

9.1 Introduction

There are several possible ways of solving a particular problem: the person who recognises teh problem solves it alone. This only succeeds however, if that person has all the necessary skills, the time and particularly the motivation to do it. When the problem reaches a certain size and complexity, the person will realise that they need help to solve it. So one, two or several employees are made available and they attempt to solve the problem together. It should be noted that the difficulty of achieving a common solution increases disproportionately to the group size. For example, the cost of communication withtin large grops should not be neglected. Without an organised structure.the achievement of a problem solution can easily run into difficulties. To prevent this, you can use project management tools developed in management science.

Here, by a „project“ we mean a fairly complex and unique problem (such asa research project or project in a crisis situation) which is to be solved within a set time and with given resources.

9.2 The Phases of a Project

The most important steps in the course of the project are in fact those of planning and preparation, because only when enough time and trouble are put into these steps, is an efficient project realization possible. Basically the following stages can be identified.

Problem
Formulating the problem
Method
Rough analysis
Specification of the requirements
    Detailed analysis
Specification

Programming

Programme

Testing

Arranging the organisation

Adjusted programme

Arranging the organisation

Assumption of data

Parallel operation

Computer application

The first task of the entire project must be to prepare a framework for the processes within the project. As follows:

Description of the project

Job specification: quantitative and qualitative determination of the wanted result

Job definition: Summary representation of the emphases and the means to be used

Procedure description: Kind, extent and result of the necessary operation steps

Dates (at the best with information of intermediate appointments)

Budget

Assignment of the project to specific persons

9.3 Information and Documentation

One of the most important aids to the planning and checking of a project is the information and documentation system. The checking is not, however, intended to stimulate employees to the job, but it is supposed to recognize possible troubles or delays in time so that countermeasures can be taken. For this purpose it is necessary to make orders, outcomes and partial tasks accessible to all participants; at the best in hard-copy form. This happens through the preparation of a software specification.

9.4 The software specification

The software specification contains the hard-copy fixing the points which the program or system to be created have to fulfill. It represents a list of all performance data and assumptions taken as a basis due to which at project end the success of the project can be found. In addition responsibility, dates, costs and all remaining restrictions are contained. Every necessary modification of the software specification has to be documented and approved by the customer.

For example among other things a delivery specification could contain the following points

Elements of a software specification:

1. General target description
2. Actual description of the system
   Construction and function
   Technical quality (Software and Hardware)
   Experiences
3. Task of the successor system
   Workflow description
   Input/output of data
   Data processing
4. Files
   File type
   Input/output standard
   Programming
   Editor
   Compiler/Linker
   Debugging aid
   Discussion of different solutions
   Limiting conditions
   Solution A: Advantages and disadvantages
   Solution B: Advantages and disadvantages
5. Functions and specifications
   Environment
   Hardware and Software
6. Ordering-extent and delivery
   Amount
   Dates
7. Handling of the project
   Date
   Place
8. Service proofs
9. Range of services
   System responsibility
   Scope of delivery and conditions
   Documentation
   Training
10. Valuation
11. Test/acceptance condition

9.5 Software documentation

A special problem during the preparation of software is their documentation. One can put forward the point of view that the source text of every program is really already documented and further information is superfluous. Everyone however that has ever attempted a program written by someone else knows how difficult it can be.

The larger the project is, the more likely it will be that complex software will be used. Therefore several programmers will normally participate in the preparation. With large programs or software packages software documentation is not only necessary at the end of the project, but also during the preparation, in order to be able to link the all the program without problems. In order each programmer knows precisely what he/she has to do, it is necessary to define common standards for the documentation before the allocation of the programming job.
Basically, the following tasks should to be observed in writing the software documentation, possibly on specific forms:

Elements of software documentation
Project name
Programme name/module number
Purpose of the program
Preparation / change (declare reference programme)
Generator
Date
necessary hardware and systems software
Module structures
Module interfaces (Entries, exits and/or entries /exits)
Algorithms and procedures to be used
Restrictions during the utilization as well as other indications of „special features“
Test logs

The sum of the documentation of the individual modules shows the entire documentation on completion of the software development. If this procedure is followed, it is possible to ensure that all aspects of the project are completed in time and to the required standard. this is particularly important as time is often limited at the end of a project. Well planned documentation also makes maintenance and upgrades easier to
carry out.

Chapter 10: Pointer

Aim of this chapter: The student will get an introduction to the concepts of pointers, how they are defined and how they can be used.

In this chapter we will learn to get direct access to the computers RAM through a pointer. We will learn, how to point to a specific part at the computers memory and how to read and set data there.

10.1 Definition

Pointers are variables which contain the address (reference) of another variable ("point on another variable").
By the instruction

```
char *ptr;
```

a pointer is declared which points at a variable of the type char.

On pointer the following operations are allowed:
a) assignment
b) dereferencing
c) equality and disparity

Although a pointer variable always contains an address, the data type must be declared at which it points. With it is guaranteed that the pointer points, e.g., after an incrementing, really, at the next element.

Annotation:
In the pointer arithmetic it is not counted in byte, but in units. So ptr+1 points always at the next element, but not on the following address.

Examples:

```
// a pointer of type integer
int *int_Zahl;
// a pointer of type edit box
TEdit *edi_myEditBox;
```

10.2 Referencing

The address of a memory cell is called reference. The Referencing -or

addressing operator "&" yields the address of a variable.
Example:

```
reference (&v);
```

Hands over the address of the memory cell with the name `v` to the function **reference**.

10.3 Dereferencing

Dereferencing is called the access to the variable value by the address. By Dereferencing of a variable one gets the content of the memory cell(s). The interpretation of the memory cell(s) depends on the data type.

The Dereferencing operator "*" yields the content of an address.

Example:

```
name = *ptr;
```

hands over the content of the memory cell(s) at which `ptr` points to a variable name.

10.4 NULL- and Void-Pointer

A `NULL` pointer is a pointer which practically points to nowhere. At the moment it does not address valid data.

Example:

```
float *fp = NULL;
```

Global pointer variables are initialised with `NULL` by default, local variables contain an undefined value.

The **Void** pointer can address an arbitrary memory cell. It is not bound to any type. `Void` pointers are tools to determine data whose type is not known for the moment.

Example:

```
void *unknown;
```

10.5 Use of Pointer

10.5.1 Addresses of Variables

In C/C ++ it is possible to allow pointer to point at arbitrary variables.

Example 1: Show, how a pointer works

1. Generate a “New Application” in C++ Builder 5. Put two Edit-controls, two Label-controls (to access the edit controls) and four Button-controls on the form.
2. The captions of the labels should be “&Temp-Variable :” and “&Pointer :”. The captions of the Buttons should be “&Fill Temp-Variable”, “Show T&emp Variable Value”, “Show P&ointer Value”, and “&Clear All”.

The example-code (on the UNIT1):

3. In the declaration-part of the form define following variables (beyond Tfrm_Pointer *frm_Pointer, where frm_Pointer is the name of the form – a pointer too as you easily recognize, that we will not use in this example):

```
//the pointer
String *str_Ptr;
//a variable, where the pointer should point
String str_Temp;
```

4. In the only existing function of the Form (here: Tfrm_Pointer:Tfrm_Pointer) put following code:

```
__fastcall Tfrm_Pointer::Tfrm_Pointer(TComponent*
Owner)
: TForm(Owner)
{
//define a reference from pointer to the
//address of the variable
str_Ptr = &str_Temp;
//fill the address, where the pointer points
//to with a value (dereferencing the pointer)
*str_Ptr = "Hello World";
}
```

5. Assign the Event “OnClick” of the button “Fill Temp-Variable” with following code (in this case “btn_FillTempVariable” is the name of the button, “edi_TempVariable” the name of the edit-box):

```
void __fastcall
Tfrm_Pointer::btn_FillTempVariableClick(TObject
*Sender)
{
//fill the variable with the text typed in the
//first edit-box
str_Temp = edi_TempVariable->Text;
}
```

6. Assign the Event “OnClick” of the button “Show Temp Variable Value” with following code (in this case “btn_ShowTempValue” is the name of the button):

```
void __fastcall
Tfrm_Pointer::btn_ShowTempValueClick(TObject
*Sender)
{
//put the value of the variable back to the
//first edit-box
edi_TempVariable->Text = str_Temp;
}
```

7. Assign the Event “OnClick” of the button “Show Pointer Value” with following code (in this case “btn_ShowPointerValue” is the name of the button, “edi_Pointer” is the name of the second edit-box):

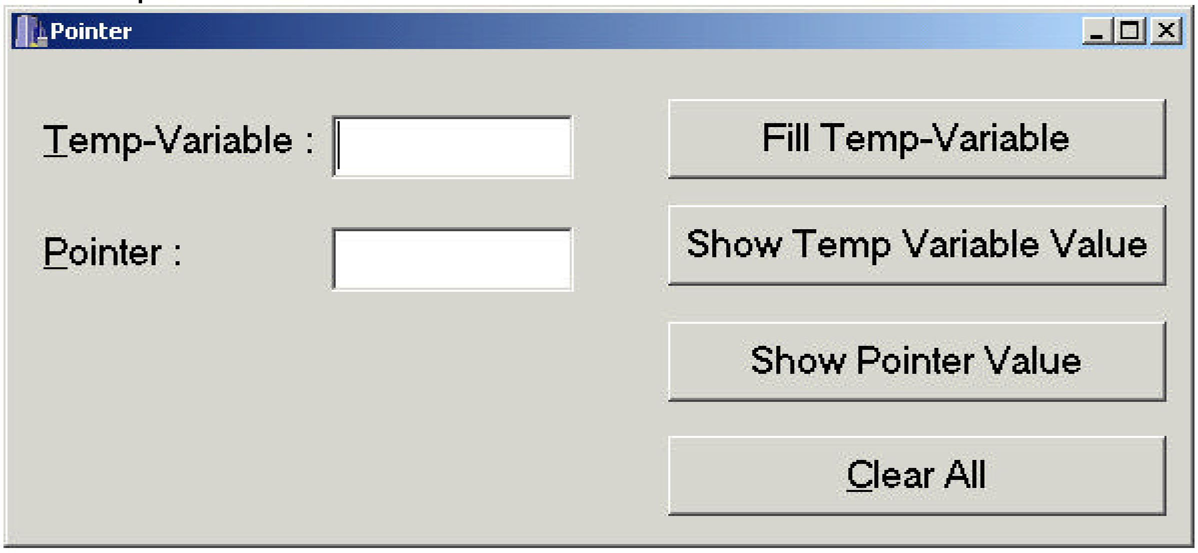
```
void __fastcall
Tfrm_Pointer::btn_ShowPointerValueClick(TObject
*Sender)
{
//fill the second edit with the value of
//the address where the pointer points to
//(dereferencing of the pointer)
edi_Pointer->Text = *str_Ptr;
}
```

8. Assign the Event “OnClick” of the button “Clear All” with following (in this case “btn_Clear” is the name of the button):

```
void __fastcall
Tfrm_Pointer::btn_ClearClick(TObject *Sender)
{
//empty both edit boxes
edi_Pointer->Clear();
edi_TempVariable->Clear();
}
```

Each time you click the buttons “Show Temp Variable Value” and “Show Pointer Value” one after the other they contain the same values, shown in the edit box. You can type a new value the first edit-box, press the “Fill Temp-Variable”-button and then the “Clear All”-button to empty the edit-boxes. Both previous buttons will show the same value again.

Here is a picture of how the form should look like:



Example 2: Using two pointers
You can advance the example and add following code:

1. At point 3. off above example add the code line:

```
//define a second pointer
String *str_Temp2;
```

2. At point 4. off above example add the code line:

```
//the second pointer gets the address of the
//temp-variable from the first pointer
str_Temp2 = &str_Temp;
```

3. At point 6. off above example change the only code line to:

```
edi_TempVariable->Text = str_Temp2;
```

This works as well as the first example.

The Examples above demonstrate the referencing and dereferencing of

variables. The example has no practical importance, however, shows the possibility of the "bending" of pointers at arbitrary memory areas.
Interesting data structures can be produced by changing, "bending" or reassigning of a pointer: linked lists, trees, batches, etc. Pointer are connections which reasonable relates separate variables to memory areas.

Example 3: Using pointer to toggle between components
This examples shows the power of pointers in a more real situation.

1. Generate a “New Application” in C++ Builder 5. Put two ListBox-controls, one Edit-control, two Button-controls and one RadioGroup-control on the form.
2. Name the controls as follows:
   Listbox1: lst_Left
   Listbox2: lst_Right
   Edit1: edi_New
   Button1: btn_Add, Caption “Add”
   Button2: btn_Clear, Caption “Clear”
   RadioGroup1: rdg_SwitchBox, Caption “Switch Box”
3. Go to the property “Items” of the RadioGroup “rdg_SwitchBox” and press String+Enter. Add the entry “& Left Listbox”, press enter for a new line and add the entry “&Right Listbox”. Close the “String-List Editor”-window.
   The RadioGroup will show two options now.

The example-code (on the UNIT1):

4. In the declaration-part of the form define following variables:

```
//the pointer of Listbox type
TListBox *lst_Temp;
```

5. In the only existing function add following code:

```
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
//initiliaze (=set) the temp-pointer with the
address
//of the first listbox
lst_Temp = lst_Left;
}
```

6. Assign the Event “OnClick” of the button “Add” with following code:

```
void __fastcall TForm1::btn_AddClick(TObject
*Sender)
{
//only if the edit-box is not empty
if (edi_New->Text != "")
{
//add a new item to the listbox and use
//the value of the edit-box
lst_Temp->Items->Add(edi_New->Text);
}
}
```

7. Assign the Event “OnClick” of the button “Clear” with following code:

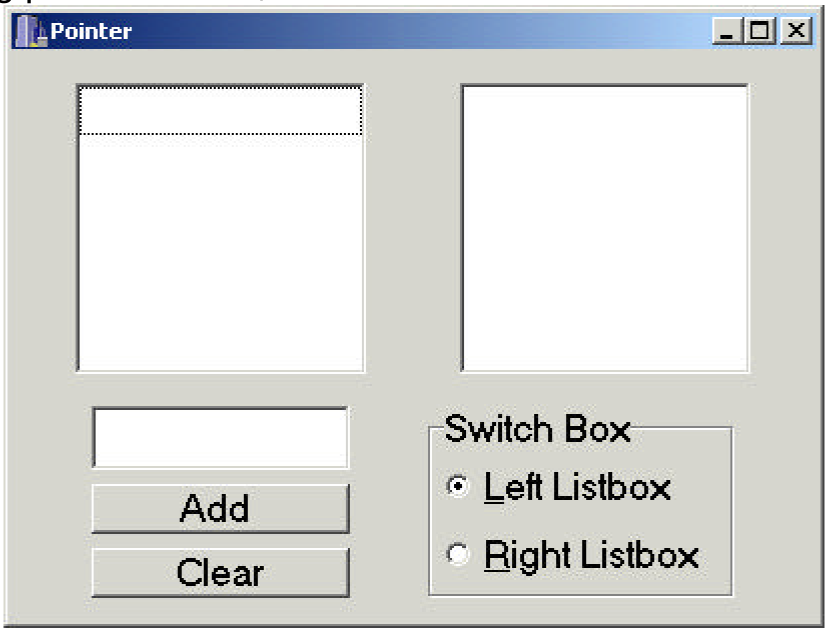
```
void __fastcall TForm1::btn_AddClick(TObject
*Sender)
{
//empty the listbox
lst_Temp->Clear();
}
```

8. Assign the Event “OnClick” of the RadioGroup “rdg_SwitchBox” with following code. The first option “Left Listbox” has the ItemIndex 0, the second option “Right Listbox” the ItemIndex 2:

```
void __fastcall
TForm1::rdb_SwitchBoxClick(TObject *Sender)
{
//if the option “Left Listbox” is activated
if( rdb_SwitchBox->ItemIndex == 0 )
{
//the pointer gets the address of the left
listbox
lst_Temp = lst_Left;
}
//if the option “Right Listbox” is activated
else
{
//the pointer gets the address of the right
listbox
lst_Temp = lst_Right;
}
}
```

In this example you can choose with witch list-box you want to work only by switching the options in the radio-group. The pointer “lst_Temp2 holds the address eider of the left or the right list-box. All commands to work with a list-box can therefore be performed on the pointer lst_Temp.

The following picture shows, how the form should look-like:



10.5.2 Pointers and Vectors

The declaration

```
int a[10];
```

defines a vector from 10 successive integer values that can be accessed by

```
a[0] to a[9]
```

With pointers also an Array can be accessed. So with

```
int *pa=&a[0]
```

the pointer pa is bent on the first element of the field a. By a simple pointer arithmetic the whole Array can be accessed:

```
a++;
*pa=120;
```

assigns a[1] the value 120.

Just could be written

```
*(pa+1)=120;
```

Annotation:

The inquiry of the address of the first element of an array with the instruction

```
int *pa = &a[0];
```

is easy readable, but unusual in C/C ++. By indicating the field name without index the address of the field is automatically generated. Therefore the following instruction will do:

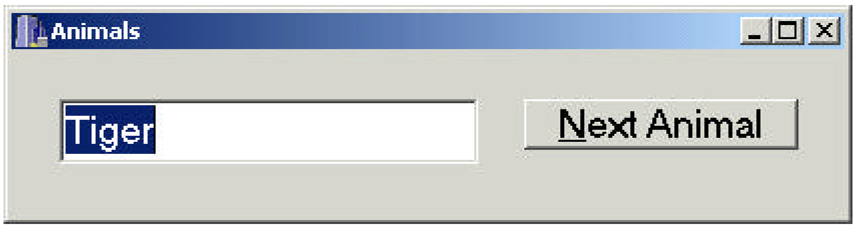
```
int *pa = a;
```

Example 4: Using a pointer as an array

1. Generate a “New Application” in C++ Builder 5. Put one Edit-control and one Button-control on the form.
2. Name the controls as follows:
   Edit1: edi_Animal
   Button1: btn_NextAnimal, Caption “Next Animal”
   Caption of Form1: “Animals”
   The example-code (on the UNIT1) starting with the pointer to the form:

```
//the code in the declaration part of the unit
TForm1 *Form1;
//an array for the names of five animals
String astr_Animals[5];
//an pointer assigned to the first element of the
array
String *str_ptr = astr_Animals;
//a control variable
int i = 0;
//---------------------------------------------
----------
__fastcall TForm1::TForm1(TComponent* Owner):
TForm(Owner)
{
//fill the fields of the array with the names
//of animals starting with field number 0
astr_Animals[0] = "Tiger";
astr_Animals[1] = "Lion";
astr_Animals[2] = "Elefant";
astr_Animals[3] = "Zebra";
astr_Animals[4] = "Hippo";
//show the first animal in the edit-box
edi_Animal->Text = *str_ptr;
}
//---------------------------------------------------
----------
void __fastcall
TForm1::btn_NextAnimalClick(TObject *Sender)
{
//increment the control variable by one
i++;
//if the control variable reaches 5 then set it
//back to 0;
if (i == 5) i = 0;
//let the pointer point to the address of the
animal
//at the location i in the array an show it in
//the edit-box
edi_Animal->Text = *(str_ptr+i);
}
//---------------------------------------------------
----------
```

The following picture shows, how the form should look-like:



10.5.3 Exercises

1. An exercise similar to example 3: create a form with three combo-boxes, one edit-box and a radio-group. Create buttons to add an item to a combo-box, to delete the last or the first item and to clear a combo-box. Use the radio-group to toggle between the three combo-boxes.
2. Create an application with three forms. The first forms contains an button and a radio-group. The second form contains a list-box filled with city-names. The third forms contains a list-box filled with postal codes. The button opens a form. The radio-group decides whether the one with city-names, or that with postal-codes.
3. Create a gamble. Use three edit-boxes and a button. On click on the button the gamble should start and create a random number in each of the edit-boxes, e.g. between 0 and 9. Write only one function to create the numbers and use a pointer to fill the different edit-boxes.

Annotation: To create random–numbers use the functions randomize() and random();

10.6 Enumerators

Aim of this subchapter:The student should learn how to name integer-values by identifiers.

In this chapter we will do first the definition of an enumeration datetype. Then we will discuss some examples.

10.6.1 Definition

With enumerators you can assign identifiers to numbers. You use the keyword “enum” to define a enumerator.
For instance:

```
enum TagDays {Mon, Tue, Wen, Thu, Fre, Sat, Sun} Day;
```

defines the enum „Days” and assingns each day an integer, starting with Mon = 0 to Sun = 6.
Also you can define a enum in the following way:

```
enum TagCoins{penny=1, twopence, nickel=penny+4, dime=10,
quarter=nickel*nickel} Coins;
```

That assigns penny =1, twopence = 2, nickel = 5, dime = 10, quarter = 25. The identifiers are called enum-constants.

10.6.2 Declaration

“Day” is a variable declare the same time as the enum is declared, which is optional.
You can declare a new Variable “today” of the type TagDays by

```
enum TagDays today;
```

10.6.3 Assignment

You can assign the special enumerator only with constants of itself. You cannot assign it with integer values.
E.g.:

```
Day=Thu;  // right
Day=1;    // wrong, even thow Thu has the value 1
```

But you can assign each integer variable an enum-constant, e.g:

int today = Thu;

Example 5: Using an enum to specify elements

1. Create a “New Application”
2. Define the form exactly as in example 4:
3. Change the code of unit1 to

```
//the code in the declaration part of the unit
TForm1 *Form1;
//an array for the names of five animals
String astr_Animals[5];
//define a enum with a value for each animal,
//where etiger = 0 and ehippo = 4
enum TagAnimals {etiger, elion, eelefant, ezebra,
ehippo} Animals;
//a control variable
int i = 0;
//-------------------------------------------------
----------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
  //fill the fields of the array with the names
  //of animals using the enum Animals
  astr_Animals[etiger] = "Tiger";
  astr_Animals[elion] = "Lion";
  astr_Animals[eelefant] = "Elefant";
  astr_Animals[ezebra] = "Zebra";
  astr_Animals[ehippo] = "Hippo";
  //assign the edit-box the first animal
  //using the enum-constant etiger
  edi_Animal->Text = astr_Animals[etiger];
}
//-------------------------------------------------
----------
void __fastcall
TForm1::btn_NextAnimalClick(TObject *Sender)
{
  //increment the control variable by one
  i++;
  //if the control variable reaches 5 then set it
  //back to 0;
  if (i == 5) i = 0;
  //show the animal-name at the field i
  edi_Animal->Text = astr_Animals[i];
}
//-------------------------------------------------
----------
```

10.7 Structures

Aim of this subchapter: The students should learn what structures are, how they are defined and for what they are used.

In this chapter we will do the definition of a structure. Then we will show how, they are used and filled with values.

10.7.1 Definition

A structure is used to store values of different type, that are not independent form another. The typical example for an struct is a record used in a database, e.g. an address. You use the keyword struct to define a structure.
You can define a structure for an address by:

```
        struct TagAddress
        {
           String Name;
           int Age;
           float Size;
         } Address;
```

10.7.2 Declaration

With the above definition of the structure also a variable “Address” is declared, which is optional.
You can declare new variables of the type TagAddress with

```
struct TagAddress MyAddress;
```

You also can declare pointers and arrays of type struct.
E.g.

```
struct TagAddress *ptr_Address; //a pointer of
type TagAddress
struct TagAddress FamilyAddresses[5];    //an
array with 5 elements of type TagAddress
```

10.7.3 Declaration with “typedef”

With the keyword “typedef” you are able to rename a structure.
E.g.

```
typedef struct TagAdress OLDADRESS;
```

But you can also use it to define a new structure:

```
//typedefiniton
```

```
typedef struct {…..} NEWADRESS;
//declare variables
NEWADRESS Friends[10], GirlFriend, *BoyFriend;
```

Annotation:
You can use “typedef” also for enumerators in the same way.

10.7.4 Assignment to elements

You can use the “.”(dot)-Operator or the “->”(arrow)-operator to assign values to the elements of a structure or read the values.
If you want to store values to the variable “MyAdress” of type TagAdress and using the dot-operator you can do as follows:

```
//using the dot-operator
struct TagAddress MyAddress;
MyAddress.Name = “Jane Somebody”;
MyAddress.Age = 25;
MyAddress.Size = 1.7;
```

If you want to use the arrow-operator if you define a pointer first. Then you assign the pointer a structure of the same type with the keyword “new”. After you used the Pointer you must destroy it with the keyword “delete”.

```
//declare a pointer of type struct TagAdress,
(only the memory is allocated)
struct TagAddress *MyAddress;
//assign a “real” variable of type struct
TagAdress
MyAdress = new (struct TagAddress);
//assign values to the elememts
MyAddress->Name = “Jane Somebody”;
MyAddress->Age = 25;
MyAddress->Size = 1.7;
//destroy the pointer
delete MyAdress;
```

Example 6: Using a Struct to store Records

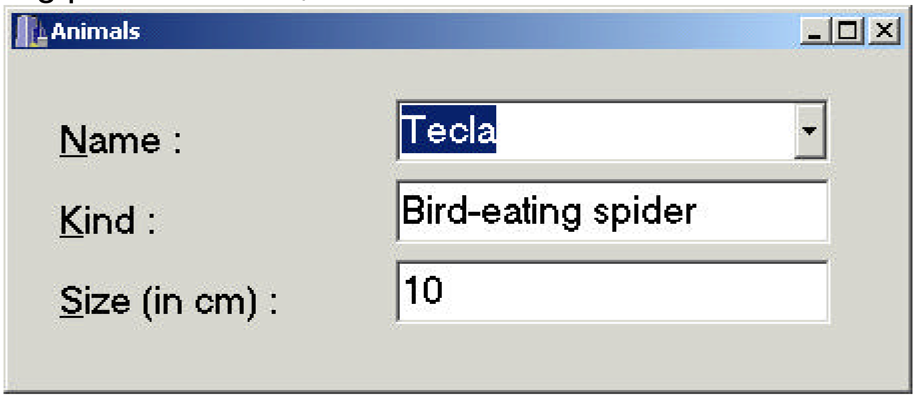
1. Create a “New Application”.
2. Put a ComboBox, two Edits and three Labels on the form.
3. Arrange the components so that the labels are left of the combobox and the edits.
4. Name the components as follows:
   ComboBox1: cmb_AnimalName
   Edit1: edi_AnimalClass

Edit2: edi_AnimalSize
Leave the Labels as they are.

5. Set the Caption of the Labels:
   Label1: Caption "&Name"
   Label2: Caption "&Kind"
   Label3: Caption "&Size (in cm)"
6. Use the Property FocusControl to connect the Labels with the corresponding component.
7. Write the following code in the unit1-window:

```
TForm1 *Form1;
//define a struct Animals
struct Animals
{
String Name;
String Class;
int Size;          // in cm

};
//declare a array of type Animals
struct Animals MyAnimals[4];
//-------------------------------------------------
----------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
//1. animal is a dog
MyAnimals[0].Name = "Pluto";
MyAnimals[0].Class = "Dog";
MyAnimals[0].Size = 70;
//2. animal is a cat
MyAnimals[1].Name = "Carlo";
MyAnimals[1].Class = "Cat";
MyAnimals[1].Size = 30;
//3. animal is a bird
MyAnimals[2].Name = "Twiggy";
MyAnimals[2].Class = "Canary";
MyAnimals[2].Size = 7;
//4. animal is a spider
MyAnimals[3].Name = "Tecla";
MyAnimals[3].Class = "Bird-eating spider";
MyAnimals[3].Size = 10;

//fill the combobox with the name of the animals
for (int i = 0; i < 4; i++)
{
cmb_AnimalName->Items->Add(MyAnimals[i].Name);
}
}
//-------------------------------------------------
----------

void __fastcall
TForm1::cmb_AnimalNameChange(TObject *Sender)
{
//if you choose an animal in the combobox, the
//editboxes show the class and the sizw of the
animal
edi_AnimalClass->Text =
MyAnimals[cmb_AnimalName->ItemIndex].Class;
edi_AnimalSize->Text =
MyAnimals[cmb_AnimalName->ItemIndex].Size;
}
//-------------------------------------------------
---------
```

The following picture shows, how the form should look like:



Name : Tecla

Kind : Bird-eating spider

Size (in cm) : 10

10.7.5 Exercises

1. Create a form and add an edit-boxes for name, address, etc. Use an array of structures to store name and address of more than one person. Use buttons to go forward and backward in the structure and show the values of the current address in the form. Use buttons to add and delete names and addresses form the structure.
2. Use exercise 1. Add a field "Gender" to form and structure. Create a button. On click on the button a second form should open, where the name and the address is written in form of the title of a letter. Watch the form of address concerning gender (Mr., Ms.).
3. Use a struct to store the name of music-tiles and the filename of the

title. Create a form with a combo-box and a button. The combo-box should hold the titles. On click on the button the music-title should be played.
Annotation: Use the TMediaPlayer component to play the music-files.

10.8 Object Oriented Concept

Aim of this subchapter: The student should learn the concept of object oriented programming. She should be able to declare a class, write functions for that class, define an object and use the functions of that object.She also should be able to use constructor and deconstructor of an object. She should be able to describe and use inheritance, polymorphism and overloading.She should know for which the terms “event”, “method”, and “property” in visual programming environments stands for.

First we will start with the definition of object oriented programming and why we should do all the additional work. We will see that all starts with a class defined in a header-file. We will write functions for that class and learn about the use of the special constructor- and deconstructor-functions. Next we will learn what the concept of inheritance, polymorphism, and overloading is and how to use them.
At last we will do a look to visual developing environments and how events, methods, and properties are defined there.

10.8.1 Definitions

10.8.1.1 Abstract Datatyp

Up to now only simple data types as char, int, float, as well as pointer and complex data types as array, struct, union (an similar type to struct, but you have only access to one element at the same time) and enum used. For the object-oriented programming a new data type is introduced, the abstract data type.

Definition:
An abstract data type is a user-defined model struct, union or class, which can accept data elements and element functions. They can be both declared both private and public.

The advantage of a such data structure lies in the strict compliance and unchangeability of modular parts of the program. With extensive software packages, written in procedural programming languages it can easily happen, that for example, through adaptation and enlargement at the beginning exactly defined interfaces after and after are 'softened' and become implementation-dependent. The shared modules and programming units are meshed.

Annotation:
The terms “class” and “abstract data type” are used synonymous.

10.8.1.2 Encapsulation

During the traditional programming dates are defined, and also functions are defined to which dates will be handed over and which manipulate dates, and/or deliver dates. At the object-oriented programming together with the dates also functions can be defined which manipulate the dates. This common definition refers to as “Encapsulation”.

10.8.2 Declaration of a Class

The declaration resembles a “struct”. It is introduced with the keyword “class”, then the name of the class (named “sphere”) follows. Curly brackets surround the elements in the class, that are split up in sections “private:” and “public:”

Example:

```
class sphere
{
   private:
   float f_radius;
   float f_volumn;
   float PI;
  public:
   void setRadius (float radius);
   float getRadius (void);
   float getVolumn (void);
   void calculateVolumn (void);
};
```

A class declaration must be finished with a semicolon !
In a class are variables and functions as elements allowed.
The declaration of the class is stored in a companion file to each unit-file, the header-file. The header-file has the same name as the unit-file, but another file-extension “.h”, e.g. the “sphere.h” is associated to the “sphere.cpp”.
You can open the header-file-window by pressing CTRL + F6 in the unit-window.

10.8.2.1 private and public:

Elements of a private section (private:) may be used exclusively from functions of the class. Thus they are protected before illicit insight or modification from outside the class.
every statement can obtain from outside onto elements of the public section (public:). With that public:-variables can be changed from outside.
There no rule according to which certain elements of a class must stand in a specific section! Both functions as also variables can either be declared public: or private:. In order to guarantee a reliable protection of the variables before outside access, it is to be recommended to put them into the private section, and to make available small element functions which can access these variables.
Classes can be declared also with the codeword “struct” (a class is a structure which contains also functions). The difference to a class exists in the handling of the elements, that belong to no separately declared section (private: or public:). While they are at default public in a struct , they always become private Elements in a class by default.

Following examples clarify the difference between struct- and class-declarations:

```
struct AClass
{
   int a;  // variable public accessible
   int getValue(); // no public:-instruction
   necessary
};


class BClass
{
   int b;  // private variable
   int getValue(); // function not accessible
   from outside
};
```

10.8.3 The Object

The class is a declaration of a new type. In order to use the element functions and arrays of the class an object must are defined.

Example:
The function “main” could use the above defined class as follows:

```
void main()
{
   sphere football;

   football.setRadius(15);
   football.calculateVolumn();
}
```

An object (named „football“) of the class "sphere“ is defined. The call of the functions occurs just the same way as the access on the elements of a struct. The function name is separated from the object name by a point.

Annotation:
In the literature the “object” is also called an “instance”.

10.8.4 The definition of element functions

The function declarations within a class are only prototypes. They must be defined in further result.

Example:
The function “setRadius”, that reads the radius for the sphere, could look as follows:

```
void sphere::setRadius (float radius)
{
   f_radius=radius;
}
```

It assigns to the private class variable “f_radius” the value of radius handed over to it.

The definition looks like the traditional functional definition, only the name of the class and two cola (sphere::) stands here before the function name. This tells the compiler unambiguously, that it an element function of the class “sphere”. It is possible that also other classes can contain a function with the name “setRadius” which however does not have anything to do with the function sphere::setRadius.

10.8.5 The Constructor

Each variable should be assign before their use to a reasonable value, so also the elements of a class. From this basis at least one function should be defined that initializes of the interiors of a class. Because this is very

important and stands beside other class components, the element functions which establish the data elements of an object in this manner, are called by a specific name: Constructor.
A "Constructor" must be characterized as such since C++ burdens it with even further tasks. The identification consists in his calling as its name is the same as the class whose objects it is supposed to initialize. The state of the argument list gives information to the compiler about additional qualities of the respective constructor. A parameter-slack constructor is for example the so-called "default constructor". It is always called if the program enters the scope of application of an object and the object declaration does not have any initializer. That means, the call occurs before the first instruction of the program that is the scope of application of the object.
Constructors are called automatically and can not be called arbitrarily as other functions from the programme. However, constructors can call further functions. Constructors can not return any value, either void!

Example:
For the class of the push-button a constructor could look as follows:

```
class sphere
 {
   private:
    float f_radius;
    float f_volumn;
    float PI;
   public:
    sphere();  //constructor
    void setRadius (float radius);
    float getRadius (void);
    float getVolumn (void);
    void calculateVolumn (void);
 };
```

10.8.6 The Deconstructor

Since the lifetime of every object is restricted, there is also an automatic disassembly system with corresponding element functions. These functions are called destructors. They have the task to free all elements belonging to the object – i.e. dynamically reserved memory blocks. The compiler produces destructor-calls automatically.
The lifetime of objects ends not after the destructor was called. If the object leaves the scope of application the destructor is called automatically (with

static objects!).
A destructor wears the name of its class with a leading tilde (~). Destructors have no return type (even not void) and an empty argument list. There is exactly one destructor per class. Was no destructor defined and the class requires a destructor, the compiler produces one automatically.
Default destructors are always public, that is, they stand in the public:-section of the class declaration. Every destructor calls implicitly destructors of its data elements after the own instructions were worked off, and in reverse precedence of their declaration.

Annotation:
To write calls of destructors in the destructor by oneself may cause critical consequences.
If necessary twofold destructor-calls release memory areas twice, which can lead to crashes.

Here some features of destructors:
1. There is only one destructors in a class
2. Destructors do not use parameters.
3. Destructors do not have any return value, not even void.
4. Destructors are optional. If a class has nothing to clean after its action, it does not need a destructor.
5. Destructors are called automatically. Programmes never call destructors.

Example:

```
class sphere
 {
   private:
    float f_radius;
    float f_volumn;
    float PI;
   public:
    sphere();  //constructor
    void setRadius (float radius);
    float getRadius (void);
    float getVolumn (void);
    void calculateVolumn (void);
    ~sphere()  //deconstructor;
 };
```

Example 8: The class “sphere”

1. Create a “New Application”.
2. Put a two Labels , two Edits and five Buttons on the form.
3. Arrange the components so that the labels are left of the edits. The buttons may be everywhere.
4. Name the components as follows:
   Edit1: edi_radius
   Edit2: edi_volumn
   Button1: btn_getRadius
   Button2: btn_setRadius
   Button3: btn_getVolumn
   Button4: btn_calculateVolumn
   Button5: btn_ClearAll
   Leave the Labels as they are.
5. Set the Caption as follows:
   Label1: &Radius
   Label2: &Volumn
   btn_getRadius: Get R&adius
   btn_setRadius: Set Ra&dius
   btn_getVolumn: Get V&lumn
   btn_calculateVolumn: Calculate Vo&lumn
   btn_ClearAll: &Clear All
6. with menu “Files”-“New…” add a new unit to the unit-window.
7. Save the unit using the name “sphere”.
8. Press CTRL + F6 within the “sphere.cpp”. The file “sphere.h” is openend.
9. Add following code to the header file:

```
class sphere
 {
   private:
    float f_radius;
    float f_volumn;
    float PI;
   public:
    sphere();  //constructor
    void setRadius (float radius);
    float getRadius (void);
    float getVolumn (void);
    void calculateVolumn (void);
    ~sphere()  //deconstructor;
 };
```

10. Save the header-file.

11. Return to the “sphere.cpp”-window.
12. Add following code declaration section of the cpp-file:

```
//includes a libray with mathmatical functions
//we will use the function Power() from there
#include <math.hpp>
```

13. Add following code the cpp-file:

```
//the constructor of the sphere, executed when
the
//object is generated
sphere::sphere()
{
 //inialize all private variables
 f_radius = 0;
 f_volumn = 0;
 PI = 3.14;
}
//-------------------------------------------------
---------
//the deconstructor of the sphere, executed when
the
//object is destroyerd
sphere::~sphere()
{
  //nothing to do in this case
}
//-------------------------------------------------
---------
//sets the radius of the sphere
void sphere::setRadius (float radius)
{
 f_radius = radius;
}
//-------------------------------------------------
---------
//returns the current value of the radius
float sphere::getRadius (void)
{
  return f_radius;
}
//-------------------------------------------------
---------
```

```
//returns the current value of the volumn of the
sphere
float sphere::getVolumn (void)
{
  return f_volumn;
}
//-----------------------------------------------
---------
//calculates the volumn of the sphere
void sphere::calculateVolumn (void)
{
  f_volumn = 4 * Power(f_radius, 3) * PI / 3;
}
//-----------------------------------------------
---------
```

14. Save the sphere.cpp-file.
15. Activate the unit-window of the form (named “unit1” by default).
16. Open menu “File”–“Include Unit Hdr”.
17. Choose sphere.h and quit with OK.

The sphere.h now is added as

```
#include “sphere.h”
```

to the unit of the form and we can use our class “sphere” now.

18. Write the following code in the declaration-section of the unit-window (beyond TForm1 *Form1;)

```
//declare an object "football" of the class
"sphere"
sphere football;
```

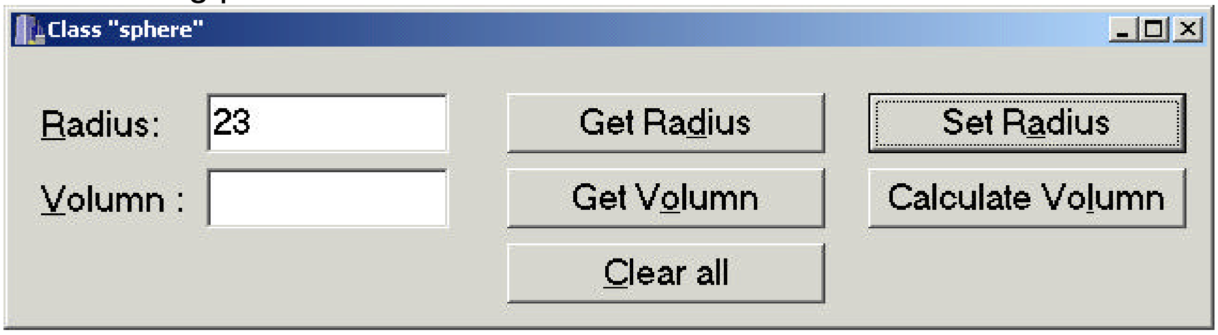
19. Add the following code to the rest of the unit-window:

```
void __fastcall
TForm1::btn_setRadiusClick(TObject *Sender)
{
  football.setRadius ( edi_radius-
  >Text.ToDouble());
}
//-----------------------------------------------
----------
void __fastcall
TForm1::btn_getRadiusClick(TObject *Sender)
{
  edi_radius->Text = football.getRadius();
}
```

```
//-----------------------------------------------------
----------
void __fastcall
TForm1::btn_getVolumnClick(TObject *Sender)
{
  edi_volumn->Text = football.getVolumn();
}
//-----------------------------------------------------
----------
void __fastcall
TForm1::btn_setVolumnClick(TObject *Sender)
{
  football.calculateVolumn();
}
//-----------------------------------------------------
----------
void __fastcall TForm1::btnClearAllClick(TObject
*Sender)
{
  edi_radius->Clear();
  edi_volumn->Clear();
}
//-----------------------------------------------------
----------
```

20. Now you can test the class “sphere” compiling the form and try some values for the “radius”. You will recognize, that if the edit-box for the radius is empty or holds a string and you press the button “Set Radius”, the application generates an exception. We will solve this problem in the following chapter.

The following picture shows how the form should look like:



10.8.7 Exercises

1. Design a class “circle”, that calculate all values of a circle
   Input-values: radius, diameter, Pi
   Output-values: radius, diameter, Pi, area, circumfence.

Create a simple form to text the class.
2. Use exercise 1 of chapter 3.5 and solve the problem as a class.
3. Use exercise 3 of chapter 3.5 and solve the problem as a class.
4. Design a class “Birds”. The class should contain many properties of birds and things they can do, that fit to all birds, e.g. that they have feathers, that they have a colour, some of them can fly, some of them not, their size, their wingspan, what they eat, and so on. Create a form to design birds, e.g. a chicken, a pigeon, even birds, which does not exist.

10.9 Overloading

You can add to the definition of a class a function with the same name as an existing function, that differs only in the type or number of parameters. This is called Overloading of a function.
Both types of function now can be used. The class itself decides which kind of the function has to be used.

Example:
You add following function to the public-section of the class-definiton of “sphere” in the header-file

```
void setRadius (String radius);
```

This overloads the function setRadius. Now we can call the Function setRadius even if the passed parameter is of the type String.

Example 8: Overlaod a function of class sphere
1. Open the sphere.h-file.
2. Add the following definition of a function to the public-section:

```
void setRadius (String radius);
```

3. Add the following definition of a function to the private section:

```
int checkRadius(String radius)
```

   This function will check, if the passed string can by cast to a float.
4. Save the sphere.h-file and activate the sphere.cpp-window.
5. Add following code to the sphere.cpp-file:

```
//the overloaded function for a string-parameter
void sphere::setRadius (String radius)
{
  //if the passed radius is a number cast it to
    float
  //and assign it to the radius. otherwise set
    the radius
  //to 0
  if (checkRadius(radius))
  {
   f_radius = radius.ToDouble();
  }
  else
 {
   f_radius = 0;
 }
}
//--------------------------------------------------
----------
//checks, if the passed string can be cast to a
float
int sphere::checkRadius(String rstr_radius)
{
  //a valid string may only contain this
    characters
    String str_Numbers = "1234567890,";
  //if the string is empty it is invalid; return
    false
  if (rstr_radius == "") return 0;
  //check each character in the passed string
  //if its a valid character. if not return false
  for (int i = 1; i <= rstr_radius.Length();i++)
 {
   //take the character at position i (the
   //Function Substring(i,1) cuts 1 character at
   //postion i) in the passed string
"rstr_radius"   and //check, if it is in the
string
  "str_numbers". if //not the function Pos()
  returns 0.
  if(str_Numbers.Pos(rstr_radius.SubString(i, 1))
  == 0)
  {
   return 0;
  }
 }
 //if all checks passed, return true
 return 1;
}
```

6. Activate the unit-window of the form and change the code of the btn_setRadius-Onclick event to:
void __fastcall TForm1::btn_setRadiusClick(TObject *Sender)

```
{
  //football.setRadius ( edi_radius-
  >Text.ToDouble());
  football.setRadius ( edi_radius->Text);
}
```

7. Run the application again. Now you can type any value in the radius-editbox and press the “Set Radius”-button. You would not get an exception any longer.

10.10 Inheritance

Sometimes you want to add more function to a class, but you must not change it because it is used in the current form. You can define a new class that inherits all function of the existing class and gets some new function too or redefines existing function of the old class. The original class is called the “base class”, the follower is called the “derived class”.
You define inheritance at the definition of the class, when you add a colon two the class-name, followed by the specifier public and the name of the base-class.

Example:

```
class B: public A
 {
  // ...
 };
```

Class A is base-class of class B.

10.10.1 Multiple-inheritance

If on class has two or more base classes this is called multiple inheritance.

Example:

```
class C: public A, public B
{
  // ...
};
```

Classes A and B are base-classes of class C.

10.10.2 Protected-elements

beneath the sections private and public there may exist another section : protected. This takes only into account if you use inheritance. The protected section of the base class is visible for the derived class. From outside the classes the elements of the protected section are inaccessible.

Example:

```
class sphere
{
  private:
   float f_radius;
   float f_volumn;
  protected:
   float PI;
   int checkRadius(String radius);
  public:
   sphere();
   void setRadius (float radius);
   float getRadius (void);
   float getVolumn (void);
   void calculateVolumn (void);
   void setRadius (String radius);
   ~sphere();
};
```

10.10.3 Overriding

You can define a function in of the base-class completely new in a derived-class. The function is defined exactly as in the base-class, but the code is different. This is called Overriding. Overriding is similar to Overloading.

Example:
A class newsphere , derived from the class sphere, defines a function checkRadius, that is already defined in the base-class sphere.

```
class newsphere : public sphere
{
  protected:
   int checkRadius(String radius);
};
```

Example 9: Inheritance of a class and overriding of functions

1. Add a new unit-window.
2. Save it and name it “newsphere”.
3. Open the header file “newsphere.h”
4. To cross-refer to the base-class ad the line:

```
#include "sphere.h"
```

5. Define the derived class “newsphere” as follows:

```
class newsphere : public sphere
{
 private:
 protected:
  int checkRadius(String radius);

 public:
  newsphere();
  void setPi(String Pi);
  float getPi(void);
  void setRadius (String radius);
  ~newsphere();};
```

6. Save the header-file
7. Open the sphere.h file:
8. Add a new section “protected” to the class definition of sphere and move the variable PI and the function checkRadius there.

```
protected:
  float PI;
  int checkRadius(String radius);
```

9. Activate the “newsphere.ccp” window.
10. Add following code:

```
//the constuctor of newsphere
newsphere::newsphere()
{
   //nothing to do
}
//------------------------------------------------
-----------
//the deconstructor of newsphere
newsphere::~newsphere()
{
   //nothing to do
}
//---------------------------------------------
-----------
//a new function, that lets the user set a value
for Pi
void newsphere::setPi(String Pi)
{
//call  the  function  checkRadius  of  the  class
newsphere
    if (checkRadius(Pi))
    {
    PI = Pi.ToDouble();
    }
}
//---------------------------------------------
-----------
//a new function, that returns the current value
of Pi
float newsphere::getPi(void)
{
    return PI;
}
//---------------------------------------------
-----------
//an  overrided  function  to  set  a  radius  of  the
sphere
void newsphere::setRadius (String radius)
{
    //call  the  function  checkRadius  of  the  class
    newsphere
    if (checkRadius(radius))
    {
        //call the function of the class sphere
        sphere::setRadius(radius);
    }
    else
    {
        //call the function of the class sphere
        sphere::setRadius(0);
    }
}
//---------------------------------------------
-----------
//an overrided function to check the radius
```

```
int newsphere::checkRadius(String rstr_radius)
{
    //variable  to  count  how  often  a  colon  is
    found
    int colonFound = 0;

    // first check the value with the function of
    the base-class
    if (!sphere::checkRadius(rstr_radius))
    {
        return 0;
    }
    else
    {
        //check each character of the string
        for     (int     i     =     1;     i     <=
        rstr_radius.Length();i++)
        {
            //if  it  is  a  colon,  increment  the
            counter
            if(rstr_radius.SubString(i, 1) == ",")
            {
                colonFound++;
            }
        }
    }
    //  if  more  than  one  colon  were  found,  then
    return false
    if (colonFound > 1) return 0;
    //if all checks passed, return true
    return 1;
}
```

11. Activate the form.
12. Add a Edit and two Buttons to the form.
13. Name them as follows:
    Edit1: edi_Pi
    Button1: btn_getPi
    Button2: btn_setPi
14. Activate the unit-window of the form.
15. Change the include statement from sphere.h to newsphere.h:

```
#include "newsphere.h"
```

16. Change the code of the unit as follows:

```
//declare an pointer of an object "football" of
the class "sphere"
newsphere *football;
//sphere football;
//-------------------------------------------------
----------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
 //assign the pointer with an real newsphere
 object
 football = new newsphere;
}
//-------------------------------------------------
----------
void __fastcall
TForm1::btn_setRadiusClick(TObject *Sender)
{
 //football.setRadius ( edi_radius-
 >Text.ToDouble());
 football->setRadius ( edi_radius->Text);
}
//-------------------------------------------------
----------
void __fastcall
TForm1::btn_getRadiusClick(TObject *Sender)
{
 edi_radius->Text = football->getRadius();
}
//-------------------------------------------------
----------
void __fastcall
TForm1::btn_getVolumnClick(TObject *Sender)
{
 edi_volumn->Text = football->getVolumn();
}
//-------------------------------------------------
----------
void __fastcall
TForm1::btn_setVolumnClick(TObject *Sender)
{
 football->calculateVolumn();
}
```

```
//--------------------------------------------------
----------
void __fastcall TForm1::btnClearAllClick(TObject
*Sender)
{
 edi_radius->Clear();
 edi_volumn->Clear();
 edi_Pi->Clear();
}
//--------------------------------------------------
----------

void __fastcall TForm1::btn_setPiClick(TObject
*Sender)
{
 football->setPi(edi_Pi->Text);
}
//--------------------------------------------------
----------

void __fastcall TForm1::btn_getPiClick(TObject
*Sender)
{
 edi_Pi->Text = football->getPi();
}
//--------------------------------------------------
----------
```

10.11 Polymorphism

Sometimes it is not clear which function, that of the basic or that of the derived class, has to be used y a programme. This may only occur, if you use pointers to declare a handle to a class, as done in example 9.
In other words, the compiler does not know at compiling time, which class the object is assign to.
The problem is fixed if you define functions as “virtual” function.

Example:
Lets use our classes sphere and newsphere.
We assign a pointer to the basic class

```
sphere *ball = new sphere;
```

If we call the function

```
ball->setRadius(4);
```

everything is all right and the right function is call.
If we assign the pointer this way:

```
sphere *ball = new newsphere;
```

and then call the function

```
ball->setRadius(4);
```

the function of class sphere is called, although there is a equal function in class newsphere and this should be called.
You can handle the problem by assigning the pointer writing

```
newsphere *ball = new newsphere;
```

but in many cases you don't know that at design-time.

Another, better solution for this problem is to define setRadius as a virtual function at least in the basic class sphere, better in all derived classes too:

```
virtual void setRadius(float radius);
```

If this is done the right function is called even if the compiler does not know which object is used at runtime.

10.11.1 Exercises

1. Use the class "birds" of exercise 4 of chapter 4.7 to create a derived class "predator".
2. Use exercise 2 of chapter 4.7 and create a derived class which can create a mail-address, as described in exercise 2 of chapter 3.5.

10.12 StringList

Aim of this sub chapter:The Students should learn about the concept of StringList and how they are used to manage text. Also they should learn, how to open a text-file and save text in a file.

In this chapter we will look at how to use a stringlist. We will read the strings of a stringlist, read a file into a stringlist and save a stringlist as a file.

10.12.1 Definition

The Stringlist is a component that stores text. You may consider the Strings as lines in a text or as fields in an array. A stringlist therefore is a text with one or more lines. You can access the different lines as the elements in a field or items by their index. You can add new strings, delete strings or empty the whole stringlist.
The stringlist is a hidden component, that means, you may not add it to a

form from the component-list.
To define a Stringlist you add following code to a function, that allocates memory for the stringlist
TStringList *MyText;
To assign the allocated memory with a real stringlist you have to do as follows
MyText = new TStringList;

10.12.2 Methods and Properties

Now you can use the stringlist:

```
MyText->Add(“new itemtext”);
```

will add the string “new itemtext” to the stringlist.

```
MyText->Delete(index);
```

will delete a stringg at position “index”.

```
MyText->Clear();
```

will empty the whole stringslist.

```
MyText->Count;
```

will count, how many strings a stringlist contains.

```
MyText->String[i];
```

will read the string at the position i, starting with 0. you can also use this property to set the string-text at position i.
You can open a text-file and store the lines in a stringlist with following code-line:

```
MyText->LoadFromFile(filename);
```

or you store the content of a stringlist in a file

```
MyText->SaveToFile(filename);
```

10.12.3 Similar Components

There are two components, that work in the same way as the Stringlist, because they use the same basic class, TStrings.
The first of these components is the TRichEdit, that uses lines to handle the text. A user can work on the text shown in a RichEdit-component.
The second is the TStringGrid-component, that uses Cells, that are bound to columns and rows. Here also a user can alter the text in the cells.
In both cases you can open a file and show the content in the component and also save the changes, a user made.

Example 10: Using a Stinglist to open a file

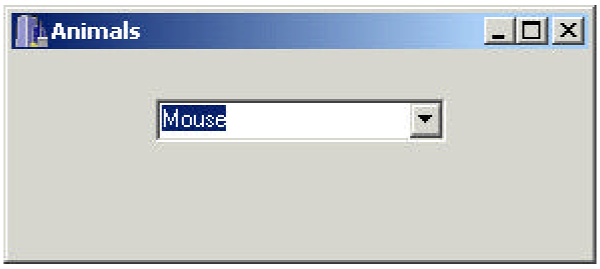
1. Create a new application.
2. Change the caption of the form to “Animals”.
3. Add a combo-box to the form.

4. Name the combo-box “cmb_Animals”, clear the property Text.
5. Create a text-file in the windows editor and write the names of animals in it.
   Save the file in the folder of the project using the name “animals.txt”
6. Assign the event OnCreate of the form and add following code to the function FormCreate

```
//define an pointer as StringList
TStringList *stl_Animals;

//assigen the pointer to a real Stringlist
stl_Animals = new TStringList;

//read a text-file an store the lines in the
stringlist
//the textfile contains names of animals one
beneath another
stl_Animals->LoadFromFile("animals.txt");
//add each animal of the textfile to the combobox
for (int i = 0;i<stl_Animals->Count;i++)
{
  cmb_Animals->Items->Add(stl_Animals-
  >Strings[i]);
}
//unassign the pointer
delete stl_Animals;#include "sphere.h"
```

If you run the application the file is opened, the animal-names are read and the combo-box is filled.

The next picture shows how the form should look like:



10.12.4 File dialogs

In this example there may occur an exception, when the file with the animal-names is not found.
you can work this through by using the open-file-dialog, which is simply a component TOpenDialog.

Example 11: Using a file-dialog to open a file

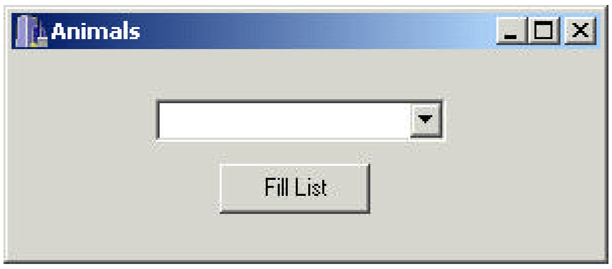
1. Add a button to the form of example 10.
2. Add a component TOpenDialog to the form.
   You may place it everywhere, because it is not visible at runtime.
3. Change the name of the opendialog to “dlg_OpenFile”.
4. Add a combo-box to the form.
5. Name the combo-box “cmb_Animals”, clear the property Text.
6. Add a button to the form.
7. Name the button “bnt_FillList”, set the caption to “Fill List”
8. Move the whole code from the FormCreate-function to the event-function OnClick of the button and add some new lines as follows.

```
void                                   __fastcall
TForm1::bnt_FillListClick(TObject *Sender)
{
   //define an pointer as StringList
   TStringList *stl_Animals;

   //set  the  filefilter  in  the  open-dialog  so
   that
   //only text-files appear in the dialog
   dlg_OpenFile->Filter     =     "Text     files
   (*.txt)|*.TXT";
   //open the file-dialog-window
   if(dlg_OpenFile->Execute())
   {
      //assign the pointer to a real Stringlist
      stl_Animals = new TStringList;

      //read  a  text-file  opend  by  the  open-
      dialog
      //an store the lines in the stringlist the
      //textfile contains names of animals one
      //beneath an other
      stl_Animals->LoadFromFile(dlg_OpenFile-
      >FileName);
      //add  each  animal  in  the  textfile  to  the
      combobox
      for (int i = 0;i<stl_Animals->Count;i++)
      {
         cmb_Animals->Items->Add(stl_Animals-
         >Strings[i]);
      }
      //unassign the pointer
```

```
                delete stl_Animals;
            }
        }
```

Now you can specify, where the file is saved, that you want to open for the animal-names.

The following picture shows how the form should look like:



10.12.5 Exercises

1. Create an application with an TMemo-component. Use the Lines-property of TMemo for file handle. Create the menus “File Open” and “File Save” to open and save text-files.
2. Create an application with an TImage-component. Use the Picture-property of TImage for file handle. Create the menu “File Open” to open BMP-files.

10.13 Filestreams

Aim of this subchapter: The Students should learn about the concept of filestreams and how they are used to manage output and input to files.

In this chapter we will get an overview on filestreams. We will use filestreams to read and write files.

10.13.1 File-Operations with Streams

If a file is opened, one must distinguish on principle between the text mode and the binary mode.

10.13.1.1 The Text Mode

The text mode means, that during the input of the dates the CR/LF sequence is transformed into the '\n' character. Vice versa during the writing a '\n' character is transformed into a CR/LF sequence.

10.13.1.2 The Binary Mode

In the binary mode the automatic transformation remain undone. The characters are read and written unchanged.

10.13.1.3 The classes „ifstream“ and „ofstream“

In order to be able to use streams for files, the header file <fstream.h> must be bound.
With the classes ifstream and ofstream a direct connection to a file is established. Ifstream opens a file for the input, ofstream for the writing, both by default in text mode!

Example: Coping a file in text mode

```
#include <fstream.h>


int main (void)
{
 char ch;

 ifstream inf ("FILE.IN");
 ofstream outf ("FILE.OUT");

 if (!inf) cerr << "Error in Infile";
 if (!outf) cerr << "Error in Outfile";

 while (inf.get (ch))
 outf.put (ch);
}
```

In the example above characters are read from the file FILE.IN and writes them into the file FILE.OUT for so long until from the file FILE.IN no more characters can be read.

File streams are input/output streams as cin and cout. Therefore here also the stream operators "<<" and ">>" are used. In the previous example line 14 and 15 could be adjusted to this notation:

```
while (inf >> ch)
  outf << ch;
```

However this method has the disadvantage, that line feeds are ignored.
If the file is supposed to be opened in the binary mode, the in- and outfile constructors allow to declare a file stream, and to join this with a file afterwards. Additional mode bits can be indicated by that during the file

open.

Example: File open in binary mode

```
// generate input-stream
ifstream inf;

...

// establish connection
inf.open ("FILE.IN", ios::binary);

...

inf.close();
```

The example above opens a stream to the file FILE.IN in binary mode. The file mode bit is optional. The mode can be taken from the following table.

MODE-BIT	ACTION
ios::app	Appends dates, writes always to the end
ios::ate	Positions onto the end of file after open
ios::in	Opens for input (implicitly for ifstream)
ios::out	Opens for output (implicitly for ofstream)
ios::binary	Opens the file in binary mode
ios::trunc	Overwrite the content of the file if it exists. (implicit, if ios::out is indicated, and neither ios::app nor ios::ate is given)
ios::nocreate	Error at open, if the file does not exist
ios::noreplace	Error at open, when the file exists, and neither ios::app nor ios::ate is given

10.14 Error-handling and Exceptions

Aim of this subchapter: The Students should learn how to deal with exceptions.

In this chapter we will use the keywords try and catch to intercept the occurrence of an execption.

10.14.1 Definition

Sometimes it happens, that a function uses values, that does not fit. May

be they are to big, to small, not of the expected type or even missing. In this case you can use the statement “try” to cause the application to try to execute a code, but if it fails, “catch” the exception generated and do something else, e.g. pop up a warning.
In general the statements is used as follows:

```
try
{
    //one or more lines of code
    //which has to be tried
    //if ok, than execute it
}
catch(…)
{
    //code of what to do, if an error occurred
}
```

Example 12: Error-handling of file-open

1. Use the example 10.
2. Change the code of the FormCreate-function as follows:

```
void     __fastcall     TForm1::FormCreate(TObject
*Sender)
{
    TStringList *stl_Animals;

    //here starts the “try”-statement
    try
    {
        stl_Animals = new TStringList;
        stl_Animals->LoadFromFile("animal.txt");
    }
    //if an exception occurs an error-warning
    //is  shown  in  a  message  box  and  the  rest  of
    the //function is not executed
    catch(...)
    {
        Application->MessageBox("File  'animal.txt'
        not found", "Error", MB_OK);
        return;
    }
    for (int i = 0;i<stl_Animals->Count;i++)
    {
        cmb_Animals->Items->Add(stl_Animals-
        >Strings[i]);
    }
    delete stl_Animals;

}
```

3. If you run the application you will get an exception, because the file “animal.txt” does not exist. The exception occurs but the result is not the expected.
   There are two possible ways to test this application:
   a) run “make” or “build” and execute the EXE-file
   b) go to menu Tools – Debugger Options, register “Language Exceptions”, and deactivate “Stop on Delphi Exceptions”. Run the application again.

10.14.2 Exercises

1. Check the exercises of previous chapters if you can add error-handling, e.g. on file-open, file-save, parameter-exchange with wrong type, etc.

Chapter 11: Advanced Module 1 Visual Component Library (VCL)

Aim: In this chapter you will learn what the VCL is and how to use it.

The following subjects will be explained in this chapter:

1. The Borland Visual Component Library (VCL).
2. A short overview of the VCL.
3. Structure of the VCL.
4. The application and form classes.
5. The string class.
6. The sets.
7. The component classes

11.1 Introduction

The environment of Borland C++ Builder offers the programmer a library of classes (framework) called **VCL** (Visual Component Library).
To understand the concept and the usefulness of a framework we can relate it to the concept of house building. There are two possibilities:
Building the house from scratch, brick by brick, tile by tile etc
Building using prefabricated units.

There is no doubt which option we would choose when dealing with IT.
A framework is a collection of prefabricated modules, ready for use and adaptation according to the programmer's needs.
Every modern environment has its own framework: Visual C++, for instance, uses "MFC" (Microsoft Foundation Class Library).
VCL was born out of **Delphi**, the basis of all modern Borland environments. Delphi uses **Object Pascal**, which derives from **Pascal**; as a consequence **VCL** is written in **Object Pascal** and adapted for **C++ Builder**. **VCL** is the core of Borland C++ Builder.
How can you find **VCL** in Borland C++ Builder?
Quite easily: any operation in BCB is linked to VCL; every time you insert and manipulate a form, a button, an editbox etc you create new instance of components and **VCL** and has an effect on them. When you insert a button on a form you create an instance of the **TButton** class, called **Button1** by default. **Button1** has inherited the characteristics of the component **VCL TButton** and thanks to the work of the programmer acquires its own personality and behaviour.

11.2 A Short Overview of the VCL

VCL has a hierarchical structure like a pyramid. Despite their name, the components of the Visual Component Library are not necessarily visual; **TTimer** (a component generating a regularly timed event), is not visual, for instance.
The **VCL** components (with varying properties) can be used, according to the particular program, either in the development phase or in the runtime phase or both.

When you insert a component on the form, making use of visual programming possibilities offered by the environment, BCB automatically writes the related code in C++. When instead you want to create and activate a component in runtime it is necessary to write directly the code in C++ yourselfas in the following example:

```
TOpenDialog* dlg = new TOpenDialog(this);
dlg->Title = "Open a new file";
dlg->Execute();
```

All the **VCL** objects must be allocated dynamically. Their creation takes place by using the operator **new**.

The only exceptions are the Windows structures reproduced in **VCL** as **TPaint**:

```
TPaint Paint(10, 10, 200, 200);
```

In this case the allocation takes place in a static way and the parameters give the position of the component.

The **VCL** classes do not have standard parameters for their functions.
If, for instance, you want to display a message it is necessary to define: the text of the message, the title of the window and the buttons to display (flag):

```
Application->MessageBox("This is a message",
"Message", MN_OK);
```

In other environments:

```
MessageBox("This is a message ")
```

11.3 The Structure of the VCL.

It is not necessary to know by heart all the details of the structure of the **VCL**; but it is important to have an idea of its structure.
Below you will find a tree menu of the **VCL** pyramid.

```
TObject;
-TPersistent;
-  -  TComponent;
-  -  -  [non visual components];
-  -  -  -  TTimer, etc.;
-  -  -  [visual components];
-  -  -  -  TControl;
-  -  -  -  -  TGraphicControl;
-  -  -  -  -  -  TShape, etc.;
-  -  -  -  -  -  -  TWinControl;
-  -  -  -  -  -  -  -  TButton, etc.
```

You are already familiar with the prefix "**T**",after meeting it so many times in **TButton**, **TEdit**, **TLabel**... etc.
At the top of the pyramid you find **TObject**, ancestor of all the classes; this is quite obvious, since we are dealing with object programming.
TPersistent handles the saving in files and memory of all the other components **TComponent** is the base class for all the components; the non visual components derive from **TComponent**, the visual components from the next class **TControl**.
The base classs for the graphic components and the class for Windows controls derive from **Tcontrol.**

When you insert forms in a project, their components and their controls, a pointer to the related class is created in BCB code; the name of the variable pointer comes from the property **Name** of the component itself.

11.4 The application and form classes.

In **VCL** the application class (**TApplication**) and the forms (**TForm**) derive directly from **Tcomponent.**

TApplication encapsulates the basic operations of a Windows program and handles operations like:
handling of the application icon;
display of online help;
display of messages.

Some properties of **TApplication** (**Icon**, **HelpFile** and **Title**), can be set within the page **Application** of the window **Project Options**.

TForm encapsulates the basic operations of any possible window and, as you have already seen in the previous chapters, includes a great number of properties, methods, events.

11.5 The string class.

VCL includes the Class **AnsiString**, simply renamed **String**, which emulates the type **Pascal long string** (as already mentioned, **VCL** is written in **Object Pascal**). This class is used in the majority of **VCL** components and allows all kinds of operations on the strings.

Some components, originally written to be used with **Delphi**, need the class **SmallString**.

11.6 The set class

Some data types are present in **Pascal** and absent in **C++** (or viceversa); one of these is the **set** function. Sets are often used in **VCL**, it is important to understand how they work.

To understand what a **set** is you can use the property **Style** of the class **TFont**. This property can include one or more of the following values: **fsBold**, **fsItalic**, **fsUnderline** and **fsStrikeout**.
As already mentioned, **C++** does not include **sets**, the solution was therefore a class **template** obviously called **set** emulating this type. Usually the attributes of the text are set in the development phase but it may be necessary to do this at runtime.
Let's suppose you want to add the attributes bold and italic to the style of the text of (**TEdit**).

```
TFontSyles Styles;
Styles << fsBold << fsItalic;
Edit1->Font->Style = Styles;
```

The operator **<<** is used to add elements to the set; it is also necessary to assign the set to the property **Font->Style** of the component to implement the desired style.

If you now wish to remove the **fsItalic** from the set and consequently from the text in the editbox, you will have to write the following code:

```
Styles >> fsItalic;
Edit1->Font->Style = Styles;
```

The operator **>>** is used to delete elements from the set.

To know if an element is part of a set, let us say **fsBold**, you will use the method **Contains()**:

```
Bold = Edit1->Font->Style.Contains(fsBold);
if (Bold) do something();
```

You may sometimes need to make a set empty; you will use the method **Clear**:

```
Edit1->Font->Style.Clear();
```

11.7 The component classes

Below you find a list of classes deriving from **TComponent**.

11.7.1 Standard Control classes

In addition to all the standard Windows controls you have met so far **TEdit**, **TBbutton**, etc.), the following also belong to this group:
TPanel, container for tool bars, status bars and others;
TBitBtn and **TSpeedButton**, buttons containing images;
TImage, to visualise images on a form;
TBevel, to create squares and lines on a form;
TStringGrid and **TDrawGrid**, to present information in table form.

11.7.2 Custom controls classes

VCL includes a group of classes that encapsulate the main controls for **32 bit di Windows**, for instance **TRichEdit**; some of these classes are quite complex and will be dealt with at a later stage.

11.7.3 Standard Dialog box classes

The group of classes related to the dialogue windows includes all the main Windows dialogue windows. The following belong to this group: **TOpenDialog**, **TSaveDialog**, **TOpenPictureDialog**, **TSavePictureDialog**, **TFontDialog**, **TColorDialog**, **TPrintDialog**, **TPrinterSetupDialog**, etc.

All the components of this group are non visual, consequently there is no interface to be displayed while programming.

11.7.4 System classes

The group of system components includes visual and non visual classes. The following belong to this group: - **TTimer**, the system Timer of Windows; its only event is **OnTimer**, it is triggered by default every time the timer is started. The interval between the events **OnTimer** is set through the component property **Interval**. **TTimer** is a non visual component;
- **TMediaPlayer**, to reproduce multimedia files;
- **TPaintBox**, to paint.

11.7.5 GDI classes

The **GDI** (Graphics Device Interface) classes are used to visualise images and text in Windows windows; they are not associated to any fixed component but many components use instance of these classes as properties. For instance, the property **Font**, which has many components, is instance of the **TFont** class.
Among the **GDI** classes we find:
TCanvas, to draw and to display images and text.
TBrush, represents a board to be used as an area to be filled with colours or with an image.
TBitmap, handles bitmap images,
TFont handles font properties,

11.7.6 Utility classes

In the group of utility classes we find:
TIniFile, to read and write windows configuration files (**.INI**);
TRegistry and **TRegKeyInfo**, to handle windows configuration register;
TStringList, to read and store lists of strings.
TList, to create an array of any type of object. The array automatically adjusts when inserting or deleting objects;
TStream, TFileStream, TmemoryStream and TResourceStream, for stream reading and writing.

11.7.7 Database classes

There is a group of classes that need to interact with a database. This will be the subject of the next chapter.

11.8 Working with the VCL data Module

Aim of this subchapter: In this chapter you'll learn how to work with the VCL data module

The following subjects will be explained in this subchapter:

1. Creating a VCL database application
2. Overview of database application architecture
3. Creating a new VCL application
4. Setting up data access components
5. Setting up the database connection
6. Setting up the unidirectional dataset
7. Setting up the provider, client dataset, and data source
8. Designing the user interface
9. Creating the grid and navigation bar
10. Adding a button
11. Writing an event handler
12. Writing the Update Now! command event handler
13. Writing the FormClose event handler

11.8.1 Creating a VCL database application

This exercise guides you through the creation a VCL C++ application that lets you view and update an example of an employee database. This exercise is written for product editions that include the database components (Professional Edition). It sets up database access that requires features not available in the Personal edition. In addition, you must have InterBase installed to successfully complete this exercise.
When you install BCB by default you will find the Interbase installation.

11.8.2 Overview of database application architecture

The architecture of a database application may seem complicated at first, but the use of multiple components simplifies the development and maintenance of actual database applications.
Database applications include three main parts: the user interface, a set of data access components and the database itself. In this exercise, you will create a dbExpress database application. Other database applications have a similar architecture.
The user interface includes data-aware controls such as a grid so that users can edit and post data to the database. The data access components include the data source, the client dataset, the data provider, a

unidirectional dataset, and a connection component. The data source acts as a conduit between the user interface and a client dataset. The client dataset is the heart of the application as it contains a set of records from the underlying database that are buffered in memory. The provider transfers the data between the client dataset and the unidirectional dataset, which fetches data directly from the database. Finally, the connection component establishes a connection to the database. Each type of unidirectional dataset uses a different type of connection component.

11.8.3 Creating a new VCL application

Before you begin the exercise, create a folder to hold the source files. Then create and save a new project.

1. Create a folder called **Probiq** to hold the project files you'll create while working through this exercise.
2. Begin a new Project. Choose File|New|Application to create a new project.
3. Choose File|Save All to save your files to disk. When the Save dialog appears, navigate to your **Probiq** folder and save each file using its default name.

Later on, you can save your work at any time by choosing File|Save All. If you decide not to complete the exercise in one sitting, you can open the saved version by choosing File|Reopen and selecting the exercise from the list.

11.8.4 Setting up data access components

Data access components represent both data (datasets) and the components that connect these datasets to other parts of your application. Each of these data access components points to the next lower component. For example, the data source points to the client dataset, the client dataset points to the provider, and so forth. Therefore, when you set up your data access components, you add the components in the proper order starting from the Connection layer.
In the following topics, you will add the database components to create the database connection, unidirectional dataset, provider, client dataset, and data source. Afterwards, you will create the user interface for the application. These components are located on the window Component List.

Tip: It is a good idea to isolate your user interface on its own form and place the data access components in a data module. However, to make things simpler for this exercise, you will place the user interface and all the components on the same form.

11.8.5 Setting up the database connection

The dbExpress page contains a set of components that provide fast access to SQL database servers.
You need to add a connection component so that you can connect to a database. The type of connection component you use depends on what type of dataset component you use. In this exercise you will use the TSQLConnection and TSQLDataSet components.
To add a dbExpress connection component:

1. Choose the TSQLConnection component from the Component List (see chapter 4.4., for how to reach the Component List). The component is called SQLConnection1 by default.
2. In the Object Inspector, set its ConnectionName property to IBLocal (it's on the drop-down list).
3. Set the LoginPrompt property to false. (By setting this property to false, you won't be prompted to log on every time you access the database.)

   TSQLConnection component has a Connection Editor. To display the Connection Editor associated with it select the Object Tree View from the Windows Menu; select the Connection component (SQLConnection1); open the contextual menu and select Edit Connection Property.

   You use the Connection Editor to select a connection configuration for the TSQLConnection component or edit the connections stored in the dbxconnections.ini file. Any changes you make in the dialog are written to that file when you click OK. In addition, when you click OK, the selected connection is assigned as the value of the SQL Connection component's ConnectionName property.

4. In the Connection Editor, specify the pathname of the database file called employee.gdb on your system. In this exercise you will connect to a sample InterBase database, employee.gdb, that is provided with C++Builder. By default, the InterBase installation places employee.gdb in C:\Program Files\Common Files\Borland Shared\Data. Put this pathname in the value field associated with the key named Database.

5. Check the User_Name and Password fields for acceptable values. If you have not altered the default values, you do not need to change

the fields. If database access is administered by someone else, you may need to get a username and password to access the database.

6. When you have finished checking and editing the fields, click OK to close the Connection Editor and save your changes.

   These changes are written to the dbxconnections.ini file and the selected connection is assigned as the value of the SQL Connection component's ConnectionName property

7. Choose File|Save All to save your project.

11.8.6 Setting up the unidirectional dataset

A basic database application uses a dataset to access information from the database. In dbExpress applications, you use a unidirectional dataset. A unidirectional dataset reads data from the database but doesn't update data.
To add the unidirectional dataset:

1. Choose the TSQLDataSet from the Component List.
2. In the Object Inspector, set its SQLConnection property to SQLConnection1 (the database connection created previously).
3. Set the CommandText property to "select * from SALES" to specify the command that the dataset executes.
4. Set Active to true to open the dataset.
5. Choose File|Save All to save the project.

11.8.7 Setting up the provider, client dataset, and data source

Provider components are the way that client datasets obtain their data from other datasets. The provider receives data requests from a client dataset, fetches data, packages it, and returns the data to the client dataset. The provider receives updates from a client dataset and applies them to the database server.

To add the provider:

1. Choose a TDataSetProvider component from the Component List
2. In the Object Inspector, set the provider's DataSet property to SQLDataSet1.

The client dataset buffers its data in memory. It also caches updates to be sent to the database. You can use client datasets to supply the data for data-aware controls on the user interface using the data source component.

To add the client dataset:

1. Choose a TClientDataSet component from the Component List
2. Set the ProviderName property to DataSetProvider1.
3. Set the Active property to true to allow data to be passed to your application.

A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on a form.

To add the data source:

1. Choose a TDataSource component from the Component List
2. Set the data source's DataSet property to ClientDataSet1.
3. Choose File|Save All to save the project.

So far, you have added the nonvisual database infrastructure to your application. Next, you need to design the user interface.

11.8.8 Designing the user interface

Now you need to add visual controls to the application so that your users can view the data, edit it and save it. In the Component List there are a set of data-aware controls that work with data in a database and build a user interface. You'll display the database in a grid and add a few commands and a navigation bar.

11.8.9 Creating the grid and navigation bar

To create the interface for the application:

1. You can start by adding a grid to the form. Choose a TDBGrid component from the Component List and put it onto the form.

2. Set DBGrid's properties to anchor the grid.
3. Locate the property **Anchors** in **OI,** open the context menu, select **Expand** and set the values of the sub-properties **akLeft, akTop,akRight** and **akBottom,** to **True.**

4. Align the grid with the bottom of the form by setting the Align property to alBottom. You can also enlarge the size of the grid by dragging it or setting its Height property to 400.

5. Set the grid's DataSource property to DataSource1. When you do this, the grid is populated with data from the employee database.If the grid doesn't display data, make sure you've correctly set the properties of all the objects on the form, as explained in previous instructions.
6. The DBGrid control displays data at design time while you are working in the IDE. This allows you to verify that you've connected to the database correctly. You cannot, however, edit the data at design time; to edit the data in the table, you'll have to run the application.

7. Choose the TDBNavigator control from the Component List and put it onto the form. A database navigator is a tool for moving through the data in a dataset (using next and previous arrows, for example) and performing operations on the data.

8. Set the navigator bar's DataSource property to DataSource1 so the navigator is looking at the data in the client dataset.

9. Set the navigator bar's ShowHint property to true. (Setting ShowHint to true allows Help hints to appear when the cursor is positioned over each of the items on the navigator bar at runtime.)

10. Choose File|Save All to save the project.

11. Press F9 to compile and run the project. You can also run the project by choosing Run from the Run menu.

When you run your project, the program opens in a window like the one you designed on the form. You can test the navigation bar with the employee database. For example, you can move from record to record using the arrow commands, add records using the + command, and delete records using the - command.

Tip: If you should encounter an error while testing an early version of your application, choose Run|Program Reset to return to the design-time view.

11.8.10 Adding a button

This section describes how to add an Update Now button to the application. This button is used to apply any edits that a user makes to the database, such as editing records, adding new records, or deleting records.
To add a button:

1. From the **Component List** choose a **TButton** and put it onto the form;
2. you will position the button on the right at the top of the form;
3. from the **Component List** choose a **TActionList** and put it onto the form
4. from the menu **Windows,** select **Object TreeView**;
   1. 5.in the window **Object TreeView** select **ActionList1**;
5. open the context menu and select **Action List Editor**;
6. open the context menu again and select **New Action**;
7. go back to the button properties and on the property **Action** select **Action1**.

Set the button caption property to Update Now! When you run the application, the button will appear with the colour grey until an event handler is added to make it work.

11.8.11 Writing an event handler

Most components on the Component palette have events, and most components have a default event. A common default event is OnClick, which gets called whenever a component, such as TButton, is clicked. If you select a component on a form and click the Object Inspector's Events tab, you'll see a list of the component's events.
For more information about events and event handlers, see chapter 4.6.4

11.8.12 Writing the Update Now! command event handler

First, you'll write the event handler for the Update Now! button:

Write the following Action1Execute Function in the Unit1.cpp module

```
void __fastcall TForm1::Action1Execute(TObject
*Sender)

  {


if(ClientDataSet1->State == dsEdit ||
ClientDataSet1->State == dsInsert)

      ClientDataSet1->Post();
    ClientDataSet1->ApplyUpdates(-1);
  }
```

This event handler first checks to see what state the database is in. When you move off a changed record, it is automatically posted. But if you don't move off a changed record, the database remains in edit or insert mode. The if statement posts any data that may have been changed but was not passed to the client dataset. The next statement applies updates held in the client dataset to the database.

Note: Changes to the data are not automatically posted to the database when using dbExpress. You need to call the ApplyUpdates method to write all updated, inserted, and deleted records from the client dataset to the database.

11.8.13 Writing the FormClose event handler

Finally, you'll write another event handler that is invoked when the application is closed. The application can be closed either by using File|Exit or by clicking the X in the upper right corner. Either way, the program checks to make sure that there are no pending updates to the database and displays a message window asking the user what to do if changes are pending.
You could place this code in the Exit event handler but any pending database changes would be lost if users chose to exit your application using the X.

1. Click the main form to select it (rather than any specific object on it);
2. Select the Events tab in the Object Inspector to see the form events;
3. Select the event **OnClose**;
4. Press **Ctrl+Enter** and go to the code window and insert the following code lines:

```
void __fastcall TForm1::FormClose(TObject
*Sender, TCloseAction &Action)

{
```

Right where the cursor is positioned (between the braces), type:

```
Action = caFree;
   if(ClientDataSet1->State == dsEdit ||
ClientDataSet1->State == dsInsert)
       ClientDataSet1->Post();
   if(ClientDataSet1->ChangeCount > 0) {
      int Ret = Application->MessageBox("You have
pending updates. Do you want to write them to the
database?", "Pending Updates", MB_YESNOCANCEL);


      if(Ret == IDYES)
        ClientDataSet1->ApplyUpdates(-1);

     else
        if(Ret == IDCANCEL)
           Action = caNone;
}
```

This event handler checks the state of the database. If changes are pending, they are posted to the client dataset where the change count is increased. Then before closing the application, a message box is displayed that asks how to handle the changes. The reply options are Yes, No, or Cancel. Replying Yes applies updates to the database; No closes the application without changing the database; and Cancel cancels the exit but does not cancel the changes to the database and leaves the application still running.

To finish, choose File|Save All to save the project. Then press F9 to run the application.

That's it! You can try out the application to see how it works.

11.9 Creating a SDI application

Aim of this subchapter: In this chapter you'll learn how to create an Single Document Interface (SDI) application. This will be explained by an example how to creat a text-editor.

11.9.1 Creating a text editor with an SDI application

In this chapter the following subjects will be explained:

1. Creating a text editor-a tutorial
2. Starting a new application
3. Setting property values
4. Adding components to the form
5. Adding support for a menu and a toolbar
6. Adding images to the Action Manager
7. Adding actions to the Action Manager
8. Adding standard actions to the Action Manager
9. Adding a toolbar
10. Clearing the text area
11. Writing event handlers
12. Creating an event handler for the New command
13. Creating an event handler for the Open command
14. Creating an event handler for the Save command
15. Creating an event handler for the Save As command
16. Creating a Help file
17. Creating an event handler for the Help Contents command
18. Creating an event handler for the Help Index command
19. Creating an About box
20. Completing your application

Note:
This tutorial is for the professional edition of C++ Builder.

11.9.1.1 Starting a new application

Before beginning a new application, create a directory to hold the source files:

1. Create a directory called **TextEditor** on your computer.
2. Begin a new project by choosing **File|New|Application** or use the default project that is already open when you started C++ Builder.

Each application is represented by a project. When you start C++ Builder, it creates a blank project by default, and automatically creates the following files:

- **Project1.cpp:** a source-code file associated with the project. This is called a project file.
- **Unit1.cpp:** a source-code file associated with the main project form. This is called a unit file.
- **Unit1.h:** a header file associated with the main project form. This is called a unit header file.
- **Unit1.dfm**: a resource file that stores information about the main project form. This is called a form file.

Each form has its own unit (**Unit1.cpp**), header (**Unit1.h**), and form (**Unit1.dfm**) files. If you create a second form, a second unit (**Unit2.cpp**), header (**Unit2.h**), and form (**Unit2.dfm**) file are automatically created.

3. Choose **File|Save All** to save your files to disk. When the Save As dialog box appears:

   Navigate to your **TextEditor** folder.
   Save **Unit1** using the default name **Unit1.cpp**.
   Save the project using the name **TextEditor.bpr**. (The executable will be named the same as the project name with an .exe extension.)

   Later, you can resave your work by choosing **File|Save All**.
   When you save your project, C++ Builder creates various additional files in your project directory. Do not delete these files.

11.9.1.2 Setting property values

When you open a new project, C++ Builder displays the project’s main form, named **Form1** by default. You’ll create the user interface and other parts of your application by placing components on this form.
Next to the form, you’ll see the **Object Inspector**, which you can use to set property values for the form and the components you place on it. When you set properties, C++ Builder maintains your source code for you. The values you set in the **Object Inspector** are called design-time settings.

1. Find the form’s Caption property in the **Object Inspector** and type "**Text Editor Tutorial**" replacing the default caption '**Form1**". Notice that the caption in the heading of the form changes as you type.
2. Run the form now by pressing **F9**, even though there are no

components on it.
3. To return to the design-time view of Form1, press **Alt+F4**.

11.9.1.3 Adding components to the form

Before you start adding components to the form, you need to think about the best way to create the user interface (UI) for your application. The UI is what allows the user of your application to interact with it and should be designed for ease of use.
C++ Builder includes many components that represent parts of an application: menus, toolbars, dialog boxes, and many other visual and nonvisual program elements.

The text editor application requires an editing area, a status bar for displaying information such as the name of the file being edited, menus, and a toolbar with buttons for easy access to commands. The beauty of designing the interface using C++ Builder is that you can experiment with different components and see the results right away. This way, you can quickly prototype an application interface.
To start designing the text editor, add a text area and a status bar to the form.

1. To create a text area, first add a **RichEdit** component. Choose **View|Component List**, select **TRichEdit** and press the **Add to form** button; then press **Esc** to close the **Component List**.
2. Each C++ Builder component is a class; placing a component on a form creates an instance of that class. Once the component is on the form, C++ Builder generates the code necessary to construct an instance of the object when your application is running.
3. With the **RichEdit** component selected, in the **Object Inspector**, select the **Align** property and set it to **alClient** (select from items).
4. The **RichEdit** component now fills the entire form so you have a large text editing area. By choosing the **alClient** value for the **Align** property, the size of the **RichEdit** control will vary to fill whatever size window is displayed even if the form is resized.
5. Choose **View|Component List**, select **TStatusBar** and press the **Add to form** button; then press **Esc** to close the **Component List**.
6. With the **StatusBar** component selected, in the **Object Inspector**, select **Panels** property and press **Ctrl+Return** to open the **Editing StatusBar1->Panels** dialog box.
7. **Right-click** and choose **Add**, to add a panel to the status bar. The panel will display the path and file name of the file being

edited by your text editor.

8. In the **Object Inspector**, set the **Text** property to "**untitled.txt**". When you use the text editor, if the file being edited is not yet saved, the file name will be "**untitled.txt**".
9. Choose **Window|Editing StatusBar1->Panels**, Then close it.
10. Now the main editing area of the user interface for the text editor is set up.

11.9.1.4 Adding support for a menu and a toolbar

For the application to do anything, it needs a menu, commands, and, for convenience, a toolbar. Though you can code the commands separately, C++ Builder provides an action list or an action manager to centralize the actions, images, and code for menu commands and toolbar buttons.
By convention, the actions that are connected to menu commands have a top-level menu name and a command name. For example, the **FileExit** action refers to the **Exit** command on the **File** menu. The following table lists the kinds of menu commands your text editor application needs and whether the action has an associated toolbar button:

Menu	**Command**	**On Toolbar?**	**Description**
File	New	Yes	Creates a new file.
File	Open	Yes	Opens an existing file for editing.
File	Save	Yes	Saves the current file to disk.
File	Save As	No	Saves a file using a new name (also lets you save a new file using a specified name).
File	Exit	Yes	Quits the editor program.
Edit	Cut	Yes	Deletes text and stores it in the clipboard.
Edit	Copy	Yes	Copies text and stores it in the clipboard.
Edit	Paste	Yes	Inserts text from the clipboard.
Help	Contents	No	Displays the Help contents screen from which you can access Help topics.
Help	Index	No	Displays the Help index screen.
Help	About	No	Displays information about the application in a box.

11.9.1.5 Action Manager editor and Action List editor differences

Depending on your edition of C++ Builder, there are two ways to manage actions and images for your menus and toolbar. All editions of C++ Builder include the Action List editor, which provides a location to centralize the response to user commands. The Action List editor is part of the Borland

Component Library for Cross Platform (CLX) and should be used instead of the Action Manager editor if migrating to a different platform (such as Linux) is a future possibility.

The Action Manager editor provides some special functionality, but is only available as part of the Visual Component Library (VCL), which is specific to the Windows platform. Using the Action Manager editor Customize dialog can provide menu actions that are customizable by the end user, and that have some of the properties of Microsoft Office (such as having seldomly used menu items hidden from view). Additionally, the Action Manager provides a more rapid development process, as actions can simply be dragged from the Action Manager Customize dialog to the menu component on the form.

Warning:
This tutorial uses the Action Manager Customize dialog for Enterprise and Professional editions of C++ Builder. If you have the Enterprise or Professional edition, proceed to "Adding menu and toolbar images (Enterprise and Professional)”.

11.9.1.6 Adding images to the Action Manager

In this section, you’ll add images for use with Action Bands.
In many cases you would add an **ImageList1** component to your form and import your own images. For this tutorial we will save time by importing the image list that was used to create the C++ Builder IDE. Unless you add your own graphics, the **ImageList1** on the **Component List** will use default images for standard actions.
To add the existing image list:

1. If you installed C++ Builder to the default directory, choose **File|Open** and select **C:\Program Files\Borland\CBuilder6\Source\vcl\actnres.pas**. To see this file, set Files of type: to (**Any file. *.***) in the Open dialog.

2. Select the **ImageList1** component and copy and paste it to your form. It is a nonvisual component, so it doesn’t matter where you paste it.

Note:
To copy **ImageList1**, **right-click** the component, and click **Edit|Copy**. On your form, **right-click** and choose **Edit|Paste**.

3. Close the **StandardActions window**.

4. Choose **Window|Object TreeView**, select **ImageList1 component**, then **right-click** and choose **ImageList Editor** to display all the possible images you can use.

The image index properties for both standard actions and the **ImageList1** we have added include:

Command	**ImageIndex property**
Edit|Cut	0
Edit|Copy	1
Edit|Paste	2
File|New	6
File|Open	7
File|Save	8
File|SaveAs	30
File|Exit	43
Help|Contents	40

11.9.1.7 Adding actions to the Action Manager

You first add the Action Manager and then add the actions.

1. Choose **View|Component List**, select **TActionManager** and press the **Add to form** button; then press **Esc** to close the **Component List**. Because it is nonvisual, you can place it anywhere on the form.

   Tip:
   To display the captions for nonvisual components you drop on the form, choose **Tools|Environment Options**, choose the **Preferences** page, check **Show component captions**, and click **OK**. Also, hovering over a component with the mouse will display its name.

2. With **ActionManager1** selected on the form, set the **Images** property in the **Object Inspector** to **ImageList1**.
3. Next you'll add the actions to the **Action Manager** and set their properties. You will add both nonstandard actions for which you set all the properties, and standard actions, which have their properties automatically set.
4. Choose Window|Object TreeView, select ActionManager1, then right-click and choose Customize. The Editing Form1-

>ActionManager1 dialogue box, or Action Manager editor, appears.

5. Right-click on the Action Manager editor and choose New Action.
6. Make sure No Category is selected, in the Actions list and Action1 is selected under Actions. In the Object Inspector, set the following properties:

- After **Caption**, type "**&New**". Note that typing an ampersand before one of the letters makes that letter a shortcut to accessing the command.
- After **Category**, type "**File**" (this organizes the File commands in one place).
- After **Hint**, type "**Create file**" (this will be the Help tooltip).
- Make sure the **ImageIndex** is set to **6** (This should match the image list we imported. You can also click the down arrow and select the proper image).
- After **Name**, type "**FileNew**" (for the File|New command) and press Enter to save the change.

7. Make sure File is selected in the Editing **Form1->ActionManager1** window. **Right-click** on the **Action Manager editor** and choose **New Action**.
8. In the **Object Inspector**, set the following properties:

- After Caption, type "&Save".
- Make sure the Category is set to "File".
- After Hint, type "Save file".
- After ImageIndex, select image 8.
- After Name, enter "**FileSave**" (for the File|Save command).

9. **Right-click** on the **Action Manager editor** and choose **New Action**.
10. In the **Object Inspector**, set the following properties:

- After Caption, type "&Index".
- After Category, type "Help".
- No ImageIndex is needed. Leave the default value.
- After Name, enter "HelpIndex" (for the Help|Index command).
- Right-click on the Action Manager editor and choose New Action.
- In the Object Inspector, set the following properties:
- After Caption, type "&About".
- Make sure Category says "Help".
- No ImageIndex is needed. Leave the default value.
- After Name, enter "HelpAbout" (for the Help|About command).

13. Keep the **Action Manager** Customize dialog on the screen.
14. Save your work by choosing **File|Save All**.

11.9.1.8 Adding standard actions to the Action Manager

Next you'll add the standard actions (open, save as, exit, cut, copy, paste, and help contents) to the action manager.

1. The **Action Manager editor** should still be displayed. If it's not: choose **Window|Object TreeView**, select **ActionManager1**, **right-click** and choose **Customize** to open it.
2. **Right-click** on the **Action Manager editor** and choose **New Standard Action**.
3. The **Standard Action Classes** dialog box appears.
4. Scroll to the **Edit** category and use the **Ctrl** key to select **TEditCut**, **TEditCopy**, and **TEditPaste**. Click **OK** to add these actions to a new **Edit** category in the **Categories list** of the Editing **Form1->ActionManager1** dialog box.
5. **Right-click** on the **Action Manager editor** and choose **New Standard Action**.
6. Scroll to the **File** category and select **TFileOpen**, **TFileSaveAs**, and **TFileExit**. Click **OK** to add these actions to the **File** category.
7. **Right-click** on the **Action Manager editor** and choose **New Standard Action**.

8. Scroll to the **Help** category and select **THelpContents**. Click **OK** to add this action to the **Help** category.

   Note:
   The custom **Help|Contents** command displays a Help file always showing the Help Contents tab. The standard **Help|Contents** command brings up the last tabbed page that was displayed, either Contents, Index, or Find.

   Now you've added all the standard actions you need for your application. The standard actions have their properties set automatically, including the image index. You can change the image index to display a different image.

9. You can change the image for standard actions if you prefer. For instance, select the **File|Open** action in the Editing **Form1->ActionManager1** dialog box. Now, change the default image to image 7 in the list.

10. Choose the Close button to close the Action Manager editor.
11. Choose **File|Save All** to save your changes.

11.9.1.9 Adding a toolbar

Since you've set up actions in the **Action Manager** Customize dialog, you can add some of the same actions that were used on the menus to an action band toolbar, which will resemble a Microsoft Office 2000 toolbar when you're finished with it.

1. Choose **View|Component List**, select **TActionToolBar** and press **Add to form** button; then press **Esc** to close the **Component List**. A blank Action Band toolbar appears under the menu bar.
2. If the **Action Manager editor** isn't displayed, open it (choose **Window|Object TreeView**, select **ActionManager1**, **right-click** and choose **Customize**), and select **File** in the **Categories list**. In the **Actions list**, select **New**, **Open**, **Save**, and **Exit** and drag these items to the **toolbar**. They automatically appear as buttons with each assigned image.
3. In the **Action Manager Customize** dialog, drag the **Edit** category to the **toolbar**. All of the **Edit** commands should appear on the toolbar.

   Note:
   If you drag the wrong command onto the toolbar, you can delete it. You can select the item in the **Object TreeView** and press **Del**.
4. Choose File|Save All to save your changes.
5. Press **F9** to compile and run the project.

   Tip:
   When you run your project, C++ Builder opens the program in a runtime window like the one you designed on the form.

   Your text editor already has lots of functionality. If you select text in the text area, the Cut, Copy, and Paste buttons should work. The menus and toolbar buttons work although some of the commands are grayed out. To activate some of the commands you will need to write event handlers.

6. To return to design mode, press **Alt+F4**.

11.9.1.10 Clearing the text area

When you ran your program, the name **RichEdit1** appeared in the text area. You can remove that text using the **String List editor**. If you don't clear the text now, the text should be removed when initializing the main form in the last step.
To clear the text area:

1. On the main form, choose the **RichEdit1** component.
2. In the **Object Inspector** select **Lines** property, press **Ctrl+Enter** to display the **String List editor**.
3. In the **String List editor**, select and delete the text "**RichEdit1**" and click **OK**.
4. Save your changes and run the program again.

The text editing area is now empty when the main form is displayed.

11.9.1.11 Writing event handlers

Up to this point, you've developed your application without writing any code. By using the **Object Inspector** to set property values at design time, you've taken full advantage of C++ Builder's RAD environment. In this section, you'll write functions called event handlers that respond to user input while the application is running. You'll connect the event handlers to the items on the menus and toolbar, so that when an item is selected your application executes the code in the handler.

For nonstandard actions, you must create event handlers. For standard actions, such as the **File|Exit** and **Edit|Paste** commands, the events are included in the code. However, for some of the standard actions, such as the **File|Save As** command, you can write your own event handler to customize the command.
Because all the menu items and toolbar actions are consolidated in the **Action Manager**, you can create the event handlers from there.

11.9.1.12 Creating an event handler for the New command

To create an event handler for the New command:

1. Choose **View|Units** and select **Unit1** to display the code associated with **Form1**.
2. First, you need to declare a file name that will be used in the event handler, adding a custom property for the file name to make it globally accessible from other methods. Open the **Unit1.h** file by **right-clicking** in the **Unit1.cpp** file in the Code editor and

choosing **Open Source/Header File**. In the header file, locate the public declarations section for the class **TForm1**, and on the line after:

```
public: // User declarations
```

type:

```
AnsiString FileName;
```

3. Press F12 to go back to the main form.
4. Go to Object Inspector and choose FileNew from your component.
5. 5. Select the **Events** tab, then choose **OnExecute** and press **Ctrl+Return**.
6. 6. Right where the cursor is positioned in the Code editor (between { and }), type the following lines:

```
RichEdit1->Clear();
FileName = "untitled.txt";
StatusBar1->Panels->Items[0]->Text = FileName;
```

7. Choose **File|Save All**.

11.9.1.13 Creating an event handler for the Open command

To open a file in the text editor, you want a standard Windows Open dialog box to appear. You've already added a standard **File|Open** command to the **Action Manager**, which automatically includes the dialog box. However, you still need to customize the event handler for the command.

1. Go to **Object Inspector** and choose **FileOpen1** from your component.
2. Choose the **Dialog** property, press **Tab** and then **Right** to expand its properties. **Dialog** is a referenced component that creates the Open dialog box. C++ Builder names the dialog box **FileOpen1->OpenDialog** by default. When **OpenDialog1's** Execute method is called, it invokes the standard dialog box for opening files.
3. Choose and set the **DefaultExt** property to "**txt**".
4. Choose **Filter** and press **Ctrl+Return** to display the **Filter editor**.
   - In the first row of the **Filter Name** column, type "**Text files (*.txt)**". In the **Filter** column, type "***.txt**".
   - In the second row of the **Filter Name** column, type "**All files (*.*)**" and in the **Filter** column, type "***.***".
   - Click **OK**.

5. Set Title to "Open file". These words will appear at the top of the Open dialog box.
6. Go to the Events tab. Choose OnAccept and press Ctrl+Return to open a Code Editor.
7. Right where the cursor is positioned (between { and }), type the following lines:

```
RichEdit1->Lines->LoadFromFile (FileOpen1->Dialog
->FileName);
FileName = FileOpen1->Dialog->FileName;
StatusBar1->Panels->Items[0]->Text = FileName;
```

Tip:
You can use the Code Insight tools to help you write your code faster. For example, after you type the arrow (->)after RichEdit1, the code completion dialog box appears. Type an “l” so that Lines : TStrings; appears at the top of the dialog box. Press Enter or double-click it to add it to your code.
That’s it for the File|Open command and the Open dialog box.

11.9.1.14 Creating an event handler for the Save command

To create an event handler for the **Save** command:

1. Go to **Object Inspector** and choose **FileSave** from your component.
2. Select **Events** tab, then choose **OnExecute** and press **Ctrl+Return** to create an event handler.
3. Right where the cursor is positioned (between { and }), type the following lines:

```
if (FileName == "untitled.txt")
   FileSaveAs1->Execute();
else
   RichEdit1->Lines->SaveToFile(FileName);
```

This code tells the text editor to display the **Save As** dialog box if the file isn’t named yet so the user can assign a name to it. Otherwise, it saves the file using its current name. The **Save As** dialog box is defined in the event handler for the **Save As** command. **FileSaveAs1 BeforeExecute** is the automatically generated name for the **Save As** command.
That’s it for the **File|Save** command.

11.9.1.15 Creating an event handler for the Save As command

When **SaveDialog's** Execute method is called, it invokes the standard Windows **Save As** dialog box for saving files. To create an event handler for the **Save As** command:

1. Go to **Object Inspector** and choose **FileSaveAs1** from your component.
2. Choose **Dialog** property, press **Tab** and then **Right** to expand its properties. **Dialog** references the **Save As** dialog box component and displays the **Save As** dialog box's properties.
3. Choose and set **DefaultExt** to "**txt**".
4. Choose **Filter** and press **Ctrl+Return** to display the **Filter editor**. In the **Filter editor**, specify filters for file types as in the **Open dialog** box.

    - In the first row of the **Filter Name** column, type '**Text files (*.txt)**". In the **Filter** column, type "***.txt**".
    - In the second row of the **Filter Name** column, type "**All files (*.*)**" and in the **Filter** column, type "***.***".
    - Click **OK**.

5. Set Title to "Save As".
6. Select Events tab, then BeforeExecute and press Ctrl+Return to create an event handler.
7. Right where the cursor is positioned in the Code editor, type the following line:

```
FileSaveAs1->Dialog->InitialDir = ExtractFilePath
(Filename);
```

8. The Events tab should still be displayed. Select OnAccept and press Ctrl+Return to create an event handler.
9. Where the cursor is positioned, type the following lines:

```
FileName = FileSaveAs1->Dialog->FileName;
RichEdit1->Lines->SaveToFile(FileName);
StatusBar1->Panels->Items[0]->Text = FileName;
```

10. Choose File|Save All to save your changes.
11. To see what the application looks like so far, press **F9**.
12. To return to design mode, press **Alt+F4**.

11.9.1.16 Creating a Help file

It's a good idea to create a Help file that explains how to use your application. C++ Builder provides Microsoft Help Workshop in the C:\Project Files\Borland\CBuilder6\Help\Tools directory, which includes information on designing and compiling a Windows Help file. In the sample text editor application, users can choose Help|Contents or Help|Index to access a Help file with either the contents or index displayed.
Earlier, you created HelpContents and HelpIndex actions in the Action Manager or Action List editor to display the Contents tab or Index tab of a compiled Help file. You need to assign constant values to the Help parameters and create event handlers that display what you want.

To use the Help commands, you'll have to create and compile a Windows Help file. Creating Help files is beyond the scope of this tutorial. However, you can download a sample rtf file "**TextEditor.rtf**", Help file "**TextEditor.hlp**" and contents file "**TextEditor.cnt**":

1. In Windows Explorer, from your **C:\Program Files\Borland\CBuilder6\Help directory, open B6X1.zip**
2. Extract and save the "**.hlp**" and "**.cnt**" files to your **Text Editor** directory.
   Note:
3. You can also use any "**.hlp**" or "**.cnt**" file (such as one of the C++ Builder Help files and its associated "**.cnt**" file) in your project. You will have to copy them to your project directory and rename them as "**TextEditor.hlp**" and "**TextEditor.cnt**" for the application to find them.

11.9.1.17 Creating an event handler for the Help Contents command

To create an event handler for the **Help Contents** command:

1. Go to **Object Inspector** and choose **HelpContents1** from your component.
2. Select **Events** tab, then choose **OnExecute** and press **Ctrl+Return** to create an event handler.

3. Right after where the cursor is positioned, type the following lines:

```
const static int HELP_TAB = 15;
const static int CONTENTS_ACTIVE = -3;
Application->HelpCommand(HELP_TAB,
CONTENTS_ACTIVE);
```

This code assigns constant values to the HelpCommand parameters. Setting HELP_TAB to 15 displays the Help dialog and setting CONTENTS_ACTIVE to -3 displays the Contents tab.

Note:
To get Help on the HelpCommand event, put the cursor next to HelpCommand in the editor and press F1.
That's it for the Help|Contents command.

11.9.1.18 Creating an event handler for the Help Index command

To create an event handler for the **Help Index** command:

1. Go to **Object Inspector** and choose **HelpIndex** from your component.
2. Select **Events** tab, then choose **OnExecute** and press **Ctrl+Return** to create an event handler.
3. Right after where the cursor is positioned in the text editor, type the following lines:

```
const static int HELP_TAB = 15;
const static int INDEX_ACTIVE = -2;
Application->HelpCommand(HELP_TAB, INDEX_ACTIVE);
```

This code assigns constant values to the HelpCommand parameters. Setting HELP_TAB to 15 again displays the Help dialog box and setting INDEX_ACTIVE to -2 displays the Index tab.
That's it for the Help|Index command.

11.9.1.19 Creating an About box

Many applications include an About box which displays information on the product such as the name, version, logos, and may include other legal information including copyright information.

You've already set up a Help About command in the Action Manager or Action List editor.
To add an About box:

1. Choose **File|New|Other** to display the **New Items** dialog box.
2. Select the **Forms** tab, select the **About Box** icon and press **Return**.
1. A predesigned form for an **About box** appears.
2. Select the form itself and in the **Object Inspector**, click the **Properties** tab and change its **Caption** property to "**About Text Editor**".
3. Go to "**About**" form (notice it is now called About Text Editor). To change each value on the form, press Tab and type the new value.

   - Change Product Name to "**Text Editor**".
   - Change Version to "**Version 1.0**".
   - Change Copyright to "**Copyright 2002**".

4. Save the **About box** form by choosing **File|Save As** and saving it as **About.cpp**.In the C++ Builder Code editor, you should have several files displayed: **Unit1.cpp**, **Unit1.h**, **About.cpp**, and **ActnRes**. You don't need the **ActnRes** unit but you can leave it there.
5. Click the **Unit1.cpp** tab and scroll to the top of the Code editor. Add an include statement for the **About** unit to **Unit1**. Choose **File|Include Unit Hdr** and then select **About** and click **OK**. Notice that `#include About.h` has been added to the top of the .cpp file.
6. Press **F12** to return to design mode. Go to **Object Inspector** and choose **HelpAbout** from your component.
7. Select **Events** tab, then choose **OnExecute** and press **Ctrl+Return** to create an event handler. Right where the cursor is positioned in the Code editor, type the following line:

```
AboutBox->ShowModal();
```

   This code opens the About box when the user clicks Help|About. ShowModal opens the form in a modal state, a runtime state when the user can't do anything until the form is closed.

8. Choose **File|Save All**.

11.9.1.20 Completing your application

The application is almost complete. However, you still have to specify some items on the main form. To complete the application:

1. Press **F12** to locate the main form.
2. Go to **Object Inspector** and choose **Form1** from your component.
3. Select the **Events** tab, choose **OnCreate** and press **Ctrl+Return**.
4. Right where the cursor is positioned in the Code editor, type the following lines:

```
Application->HelpFile =
ExtractFilePath(Application->ExeName) +
"TextEditor.hlp";
FileName = "untitled.txt";
StatusBar1->Panels->Items[0]->Text = FileName;
RichEdit1->Clear();
```

   This code initializes the application by associating a Help file, setting the value of FileName to untitled.txt, putting the file name into the status bar, and clearing out the text editing area.

5. Choose **File|SaveAll** to save your changes.
6. Press **F9** to run the application.

Congratulations! You're done.

11.10 Creating an MDI application

Aim of this subchapter: In this chapter you will learn what a Multiple Document Interface (MDI) is.

11.10.1 What is a MDI application.

We shall now explain and describe an MDI application

Besides Single Document Interfaces (SDI) it is also possible to create Multiple Document Interfaces (MDI), with more than one window.
We know that the windows of a user's interface can be minimized, enlarged, reduced to icons and moved. All this is possible when the windows reside in an environment containing other windows: desktop windows are the best known example.
The facility to contain more than one window marks the difference between an SDI and a MDI. An SDI application cannot contain other windows, in an MDI application the main window also called “parent” window contains one or more “child” windows.
The main window usually contains a menu bar and possibly tool bars. The child window moves only inside the main window. When an SDI or an MDI is reduced to icons, the icons will be visible on the Windows application bar; when an MDI child window is reduced to an icon, this icon will appear below the main MDI window.

11.10.2 How to create an MDI

To create an MDI we shall have to modify the property **FormStyle** of the forms under consideration. For the parent window you will select the value **fsMDIForm**, for the child windows **fsMDIChild**.
An example is the best way to show how to create an MDI.

11.10.2.1 Creating an MDI application.

You will now create a simple MDI application with the aim of displaying and saving bitmap images.
You will need to do the following:

1. create the main Form (parent MDI)
2. create a simple menu bar
3. insert the dialogue windows Open and Save as found on the menu bar
4. implement the code of the items of the above menu bar
5. create the child MDI form

6. run and test the program

Start by opening BCB as usual: you will find an empty form in front of you.

1. To create a parent Form use the properties of the default BCB empty form Form1

   - **Name** = “**MainForm**”;
   - **Caption** = “**View bitmap**”;
   - **Height** = **600**;
   - **With** = **800**;
   - **FormStyle** = “**fsMDIForm**” (select from the list).

2. to insert a menu bar on the parent Form you will
   - position a **TmainMenu component on the MainForm**
   - build the menu bar (items property) as follows:
     + **&File**
     + **&Open**
     + **&Save As**
     + **&Window**
     + **&Tile**
     + **&Cascade**
     + **&Arrange All**

3. To insert the dialogue windows Open and Save on the menu bar:
   - add a component **TOpenDialog** to **MainForm**;
   - insert a component **TsaveDialog** to **MainForm**;
   - modify the property **Name** of "**OpenDialog1"** to “**OpenBitmap**”;
   - modify the property **Title** of "**OpenDialog"** to “**Open Bitmap**”;
   - modify the property **Name** of "**SaveDialog1"** to “**SaveBitmap**”;
   - modify the property **Title** of "**SaveDialog"** to “**Save Bitmap**”.

4. To implement the code of the items of the menu bar, for each item:
   - reach **OI (Object Inspector),**
   - select the component on which to operate,
   - **Ctrl+Tab** to open the window **Events**
   - reach the event **OnClick** and press **Ctrl+Enter** to open the window of code“**Unit1.cpp**” (you will find the cursor in the correct place for inserting the following fragments code)
     In the empty line between the brackets of the event **OnClick** of the menu item **Open1** insert the following code:

```
if (OpenBitmap->Execute())
  {
```

```
  TChild* child = new TChild(this);
  child->Image->Picture->LoadFromFile(OpenBitmap-
>FileName);
  child->ClientWidth = child->Image->Picture-
>Width;
  child->ClientHeight = child->Image->Picture-
>Height;
  child->Caption = ExtractFileName(OpenBitmap-
>FileName);
  child->Show();
  }
```

In the empty line between the brackets of the event **OnClick** of the menu item **SaveAs1** insert the following code:

```
  TChild* child =
dynamic_cast<TChild*>(ActiveMDIChild);
  if (!child) return;
  if (SaveBitmap->Execute());
  {
  child->Image->Picture->SaveToFile(SaveBitmap-
>FileName);
  }
```

In the empty line between the brackets of the event **OnClick** of the menu item **Tile1** insert the following code:

```
  {
  Tile();
  }
```

In the empty line between the brackets of the event **OnClick** of the menu item **Cascade1** insert the following code:

```
  {
  Cascade();
  }
```

In the empty line between the brackets of the event **OnClick** of the menu item **ArrangeAll1** insert the following code:

```
  {
  ArrangeIcons();
  }
```

5. To create the child Form MDI:
   - insert a new Form in the project by selecting **File\New Form**;
   - assign the value "**Child**" to the property **Name** (you can ignore the property **Caption** since the name of the displayed image will appear automatically at runtime).
   - assign the value "**fsMDIChild**" to the property **FormStyle** select from the list).
   - The Form will now be treated as child form MDI

   You will now realise that **OI (Object Inspector)** will allow you to operate only on the active form and on its components; to move from one form to the other:

   - open View\Project Manager;
   - from the menu tree select the form on which you want to operate
   - press Enter

   To insert and set a component **TImage** in the Form "**Child**" (this component will display the bitmaps)

   - insert a component Timage on the Form Child;
   - assign the value "Image" to the property Name;
   - assign the value "True" to the property NameStretch
   - assign the value "**alClient**" to the property **Align**

   You will now do the following operations:

   - choose File\Save and save the unit of the form Child naming it "MDIChild";
   - go to the code editor (F12), with Ctrl+Tab select the page related to Unit1.cpp (MainForm);
   - open File\Include Unit Hdr..., select MDIChild and press Enter (compiler will now be able to refer to the object **TChild)**.

7. Now, you can save the entire project ; the elements not saved so far will keep their default names; press F9 to compile and execute the program. Now you can try to open and save the bitmap images and use the options to display the windows.

The above program gives you an idea of the functioning of a MDI application. There are however some problems involved. For example, when you try to open a non graphic file you will generate an access violation; in addition an empty child window will open when you start the program; also if you close a child window it will actually be reduced to an icon.

Chapter 12: Advanced Module 2 Data Base development with Borland C++ Builder

Aim of this chapter: In this chapter you'll learn what a relational databnase is, and how to develop such a database with Borland C++ Builder 5.0

12.1 Concept of Relational Databases

Aim of this subchapter: **The student should know how the principles of rational databases. He/she should be able to design a database beginning with the analysis of the real objects and how to map them into tables.**

In this chapter we will learn to design a relational database as a picture of real objects and how they are related. We will learn about primary, secondary and foreign key and their functions in sorting and the uniqueness of records in a table. Also we will see, that there are three ways to relate tables, called 1:m, m:m, and 1:1-relations

12.1.1 Definition of a Databases

- What is a database, a table, a record?
- Tables, queries, forms, reports
- Fields, data-types
- What are entities, attributes, tuples

12.1.2 Definition of Relational Databases

- Definition of a relational database (master-detail)
- Difference between relational and hierarchical databases
- Examples of relational databases
- Design of relational databases
- Normalization

12.1.3 Keys

- Use of keys
- Kinds of keys
- How to define keys

12.1.4 Kinds of Relationships

- The purpose of relationships
- Kinds of relationships
- Referential integrity

12.2 Further exercises:

1. Design a database for telephone-numbers and emails. A person can get arbitrary telephone-numbers and email-addresses.
2. Design a database for renting of music-cds (customers, cds, renting)
3. Design a database for the sales of music-cds (customers, cds, orders, order-details, accounts)
4. Design a database that holds the data of a school (teachers, pupils, classes, school subject, …)

12.3 ODBC, BDE Administrator and Database-Desktop

***Aim of this subchapter:* The student should know how to make databases available to applications created by Borland C++ Builder 5 (BCB5). He/she should know how to work with the BDE Administrator. Also he/she should know how to create a database with the Database-Desktop. He/she also should know how to create connections to databases through ODBC.**

In this chapter we will start with the Database-Desktop and create tables and keys. Then we will create a connection to this databases using the BDE Administrator. We also will use the ODBC to create connections to databases.

12.3.1 Creating Databases

- Getting to know the Database-Desktop-application.
- Create tables, define fields, set the data type.
- Define keys
- Set referential integrities between tables

12.3.2 Create Connections to Databases

- Getting to know the BDE Administrator.
- Creating a new connection to a database

12.3.3 Create Connections to Databases using ODBC

- Definition of ODBC-Drivers
- Native drivers versus ODBC-Drivers
- Create a new ODBC-connection with the ODBC-Administrator of the Control-Panel

12.4 Further exercises:

5. Use the exercises of the previous chapter to create tables and connections to the databases.

12.5 Access to a Database

***Aim of this subchapter:* The student should know how to get access to databases in Borland C++ Builder 5 (BCB5). He/she should know about the components used to define access to a single table database and to databases in a relationship-model.**

In this chapter we will start with an overview of the data-access-components of BCB5. We will see how the access is managed through the components TTable or TQuery and TDataSource.
Finally we will show how to display data in a form.

12.5.1 Access to a Database through TTable and TDataSource

- What components have to be used to get access to a database
- Add a TTable-component to a form.
- Create a connection to a database.
- Set the properties of a table-object.
- Add a TDataSource-object to a form.
- Create a connection to a Table-object.
- Set the properties of a DataSource-object.
- Create calculated and lookup-fields (double-click on a table or query-object)

12.5.2 To Show Data in a Form

- Examine data-controls
- Use TDBText, TDBEdit, TDBMemo to show data.
- Use TDBListbox and TDBCombobox to show data.
- Use TDBRadioGroup and TDBCheckBox to show data.
- Use a TDBGrid to show data.
- Navigate between records.
- Add, edit and delete records.

12.5.3 Use a Data-Module

- Definition of a Data Module
- Use of a Data Module with more than one form

12.6 Further exercises:

Use the exercises of previous chapters to create forms for single tables.

12.7 Creating Queries with TQuery

***Aim of this subchapter:* The student should learn to create queries in BCB. He/she also should learn how to create SQL-statements.**

In this chapter we will deal with the component TQuery, which is used to create Queries. Because SQL is used with TQuery we will start with a short introduction to SQL and learn about the statement. SELECT

12.7.1 Definitions

- Definition of SQL
- Use of TQuery to introduce SQL-statements to BCB

12.7.2 Introduction to SQL: SELECT

- The Syntax of SQL.
- Simple queries using SELECT.
- The keywords WHERE and ORDER.
- Using JOIN to define queries with multiple tables.
- Using parameters in query.

12.7.3 Use of TQuery to Create Queries

- Adding a TQuery-component to a form.
- Properties of a query-object.
- Defining SQL-statements for a query-object.
- Using TQuery to define master-detail forms.

12.8 Further exercises:

Use the exercises of the previous chapters to create SQL-statements and master-details forms.

12.9 Designing a Report

***Aim of this subchapter:* The student will learn how to print data from one or more tables, and how to sort and group it.**

In this chapter we will see how to prepare data for printing with the report-component. We will work with some more components, so we can manage data-display in a report.

12.9.1 Definitions

- Definition of a report and how to use it.
- Components which have to be used for a report.

12.9.2 The Component TQuickReport

- Add a TQuickReport to a form.
- Properties of a quick-report-object.
- The TQRPreview-component for previewing
- The TQRPrinter-component for printing

12.9.3 Components to Display Data in a Report

- Place components on a report: TQRBand
- Show fixed text: TQRLabel and TQRMemo
- Show the content of fields: TQRDBText
- Calculate in a report: TQRDBCalc
- Page-numbers, date and time: TQRSysData
- Master-detail-databases: TQRDetailLink
- Grouping records in a report: TQRGroup

12.10 Further exercises:

Use the exercises of the previous chapters to create reports.

12.11 Update SQL

***Aim of this subchapter:* The student will learn, how to change data in tables with master-details relationships.**

First we will learn about the SQL-statements update, delete and insert and how to use them. Then we will introduce the updateSQL-object which is used to work with these SQL-statements.

12.11.1 The Instruction UPDATE, DELETE, INSERT

- Using the SQL-statements UPDATE, DELETE and INSERT

12.11.2 The component TUpdateSQL

- Use TUpdateSQL to change, delete or insert data
- Add a TUpdateSQL-component to a form
- Define the SQL-statement

12.12 Further exercises:

Use the exercises of the previous chapters to update data in a master-detail-relationship.

Chapter 13: Examples for the final project after the Basic section 1:

Examples for exams:

General hints:

- The examples are of different difficulty
- Some examples have additional features which may added or omitted to the problem definiton.

13.1 Example 1: Hey what's your new name?

The following is an excerpt from a children's book, "Captain Underpants and the Perilous Plot of Professor Poopypants," by Dave Pilkey:

Use the third letter of your first name to determine YOUR new first name:
A = poopsie
B = lumpy
C = buttercup
D = gidget
E = crusty
F = greasy
G = fluffy
H = cheeseball
I = chim-chim
J = stinky
K = flunky
L = booger
M = pinky
N = zippy
O = goober
P = doofus
Q = slimy
R = loopy
S = snotty
T = tulefel
U = dorkey
V = squeezit
W = oprah
X = skipper
Y = dink
Z = zsa zsa

Use the second letter of your last name to determine the first half of your new
last name:

A = diaper
B = toilet
C = giggle
D = burger
E = girdle
F = barf
G = lizard
H = waffle
I = cootie
J = monkey
K = potty
L = liver
M = banana
N = rhino
O = bubble
P = hamster
Q = toad
R = gizzard
S = pizza
T = gerbil
U = chicken
V = pickle
W = chuckle
X = tofu
Y = gorilla
Z = stinker

Use the fourth letter of your last name to determine the second half of your new last name:
A = head
B = mouth
C = face
D = nose
E = tush
F = breath
G = pants
H = shorts
I = lips
J = honker
K = butt
L = brain
M = tushie
N = chunks
O = hiney

P = biscuits
Q = toes
R = buns
S = fanny
T = sniffer
U = sprinkles
V = kisser
W = squirt
X = humperdinck
Y = brains
Z = juice

Tasks:

1. Create a form where somebody can type his or her name
2. Clicking on a button the new name should be created. Each click of the same button should add the next part of the new name.
3. The form should contain a “new”-button, so that somebody could type a new name.
4. Then there should be a second form, that can be opened by a button from the first form. In this form the lists of values above should be displayed in three list-boxes.
5. Beside the buttons all commands should be reached by a menu and with shortcuts.

13.2 Example 2: Hangman

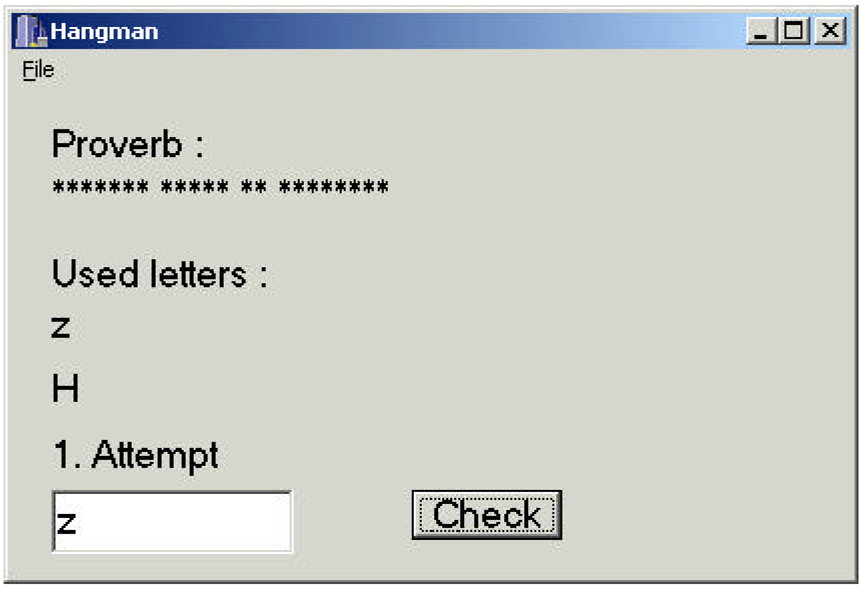
Hangman is a game to guess words or proverbs:
A word or proverb is randomly chosen from a list. All letters are converted to dots and each space remains a space. These dots and spaces are shown in a field in the form. Then somebody can enter a letter in an edit-field. Clicking on a button it is checked, to see if this letter is in the searched word or proverb. If it is, then the corresponding dots in the first field are changed to this letter. If the letter is not found, there should be a message and one letter is added to the word “HANGMAN”.
The game ends when either the word is guessed or the word HANGMAN is completed and there are still some dots left.
The commands “New game” and “Exit” should be called by a menu and with shortcuts.

Note: Use the functions randomize() and random(limit) to choose a word or proverb.
Some additional features:

- The attempts are counted and shown in the form.
- A list of letters that have already been used is shown.
- A letter may not be used a second time. If somebody does that a

warning is shown. This attempt may be counted as right or wrong.

The following picture shows what the form could look like:



Proverb :
******* ***** ** ********

Used letters :
z
H
1. Attempt
z

13.3 Example 3: Hanoi Towers

In this game we have three towers made of disks of different size (we may represent them in our examples as numbers). You can move the disks from one tower to another, but you should never place a bigger disk on top of a smaller disk (in our case a bigger number above a smaller number). You can only move one disk at a time.
You should move the whole tower of disks from one place to another.
Tasks:
- Create a form with three fields (e.g. list-boxes).
- At the start of the game, you will be asked how big the tower should be.
- With buttons you can move the “disks” from one tower to another, one at a time.
- If the move is not allowed a warning should be shown.
- At the end a message-box for the success should be shown.
- Additional feature:
- A counter should count the time it takes to finish the game. For this feature use the TTimer-component.

13.4 Example 4: Calculator

The task is to design a calculator:
Tasks:
- Add numbers with buttons and by shortcut
- A field, that display the numbers and calculation
- Calculation for plus, minus, multiply and divide

- Percent-calculation.
- Button for new calculation (C-button)
- Button for clear display, because of a wrong number. The calculation is not stopped (CE-button)
- Button for remove the digit added last.
- Additional features:
- Calculation with brackets.
- Calculation of power and root.
- Memory for numbers
- Example 5: Play music
- You have a list of music tracks, representing *.wav files (some of them are installed during the windows-installation).
- Somebody will choose a track from a list. There are buttons for start, stop, and pause the track.
- There also should be the opportunity to go to the next or previous track.
- Use the TMediaPlayer-component to play the tracks.
- The commands may be activated by buttons, by a menu or by shortcuts.
- Additional features:
- One could create his or her own list of tracks using the existing tracks which can be played.
- Shuffle play: play tracks randomly. Use the function randomize() and random(limit) for this.
- Repeat: Play a track over and over again.

13.5 Example 6: Speak numbers

For this example first you have to record someone speaking numbers and words, eg. one, two, three, …, eighteen, nineteen, twenty, thirty, …, hundred, and, thousand, million etc as wave-files.
In a edit-box in a form somebody can type a number. On clicking on a button the number is spoken by adding the corresponding wave-files.
Use the TMediaPlayer-component to playback the numbers.

Chapter 14: Examples for the final project after the Basic section 2:

14.1 Example 1: CD-Database

Create a media player which may store the title of a music-CD and the titles of the track in a file.
Features:

- Identify a music-CD
- Read the number of tracks
- Play a track, stop, pause, next, previous
- Random play of tracks, custom defined list of tracks, repeat a track, repeat a whole music-CD
- Add and edit title and performer of a CD.
- Store this information in a file
- If a Music-CD is used, the player should automatically check, if there is already an entry in the “database” for this CD

Implement all the commands concerning playing one or more tracks as a class. Implement as a second class all the commands concerning the “database” of CD-title, performer and track-titles, including file-open, -save and –close.

14.2 Example 2: Database for Addresses

Create a “database” for people and their addresses. Use a simple text-file to store the addresses.
Features:

- Create a form to display, edit, delete, and add new addresses.
- Create a file-access to store the addresses.
- The user may use different address-files.

Implement the file-access as a class. This class should also be able to identify a address in the file and display it on the form.
Additional Feature:

- A person may have more than one address. So names and addresses have to be stored in different files. Use ID-numbers to identify which address belongs to which person.
-

Implement the file-access as a base class. Create two derived classes, one for the access to the people-file and one for the access to the address-file.

14.3 Example 3: Calendar

Create a simple calendar. The calendar should show a whole month. The user should be able to choose the year and the month. Also she or he should be able to move to the next or previous month. By default the calendar should show the current month of the current year.
The calendar should show Saturdays and Sundays in a different colour from the other working days.

Create the calendar ion the basis of the StringGrid-component.
Implement the calculation of year, month, Saturdays and Sundays as a class.
Additional Features:

- The user may enter dates, that are stored in a file. The date should include begin, end, and subject.
  You may show the dates in an additional grid, that is updated each time the user selects another day in the calendar-grid. The calendar-grid should show that there are one ore more dates this day.
  You may also show the dates in a second form, that is opened if somebody double-clicks on the day or presses Control+Return.
- Implement holidays. There isare different kinds of holidays: church festivals partly depends on the Easter. There are function to calculate the weekend, where Easter lies in a particular year. They are widely published in the internet. The other church festivals and bank holidays have a fixed day. Gice the user the chance to fix new holidays and store them in a file. Implement the file-access as a class. Implement the calculation of holidays as a derived class with base classes for the initial calculation of the calendar and the file-access.

14.4 Example 4: Hangman

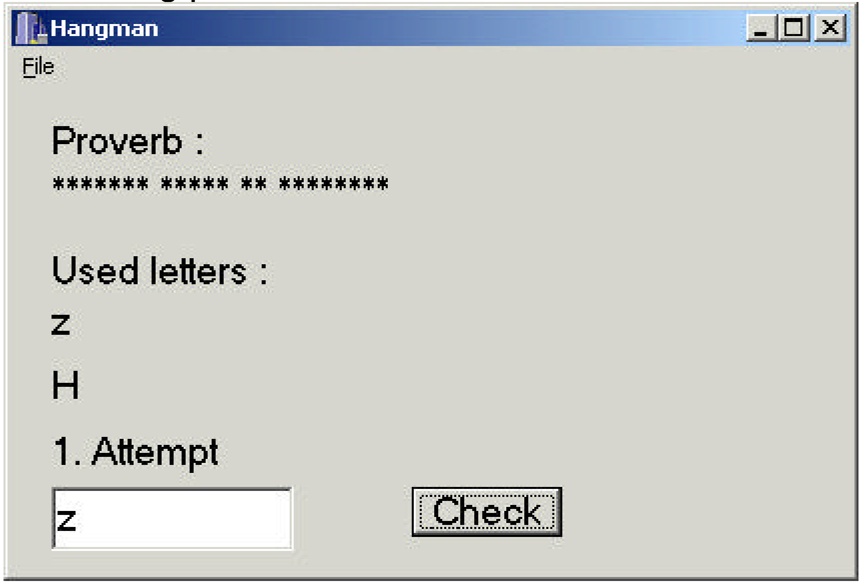
Hangman is a game to guess words or proverbs:
A word or proverb is randomly chosen from a list. All letters are converted to dots and each space remains a space. This dots and spaces are shown in a field in the form. Then somebody can enter a letter in an edit-field. On clicking on a button it is checked, to see if this letter is in the word or proverb. If it is, then the corresponding dots in the first field are changed to this letter. If the letter is not found, there should be a message and one letter is added to the word "HANGMAN".
The game ends when either the word is guessed or the word HANGMAN is completed and there are still some dots left.
The commands "New game" and "Exit" should be called by a menu and with shortcuts.

Note: Use the functions randomize() and random(limit) to choose a word or proverb.
Additional tasks:

- The attempts are counted and shown in the form.
- A list of letters which have already been used is shown.
- A letter may not be used a second time. If somebody does that a warning is show. This attempt may be counted as right or wrong.
- The word or proverbs should be stored in a file. When somebody starts a new game, the file should be opened and one of the entries randomly chosen.
- In a second form the user can edit the words and proverbs of the file. They are displayed in a list. If the user marks one of them, he or she can edit it and save the changes. Also he or she may add new words and proverbs. On close of form the changes are stored in a file.

Main tasks:
- Implement the file access as a class: read, write, file-access, etc.
- Implement all steps concerning the game as a class: new proverb, check letter, list of used letters, number of attempts, etc.

The following picture shows how the form could look like:



Proverb :
******* ***** ** ********

Used letters :
Z
H

1. Attempt

Z

14.5 Example 5: Calculator

The task is to design a calculator:
Tasks:
- Add numbers with buttons and by shortcut

- A field, that displays the numbers and calculation
- Calculation for plus, minus, multiply and divide
- Percent-calculation.
- Button for new calculation (C-button)
- Button for clear display, because of a wrong number. The calculation is not stopped (CE-button)
- Button for remove the last added digit.
- Calculation with brackets.
- Calculation of power and root.
- Memory for numbers

Implement input, calculation and memory in three different classes

Chapter 15: Examples for the final project after the Advanced Module 1 (VCL):

15.1 Example 1: A Simple Text-Editor

Design a text-editor. The editor should be able to edit simple text.
Features of the editor:

- Open and close one text file (SDI).
- Save and Save As.
- Select text
- Delete, copy and move text (cut, copy, paste)
- Define the length of a line

Implement the file-access and the file-edit as different classes
Additional features:

- Show, in which row and column the cursor is resting
- Count the number of words and the number of characters in the file
- Undo the last edit-command
- Print the file
- Page setup

Implement printing as a class.

15.2 Example 2: A Complex Text-Editor

Design a text-editor. The editor should be able to edit simple text.
Features of the editor:

- Open and close more than one file (MDI).
- Save and Save As.
- Select text
- Delete, copy and move text (cut, copy, paste), also from one opened file to another.
- Define the length of a line

Implement the file-access and the file-edit as different classes
Additional features:

- Show, in which row and column the cursor is resting
- Count the number of words and the number of characters in the file
- Undo the last edit-command
- Print the file

- Page setup

Implement printing as a class.

15.3 Example 3: A Simple Spreadsheet-Editor

Design a spreadsheet-editor. The editor should be able to edit simple entries, identify number- and text-entries, do simple calculations (plus, minus, multiply, divide).
Features of the spreadsheet-editor:

- Open and close one file (SDI).
- Save and Save As.
- Enter and edit numbers and text in a cell
- Do simple calculations: plus, minus, multiply, divide
- Show errors in a formula as errors.
- Delete, copy and move text (cut, copy, paste).

Use the StringGrid-component for the spreadsheet.
Implement the file-access, the file-edit, and the calculation as different classes.
Additional features:

- Show, in which row and column (cell) the cursor is resting
- Undo the last edit-command
- Print the file
- Page setup
- Define a function “Sum”, which can add the contents of selected cells.

Implement printing as a class.

15.4 Example 4: A Complex Spreadsheet-Editor

Design a spreadsheet-editor. The editor should be able to edit simple entries, identify number- and text-entries, do simple calculations (plus, minus, multiply, divide).
Features of the spreadsheet-editor:

- Open and close more than one file (MDI).
- Save and Save As.
- Enter and edit numbers and text in a cell
- Do simple calculations: plus, minus, multiply, divide
- Show errors in a formula as errors.
- Delete, copy and move text (cut, copy, paste), from one file to another.

Use the StringGrid-component for the spreadsheet.
Implement the file-access, the file-edit, and the calculation as different classes.
Additional features:

- Show, in which row and column (cell) the cursor is resting
- Undo the last edit-command
- Print the file
- Page setup
- Define a function “Sum”, which can add the contents of selected cells.

Implement printing as a class.

Chapter 16: Examples for the final project after the Advanced module 2 (Databases):

16.1 Example 1: CD-Rom Database

Create a media player which may store the title of a music-CD and the titles of the tracks in a file.
Features:

- Identify a music-CD
- Read the number of tracks
- Play a track, stop, pause, next, previous
- Random play of tracks, custom defined list of tracks, repeat a track, repeat a whole music-CD
- Add and edit title and performer of a CD.
- Store this information in a database, having two files, one for the CD-titles and performers and a second for the track titles.
- Use the dBase, Paradox or Access-format for the database.
- Work out a database-design.
- Create the tables.
- Create a form to show the content for the database.
- If a music-CD is used, the player should automatically check, if there is already an entry in the database for this CD

Implement all commands concerning playing one or more tracks as a class.
Implement access to the database as a second class.

16.2 Example 2: Address-Database

Create a database for people and their addresses. People may have more than one address, an address may be connected to more than one person.
Use the dBase, Paradox or Access-format for the database.
Features:

- They user may connect existing people and addresses.
- Work out a database-design.
- Create the tables.
- Create a the form.

Implement access to the database including editing of records as a class.

16.3 Example 3: To-Do-List

Create a database for to-dos. A to-do may include subject, begin, end,

done, done-date, estimated amount of time to finish the to-do, used amount of time, percent of completion etc.
Features:

- Create a form to display, edit, delete, and add new to-dos.
- To-dos should be categorised into future, actual and should be finished by now.
- The user may choose, to view to-dos of today, this week, this month or all
- Finished to-dos should only be displayed in an extra list “finished”.
- Use the dBase, Paradox or Access-format for the database.
- Work out a database-design.
- Create the tables.

Implement access to the database including editing of records as a class.

16.4 Example 4: Organize Music-Collection

Create a database to organize a music-collection. Beside the title, tracks and performers there should be fields for considered the kind of device (CD, LP, MC, etc.), the kind of music (classic, pop, country, etc.) and similar categories.
Features:

- Create a form to display, edit, delete, and add new music.
- All category-lists should be editable by the user.
- Use the dBase, Paradox or Access-format for the database.
- Work out a database-design.
- Create the tables.

Implement the access to the databases including editing of records as one or more classes (base-classes, derived classes).