



METIER

Graduate Training Course in „Ecological Modelling“

22 May – 2 June 2008
Leipzig & Bad Schandau

*Material of the Precourse
„Introduction into Programming with
Borland C++ Builder“*

Course Team:

Michael Müller
Florian Hartig

Sten Zeibig
Oliver Jakoby

Helmholtz Centre for Environmental Research – UFZ, Leipzig
Department of Ecological Modelling



Version: 20.05.2008

Contact: tamara.muenkemueller@ufz.de, sten.zeibig@ufz.de, oliver.jakoby@ufz.de,
michael.mueller@ufz.de

Helmholtz Centre for Environmental Research - UFZ
Department of Ecological Modelling
Permoserstraße 15
04318 Leipzig, Germany
Tel.: +49 341 1279

Contents

1	Introduction	3
1.1	How to use this script	3
1.2	Why the Borland Builder and C++	3
2	Starting to work with the Borland Builder	5
2.1	First steps	5
2.2	How to save and copy a program	10
2.3	Comparisons	11
2.4	Memo	12
2.5	Loops	13
2.6	Random numbers	16
2.7	Charts	19
2.8	Global and local variables	20
2.9	Functions	21
2.9.1	Functions without arguments	21
2.9.2	Functions with arguments	22
2.9.3	Functions with pointer arguments	22
2.9.4	Functions with a <i>return</i> -statement	23
2.9.5	Combinations:	23
2.10	Output files: Writing in a text file	23
2.11	Handling errors or things you should check before asking for help	24
2.12	Changing your program while it is running: pause, stop, exit	24
3	Examples	27
3.1	Sheep farm	27
3.2	Predator prey	30
3.3	Carnivorous plant	35

1 Introduction

1.1 How to use this script

Following with the course structure we want to guide you through learning some basic programming skills step by step. The text is organised via tasks, hopefully helpful comments and solutions. We suggest to answer the questions and tasks in this script (following the ‘**Task:**’) before going on to the answers and solutions (‘**Solution:**’). Sometimes you will find some helpful comments (‘**Help:**’) following the questions. In between you will find some Tips (‘**Tips**’) and lots of Does & Don’ts. These are suggestions which might help you to avoid some tricky traps. Some of these suggestions seem to be unnecessary for simple coding. However, if you are strict in following these ‘rules’ you will have it much easier later on. Before we start with the course itself we’ll give some reasons for our choice of the used programming language and environment. You are very welcome to give criticism, comments and suggestions concerning this manuscript which may help us to improve it. We hope you will enjoy working with this script!

This script is based on the Winter School on Ecological Modelling which is a yearly event and organized by the Department of Ecological Modelling, UFZ (<http://www.winterschule.ufz.de>). Great effort has been put into the former winter schools and many PhD students and colleagues contributed to their success. We would like to thank everybody who was involved as this script would not have been possible without all the preparatory work. We also thank the volunteers for proof reading the first draws and last but not least the winter school attendees for their suggestions on how to improve the course. Now, we can profit from all this for the METIER-course on ecological modelling.

1.2 Why the Borland Builder and C++

Why are you going to learn C++? Certainly, other programming languages exist, that are easier to learn. So what makes C++ a good choice?

C++ is :

- Fast. With today’s compiler’s, C++ compiles to 95% of the speed of hand-written assembly.
- Efficient. Using C++ specific techniques (pointer) one may write very efficient program code.
- Applicable for IBM. The object oriented paradigm of C++ is well suited for implementations of individual based models (IBM), which are often used in theoretic ecology.

- Widespread. Most applications are written in C++. Also, many (also free) compilers are available.

See also: <http://www.bergen.org/AAST/Projects/Cpp/why.html>.

These are good reasons for using C++. But note: The last two advantages do not hold for all compilers. Some of them (e.g. Borland Builder) provide compiler specific libraries. Code using these libraries is not compilable with other compilers. Nevertheless we decided to use it for two very good reasons.

The Borland Builder (at least today) is:

- User-friendly. It is one of the handiest coding environment.
- Easy. Graphical elements can be easily applied by drag and drop.

The C++ Borland Builder is a popular rapid application development environment produced by Borland for writing programs in the C++ programming language. C++ Builder includes tools that allow true drag-and-drop visual development, which makes programming much easier at the beginning.

2 Starting to work with the Borland Builder

2.1 First steps

1. Open the Borland Builder.
2. Create a new Project using File/New/VCL Forms Application - C++Builder.
3. Save your code using 'File/Save all'; you need to save different files for one program or project (you will automatically be asked to save different files). Always save everything in one folder, and only save one program in one folder! You will get into trouble if you mix up several projects in one folder. Avoid it!
4. Now we can start: One important button is the green play button (see the red circle in figure 2.1), press this button.
5. What you can see now: After pressing the play button your program starts. It is a very simple and boring one because it does nothing – but it is a real program. Close your program now (by using 'CTRL + F2' or pressing the cross in the upper right corner). You will again see the template for the interface of your program (called 'Form1'; you can recognise the template by the dotted background and the running program by the plain grey background). Via this template you can design the interface of your program.

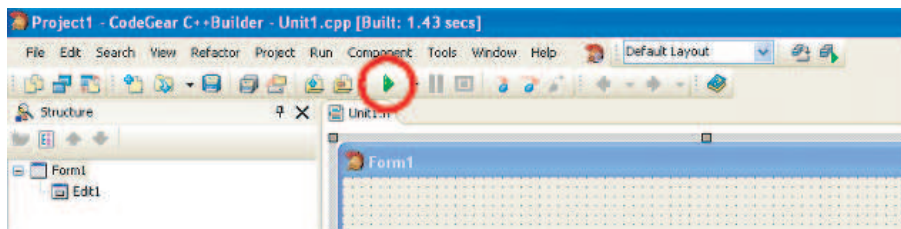


Figure 2.1: Screen shot Borland C++Builder. Pressing the green button (see red circle) makes the Builder compile and run your program.

This is boring! What else can we do? Something should happen or change...

Take an 'TEdit field' (see red circle in figure 2.2) out of the 'Standard category' (see green circle in figure 2.2) of the component palette and move it onto your form. You can change properties of this edit field in the object inspector. For example you can change the text showed in the edit field. For this purpose you have to change the property 'Text' in the object inspector, which you will usually find on the left next to your form (if you don't see the object inspector on the left

hand side of your screen, open it via 'View/Object inspector'). You can also change the property 'Width' or the property 'Caption'. Try it! Let your program run again (green play button) and see what happens.

Task: Try to change the color of the edit field on your form!

Solution: You can change this property with the object inspector. Here you define the properties your interface will have when the program starts. Note: These changes do not appear in your code.

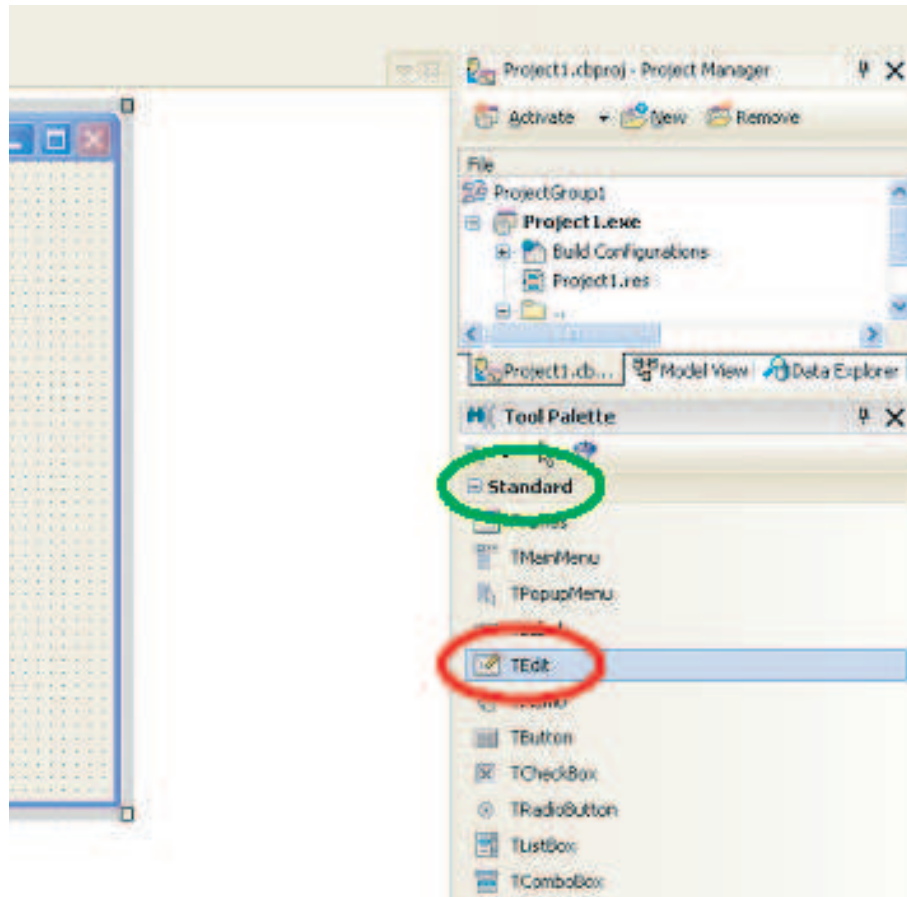


Figure 2.2: Screen shot Borland C++Builder. The 'Tool Palette' provides many items, which may be selected and inserted into your form. E.g. The 'TEdit' field (red circle) out of the 'Standard' category (green circle)

What else do we want? Something should happen. We need commands or buttons that we can click – or a menu from which we can choose something.

Take the 'OK-button' out of the standard category of the tool palette and move it onto your form. Double click this button. You will jump into the 'Unit1.cpp' window and more exactly into a function. All code you will write into this function (within the curly brackets) will be

executed when you press this button later on (while the program is running).

Tip: You can change between 'Unit1.cpp' and 'Form1' using 'F12'.

Task: Build a program that changes the text in the Edit field to 'Hello World' when you press a button.

Tip: You can access properties of an object via the arrow operator ('->'); you can assign text with the equal operator ('='); always use the semicolon (';') at the end of every command!

Solution:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Edit1 -> Text = "hello_world";
}
```

Run the program (green button). Congratulations, you built your first useful program!

What you see in the listing above is called a *function*. The first line is called its *head*. Although it looks cryptic and somehow weird there are (of course) rules behind it which will be explained a bit more detailed later. What the function does is defined in its *body*, which is written between the curly brackets. Thus the simple scheme behind is:

```
function head
{
    function body
}
```

Task: Build a program that allows you the following: you can write text in one edit-field, press the button and then this text appears in a second edit field.

Tip: Remember that you want to access the 'Text' property of the first edit field.

Solution: Create another edit-field (on the form) and write the following code (in 'Unit1.cpp'; get there by double clicking the button on the form):

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Edit2 -> Text = Edit1 -> Text;
}
```

Tip: If you ever want to delete a button from your form: First, delete the function's body; secondly, mark the button and delete it from the form. If you proceed like this, the Borland C++Builder will remove the function head and –which is important– automatically also all the associated stuff, which we did not talk about yet. If you remove the whole function yourself, you will have to do this manually. As long as you do not know exactly how to handle this –and you are probably not, otherwise you would not read this script– this is NOT RECOMMENDED!

Task: Build a program that allows you the following: You can write a word in an edit field and another word in another Edit field and when you press a button both words appear in a third edit field.

Solution:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Edit3 -> Text = Edit1 -> Text + " " + Edit1 -> Text;
}
```

Tip: You can use double slash (//...) for comments in one line and /*...*/ for longer comments. Use comments in your program generously to help yourself remembering what you meant by writing this code, not to speak about other users. Example:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //Edit1 -> Text = "hello world"; no more need for that
    Edit2 -> Text = Edit1 -> Text;
    /*You can access on properties of an object via
    the arrow operator (->); you can assign text with
    the equal operator(=); write text in parentheses ("");
    always use the semicolon (;) at the end of every command!*/
}
```

Task: Build a calculator which allows you to sum up two numbers.

Help: You will need the function StrToInt() for that. Find out what the function StrToInt() does: Mark the function and then press 'F1'. This help also works for edit fields, etc.

Solution: StrToInt() is a Borland function that changes strings (text) into integers. Since we want to get a string as output we should also change the whole result of the calculation back

to 'String'. As this is –in a mathematical sense– a unique operation it works also automatically, because the Borland compiler knows what to do. However, it is better to do it explicitly because then you are sure what is going on in your program. (Programming is not only about saying a machine what to do, but also about thinking in structures and communicate this to other people. It is easier to do this as explicitly as possible. Everything else is showing-off or mathematics or both.)

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Edit3 -> Text = IntToStr(StrToInt(Edit1 -> Text) +
                            StrToInt(Edit2 -> Text));
}
```

Tip: Even if you only change little things in your program let your program run every here and then. It is really annoying to find out after one hour programming that you created an error without having a clue when and where something went wrong!

You will learn how to use variables now. This allows you to structure your code much more effectively – especially if it is a more complex project.

Task: Build a calculator which allows you to divide two numbers.

Solution: Step1

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int var1 = StrToInt(Edit1 -> Text);
    int var2 = StrToInt(Edit2 -> Text);
    int frac = var1 / var2;

    Edit3 -> Text = IntToStr(frac);
}
```

Task: What is wrong with this program? Test it, do you find the error?

Solution: Everything behind the comma is cut since 'frac' is an integer variable! What you need is a variable of type 'double', which allows for real numbers (well, in fact double only simulates real numbers, because nothing is infinite in a machine). The type 'float' is also possible, but uses less memory and is therefore less accurate.

Even if 'frac' is a variable of type double, the program does something wrong. And this is one of the traps of C++ which is not intuitive at first sight: Whenever you divide two integer numbers the compiler creates an integer result (the 'true' result rounded down). This is the complement of the modulo-operation. You can escape this by converting one of the integers into double (write e.g. 23.0/8 or 23/8.0 instead of 23/8).

Tip: Sometimes you will need to convert an integer variable into double. For this purpose one can just write a '(double)' in front of the variable:

```
...  
  
int var1 = 23;  
int var2 = 42;  
  
double frac = (double) var1 / var2;  
...
```

Task: Find out how to change a string into a double (or float) using the help for `StrToFloat()`. Click on the 'See Also' button within the help file. Then 'repair' your program.

Solution: Step2

```
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    double var1 = StrToFloat(Edit1 -> Text);  
    double var2 = StrToFloat(Edit2 -> Text);  
  
    double frac = var1 / var2;  
    Edit3 -> Text = FloatToStr(frac);  
}
```

Don't forget to save your program!

2.2 How to save and copy a program

Now we will start a new program (or project as Borland Builder would say) which is based on the old one. Please read this carefully to avoid another and even worse trap.

1. Never use 'File/Save project as'!
2. Copy the whole folder into another folder with a file manager, e.g. Total Commander or Explorer.

3. Open your program (double click the ‘.cbproj’ file or open Borland Builder and use ‘File/Open project’).
4. Close all windows via the little cross in the upper right corner.
5. Use the project manager in the upper right corner of your screen to open your ‘Unit1.cpp’ by double clicking (if the project manager is not visible open it via ‘View/Project Manager’).

If you do not follow these rather weird rules BorlandBuilder remembers the old path of your project and saves changes there (this means in your old folder, not in your new one!). Another possibility is to copy everything to a new folder and delete the ‘.res’ file in this folder (thereby you delete the file where BorlandBuilder stores the old positions).

2.3 Comparisons

Task: Build a program that allows you the following: You type in two numbers and your program tells you, what’s the relation between the first and the second number (‘smaller’, ‘bigger’ or ‘equal’).

Help: Use the command:

```
if (x>y) { ... }
```

You only need to encapsulate your commands in curly brackets if you want to execute more than one command under the condition that the comparison $x > y$ is true.

Solution: First version

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int var1 = StrToInt(Edit1 -> Text);
    int var2 = StrToInt(Edit2 -> Text);

    if(var1 < var2) Edit3 -> Text = "smaller";
    else if(var1 > var2) Edit3 -> Text = "bigger";
    else Edit3 -> Text = "equal";
}
```

[4] **Solution:** Second version

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int var1 = StrToInt(Edit1 -> Text);
    int var2 = StrToInt(Edit2 -> Text);

    if(var1 < var2) Edit3 -> Text = "smaller";
    if(var1 > var2) Edit3 -> Text = "bigger";
    if(var1 == var2) Edit3 -> Text = "equal";
}
```

Task: What is the difference between these two solutions?

Help: You can use breakpoints to go through your program step by step. Click on the left bar in Unit1.cpp to set breakpoints (therewith you add a red bubble and mark the line reddish). If you now run the program (using e.g. 'F9' or the 'green play button') it will stop at that point. The C++Builder will switch to the 'Debug Layout'. After it stops use 'F8' or 'F7' to jump step by step trough your program. If you move the cursor on a variable you can now see the current value or content of this variable. Furthermore you can use the 'Watch List' and the 'Local Variables' on the left of your screen to follow the values of your variables. Can you now answer the above question?

Solution: Using `else` leads to jumping out of the command block if one comparison was already true. The compiler will not check whether other comparisons are also true for 'not'.

2.4 Memo

Task: Create a new program with a Memo that allows you the following: When you press a button a new line appears in the Memo.

Solution: Take the 'TMemo' out of the standard category of the tool palette and move it onto your form. Create a button and double click this button. Write the following code into the function of this button:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Memo1 -> Lines -> Append(" ... ");
}
```

'Memo1' contains 'Lines', which are accessible via the '->'-operator. The function `Append()` adds text to 'Lines'. Clicking the button several times creates several new lines. You can use the help to find out more about 'Memo' and 'Lines' (move the cursor on 'Memo1' or 'lines' and press 'F1').

Tip: Change the property 'scroll bar' of 'Memo1' in the Object Inspector (e.g. to 'ssVertical'). Now let your program run again and see what happens.

2.5 Loops

Task: Create a program to write display the first 100 of the Fibonacci numbers.

Help: You need loops for that: In a loop something happens several times. If you define an index for the loop, that increases by 1 for every run you can count the number of runs, for example. Use the following code to build a loop:

```
for (int i=0; i<100; i++){
    ... //here come the commands which should be repeated
        //several times
}
```

Tip: The command `i++` is a short version of `i=i+1;`

Excursus: The Fibonacci numbers were first studied by Leonardo Fibonacci (around 1200) to describe the (simplified) growth of a rabbit population. The rules for this model were very simple: In the first month there is one newly born couple, which gets fertile after their second month. Each fertile rabbit couple produces another couple per month. Rabbits never die!



Figure 2.3: A shell whose shape reflects the same law as the one behind the Fibonacci numbers.

In the second month we have still one pair, two in the third, than 3, 5, 8, 13, 21, Figure 2.4 illustrates this. J. Kepler pointed out that the ratio of consecutive Fibonacci numbers converges to the Golden Ratio. You can find Fibonacci sequences in many biological settings like tree branches, leaves in flowers, spiral patterns in horns and shells (see wikipedia for more details).

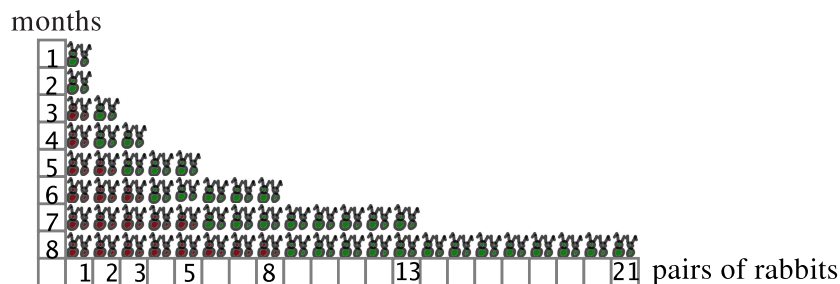


Figure 2.4: Rabbit population growth following Fibonacci numbers. Rabbits are immature for two month (light) then they become mature (dark).

Solution:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int n1 = 1;
    int n2 = 1;
    int aux;

    Memo1 -> Lines -> Clear(); /* to clear "Memo1"
    in the beginning; another function for the object
    Lines which belongs to the object "Memo1"*/

    Memo1 -> Lines -> Append(n1);
    Memo1 -> Lines -> Append(n2);

    for (int i=0; i<100; i++){
        aux = n2;
        n2 = n2 + n1;
        n1 = aux;
        Memo1 -> Lines -> Append(n2);
    }
}

```

Task: You can see that only the first numbers are right. Then you get negative values. This is because the memory used for data of type 'int' is fixed. When an integer number is too large the memory overloads and the storage of the algebraic sign becomes affected. This way the originally very big number may become a very small one and the variable contains a wrong number. All further computations are based on this wrong number (error propagation).

Tip: If you feel uncomfortable with the concept of loops use breakpoints and 'F8' to go step by step through your code.

Task: Change your Fibonacci program so that only values below 1000 are written.

Tip: You can do everything that you can do with a while loop with a for loop but sometimes a while-loop is more elegant. However, while loops also bear the problem that you can easily create never ending loooooooooooooops.

Solution: Solution 1

```
for (int i=0; i<100; i++){
    aux = n2;
    n2 = n2 + n1;
    n1 = aux;
    Memo1 -> Lines -> Append(n2);
    if (n>1000) break;
}
```

Solution: Solution 2

```
for (int i=0; i<100, n2<1000; i++){
    aux = n2;
    n2 = n2 + n1;
    n1 = aux;
    Memo1 -> Lines -> Append(n2);
    // if (n>1000) break;
}
```

Tip: It would be a possible solution to change the index 'i' of your loop to 100 if $n2 > 1000$. YET, it is very dangerous to change the index within the loop. It gets very messy in longer code. Don't do it!

Tip: Some words about the visual appearance of your code. When your programs become more complex you will easily get lost in your code, unless you structure it well and comment it adequately. It is very helpful to structure your code hierarchically. Example:

```
void __fastcall TForm1::Button1Click(TObject *Sender){
```

```

    Memo1 -> Lines -> Append(" ... ");
    /*this code is called within the function ,
    it is one level below , thus it is indented.*/
}
/*this bracket belongs to the function call;
writing it directly under the start of the
function will allow you to recognise this*/

```

Look at the solutions of the last tasks for more examples. Make your life happier, follow these suggestions!

2.6 Random numbers

If you want to create random numbers you can use the function `rand()`. Find out about `rand()` in the help file (write `rand()` in your code, place cursor on it and press 'F1'). However, keep in mind that the numbers just LOOK random. They are part of a long random looking sequence of numbers generated by a function called pseudo random generator (PRNG; There exist many PRNGs and `random()` is one of them. It is fast and bad but usually good enough. If in doubt try another generator: look for 'Marsaglia' or 'Agner Fog' in the internet for more on this topic and some implementations). If you start the program the first time you click on your 'go' button you will always get the same 'random' number. This is because the compiler takes the same part of the long random looking sequence of numbers. Using the function `randomize()` changes this in dependence on the internal computer time. Thus, the number is still not totally random but it is at least not repeatable.

Task: Use your Fibonacci program again (first save it into another folder, open it from there and don't forget to close all windows before opening them again via the Project manager!). Now write ten random numbers instead of 1000 Fibonacci numbers.

Solution:

```

__fastcall TForm1::TForm1(TComponent *Owner)
    : TForm(Owner)
{
    randomize();
}

...

void __fastcall TForm1::Button1Click(TObject *Sender)
{

```

```

int rnd;

Memo1 -> Lines -> Clear();

for(int i=0; i<10; i++){
    rnd = rand();
    Memo1 -> Lines -> Append(rnd);
}
}

```

Task: Create random numbers between 0 and 1.

Solution:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    double rnd;

    Memo1 -> Lines -> Clear();

    for(int i=0; i<10; i++){
        rnd = rand()/(double)RAND_MAX;
        //RAND_MAX is the highest number
        //you can create with rand()
        Memo1 -> Lines -> Append(rnd);
    }
}

```

Task: Find out about the differences between the functions `rand()` and `random()` using the help file. Answer the last task using `random()`. Additionally, compute the mean of the 10 random numbers. Which value would you expect?

Tip: Another useful function for random numbers is `RandG(mean, std)`. With this function you can create normally distributed data. However, you have to include the 'Math' library into your project since the compiler will not know the function otherwise. Therefore you have to include

```
#include <Math.hpp>
```

in the header of your program:

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include <Math.hpp>
/* ..... */
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
/* ..... */
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
```

Help: Use an ‘array’. An array can store more than one number (like a vector). If you do not like to think about vectors imagine a cupboard with several drawers one above the other. Declaring the array is very similar to what we did before. You only have to tell from beginning on how many numbers should be stored in the array (how many drawers your cupboard has) and which type (‘bool’, ‘char’, ‘int’, ...) these numbers should have (which entities should be stored in a single drawer). Figure 2.5 shows an array of integers, where every entry is the square of its index.

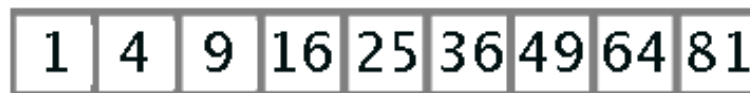


Figure 2.5: Scheme of an array of integer numbers, which are the square of its index.

```
double MyRandomNumbers[10];
```

You can get the numbers (look into the drawers) using the squared brackets.

```
double x = MyRandomNumber [ 0 ] ;
```

The above command stores the first number from the array 'MyRandomNumber' (the index of the first drawer is '0') in a new variable called 'x'. Be careful: The 10 numbers are stored in 'MyRandomNumber[0]', 'MyRandomNumber[1]',..., 'MyRandomNumber[9]' (Your drawers have the names 0, 1, 2, ..., 9). You can go through the whole array (through all drawers of your cupboard) step by step using a loop.

Tip: You can use a short version for $x = x + 2$; which is: $x += 2$; for $x = x + 1$ it is even shorter: $x++$.

Solution:

```
void __fastcall TForm1::Button1Click (TObject * Sender)
{
    double rnd [ 10 ];
    double sum = 0;

    Memo1 -> Lines -> Clear ();

    for (int i=0; i<10; i++){
        rnd [ i ] = random (10000)/(double)10000;
        sum += rnd [ i ];
        //this is a short version of sum = sum + rnd [ i ];
    }
    Memo1 -> Lines -> Append (sum / 10.0);
}
```

2.7 Charts

Task: Create a chart which shows the random numbers against the position in the array.

Help: Take a 'TChart' from the 'TeeChart Std' category of the tool palette and place it on your form. Then double click on the chart on your form. Choose 'Chart/Series/Add' in the window that opens and then double click 'Fast Line'. Borland will now add a series to the chart called 'Series1'. You can access this series via:

```
Series1 -> AddXY(xvalues , yvalues );
```

Solution:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    double rnd[10];
    double sum = 0;

    for(int i=0; i<10; i++){
        rnd[i] = random(10000)/(double)10000;
        sum += rnd[i];

        Series1 ->Add(rnd[i], "", clRed);
    }
}
```

Tip: If you want your chart to change while your program is running you need to refresh it:

```
Chart1 -> Refresh();
```

Before drawing a new chart you can clear the old one using:

```
Series1 -> Clear();
```

2.8 Global and local variables

You can declare a new variable locally or globally. A global variable is declared for the whole program and is everywhere known. You can write the declaration e.g. directly after the header. Local variables are declared within functions or loops and are only known within these functions or loops (e.g. see the variable `i` in the for-loop in the code above this text). It is better style to declare your variables locally. One big problem of global variables is the confusion that arises if

you declare a local variable and a global variable with the same name. Furthermore, it is always useful to keep things that belong to each other together.

2.9 Functions

We already used the function for the 'OK' button which appeared when we double clicked the 'OK' button on the form. We can also create our own functions. Functions connect commands which belong together. Therewith, your code will be more clearly arranged, it will be easier to test parts of your code and you can re-use the commands within the function without copying them all the time.

The principle structure of a function looks like this (write the following above the automatically created functions but beneath the header):

```
void Function_name () {  
    ... //here come the commands  
}
```

You can later call the function in your program using:

```
Function_name ();
```

There exist different possibilities to use functions. Functions can work with arguments which are assigned to them or without them and they can return something or not (see below). The following examples all do the same: They divide a numerator by a denominator.

```
// variables :  
double num=3.0;  
double denom=2.2;  
double res=0.0;
```

2.9.1 Functions without arguments

```
void divide1 () {  
    res=num/denom;
```

```
}  
...  
// call function:  
divide1 ();
```

2.9.2 Functions with arguments

In longer code it can be very helpful to write functions with arguments. This way you only need one look to know which variables your function changes and which not.

```
void divide2(double numerator , double denominator) {  
    res=numerator/denominator;  
}  
...  
// call inside another function:  
divide2(num, denom);
```

2.9.3 Functions with pointer arguments

```
void divide3(double numerator , double denominator ,  
            double & result) {  
    result= numerator/denominator;  
}  
...  
// call function:  
divide3(num, denom, res);
```

'double& result' is a reference (or an alias) for the argument. The '&' changes the variable to a reference. Without using the '&' you could not change the value of the variable result with the function (as you would only use a copy) and the call `divide3(num, denom, res)` would not change the value of the variable 'res'. For arrays it is just the name instead of '&' which gives the reference.

2.9.4 Functions with a *return*-statement

Functions of the type 'void' can change variables but give nothing back. Functions with e.g. type 'double' give back a 'double' number where they have been called.

```
double divide4(double numerator , double denominator) {  
    return numerator/denominator;  
}  
...  
// call function:  
res = divide4(num, denom);
```

2.9.5 Combinations:

```
bool divide6(double numerator , double denominator ,  
            double & result) {  
    if(denominator==0.0) return false;  
    result=numerator/denominator;  
    return true;  
}  
...  
// call the function:  
bool ok;  
ok=divide6(num, denom, res);  
if(!ok) {  
    // numerator equals 0.0 and no change of res  
    // error message:  
    Application->MessageBox("Denominator_should_not_be_zero!");  
}  
// Otherwise (ok==false) res contains now the result.
```

2.10 Output files: Writing in a text file

To analyse your program it is useful to see your results not only on the interface of your program but to write them in text documents. This allows you to store them and to further analyse them with statistic or graphic software. You can use the following code for that:

```

//You need to include the header fstream
#include <fstream>
...
/* ..... */

//define and open your output file in one step:
std::ofstream k("output.txt", std::ios::app);
//write the content of your variables in your output file:
k << Variable1 <<" ," << Variable2 <<" ," << Variable3 <<" \n";

```

2.11 Handling errors or things you should check before asking for help

You have already learned how to use breakpoints. The 'CodeGuard' is also useful for finding problems that only occur when the program is running, such as the localisation of an overflow (e.g. if you try to access values of an array outside its borders; non existing drawers and thus something out of the mess on the floor). After testing your program you should switch off the CodeGuard since it makes your program slow.

In the Borland Builder you can find the 'CodeGuard' under 'Project/Options'. Then choose the flag 'C++ Compiler/Code Guard compile support' and activate all Code Guard options. After changing something here you should compile your program again with 'Project/BuildAllProjects'. Otherwise you might get strange error messages.

2.12 Changing your program while it is running: pause, stop, exit

You can always stop your program while it is running using 'Run/Program Reset' or press 'Strg+F2'. It is also very helpful to create buttons on your interface with which you can stop, pause or close your program. You can use variables from type 'boolean' for that. Boolean variables can be either *true* or *false*. Example:

```

//declare variable kill above the functions for the buttons;
bool kill;

/* ..... */
//function for run button
void __fastcall TForm1::Button1Click(TObject *Sender)

```

```

{
    kill = false;    // initializing
    int a=0; int n=0;
    while (a==0){
        // (senseless) never ending loop :
        n++;
        Mem1->Lines->Add(IntToStr(n));
        Application->ProcessMessages();
        /*checks changes while the program is running,
        e.g. clicking buttons on the surface;
        it makes your program slow, so place it carefully*/
        if(kill){    //same as (if kill == true)
            ShowMessage("killed");
            return; //return : stop function
        }
    } // end while loop
}
/*.....*/
//function for kill button
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    kill=true;    // set flag
}
/*.....*/
//function for pause button
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    ShowMessage("Program_paused...");
}

```

3 Examples

Now we will present some more complex examples. The idea of these examples is to improve both your programming skills and your knowledge about ecological modelling.

3.1 Sheep farm

Task: Assume that there is a sheep farm. Two meadows belong to the farm. The meadows are connected via one narrow path. The farmer who is an organic farmer is confronted with a problem: He found an alien plant growing next to the path. This plant is unhealthy for his sheep. Sheep disperse this plant. The farmer can use some organic herbicides against the plant but this is expensive and only partly successful. How can you help? Think about this before you read further.

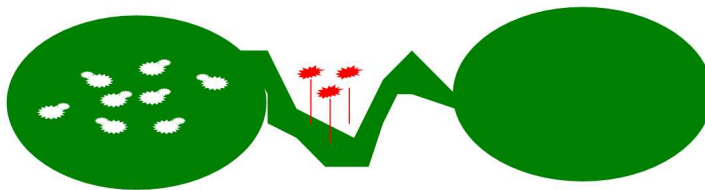


Figure 3.1: Sheepfarm 1

Tip: *Step 1:* This is much too unspecific for scientific research. We ALWAYS need a good question before we start with a model! One question could be: How many pesticides does the farmer need to suppress the plant in one year? Is that a good starting point?

Step 2: This is still much too unspecific. What should the farmer do when you give the necessary amount of pesticide to him? He still has thousands of possibilities to use the pesticides. A good question could be: The farmer has a certain amount of pesticides. How should he distribute these pesticides over his farmland?

This is a question you could answer. What do you need to know? What could be important? How could you simplify this question to start with your model? Think about this before you read further and write down possible influential factors.

Some suggestions for influential factors

- Length of the path
- Width of the path
- Number of sheep and seeds

- How often disperse sheep
- Probabilities of sheep taking up seeds and letting them fall off again

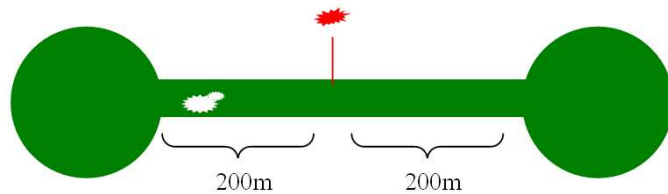


Figure 3.2: Sheepfarm 2

Step 3: Very important: You will have to simplify. It makes no sense to build a very complex model and then not understand anymore what is going on. Start with an oversimplified model and test model rules for congruency.

Some suggestions for model properties

- Assume a meta sheep or a meta sheep herd
- Assume only one dispersal event
- In both directions happens the same, thus simulate only one direction
- We need to know about probabilities (e.g. how often will a seed travel with the meta sheep? What is the germination rate of the seeds? What is the chance that the herbicide will kill the plant?)

Step 4: Think about a simple model that you could implement easily. One possibility: Distribute the amount of pesticides relative to the spatial distribution of seeds. How will the sheep distribute the seeds?

Task: Draw a flow chart which describes your model rules.

Solution: This is just one possibility among various.

Task: Write a model based on your flow chart.

Solution: Again this is only one possibility.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    const int sheep=50; //number sheep
    //start number of seeds per sheep
}
```

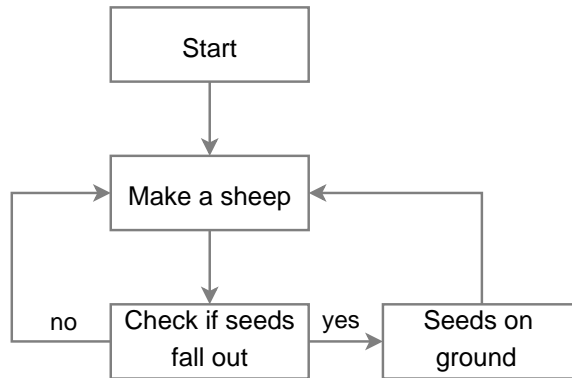


Figure 3.3: Flowchart. What's to do for every sheep.

```

const int sheep_seed=StrToInt(Edit2->Text);
//Prob. Of seeds to fall down
const float p_fall=StrToFloat(Edit3->Text);
const int steps=20;//number of steps
//number of fallen seeds per section of the path
int seeds[steps];
//number of seeds still on sheep
int seeds_travelling[sheep];
int temp;//temporary variable

//initialise sections of the path; no seeds
for(int i=0;i<steps;i++) seeds[i]=0;
//initialise the fur; number of seeds in the fur
for(int i=0;i<sheep;i++) seeds_travelling[i]=sheep_seed;
randomize();//random starting point of random number
Series1->Clear();//clear graphic
for(int i=0;i<steps;i++)//In each step ...
{
  for(int j=0;j<sheep;j++)// ... on each sheep ...
  {
    temp=seeds_travelling[j];
    for(int k=0;k<seeds_travelling[j];k++)// ... each seed
    {
      if((rand()/(float)RAND_MAX)<p_fall)
      // ... has the chance
      {
        temp--; // ... to fall ...
        seeds[i]++; // ... down.
      }
    }
    seeds_travelling[j]=temp;
  }
}
  
```

```

    }
    Series1 ->AddXY(i , seeds [ i ]); //draw
  }
}

```

Task: Test your program. How does the distribution change when you incorporate the process of taking up again already fallen seeds? What happens if sheep loose different fractions of seeds? What can you learn about how the farmer should distribute the pesticides?

Solution: It highly depends on the parameters how the seeds will be dispersed. Obviously, the amount of pesticides should be higher close to the plant and lower more far away. Regularly, sheep already lost all seeds when they come to the end of the path. The distribution you modelled here is a Weibull distribution. This distribution is typical for dispersal (see wikipedia for more details). The maximum of the distribution moves if you allow the seeds to be lifted again by other moving sheep. (If you don't allow this you'll get an exponential distribution as a special case of the Weibull distribution.)

Task: Yet, what is about rare long distance dispersal? We ignored that until now although it is very important and would change the outcome. If you like you can improve the model by including long distance dispersal and by modelling several years (reproduction of new plants, etc.)

3.2 Predator prey

Task: Create a very simple spatial predator-prey system, e.g. hare and fox.

Step 1: What do we need for that?

- A two dimensional grid which describes a part of the landscape (quadratic cells to make it easy because this fits to the array structure of the computer)
- The grid is homogeneous, no differences between grid cells
- Each cell is free, occupied by hare or by fox (three possible states)
- Properties of neighbouring cells can influence properties of focus cell (dispersal abilities of foxes and hare); all 8 neighbouring cells are neighbours:
 1. Empty cell changes to hare cell if at least a neighbours are hare (hare reproduction)
 2. Hare cell changes to fox cell if at least b neighbours are fox (fox reproduction depending on food)
 3. Fox cell changes to empty cell with certain probability c (fox mortality)

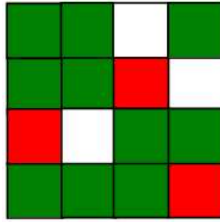


Figure 3.4: Grid. Predator-Prey

Tip: You can use the `MMColorGrid` which is a very nice graphical tool to draw grids (two-dimensional arrays). Take the ‘`MMColorGrid`’ from the ‘`Toomai`’ page of the component palette and add it to your ‘`Form1`’. Initialize the grid with the values you used to initialize your two dimensional array (You have to tell the grid which size it should have). Do this in the ‘`TForm1` constructor’ which was produced automatically (The constructor of the form gets executed once in the program so this is a good point). But caution! The `MMColorGrid` is no standard feature of the Borland Builder. It is developed at the OESA by Michael Müller. If you work in the OESA: If there is no toomai page in your component palette switch out C++ Builder and install `MMColorGrid` from `NetInstall`. If not: contact `MMColorGrid@toomai.de`.

```

__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
    //add here only things that should be done only once
    MMColorGrid1->Xcells=xcells;
    MMColorGrid1->Ycells=ycells;
}

```

Now chose ‘Object Inspector/Legend’ and click on the ‘...’. You can now add one colour for each value to the grid, e.g. white for empty cells, green for hares and red for foxes. Select ‘New/One value’. Double click the white square to chose the colour for this value and add a legend text. After adding all values close the window.

```

void Display(){
    //loop: all cells should be hare
    for(int i=0; i<xcells; i++){
        for(int j=0; j<ycells; j++){
            //here we need Form1 to tell the compiler
            where to search for the ColorGrid
            Form1->MMColorGrid1->SetXY(i,j,grid[i][j]);
        }
    }
}

```

```

    }
    Form1->MMColorGrid1->Repaint();
}

```

Step 2: Write a program which creates a grid with the following starting conditions when one presses a button: all cells hare and one fox in the middle.

Solution:

```

#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
/* states of our grid:
empty=0
hare=1
fox=2 */

// Initialization
const int xcells=20;
//this means xcells can only be equal to 20,
//you can not change this during the further code; we use this

const int ycells=20;
//arrays can only be initialised with constants
int grid[xcells][ycells];

// Starting conditions
/* ..... */
void Init_FoxMiddle(){
    //slope: all cells should be hare
    for(int i=0; i<xcells; i++){
        for(int j=0; j<ycells; j++){
            grid[i][j]=1;
        }
    }
    //the middle cell should be fox
    grid[10][10]=2;
}

```

```

/* ..... */
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//you get the framework for this
//if you build a OK button on your form
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
    Init_FoxMiddle();
}

```

Task: Write a program which creates a grid with the following starting conditions when one presses a button: all cells by random empty, hare or fox.

Solution:

```

void Init_FoxMiddle(){
//slope: all cells should be filled by random
//with nothing, hare or fox
    for(int i=0; i<xcells; i++){
        for(int j=0; j<ycells; j++){
            grid[i][j]=random(3);
        }
    }
}

```

Task: Now you need rules that tell the computer how the cell states shall change within one year (one time step). The rules are:

1. Empty cell changes to hare cell if at least a neighbours are hare (hare reproduction)
2. Hare cell changes to fox cell if at least b neighbours are fox (fox reproduction depending on food)
3. Fox cell changes to empty cell with certain probability c (fox mortality)

We would suggest to try this out by hand on a sheet of paper. Which are the traps you can jump in?

Solution: When you first change all empty cells, than all fox cells and than all hare cells you will change neighbourhood relations all the time; what happens in the cell depends on the order

of your rules. Yet, we want the changes to depend on the starting conditions but not on the already changed conditions! What can we do? One solution is to create another grid in which we write the changes.

Task: Create a function that is able to count neighbours.

Solution:

```
void CountingNeighbours(int x, int y,
                        int & HNumber, int & FNumber){

    //Before we count all numbers should be zero
    HNumber=0;
    FNumber=0;
    for(int m=x-1; m<=x+1; m++){
        for(int n=y-1; n<=y+1; n++){
            //count foxes and hare
            if (!(m==x && n==y)){
                if (grid[m][n]==1) HNumber++;
                //else if: only when it is not hare
                //check whether it is a fox
                else if (grid[m][n]==2) FNumber++;
            }
        }
    }
}
```

Task: Now build the whole program. Remember that you will need a second grid. If you have created a running program think about the borders of your grid. What happens there? One traditional solution is to create continuous borders. This means that something that disappears at the right side (top) comes in at the left side (bottom) and vice versa. How could the commands for this problem look like?

Help: A nice solution for continuous borders is the modulus (written as % in C++):

```
x = (x + xcells) % xcells;
//xcells is the horizontal size of the grid
y = (y + ycells) % ycells;
//ycells is the vertical size of the grid
```

3.3 Carnivorous plant

In this example some slightly advanced programming techniques will be presented to you. Therefore we will switch to another didactic. We don't want you to develop a program by your own but to understand and improve an existing one and to add your own ideas. But first we will start with some rudiments of theory of objective oriented programming languages as C++.

Well, implicitly we already took advantage of this concept, but now we describe it a bit. The concept of objective oriented programming allows writing program code which expresses real-world relationships in an explicit way. This ability makes programming languages like C++ extremely suitable for implementing individual based models. Consider for example a fly. A fly can be seen as a real-world object. Out of all objects in the world it can be classified as an animal. So it belongs to the class of animal objects. Moreover we can say it is an insect. Insects constitute a sub-class of the animal class. If we want to be more precise we say a fly is an object of the sub-class of insects which is called Diptera and so on.

Every affiliation to a specific class determines some properties of an object. In our example the fly shares the property to have a complex metabolism which do not base on photosynthesis with all animals. But it does not share the property having only two wings with all insects, because not all insect are Diptera. We see that there is a hierarchy of classes. The more specific a class is – is to say the further down in the hierarchy it is – the more specific are the properties of the objects belonging to the class.

This hierarchy of classes, in which every real-world object can be arranged, can be emulated in C++. The following example illustrates this. We define a class of individual 'TInd' and after that a sub-class of flies 'TFly'.

This concept of inheritance of properties and methods briefly described here is very useful for very big project – bigger than everything an ecological modeller is likely to do. So just notice this concept but don't use it. If you are not well trained in it, it could bring you unnecessarily into more trouble than you will already have when you start implementing your own model.

```
class TInd
{
    bool alive;
    short sex;
    int age;
    double weight;
};
/* ..... */
class TFly : public TInd
{
    int x_pos;
    int y_pos;
};
```

```

...
/* ..... */
TFly fly = new TFly (); // creates a new Fly
fly . x_pos = 10;      // set its position
fly . y_pos = 15;      // set its position

```

Every individual ('TInd') in our virtual world has the properties to be alive, to have a sex, age and weight. (Well, these properties are fairly arbitrarily chosen, but for principles sake they suit.) The class 'TFly' is a sub-class of 'TInd' (indicated by the colon after the declaration of 'TFly'). Hence every 'TFly object (every virtual fly) has the properties of 'TInd' as well. Is to say every fly can be alive or not and has a sex, age and weight.

But that's not all about objects. Objects of a certain class can do things, which in combination with all their other properties only they are able to do: Roughly spoken only a fly is a fly and can fly. In the terminology of C++ we say, that there are methods which belong to the class. We illustrate this by extending the above example. We give our fly objects of the 'TFly' class the opportunity to move. For this purpose we define inside the class definition of 'TFly' a function `move(void)` which we now will not call 'function' anymore but 'method'. The method itself we write down outside the class definition.

```

class TFly : public TInd
{
    int x_pos;
    int y_pos;
    void move( void );
};
/* ..... */
... //some other declarations
/* ..... */
void TFly :: move( void )
{
    ... //stuff which the method does
}

```

At this point one special type of methods should be mentioned: constructors. Constructors are called whenever an object is generated. They are used to initialize this new object. There is a standard constructor which every object of the class Object has. You don't have to care much about it, because it is used implicitly. But you can define your own constructors for your classes. Then they will be used instead of the standard constructors. In contrast to other methods a constructor has no type (You'll see this in the following example). It can be used to initialize the properties of an object. The constructor always has the name of the class (the constructor of the

class Tfly is named Tfly).

```
class Tfly : public TInd
{
    int x_pos;
    int y_pos;
    Tfly(int x, int y); //declaration of the constructor
    void move(void);
};
/* ..... */
... //some other declarations
/* ..... */
Tfly::Tfly(int x, int y)
{
    x_pos=x;
    y_pos=y;
}
/* ..... */
void Tfly::move(void)
{
    ... //stuff which the method does
}
```

These are the basic tools to start with individual based modelling in C++.

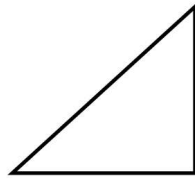
Now one helpful concept for maintenance of a population of objects (e.g. a fly population) will be presented: lists. Lists are an advanced data structure provided by the Borland C++ library (to use them you have to include 'list.h' library in the header).

To understand how lists are used it helps to see them in the context of a program. Therefore we now present you a whole program. It shows the use of classes and lists and some other things already explained in the previous sections.

Task: Try to understand the program! Extend it: Add a class for the plants following the scheme of the class 'Tfly'! Add a dynamic to the population of plants and hold it in a list following the example of the fly population! Try to modify the classes so that is biologically more realistic! Try to think of an alternative implementation of the random walk in the method `move(void)`!

Help: Examining the code you should be able to find out which elements the graphical interface of the program has. Place them on your form and generate the corresponding function heads by clicking twice on the elements as explained in the sections above. Then fill in the code from this example.

Help: For the alternative random walk consider this:



$$\cos \alpha = \frac{x}{r} \Leftrightarrow x = r * \cos \alpha$$

$$\sin \alpha = \frac{y}{r} \Leftrightarrow y = r * \sin \alpha$$

Figure 3.5: Tips for alternative random walk

C++ uses the radian instead of degree. $\frac{AngleRadian}{2\pi} = \frac{AngleDegree}{360}$

```
//fly.x is x position of the fly; xangel a direction  
fly.x = fly.x + steplength * cos(xangel);  
//fly.y is y position of the fly; yangel a direction  
fly.y = fly.y + steplength * sin(yangel);
```

Solution:

```
#include <vcl.h>
#include <list.h> //to use lists one has to include this lib
#pragma hdrstop

#include "Unit1.h"

#pragma package(smart_init)
#pragma resource "*.dfm"

//definition of the class TInd, it contains variables which
//describe the properties of an individual, and declarations
//of methods which manipulate the properties of an individual.
/*.....*/
class TInd
{
public:
    bool alive; //auxiliary variable
    short sex; //property sex
    int age; //property age
    double weight; //property weight
    TInd(void); //constructor
    void die(double m); //method for individual's dying
    void ageing(void); //method for individual's aging
};
/*.....*/
// the class TFly is now derived from the class TInd
// (which is to say a fly is an individual)

class TFly : public TInd
{
public:
    int x_pos; //x-coordinate of fly's position
    int y_pos; //y-coordinate of fly's position
    TFly(void); //constructor for an object of type fly
    void move(void); //method to move fly
    void get_caught(void); //method of getting caught
private: //this can only be modified by a method of TFly
    int direction;
};
/*.....*/
```

```

//list of flies substitutes an array, e.g.:
//TFly IndArray [100]
//the list contains the population

list<TFly> IndList;

//some global variables

const int dimension=50; //dimension of landscape
double plant_density;
int number_of_flies;
double fly_mortality;
bool landscape[dimension][dimension];
bool pause;
/* ..... */
TForm1 *Form1;
/* ..... */
//Now the methods of the classes have to be implemented

//the constructor sets initial values for the variables
//of an individual
TInd::TInd(void)
{
    alive=true;
    sex=random(2);
    age=1;
    weight=1.0;
}
/* ..... */
//an individual dies accordant to the mortality m
void TInd::die(double m)
{
    double z=(double) rand()/(RAND_MAX+1);
    if (z<m) alive=false;
}
/* ..... */
//ageing means increasing age and weight
void TInd::ageing(void)
{
    age++;
    weight+=0.1;
}
/* ..... */
//constructor of a fly

```

```

TFly :: TFly ( void )
{
    alive=true;
    sex=random(2);
    age=0;
    weight=1.0;
    x_pos=random( dimension );
    y_pos=random( dimension );
    direction=random(8);
}
/* ..... */
//the fly does a simple auto correlated random walk
void TFly :: move( void )
{
    //calculate the direction of the next step
    //for this purpose we imagine the neighbourhood of the
    //position p as follows
    /*
        _____
        | 0 | 1 | 2 |
        _____
        | 7 | p | 3 |
        _____
        | 6 | 5 | 4 |
        _____
    */

    int d=random(3)-1;
    if( direction+d==-1)direction=7;
    else direction=(direction+d)%8;

    //go into this direction
    // '||' stands for the logical or
    if( direction ==0|| direction ==1|| direction ==2) x_pos++;
    if( direction ==4|| direction ==5|| direction ==6) x_pos--;
    if( direction ==2|| direction ==3|| direction ==4) y_pos++;
    if( direction ==6|| direction ==7|| direction ==0) y_pos--;

    //merge edges , i.e. make a torus
    if( x_pos<=0)
        x_pos=dimension-1;
    if( y_pos<=0)
        y_pos=dimension-1;
    if( x_pos>=dimension)

```

```

        x_pos=0;
        if(y_pos>=dimension)
            y_pos=0;
    }
    /* ..... */
    //if a fly gets caught by a plant it dies
    void TFly::get_caught(void)
    {
        if(landscape[x_pos][y_pos]==true)
            alive=false;
    }
    /* ..... */
    //function to initialize the landscape where
    //plants and flies meet each other
    void init_landscape()
    {
        TColor col;

        //read values of the global variables from the form
        number_of_flies=StrToInt(Form1->Edit1->Text);
        plant_density=StrToFloat(Form1->Edit2->Text);
        fly_mortality=StrToFloat(Form1->Edit3->Text);

        //initialize picture size
        Form1->Image1->Picture->Bitmap->Width=15*dimension;
        Form1->Image1->Picture->Bitmap->Height=15*dimension;

        //initialize the landscape grid
        for(int i=0; i<dimension; i++)
            for(int j=0; j<dimension; j++)
                if(rand()/((double)RAND_MAX)<plant_density)
                    landscape[i][j]=true; //yes, there is a plant
                else
                    landscape[i][j]=false; //no plant
    }
    /* ..... */
    //function to display the actual state of the system
    void display()
    {
        TColor col;

        //display plants
        for(int i=0; i<dimension; i++)
            for(int j=0; j<dimension; j++)

```

```

    {
        if ( landscape[i][j]==true ) col=c1Green;
        if ( landscape[i][j]==false ) col=c1White;

        //drawing the the stuff in the appropriate colour
        //at the right point in space
        Form1->Image1->Canvas->Brush->Color = col;
        Form1->Image1->Canvas->
        Ellipse (Rect(i*10,j*10,i*10+10,j*10+10));
    }

    //display flies using an iterator for going through the list
    for( list <TFly>::iterator iter=IndList.begin();
        iter != IndList.end(); iter++)
    {
        //its the turn of this fly now
        TFly & ThisFly = *iter;

        //drawing the the fly in black
        Form1->Image1->Canvas->Brush->Color = c1Black;
        Form1->Image1->Canvas->Ellipse (Rect( ThisFly . x_pos*10,
            ThisFly . y_pos*10,
            ThisFly . x_pos*10+10,
            ThisFly . y_pos*10+10));
    }
}
/* ..... */
//auxiliary function which is used by "remove_if"
bool IndIsDead(TInd AnInd)
{
    if ( AnInd.alive ) return false;
    else return true;
}
/* ..... */
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
/* ..... */
//pushing this button generates an new population of flies ,
//and initializes the landscape
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //first clear everything

```

```

IndList.clear();
Form1->Series1->Clear();

randomize();

init_landscape();

//generate and insert flies into the list
for(int i=0;i<number_of_flies;i++)
{
    TFly NewFly; //declare a new fly
    IndList.push_back(NewFly); //add it to the list
}

display();
}
/* ..... */
//clicking this button starts the simulation
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    pause=false;

    while(!pause)
    {
        for(list<TFly>::iterator iter=IndList.begin();
            iter!=IndList.end();iter++)
        {
            TFly & ThisFly = *iter; //its the turn of this fly now
            ThisFly.die(fly_mortality); //will it die?
            if(ThisFly.alive)ThisFly.ageing(); //if not than at
            //least ageing
            if(ThisFly.alive)ThisFly.move(); //and moving
            if(ThisFly.alive)ThisFly.get_caught();
            //will it be caught by a plant?
        }

        //remove all dead flies from the list
        IndList.remove_if(IndIsDead);

        //produce some fancy output
        Form1->Series1->Add(IndList.size());
        Label1->Caption=IndList.size();
        display();
    }
}

```

```
//if population is extinct stop simulation
if(IndList.size()==0) pause=true;

//watch out for button clicks
Application->ProcessMessages();
}
}
/*.....*/
//clicking this button changes the value of "pause" to true
//and make the simulation halt.
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    pause=true;
}

```
