University of Berne **Department of Teachers Training**

Aegidius Plüss

Introduction to object-oriented programming with

C++

and the MS-Windows framework

Champ

Version 1.04

May 1997

An Introduction to Object Orientied Programming using Object Turtles.

Author:

Prof. Dr. Aegidius Plüss
University of Berne, Switzerland
E-mail: pluess@sis.unibe.ch

Developer of Champ:

Salvisberg Software & Consulting
Bellevuestrasse 18
CH-3095 Berne, Switzerland
Fax: (+41)(+31) 972'42'10
E-mail: info@salvisberg.com
WEB: www.salvisberg.com

1 Introduction

Even though there are many opinions concerning the importance of any particular theme in a basic course on computer science, it is a widely acclaimed fact that treating algorithmic processes is particularly important. Usually this is done in connection with a programming course, since only the use of a programming language allows algorithms to be formulated precisely and then be executed on a computer.

However, opinions differ strongly as to which programming language and environment to choose, and how to proceed methodologically. Modern, practically oriented schooling increasingly demands that the programming language should be one widely used, available on different computer platforms and operating systems, and that both the development and the execution of programs should take place under a graphical operating system. In addition, the programming language should on the one hand be simple enough for beginners without any previous knowledge of programming to handle; on the other hand, the same language should be applicable to teaching advanced students more complex matters, object oriented programming (OOP) in particular.

Especially because of this last demand, all script-like languages such as HyperCard and Toolbook are out of the question. And even in their most modern versions, languages popular with beginners, such as Basic or Logo, are not independent enough of a particular platform, nor have they enough features to enable the teaching of data encapsulation and OOP techniques.

From the theory of learning we have long known that introductory instruction to a new, motivating subject can have a strongly determining effect on students, so that later it becomes very hard to get rid of bad habits caused either by errors or by oversimplification. This is why already in 1969, in his famous article "Programming is considered to be harmful", Dijkstra demanded the total abolishment of programming classes in the lower grades rather than having students taught wrong programming methods.

In recent years there has been a change within computer science towards object oriented programming. This new technique is so fundamentally different from the classical imperative, functional or logical programming, that one might even speak of a change of paradigms. If the beginner is to be introduced to this new technique, or at least his later access to it is not to be blocked, it is necessary that small introductory programs be developed according to object oriented concepts right from the beginning.

To give a first idea, it is good to have students in beginners' classes understand screen elements such as buttons as *objects* with certain **properties** and **behavior** (usually the routines triggered by mouse actions). But decisive for object oriented programming is the concept of class hierarchy, within which the properties and behavior of objects can be extended through *inheritance* by class derivation.

For this reason it is dangerous to use a simple programming language in which, although it might allow the construction of screen elements with properties and behaviors, it is not possible to use inheritance.

In spite of the great variety of programming languages, it is not easy to find one that fulfills all the demands mentioned above. If both the possibility of using OOP techniques and a high rate of recognition at educational institutions are considered particularly important, the selection is limited to (Object-)Pascal, C++, Smalltalk, Oberon and Eiffel. Should, in addition, the language be available across various computer platforms and widely used by professionals around the world, C++ is a good choice for use in an introductory computer science course.

It is sufficiently known that compared to languages especially designed for teaching like Pascal or Eiffel, C++ is a complex language, a fact which repeatedly leads to the claim that it is not a suitable language for beginners. Our longterm experience in teaching programming to young students (of around 16 years of age) shows, however, that it is possible to develop a methodologically well worked-out concept for courses using C++ so that success rates are as good or better than in former courses using Logo, Basic or Pascal.

2 Turtle graphics with multiple object turtles

In our experience, it has proven to be most useful to make certain programming tools available to the beginner in the form of frameworks or libraries. Since young people in particular strongly respond to graphic elements, a graphics library that is easy to handle should be at their disposal right from the beginning. For program development under a complex graphic user interface such as Windows, elaborate libraries that suitably hide its complexity are almost a necessity.

Already in 1980, with the programming language Logo, Papert created a graphically oriented learning environment [4]. For many teachers, the Logo turtle means getting away from dry text and number oriented problems, and it corresponds to a general esthetic need for graphical animation. Unfortunately, especially with professionals Logo has not enjoyed great success. This is by and large due to three causes:

- The programming environments of most Logo versions are outdated and are not being developed any more.

- Logo focuses too much on recursions, a method which is elegant, but which is not at all easy to understand and is of minor importance in most other programming languages.

- Logo does not lend itself to numeric applications and is therefore nearly unknown by professional programmers.

Nonetheless, school experience shows that particularly due to its turtle graphics, Logo is a good language for beginners [6]. True hymns of appraisal have been written about turtlegraphics, stressing its advantage - especially for beginners' courses - of **direction oriented** over **coordinate oriented** graphics, and of its animation through the moving turtle.

In a beginners' course focusing on OOP, a better example for objects than screen turtles can hardly be imagined. They have an intuitive reality, they possess properties (color, position, direction) and behavior (go forward, turn left). It is also obvious that the turtles can gain further properties (name, etc.) and behavior (draw circle, etc.) through inheritance.

Since several turtle objects are present at the same time in an object oriented turtle graphics system, the system must be able to handle the overlapping of turtle shapes and their traces correctly.

3 Programming under a graphic oriented user interface (GUI)

During recent years, window-based user interfaces have become a standard in practice. In lower-grade computer science courses, students are trained to work with such interfaces, so that nowadays most of them never really get into contact with command line operating systems any more. Because of this fact, it is hardly motivating to then do programming in a text oriented environment. On the other hand, the graphic user interface and the handling of events or messages makes developing programs a complicated matter, which could lead to the disadvantage of having to spend more course time developing the user interface and the message handling than actually treating the algorithmically interesting parts of the program.

In the professional world, the complexity of programming under a GUI is intercepted through system calls (Application Programming Interface (API)) and class libraries or frameworks such as Microsoft Foundation Class (MFC) or Borland's Object Windows Library (OWL). The latter however requires a sound knowledge of C++ right from the start, and therefore cannot be used in beginners' courses.

All our programming courses strictly followed the principle that all source code visible to the student must be fully comprehensible with the student's actual knowledge. Code generating tools like Borland's Delphi or AppExpert, powerful for professional programmers, contain sophisticated code skeletons which could hardly be explained to beginners.

For beginners' courses and scientific applications concentrating on algorithms, a development environment is needed which automatically produces genuine GUI-oriented programs from a program code that is not or is hardly different from the traditional text-oriented form. It should also be possible to write programs under a modern operating system like MS-Windows without previous knowledge of OOP. Even for more complex user interfaces using mouse actions, dialogues, and menus, elementary OOP techniques like instantiating objects or calling member functions should be sufficient. And most important, the concept of message queues should be avoided in message handling. Instead, all events should trigger a user defined *callback function* which handles the event.

One of the advantages of programming under a modern GUI-based operating system is independence of hardware. So, for example, the screen and the printer are automatically adapted to the available devices without any modification of the program.

4 The Champ framework

The framework *Champ* was developed according to the principles shown above in order to make it possible for beginners in programming, but also for teachers and scientists to develop C++ programs with little effort, but all the same with MS-Windows' functionality fully conserved. (For Champ features, see appendix 1.)

In order to illustrate the extreme simplicity of programming with Champ, the first program will draw a diagonal line in a graphics window. It uses global functions (their names starting with a g) and the default floating point coordinate system x = 0..1, y = 0..1. The window has a system menu to copy graphics into Windows' clipboard or to any attached printer.

```
#include <champ.h>

void gmain ()
{
   ginit( "First Application" );
   gpos( 0, 0 );
   gdraw( 1, 1 );
}
```

In order to promote object oriented methods right from the beginning, it is better to see windows as instances of objects of a predefined class *CPWindow,* which means not using global functions. The following program produces a window object *myWindow* and shows it on the screen. The member functions pos() and draw() draw inside the window. Since according to the block scope of an instance the window object would automatically be destroyed as soon as gmain were quit, the program is made to wait with getch() until a key is pressed.

```
#include <champ.h>

void gmain ()
{
   CPWindow myWindow( "First Application" );
   myWindow.pos( 0, 0 );
   myWindow.draw( 1, 1 );

   getch();
}
```

The OOP version is not only more elegant, but it also makes it possible to draw several windows at the same time by producing several CPWindow instances.

```
#include <champ.h>
```

```
void gmain ()
{
   CPWindow first( "First Window" );
   CPWindow second( "Second Window" );

   first.pos( 0, 0 );
   first.draw( 1, 1 );
   second.penColor( RED );
   second.circle( 0.5, 0.5, 0.2 )
}
```

(For syntax conventions, see appendix 2.)

5 A course concept for object oriented programming with C++ using object turtles

It would be far too difficult for beginners to define C++ classes from scratch. However, they can easily use predefined objects, since creating them is not more complicated than defining a variable in a language that is not object oriented. Only after having mastered the basics of OOP using predefined classes, are the students taught to construct classes from scratch.

Our idea is to use the most motivating element of Logo, the turtle graphics system, in an introductory course on C++. This does not mean that the student programs in Logo. He simply uses C++ in a more pleasant and motivating way than is presented in most introductory textbooks. The full Logo turtle command set is supported. A turtle is "created" and appears on the screen whenever a turtle object is instantiated. Object turtles can also be dynamically created and destroyed and all problems of visibility and overlapping of multiple turtles on the screen are handled correctly. The total number of turtles is restricted only by memory size and performance degradation. The turtle graphics library is smoothly integrated into the programming environment to guarantee that the beginner does not see much difference between reserved words of the C++ language and turtle commands.

In the following it is assumed that the learner has little or no previous experience in handling computers.

Step 1: First contact with the computer

Aim: Getting acquainted with keyboard, mouse, Windows interface, Integrated Development Environment (IDE) and editor.

Procedure: The student is asked to correct or complete a (funny, stimulating) text skeleton (ASCII file) using the IDE editor.

Remarks: At the same time, this exercise is meant to be an introduction to word processing, so the most important features of a word processor should be pointed out to the student.

Step 2: Instantiation

Aim: Getting to know the development cycle of a program (using the Champ Project manager): editing, compiling, running, debugging on the exampleof a sequence. Initializing the graphics system. Creating objects. Calling member functions (sending messages).

Procedure: Create a turtle and draw a stair with 5 steps 20 units high and 20 units wide.

Remarks: The first program example shows how to create objects and how to use their behavior by calling member functions using the “point“ operator.

Default coordinates of turtle graphics: horizontal -200..200, vertical 200..200.

Block structuring is done with curly brackets (block start, block end).

Instantiation of the turtle object "john" and its "lifetime" inside a block. The term "instantiation" is preferred to "variable definition".

The compiler regards names that differ only in capitalization as different from each other.

Semicolons separate instructions. Other than that, screen design is free to a large extent, but we follow some conventions very closely.

Functions are always to be provided with round brackets, in between which the values (arguments) passed to the function are written. (Functions that do not need an argument are given empty brackets, in order to make them easily distinguishable from the other names.)

One-line comments are introduced by //, those that are more than one line long are put between /*...*/. We prefer the single line comments and reserve the multiline comments to “comment out“ whole sections when debugging programs.

Supplements: Let two stairs be drawn by two turtles "john" and "laura" using also the turtle member functions: *turtleColor*, *penColor*.

```
// LEARN02.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn02" );  // Initialize graphics window
   Turtle john;         // Instantiate object "john"

   john.forward( 20 ); // Send msg: "Walk 20 steps"
   john.right( 90 );   // Send msg: "Turn to the right"
   john.forward( 20 );
```

```
   john.left( 90 );

   john.forward( 20 );
   john.right( 90 );
   john.forward( 20 );
   john.left( 90 );

   john.forward( 20 );
   john.right( 90 );
   john.forward( 20 );
   john.left( 90 );

   john.forward( 20 );
   john.right( 90 );
   john.forward( 20 );
   john.left( 90 );

   john.forward( 20 );
   john.right( 90 );
   john.forward( 20 );
   john.left( 90 );
}


// LEARN02A.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn02a" );
   Turtle john;         // Instantiate object "john"
   Turtle laura;        // Instantiate object "laura"

   laura.turtleColor( YELLOW );
   laura.penColor( GREEN );
   laura.setPos( 200, 0 );

   john.forward( 20 );
   john.right( 90 );
   john.forward( 20 );
   john.left( 90 );

   laura.forward( 20 );
   laura.left( 90 );
   laura.forward( 20 );
   laura.right( 90 );

   john.forward( 20 );
   john.right( 90 );
   john.forward( 20 );
   john.left( 90 );

   laura.forward( 20 );
   laura.left( 90 );
   laura.forward( 20 );
   laura.right( 90 );

   john.forward( 20 );
```

```
    john.right( 90 );
    john.forward( 20 );
    john.left( 90 );

    laura.forward( 20 );
    laura.left( 90 );
    laura.forward( 20 );
    laura.right( 90 );

    john.forward( 20 );
    john.right( 90 );
    john.forward( 20 );
    john.left( 90 );
    laura.forward( 20 );
    laura.left( 90 );
    laura.forward( 20 );
    laura.right( 90 );

    john.forward( 20 );
    john.right( 90 );
    john.forward( 20 );
    john.left( 90 );

    laura.forward( 20 );
    laura.left( 90 );
    laura.forward( 20 );
    laura.right( 90 );
}
```

Step 3: Repeat structure "repeat", more than one window

Aim: Getting to know the structure "repeat".

Procedure: It is impractical to write down identical program parts repeatedly. One of the most important program structures is repetition. *repeat(n)* repeats an instruction, resp. a block of instructions n times.

Remarks: Structuring into blocks with a bracket at the beginning and at the end of the block becomes necessary as soon as more than one instruction is to be executed within the "body". Attention should be paid to the screen design (location of brackets, indents) applied here and in the following.

repeat is a structure which is not part of the programming language C++, but which is provided by Champ to be used by the beginner.

In many cases it is instructive to see the turtle doing its work. Because most computers are too fast, you may use *speed* to slow down the movement.

Additions: The global function ginit("Learn03") actually creates a CPWindow object and makes it visible. In order to cultivate object oriented thinking, it would be better to use

```
CPWindow myWindow( "Learn03" );
```

instead. However, in this case an instantiated turtle must explicitly be positioned inside the window:

```
  Turtle john;
  john.setPos( myWindow, 0, 0 );
```

Whenever a window is instantiated, it automatically becomes visible. If it is necessary to define some special properties (position, size, scrollbars etc.) before displaying the window, the line

```
    #define CP_INVISIBLE_WINDOW
```

may be added. Make the window visible or hidden any time using *show(true)* resp. *show(false)*.

It is to be remembered that the turtle vanishes from the screen as soon as the window object exits its scope. This is why

```
    getch();
```

is inserted in order to stop the execution until a key is pressed.

Let "john" draw a right stair in the window "water" and "laura" draw a left stair in the window "land".

In order to give a real Windows “look and feel“ Champ graphics windows may be easily embedded into a frame window.

Create a frame “playLand“ and let “john“ draw a stair in window „forest“ and, at the same time, “laura“ draw a circle in window „gras“.

```
// LEARN03.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn03" );

   Turtle john;
   john.speed( 20 );     // John is going slowly now

   repeat ( 5 )          // Repeat 5 times
   {
      john.forward( 20 );
      john.right( 90 );
      john.forward( 20 );
      john.left( 90 );
   }
}


// LEARN03A.CPP
```

```
#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   CPWindow grass( "Grass" );
   CPWindow forest( "Forest" );

   Turtle::speed( 100 );
   Turtle john;
   Turtle laura;
   john.setPos( forest, 0, 0 );
   laura.setPos( grass, 0, 0 );

   repeat ( 20 )
   {
      john.forward( 10 );
      john.right( 90 );
      john.forward( 10 );
      john.left( 90 );
      laura.forward( 10 );
      laura.right( 18 );
   }
   getch();
}


// LEARN03B.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   CPFrame playLand;
   CPWindow gras( "Gras" );
   CPWindow forest( "Forest" );

   Turtle::speed( 100 );
   Turtle john;
   Turtle laura;
   john.setPos( forest, 0, 0 );
   laura.setPos( gras, 0, 0 );

   repeat ( 20 )
   {
      john.forward( 10 );
      john.right( 90 );
      john.forward( 10 );
      john.left( 90 );

      laura.forward( 10 );
      laura.right( 18 );
   }
   getch();
}
```

Step 4: Repeat structure "while"

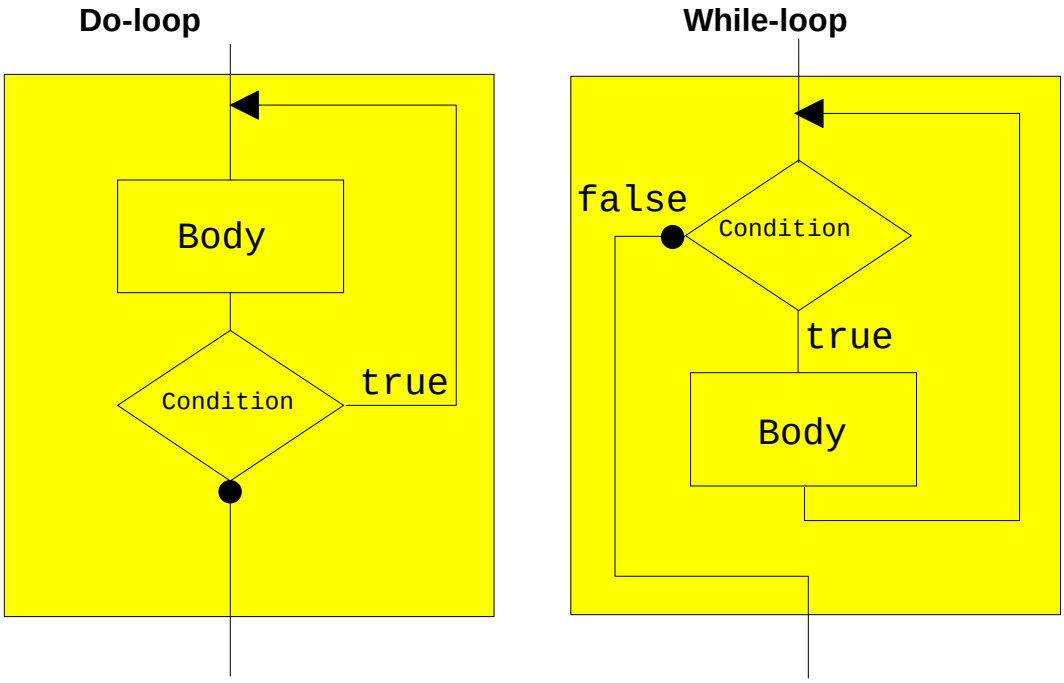
Aim: Geting to know the "while" loop.

Procedure: Draw a stair with 5 steps with a step counter.

Remarks: Introduction of an integer variable i. Instead of instantiation, this process is also called "variable definition with initialization".

The loop's condition can be at the beginning or at the end of the body:

while loop:	condition at the beginning (test before the body)
repeat loop:	condition at the end (test after the body)




```
// LEARN04.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn04" );
   Turtle john;
   int i = 0;
   while ( i < 5 )
   {
      john.forward( 20 );
      john.right( 90 );
      john.forward( 20 );
      john.left( 90 );
      i = i + 1;
   }
   getch();
```

```
}
```

Step 5: Repeat structure "do"

Aim: Getting to know the "do-while" loop.

Procedure: Draw a stair with 5 steps, doing the test after the body.

Remarks: This is a condition for the execution of the body, not a break condition.

```
// LEARN05.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn05" );

   Turtle john;
   john.speed( 40 );

   int i = 0;
   do
   {
      john.forward( 20 );
      john.right( 90 );
      john.forward( 20 );
      john.left( 90 );
      i = i + 1;
   } while ( i < 5 );

   getch();
   gend();
}
```

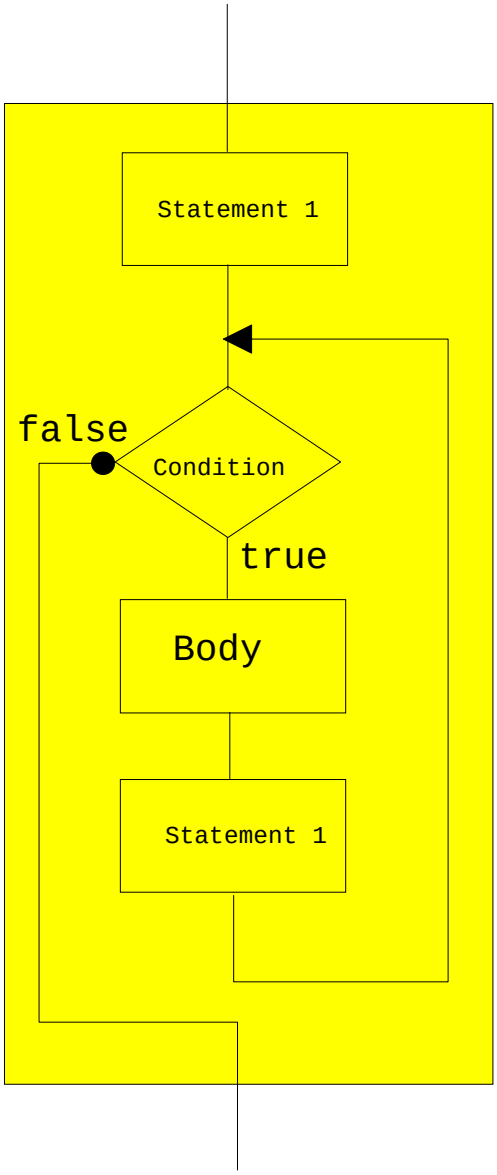
Step 6: Repeat structure "for"

Aim: Getting to know the "for" loop.

Procedure: The automatic loop counter simplifies notation.

Remarks: Basically this is a "while" loop. Please note that the condition is the running and not the break condition as is customary in other programming languages.

for (statement1; condition; statement2)




```
// LEARN06.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn06" );

   Turtle john;
   john.speed( 40 );

   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );
      john.right( 90 );
      john.forward( 20 );
      john.left( 90 );
   }

   getch();
   gend();
}
```

Step 7: "if-else" selection

Aim: Getting to know "if-else" selection.

Procedure: Draw a left or a right stair with a query.

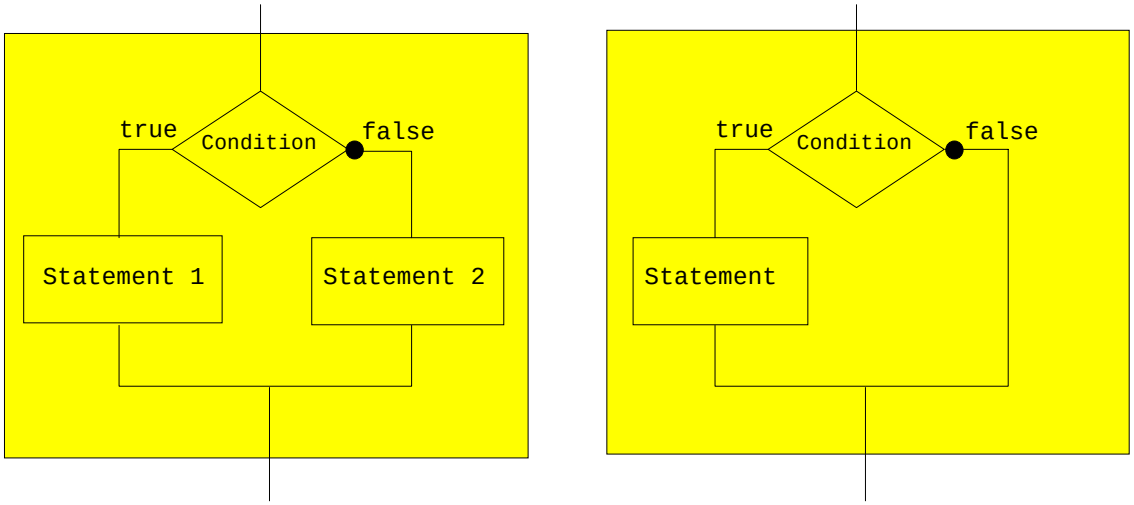
Remarks: This is a "twofold selection". (Single selections don't have the "else" part.) Here, too, flow charts may be used for further illustration.

Note the difference between the assignment with = and the condition for equality with ==, a somewhat uncustomary form of notation. However, the C++ compiler provides simple equality marks with warnings (if all warnings are switched on, the way they should be, and no special bracket techniques are used).

Text is displayed in a console window through the object *cout* (can be understood as an object, namely the text screen). Here, a message is sent with the operator <<. (An "include" of the file "iostream.h" has to be made.)

The reading of a key is generally done with the object *cin* (can be understood as an object, namely the keyboard). The character typed is assigned to the variable "answer" with the operator >>. In real Windows programs it is customary not to use console windows. In this case, a text is read through an input dialog. In order to use cin, the console window must be initalized and displayed using *cinit*.

Because they make programs confusing, deeply nested "if-else" structures are to be avoided.



Supplements: Validating the input, use of the operators &&, ¦¦, !

```
// LEARN07.CPP

#define OBJECT_TURTLE
#include <champ.h>
```

```
#include <iostream.h>
#include <conio.h>

void gmain ()
{
   ginit( "Learn07" );
   cinit( "Learn07", 10, 40 );

   Turtle john;
   char answer;
   int i;

   cout << "Left or right (l,r)? ";
   cin >> answer;

   if ( answer == 'l' )
   {
      for ( i = 0; i < 5; i++ )
      {
         john.forward( 20 );
         john.left( 90 );
         john.forward( 20 );
         john.right( 90 );
      }
   }
   else
   {
       for ( i = 0; i < 5; i++ )
       {
          john.forward( 20 );
          john.right( 90 );
          john.forward( 20 );
          john.left( 90 );
       }
   }

   getch();
   gend();
   cend();
}
```

Step 8: Multiple selection "switch"

Aim: Getting to know the multiple selection "switch".

Procedure: Draw a left or a right stair with a query, using the "switch" structure.

Remarks: After testing the condition, the program branches off to the corresponding place in the "switch" body (as if there were a jump to a label). From here, the program is executed to the end of the body, or to the next break instruction. A break instruction causes the rest of the body to be skipped.

The default section is optional. It is jumped to when none of the conditions are fulfilled.

A uniform technique of inserting blocks is to be applied, and the body should be well structured. If the same body section is to be executed on different conditions, several "case"s have to be written, one below the other.

To give a better, “look and feel“, input via the console window is replaced by an input box. Champ provides predefined (template-) classes for the standard data types. To display the box, use the member function *showModal*.

Supplements: Repeat the input after a wrong answer.

```
// LEARN08.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn08" );

   Turtle john;
   int i;
   char answer = 'r';

   CPInputChar( "Select stair", "Left or right?",
                answer, 'r' ).showModal();

   switch ( answer )
   {
      case 'l':
      case 'L':
         for ( i = 0; i < 5; i++ )
         {
            john.forward( 20 );
            john.left( 90 );
            john.forward( 20 );
            john.right( 90 );
         }
         break;

      case 'r':
      case 'R':
         for ( i = 0; i < 5; i++ )
         {
            john.forward( 20 );
            john.right( 90 );
            john.forward( 20 );
            john.left( 90 );
         }
         break;

      default:
         CP::msgBox( "Report", MB_OK,
                     "Illegal entry" );
   }
   getch();
   gend();
```

```
}


// LEARN08A.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include <iostream.h>

void gmain ()
{
   ginit( "Learn08a" );

   Turtle john;
   int i;
   char answer = 'r';
   bool isIllegal = true;

   while ( isIllegal )
   {
      CPInputChar( "Select stair", "Left or right?",
                   answer, 'r' ).showModal();
      switch ( answer )
      {
         case 'l':
         case 'L':
            for ( i = 0; i < 5; i++ )
            {
               john.forward( 20 );
               john.left( 90 );
               john.forward( 20 );
               john.right( 90 );
               isIllegal = false;
            }
            break;

         case 'r':
         case 'R':
            for ( i = 0; i < 5; i++ )
            {
               john.forward( 20 );
               john.right( 90 );
               john.forward( 20 );
               john.left( 90 );
               isIllegal = false;
            }
            break;

         default:
            CP::msgBox( "Report", MB_OK,
                        "Illegal entry" );
       }
   }
   getch();
   gend();
}
```

Step 9: Functions

Aim: Realizing that it is necessary to do modular programming (user defined functions).

Procedure: Self-contained program sections, called functions, are given a name of their own. Define *leftStair* and *rightStair* to draw the stair.

Remarks: The object "john" is instantiated (locally) within the function, which means that it exists only within the function, i.e. at the end of the function call it is destroyed.

Function declaration (also called function prototype), function definition and function call are to be clearly distinguished. Therefore we put an extra space between the name and the parameter parenthesis in the function declaration and definition.

Capitalization in names of functions should be consistent. Underscores are only used in special cases. Double underscores should be avoided because they are reserve for internal compiler use.

It is usual to omit the parameter type "void".

Function definitions can only take place on the outmost level (not inside functions).

Supplements: Let the function return the length of distance covered, and the main program print it out.

Remarks: Returning a value is done with *return*. With the execution of *return*, the function is broken off, and the value evaluated in the return statement is given back to the caller. A function is allowed to have more than one *return*, but this should be used with care because it makes programs confusing.

```
// LEARN09.CPP

#define OBJECT_TURTLE
#include <champ.h>

void leftStair ();
void rightStair ();

void gmain ()
{
   ginit( "Learn09" );
   char answer = 'r';

   CPInputChar( "Select stair", "Left or right?",
                answer, 'r' ).showModal();
```

```
   if ( answer == 'l' )
      leftStair();
   else
      rightStair();

   getch();
   gend();
}

void leftStair ()
{
   Turtle john;
   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );
      john.left( 90 );
      john.forward( 20 );
      john.right( 90 );
   }
}

void rightStair ()
{
   Turtle john;
   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );
      john.right( 90 );
      john.forward( 20 );
      john.left( 90 );
   }
}


// LEARN09A.CPP

#define OBJECT_TURTLE
#include <champ.h>

float leftStair ();
float rightStair ();

void gmain ()
{
   float length;

   ginit( "Learn09a" );
   cinit( "Learn09a", 10, 40, CPPosition( 300, 200 ) );
   char answer = 'r';

   CPInputChar( "Select stair", "Left or right?",
                answer, 'r' ).showModal();

   if ( answer == 'l' )
      length = leftStair();
   else
      length = rightStair();

   cactivate();
```

```
   cout << "Length of turtle's path: " << length;
   getch();
   cend();
   gend();
}

float leftStair ()
{
   Turtle john;
   float pathLength = 0;

   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );
      pathLength += 20;
      john.left( 90 );
      john.forward( 20 );
      pathLength += 20;
      john.right( 90 );
   }
   return pathLength;
}

float rightStair ()
{
   Turtle john;
   float pathLength = 0;

   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );
      pathLength += 20;
      john.right( 90 );
      john.forward( 20 );
      pathLength += 20;
      john.left( 90 );
   }
   return pathLength;
}
```

Step 10: Function parameters

Aim: Getting to know function parameters.

Procedure: The program becomes simpler and clearer, if it is possible to pass information of the kind of the stair when calling the function.

Remarks: Formal parameters (in function definitions and function declarations) should be distinguished from actual parameters (in function calls, sometimes called “arguments“).

The lists of parameters in the function definition and in the function declaration should be identical.

Function parameters are "calls-by-value", i.e. in the call, only the value of the parameter is passed to the function. (Actually, the value is copied to a

local temporary variable.)

Constant characters must be put between single quotation marks.

Objects (variables) instantiated in *gmain* are local to gmain and therefore "invisible" in other functions.

```
// LEARN10.CPP

#define OBJECT_TURTLE
#include <champ.h>

void stair ( char direction );

void gmain ()
{
   ginit( "Learn10" );
   char answer = 'r';
   CPInputChar( "Select stair", "Left or right?",
                answer, 'r' ).showModal();
   stair( answer );
   getch();
   gend();
}

void stair ( char direction )
{
   Turtle john;
   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );

      if ( direction == 'l' )
         john.left( 90 );
      else
         john.right( 90 );

      john.forward( 20 );

      if ( direction == 'l' )
         john.right( 90 );
      else
         john.left( 90 );
   }
}
```

Step 11: Global variables

Aim: Avoiding global variables.

Procedure: Instead of passing variables to the functions, it is also possible to use global variables. Introduce a global variable instead of a call-by-value and arbitrarily cause a side effect.

Remarks: As a rule, global variables are but an apparent simplification. In fact, they are a source of numerous errors, since any section of the program can change them.

Supplement: Introduce a global and a local variable "answer" and explain the program's faulty behavior.

```
// LEARN11.CPP

#define OBJECT_TURTLE
#include <champ.h>

char answer = 'r';                    // Global variable

void stair ();
void sideeffect ();

void gmain ()
{
   ginit( "Learn11" );
   CPInputChar( "Select stair", "Left or right?",
                answer, 'r' ).showModal();
   stair();
   sideeffect();
   stair();
   getch();
   gend();
}

void stair ()
{
   Turtle john;
   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );

      if ( answer == 'l' )
         john.left( 90 );
      else
         john.right( 90 );

      john.forward( 20 );

      if ( answer == 'l' )
         john.right( 90 );
      else
         john.left( 90 );
   }
}

void sideeffect()
{
   answer = 'r';
}


// LEARN11A.CPP

#define OBJECT_TURTLE
```

```
#include <champ.h>

char answer = 'r';                    // Global variable

void stair ();
void sideeffect ();

void gmain ()
{
   ginit( "Learn11a" );

   char answer = 'l';
   // Local variable with same name as global

   CPInputChar( "Select stair", "Left or right?",
                answer, 'l' ).showModal();
   stair();
   sideeffect();
   stair();
   getch();
   gend();
}

void stair ()
{
   Turtle john;
   for ( int i = 0; i < 5; i++ )
   {
      john.forward( 20 );

      if ( answer == 'l' )
         john.left( 90 );
      else
         john.right( 90 );

      john.forward( 20 );

      if ( answer == 'l' )
         john.right( 90 );
      else
         john.left( 90 );
   }
}

void sideeffect ()
{
   answer = 'r';
}
```

Step 12: References

Aim: Getting to know references.

Procedure: Use a reference parameter instead of a call-by-value.

Remarks: In references, the function is passed the "location" of the variable (its address). Thus, the function can change its value.

Copying variables in a call-by-value always means a loss of time and storage space. In addition to that, for some objects the copying operation is not even defined, so that references are the only possible way to pass the objects to the function.

If the reference variable is not changed in the function, it should be passed to the function in the form of a constant variable (for reasons of protection from side effects).

Supplement: If you pass a turtle “by value“, a temporary copy of the turtle is made (in a storage region called “stack“). Show that this will cause a faulty program because the temporary and not the original turtle is moved.

```
// LEARN12.CPP

#define OBJECT_TURTLE
#include <champ.h>

void stair ( Turtle & aTurtle, char direction );

void gmain ()
{
   ginit( "Learn12" );
   Turtle john;
   char answer = 'r';
   CPInputChar( "Select stair", "Left or right?",
                answer, 'r' )
      .showModal( CPPosition( 200, 200 ) );
   stair( john, answer );
   getch();
   gend();
}

void stair ( Turtle & aTurtle, char direction )
{
   for ( int i = 0; i < 5; i++ )
   {
      aTurtle.forward( 20 );

      if ( direction == 'l' )
         aTurtle.left( 90 );
      else
         aTurtle.right( 90 );

      aTurtle.forward( 20 );

      if ( direction == 'l' )
         aTurtle.right( 90 );
      else
         aTurtle.left( 90 );
   }
}


// LEARN12A.CPP
```

```
#define OBJECT_TURTLE
#include <champ.h>

void stair ( Turtle aTurtle, char direction );

void gmain ()
{
   ginit( "Learn12a" );
   Turtle john;
   char answer = 'r';
   CPInputChar( "Select stair", "Left or right?",
                answer, 'r' )
      .showModal( CPPosition( 200, 200 ) );
   stair( john, answer );
   getch();
   john.forward( 100 );
   getch();
   gend();
}

void stair ( Turtle aTurtle, char direction )
{
   for ( int i = 0; i < 5; i++ )
   {
      aTurtle.forward( 20 );

      if ( direction == 'l' )
         aTurtle.left( 90 );
      else
         aTurtle.right( 90 );

      aTurtle.forward( 20 );

      if ( direction == 'l' )
         aTurtle.right( 90 );
      else
         aTurtle.left( 90 );
   }
   getch();
}
```

Step 13: Arrays

Aim: Getting to know arrays.

Procedure: Define a turtle family as an array and move the different family members in different directions.

Remarks: The number of array elements is defined in the variable definition of the array.

The array elements are referred to through their index. Unfortunately the index can only assume integer values starting from 0.

An array index overflow is not tested for. Errors usually cause a break during the execution time of the program, possibly even a system crash.

```
// LEARN13.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Turtle14" );

   Turtle family[4];

   family[0].forward( 50 );
   family[1].left( 90 );
   family[1].forward( 50 );
   family[2].left( 180 );
   family[2].forward( 50 );
   family[3].left( 270 );
   family[3].forward( 50 );

   getch();
   gend();
}
```

Step 14: Strings

Aim: Getting to know strings as arrays.

Procedure: Introduce a family name for the turtle family and first names for its different members.

Remarks: The string is defined as a character array. The ASCII code of the characters is saved in consecutive memory field that must always be terminated with the zero byte (called “null character“).

Most string functions automatically process the terminating null character. However, it must be made sure that there is enough space for it, i.e. a string of the length n has to have room for n+1 bytes.

Constant strings are put between double quotation marks. The terminating character is automatically included.

A string cannot be filled with several characters through an assignment. Instead, the string function *strcpy* is used.

Arrays of strings are implemented as arrays with a double index.

Supplements: Fill the strings through initialization.

```
// LEARN14.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include <iostream.h>
```

```
#include <string.h>
#include <conio.h>

void gmain ()
{
   ginit( "Learn14" );
   cinit( "Learn14", 10, 40, CPPosition( 250, 200 ) );

   Turtle family[4];
   char familyName[10];
   char firstName[4][10];

   strcpy( familyName, "Smith" );
   strcpy( firstName[0], "Dad" );
   strcpy( firstName[1], "Mam" );
   strcpy( firstName[2], "Jack" );
   strcpy( firstName[3], "Su" );

   for ( int i = 0; i < 4; i++ )
   {
      cout << firstName[i] << " " << familyName
                           << " moving now..." << endl;
      family[i].left( 90*i );
      family[i].forward( 50 );
      getch();
   }
   cend();
   gend();
}


// LEARN14A.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include <iostream.h>
#include <conio.h>

void gmain ()
{
   ginit( "Learn14a" );
   cinit( "Learn14a", 10, 40, CPPosition( 250, 200 ) );

   Turtle family[4];
   char familyName[] = "Smith";
   char firstName[][10] = { "Dad", "Mam",
                            "Jack", "Su" };

   for ( int i = 0; i < 4; i++ )
   {
      cout << firstName[i] << " " << familyName
           << "moving now... " << endl;
      family[i].left( 90*i );
      family[i].forward( 50 );
      getch();
   }
   cend();
   gend();
}
```

Step 15: Pointers

Aim: Getting to know pointers.

Procedure: Move a turtle which is referred to as a pointer.

Remarks: Pointers (pointer variables) are important particularly in connection with dynamic data structures, i.e. if an object is created only during execution time and the storage space is to be released again later. (Storage space for a dynamic object is allocated on the so-called "heap" with the operator "new" and freed with the operator "delete".)

The object to which the pointer points is referred to through the dereferencation of the pointer (with the star operator).

A pointer's value does not make sense when being defined without initialization. Prior to its first dereferencation, the pointer must be given a meaningful value, either through an initialization or an assignment. If this is not done, runtime errors might cause the computer to crash.

An object referred to with a pointer should be destroyed when not used any longer (in order to release storage space). When the pointer is given a new value without the object having been destroyed beforehand, a "data corpse" is left behind and wastes storage space. (C++ does not have automatic "garbage collection").

Supplements: Use the arrow operator to access member functions and data elements of an object referred to through a pointer.

```
// LEARN15.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn15" );

   Turtle * pLaura;
   pLaura = new Turtle;  // Or simply: Turtle * pLaura
= new Turtle;

   (*pLaura).speed( 40 );
   (*pLaura).forward( 100 );
   (*pLaura).stampTurtle();
   (*pLaura).right( 90 );
   (*pLaura).forward( 100 );
   getch();
   delete pLaura;
   getch();
   gend();
}
```

```
// LEARN15A.CPP

#define OBJECT_TURTLE
#include <champ.h>

void gmain ()
{
   ginit( "Learn15a" );

   Turtle * pLaura;
   pLaura = new Turtle;

   pLaura->speed( 40 );
   pLaura->forward( 100 );
   pLaura->stampTurtle();
   pLaura->right( 90 );
   pLaura->forward( 100 );
   getch();
   delete pLaura;
   getch();
   gend();
}
```

Step 16: Menus

Aim: Getting to know an event driven program with a menu.

Procedure: Write a program that has a menu with two entries "go" and "exit". When the mouse is clicked on "go", the turtle starts moving endlessly in a circle until the program is terminated by a mouseclick on "exit".

Remarks: The menu is designed using the Resource Workshop. The workshop creates a resource file LEARN16.RC that must be included in the project.

Menu items that were defined with the Resource Workshop are selected by the program using an "identificaton", i.e. a integer number. A symbol for this number is defined in the include file LEARN16.RH that is accessible for both the Resource Workshop and to the source program.

The menu is instantiated as an object of the class CPMenu. The menu name given by the resource editor must be identical to that stated in the instantiation. The menu options have corresponding global functions that are registered with *registerMenuItem*. Every time the item is clicked, the corresponding function is called ("callback" function).

In order to assign the menu to a window, a reference to the menu object is passed when the window is instatiated.

The main program executes an endless loop,where the value of program „state“ is tested. When the system is in the „stopped“ state, the function *CP::yield* must be called in order to force the system to serve any Windows events (the operation system is not preemtive but cooperative).

Supplements: Rewrite the same program using the *addMenuItem* and *addSubMenu* functions to create a menu without using the Resource Workshop.

```
// LEARN16.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include "learn16.rh"

void do_go ();
void do_stop ();
void do_exit ();

enum { stopped, running, aborting } state = stopped;

void gmain ()
{
   CPMenu myMenu( "Learn16_Menu" );
   myMenu.registerMenuItem( IDM_GO, do_go );
   myMenu.registerMenuItem( IDM_STOP, do_stop );
   myMenu.registerMenuItem( IDM_EXIT, do_exit );

   ginit( "Learn16", myMenu );
   Turtle john;
   john.speed( 40 );

   do {
      switch ( state )
      {
         case stopped:
            CP::yield();
            break;

         case running:
            john.forward( 10 ).left( 10 );
            break;
       }
  } while ( state != aborting );

  gend();
}

void do_go ()
{
   state = running;
}

void do_stop ()
{
   state = stopped;
}

void do_exit ()
{
   state = aborting;
}}
```

```
// LEARN16A.CPP

#define OBJECT_TURTLE
#include <champ.h>

enum { stopped, running, aborting } state = stopped;

void do_go ();
void do_stop ();
void do_exit ();
void redColor ();
void greenColor ();
void yellowPenColor ();
void cyanPenColor ();

Turtle john;

void gmain ()
{
   CPMenu myMenu;
   myMenu.addMenuItem( "&Go", do_go );
   myMenu.addMenuItem( "&Stop", do_stop );
   myMenu.addSeparator();
   myMenu.addMenuItem( "&Exit", do_exit );

   CPSubMenu optionMenu =
      myMenu.addSubMenu( "Options" );

   CPSubMenu colorMenu =
      optionMenu.addSubMenu( "Turtle Color" );
   colorMenu.addMenuItem( "Green", greenColor );
   colorMenu.addMenuItem( "Red", redColor );

   CPSubMenu penMenu =
      optionMenu.addSubMenu( "Pen Color" );
   penMenu.addMenuItem( "Yellow", yellowPenColor );
   penMenu.addMenuItem( "Cyan", cyanPenColor );

   CPWindow wnd( "Learn16a", myMenu );
   john.speed( 40 );
   john.setPos( wnd, 0, 0 );

   do {
      switch ( state )
      {
         case stopped:
            CP::yield();
            break;

         case running:
            john.forward( 10 ).left( 10 );
            break;
       }
  } while ( state != aborting );

  gend();
}

void do_go ()
{
```

```
   state = running;
}

void do_stop ()
{
   state = stopped;
}

void do_exit ()
{
   state = aborting;
}

void redColor ()
{
   john.turtleColor( RED );
}

void greenColor ()
{
   john.turtleColor( GREEN );
}

void yellowPenColor ()
{
   john.penColor( YELLOW );
}

void cyanPenColor ()
{
   john.penColor( CYAN );
}
```

Step 17: Modeless dialogs

Aim: Getting to know modeless dialog boxes.

Procedure: Write a program with a "buttonbar" that should contain the buttons "go", "stop", "cont", and "exit". When "go" is clicked, the turtle starts moving in a circle, until "stop" or "exit" is clicked. "cont" lets the turtle continue the circular motion that was stopped with "stop".

Remarks: Modeless dialogs are displayed on the screen with *showModeless*. Since the function returns afterwards, events are processed exclusively with callback functions. With *close*, an open dialog can be closed.

The button bar is designed as a dialog containing buttons with the Resource Workshop. The workshop creates a file LEANR17.RC, which must be included in the project.

The link between the buttons created with the Resource Workshop and the source program is established by means of an "identification", i.e. a symbolic integer defined in LEARN17.RH.

The dialog is instantiated as an object of the class *CPModelessDialog*.

The dialog's name given by the resource editor must be identical with the name registered in the instantiation. The buttons are objects of the class *CPButton* and have a corresponding global function registered in the instantiation of the button object. Every time a button is clicked, the corresponding callback function is called.

When the member function *showModeless*() is called, the dialog is displayed on the screen. If a (reference or pointer of a) Champ window is passed as a parameter, then this window becomes the "owner" of the dialog. The constant *CP::upperLeft* "attachs" the dialog to the upper left edge of the window, so that the dialog moves together with the graphics window when it is moved on the screen.

In an endless loop, the main program moves the turtle. After "stop", the program must call *CP::yield* in order for window events still to be processed.

```
// LEARN17.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include "learn17.rh"

enum { stopped, running, aborting } state = stopped;

void do_go ();
void do_stop ();
void do_exit ();

void gmain()
{
   CPWindow wnd( "Learn17" );
   wnd.clear( LIGHTGREEN );

   CPModelessDialog btnBar( "ButtonBar" );
   CPButton   goButton( btnBar, IDB_GO,   do_go );
   CPButton stopButton( btnBar, IDB_STOP, do_stop );
   CPButton exitButton( btnBar, IDB_EXIT, do_exit );

   btnBar.showModeless( wnd, CP::upperLeft );

   Turtle john;
   john.speed( 40 );
   john.setPos( wnd, 0, 0 );

   do
      switch ( state )
      {
         case stopped:
            CP::yield();
            break;

         case running:
            john.forward( 10 ).left( 10 );
            break;
       }
   while ( state != aborting );
```

```
}

void do_go ()
{
   state = running;
}

void do_stop ()
{
   state = stopped;
}

void do_exit ()
{
   state = aborting;
}
```

Step 18: Modal dialogs, radio buttons

Aim: Getting to know modal dialog windows.

Procedure: Write a program that opens a modal dialog with 3 radio buttons when started. When the user has selected either "triangle" or "square" or "star", the corresponding shape is drawn.

Remarks: In contrast to modeless dialogs, *showModal* displays the box on screen, but the function does not return until the dialog is closed. If the dialog contains an OK button or a Cancel button with the IDs IDOK resp. IDCANCEL, the dialog is closed by a click on either of them. The function returns the ID of the button clicked. Afterwards, it is possible to check which radio button was clicked by calling *isChecked*.

The dialog is created with the Resource Workshop. Since the buttons are radio buttons (e.g. exactly one button is checked), they should be inserted in a groupbox.

The dialog is instantiated as an object of the class *CPModalDialog*. The dialog's name given by the resource editor must be identical with the one stated in the instantiation. The radio buttons are objects of the class *CPRadioButton*.

Supplements: Write a program with check buttons. All selected shapes are drawn.

```
// LEARN18.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include "learn18.rh"

void drawTriangle ();
void drawSquare ();
void drawStar ();

void gmain ()
```

```
{
   ginit( "Learn18" );
   gclear( LIGHTGREEN );

   CPModalDialog myDialog( "RadioBox" );
   CPRadioButton but1( myDialog, IDC1, IDC1, IDC3 );
   CPRadioButton but2( myDialog, IDC2, IDC1, IDC3 );
   CPRadioButton but3( myDialog, IDC3, IDC1, IDC3 );

   if ( myDialog.showModal() == IDOK )
      if ( but1.isChecked() )
         drawTriangle();
      else
         if ( but2.isChecked() )
            drawSquare();
         else
            if ( but3.isChecked() )
               drawStar();

   getch();
   gend();
}

void drawTriangle ()
{
   Turtle john;
   repeat( 3 )
      john.forward( 100 ).left( 120 );
}

void drawSquare ()
{
   Turtle john;
   repeat( 4 )
      john.forward( 100 ).left( 90 );
}

void drawStar ()
{
   Turtle john;
   repeat( 12 )
      john.forward( 100 ).left( 150 );
}


// LEARN18A.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include "learn18a.rh"

void drawTriangle ();
void drawSquare ();
void drawStar ();

void gmain ()
{
   ginit( "Learn18a" );
   gclear( LIGHTGREEN );
```

```
   CPModalDialog myDialog( "CheckBox" );
   CPCheckButton but1( myDialog, IDC1 );
   CPCheckButton but2( myDialog, IDC2 );
   CPCheckButton but3( myDialog, IDC3 );

   if ( myDialog.showModal() == IDOK )
   {
      if ( but1.isChecked() )
         drawTriangle();
      if ( but2.isChecked() )
         drawSquare();
      if ( but3.isChecked() )
         drawStar();
   }
   getch();
   gend();
}

void drawTriangle ()
{
   Turtle john;
   repeat( 3 )
      john.forward( 100 ).left( 120 );
}

void drawSquare ()
{
   Turtle john;
   repeat( 4 )
      john.forward( 100 ).left( 90 );
}

void drawStar ()
{
   Turtle john;
   repeat( 12 )
      john.forward( 100 ).left( 150 );
}
```

Step 19: Scrollbars

Aim: Getting to know scrollbars.

Procedure: Write a program that creates a CPWindow with a horizontal and a vertical scrollbar with which the position of the turtle can be adjusted.

Remarks: Scrollbars belonging to a CPWindow are instances of the classes *CPHorzScrollbar* resp. *CPVertScrollbar*. They are automatically created by calling *addHorzScrollbar* resp. *addVertScrollbar*. They take the name of a callback function that is automatically called when the scrollbar's thumb changes its position.

In order to add the scrollbars before the window is displayed, insert the following line before including champ.h

```
#define CP_INVISIBLE_WINDOWS
```

By default, the callback function is called once, when the thumb reaches its final position. *setTrack(true)* causes the callback function to be invoked whenever the the thumb reaches a new position value. This may cause problems if the callback function does not return fast enough.

Supplements: Write a program with a scrollbar in a modeless dialog window. The turtle should continually move in a circle. Its velocity should be adjustable by means of the scrollbar.

```
// LEARN19.CPP

#define CP_INVISIBLE_WINDOWS
#define OBJECT_TURTLE
#include <champ.h>

void horzProc ( int pos );
void vertProc ( int pos );

Turtle john;
int horzPos = 0;
int vertPos = 0;

void gmain ()
{
   CPWindow wnd( "Learn19" );

   wnd.addHorzScrollbar( horzProc );
   wnd.horzScrollbar().setRange( -150, 150, 1, 10 );
   wnd.horzScrollbar().setPosition( 0 );
   wnd.horzScrollbar().setTrack( true );
   wnd.addVertScrollbar( vertProc );
   wnd.vertScrollbar().setRange( -150, 150, 1, 10 );
   wnd.vertScrollbar().setPosition( 0 );
   wnd.vertScrollbar().setTrack( true );

   wnd.show();   // Display window with scrollbars

   john.setPos( wnd, 0, 0 );
   getch();
}

void horzProc ( int pos )
{
   if ( horzPos < pos )
      john.setHeading(90);
   else
      if ( horzPos > pos )
         john.setHeading(270);

   john.setPos( pos, -vertPos );
   horzPos = pos;
}

void vertProc ( int pos )
{
   if ( vertPos < pos )
      john.setHeading( 180 );
```

```
   else
      if ( vertPos > pos )
         john.setHeading( 0 );

   john.setPos( horzPos, -pos );
   vertPos = pos;
}


// LEARN19A.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include "learn19a.rh"

void scrollbarProc( int pos );
Turtle john( RED );

void gmain ()
{
   CPWindow wnd( "Learn19a" );
   wnd.clear( LIGHTGREEN );

   CPModelessDialog myDialog( "ScrollBox" );
   CPScrollbar myBar( myDialog, IDC_BAR, scrollbarProc
);
   myBar.setRange( 1, 500 );
   myBar.setPosition( 250 );
   myDialog.showModeless( wnd, CP::lowerLeft );

   john.setPos( wnd, 0, 0 );
   john.speed( 250 );

   while ( !CP::kbhit() )
   {
     john.forward( 10 );
     john.left( 10 );
   }
}


void scrollbarProc ( int pos )
{
   john.speed( pos );
}
```

Step 20: Keyboard events

Aim: Learning how keyboard events are used.

Procedure: Write a program to move a turtle with the cursor keys.

Remarks: Keyboard events can be evaluated through an object of the class *CPKeyboard*. Using *attach*, the keyboard object “belongs“ to the corresponding window, e.g. the function passed to *registerCallback* is called if the window has the focus and a key is pressed.

The parameter values passed to the callback functions may be used to determine the kind of event occured.

```
// LEARN20.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include <cpkbd.h>

#define CURSORLEFT  37
#define CURSORUP    38
#define CURSORRIGHT 39
#define CURSORDOWN  40
#define ESCAPE  27

void kbdProc ( CPKeyboard::Event event,
               unsigned keycode );

CPTurtle john( RED );

void gmain ()
{
   CPWindow wnd( "Learn20" );
   CPKeyboard kbd;
   kbd.registerCallback( kbdProc );
   wnd.attach( kbd );
   john.setPos( wnd, 0, 0 );

   wnd.pos( 20, 0 );
   wnd.text("Press cursor keys to move");
   wnd.pos( 20, -20 );
   wnd.text("or <Esc> to quit");

   CP::run();
}


void kbdProc ( CPKeyboard::Event event,
               unsigned keycode )
{
   if ( event == CPKeyboard::keyDown )
   {
      switch ( keycode )
      {
         case CURSORLEFT:
            john.setHeading( 270 );
            break;

         case CURSORRIGHT:
            john.setHeading( 90 );
            break;

         case CURSORUP:
            john.setHeading( 0 );
            break;

         case CURSORDOWN:
           john.setHeading( 180 );
           break;
```

```
            case ESCAPE:
               CP::quit();
               break;
         }
         john.forward( 5 );
      }
   }
```

Step 21: Mouse events

Aim: Learning how mouse events are used.

Procedure: Write a program that causes a new member of a turtle family (maximum of six members) to be produced when the mouse is double-clicked (left button) on the cursor position. Further, the program should enable you to "catch" a turtle by clicking the left mouse button near it, and then to move it by dragging it with the mouse (during which the cursor shape should change to a cross).

Remarks: To use the mouse in a given window, the constructor of CPMouse takes (a reference to) the window. An OR-mask given to *registerEvent* lets you enable the particular events that calls the given callback function. For efficency reasons only a minimum number of events should be enabled.

The callback function receives a reference to the mouse instance to let you retrieve the information on which event happend.

In order for mouse events outside the active window to be registered as well, the mouse must be "caught" with

```
setCapture();
```

In order for the mouse to be at the disposal of other programs again, particularly the Windows environment,

```
releaseCapture();
```

must be called.

When the turtles are global instances, Champ uses *setPos* to set the turtle coordinate system. Because the mouse coordinates are retrieved before calling *setPos*, the coordinate system must be set explicitly at the beginning of the program.

Supplements: Write a program for drawing a line with the left mouse button (clicking and pulling produces a rubber band line).

Remarks: While pulling the rubber band line, the drawing mode XOR must be used with

```
setRop2( R2_XORPEN );
```

Further, the pen color must be adapted to the background color using the XOR operator. When the mouse button is released, the line is drawn with

```
    setRop2( R2_COPYPEN );
```

in order to avoid holes where the line crosses other lines.

```
// LEARN21.CPP

#define OBJECT_TURTLE
#include <champ.h>

#define FAMILYSIZE 6

void mouseProc ( CPMouse & aMouse );
Turtle family[FAMILYSIZE];

void gmain ()
{
   CPWindow wnd( "Learn21" );
// Because turtles are global (setPos comes too late)
   wnd.window( -200, 200, -200, 200 );
   CPMouse micky( wnd );
   micky.registerEvents( CPMouse::lPress |
                         CPMouse::lRelease |
                         CPMouse::lDClick |
                         CPMouse::rDClick |
                         CPMouse::move,
                         mouseProc );
   wnd.pos( -100, 100 );
   wnd.text( "Right mouse double click to quit" );
   CP::run();
}

void mouseProc ( CPMouse & aMouse )
{
  static int member = 0;
  static int caughtMember = -1;  // No turtle caught
  const float xRegion = 10;
  const float yRegion = 10;
  int i;

  switch ( aMouse.event() )
  {
     case CPMouse::lPress:
     // Called also once when clicked
     // or once(!) when doubleclicked
        for ( i = 0; i < member; i++ )
        {
           if ( fabs( aMouse.userX() -
                family[i].xCor() ) < xRegion &&
              ( fabs( aMouse.userY() -
                family[i].yCor() ) < yRegion ) )
           {
              caughtMember = i;
              aMouse.setCapture( aMouse.window() );
              // In order to get a lRelease event when
              // released outside the turtle window
```

```
                aMouse.setCursor( IDC_CROSS );
                break;
             }
          }
          break;

       case CPMouse::lRelease:
       // Called also once when clicked
       // or twice when doubleclicked
          if ( caughtMember != -1 )
          {
             caughtMember = -1;    // No turtle caught
             aMouse.releaseCapture();
             // Mandatory
             aMouse.setCursor( IDC_ARROW ); //
          }
          break;

       case CPMouse::lDClick:
          if ( member < FAMILYSIZE )
          {
             family[member].turtleColor( member+1 );
             family[member].setPos( aMouse.window(),
             aMouse.userX(), aMouse.userY() );
             member++;
             if ( member == FAMILYSIZE ) {
                aMouse.window().pos( aMouse.userX()+20,
                                     aMouse.userY() );
                aMouse.window()
                   .text("Last member of family");
             }
          }
          break;

       case CPMouse::move:
          if ( caughtMember != -1 )
          // A turtle is caught
             family[caughtMember]
                .setPos( aMouse.userX(),
                         aMouse.userY() );
          break;

       case CPMouse::rDClick:
          CP::quit();
          break;
    }
}


// LEARN21A.CPP

#include <champ.h>
#define PENCOLOR WHITE
#define BACKGROUND LIGHTBLUE
#define XORCOLOR PENCOLOR ^ BACKGROUND

void drawLine ( CPMouse& aMouse );

void gmain ()
{
```

```
   CPWindow wnd( "Learn21a" );
   CPMouse mickey( wnd );
   mickey.registerEvents( CPMouse::move |
                          CPMouse::lPress |
                          CPMouse::lRelease,
                          drawLine);
   wnd.clear( BACKGROUND );
   CP::run();
}


void drawLine ( CPMouse & aMouse )
{
   static bool isRubberBand = false;
   static float x0, y0, x1, y1;
   CPWindow & wnd = aMouse.window();

   switch ( aMouse.event() )
   {
      case CPMouse::move:
         if ( isRubberBand )
         {
            // Erase old line
            wnd.pos( x0, y0 );
            wnd.draw( x1, y1 );
            // Draw new line
            x1 = aMouse.userX();
            y1 = aMouse.userY();
            wnd.pos( x0, y0 );
            wnd.draw( x1, y1 );
         }
         break;

      case CPMouse::lPress:
         wnd.setRop2( R2_XORPEN );
         wnd.penColor( XORCOLOR );
         aMouse.setCapture( wnd );
         isRubberBand = true;
         aMouse.setCursor( IDC_CROSS );
         x0 = x1 = aMouse.userX();
         y0 = y1 = aMouse.userY();
         break;

      case CPMouse::lRelease:
         wnd.setRop2( R2_COPYPEN );
         wnd.penColor( PENCOLOR );
         wnd.pos( x0, y0 );
         wnd.draw( x1, y1 );
         aMouse.releaseCapture();
         isRubberBand = false;
         aMouse.setCursor( IDC_ARROW );
         break;
   }
}
```

Step 22: Text windows

Aim: Getting to know formatting text screen output using cout.

Procedure: Study the sample program that demonstrates the format possibilites of cout and some features of cin.

Remarks: Even though "genuine" Windows programs do not use text windows, it can very well be useful to use them in programming courses, for debugging purposes or in scientific applications, since they may greatly simplify programs. The exemplary program shows the main format possibilities.

```
// LEARN22.CPP

#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <champ.h>

void gmain ()
{
   cinit("Learn22", 20, 80);

   int i, m, n;
   long k;
   char ch;
   float x;
   double y;
   char word[10];

   cout << "Demonstration of some features of cout, "
           "cin stream objects "(this is a long line)";

// extraction operator
   cout << "\n\nEnter integer: ";
// insertion operator
   cin >> i;
// concatenate
   cout << "Got: " << i << " (dec)" << endl;
// octal from now on
   cout << oct << "     " << i << " (oct)" << endl;
// hex from now on
   cout << hex << "     " << i << " (hex)" << endl;
// uppercase from now on
   cout << setiosflags( ios::uppercase );
// hex in uppercase
   cout << "     " << i << " (HEX)" << endl;

   cout << "\n\nEnter long: ";
   cin >> k;
// now decimal again
   cout << dec << "Got: " << k << " (dec)" << endl;
   cout << oct << "     " << k << " (oct)" << endl;
   cout << hex << "     " << k << " (hex)" << endl;
// reset to decimal
   cout << dec;

   cout << "\nEnter decimal: ";
   cin >> x;
   cout << "Got: " << x << " (normal)" << endl;
// exponential format
   cout << setiosflags( ios::scientific );
```

```
   cout << "Got: " << x << " (scientific)" << endl;
// fixpoint format
   cout << setiosflags( ios::fixed );
// show decimal point
   cout << setiosflags( ios::showpoint );
// two decimals
   cout << setprecision(2);
   cout << "Got: " << x << " (normal, 2 digits)"
        << endl;
// ten decimals
   cout << setprecision( 10 );

   cout << "\nEnter double: ";
   cin >> y;
   cout << "Got: " << y << " (normal)" << endl;
   cout << setiosflags( ios::scientific );
   cout << "Got: " << y << " (scientific)" << endl;
   cout << resetiosflags( ios::scientific );

   cout << "\nEnter character: ";
   cin >> ch;
   cout << "Got: " << ch << endl;

   cout << "\nEnter string: ";
// limit to 9 char + '\0'
   cin >> setw( 10 ) >> word;
   cout << "Got: " << word << endl;

   cout << "\nDisplay random integer matrix now..."
        << endl << endl;
   for ( m = 0; m < 3; m++ )
   {
      for ( n = 0; n < 5; n++ )
         cout << setw( 10 ) << random( 1001 );
      cout << endl;
   }

   cout << "\n\nDisplay random float matrix now..."
        << endl << endl;
   cout << setprecision( 7 );
   for ( m = 0; m < 3; m++ )
   {
      for ( n = 0; n < 5; n++ )
         cout << setw( 16 ) << random( 1001 ) / 1000.0;
      cout << endl;
   }

   cout << "\nPress any key...";
   ch = getch();
// unbuffered output on stderr
   cerr << endl << "\nYou entered key: " << ch;

   cout << "\n\nPress any key to quit ";
   getch();
   cend();
}
```

Step 23: VBX controls

Aim: Learning to use pre-defined VBX controls.

Procedure: Write a program that contains two pre-defined switch elements of the size of 17 x 24 pixels in a dialog of the size of 34 x 24 pixels. As long as one of them is switched on, the turtle endlessly moves in a circle. When the second one is switched on, the program is terminated.

Remarks: Microsoft first standardized the use of controls in connection with the programming language Visual Basic. In order for VBX elements to be used, the corresponding library files (with the additions .VBX or .DLL) must be available. The distribution of Borland's C++ contains the file SWITCH.VBX that let you include dual position switches very easily. (Many software companies sell VBX libraries.)

In the Resource Workshop SWITCH.VBX should be added under the menu option "Install file ¦ library". The switch element is then shown among the tool icons and can be inserted into a dialog like any other control.

In the program, VBX elements are treated similarly to other controls. First an instance of CPVbxControl is created and the file name (with path) of the library file is given. When an object is instantiated, the library is automatically loaded (if this has not yet been done), and when the last object that needs the library is destroyed, the library is automatically unloaded.

Usually, VBX elements possess properties documented by the producer of the library. They are often referred to with a name, i.e. a string. The state of a property is checked with the member function *getPropbyName*.

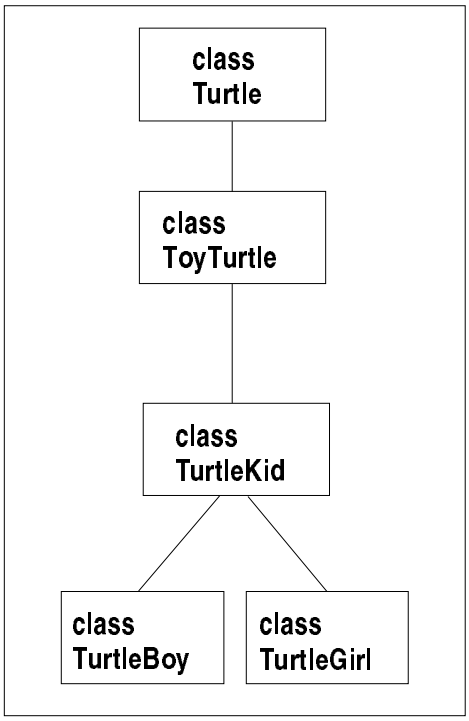
In order to have access to the VBX support, it is necessary to include the header file <cpvbx.h>. You must also check the VBX checkbox in the Target Expert (press the right mouse button while the cursor is on the EXE node in the project tree.)

```
// LEARN23.CPP

#define OBJECT_TURTLE
#include <champ.h>
#include <cpvbx.h>
#include "learn23.rh"

enum {stopped, running, aborting} state = stopped;

void gmain ()
{
   long on = 0;;

   CPModelessDialog myDialog( "LEARN23_DIALOG" );
   CPVbxControl switch1( "switch.vbx", myDialog,
                         IDC_BISWITCH1 );
   CPVbxControl switch2( "switch.vbx", myDialog,
                         IDC_BISWITCH2 );
```

```
   CPWindow wnd( "Learn23" );
   Turtle john;
   john.setPos( wnd, 0, 0 );
   myDialog.showModeless( wnd, CP::lowerLeft );

   do
   {
      switch1.getPropByName( "On", &on );
      if ( on )
         state = running;
      else
         state = stopped;

      switch2.getPropByName( "On", &on );
      if ( on )
         state = aborting;

      switch ( state )
      {
         case stopped:
            CP::yield();
            break;

         case running:
            john.forward( 10 ).left( 10 );
            break;
       }
   } while ( state != aborting );
}
```

Step 24: Class declaration

In the following, a class hierarchy will be constructed as follows:



Aim: Learning how to construct a class through derivation.

Procedure: Using the class *Turtle*, define a class *ToyTurtle*, the objects of which have the ability to draw a square with the member function *shape* in addition to the abilities of the Turtles.

Remarks: In the class definition, the base class is provided with the addition *public*, following a colon. This grants objects of the derived class *ToyTurtle* exactly the same access to the base class as objects of the base class itself.

Objects of the derived class are at the same time objects of the base class, so "ToyTurtles are Turtles" ("is-a" relation). Thus, ToyTurtles "inherit" all the Turtles' abilities and properties.

The member functions of the class are only declared and not defined in the class definition (the class contains a function prototype). The function definition is outside the class definition and must therefore be given a scope. This is done with the double colon (scope operator).

```
// LEARN24.CPP

#define OBJECT_TURTLE
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}


void gmain ()
{
   ginit( "Learn24" );
   Turtle::speed( 100 );

   // John is a Turtle only
   Turtle john;
   john.turtleColor( GREEN );
   john.forward( 100 );
   john.right( 90 );
   john.forward( 100 );

   // But Laura is a Turtle and a ToyTurtle
   ToyTurtle laura;
   laura.turtleColor( YELLOW );
```

```
   laura.left( 45 );
   laura.forward( 100 );
   laura.shape();       // Laura can draw a shape!

   getch();
   gend();
}
```

Step 25: Private data elements

Aim: Understanding how important private data elements are.

Procedure: Define a class *TurtleKid* derived from *ToyTurtle*, the instances of which can learn a name and then write it out.

Remarks: The names are "protected" towards the outside world with the access specification "private", i.e. only member functions of their class have reading and writing access to these names.

For data encapsulation it is highly important that, as well as possible, all data elements belong to the private section of the class. This way, the possibility of an illicit access from outside the class, which would usually lead to serious program mistakes, is excluded.

The graphical overlapping of the turtles is automatically handled. However, if other graphic elements (also writing) are to be written over the turtles, then the turtles must be taken away beforehand with *lift* and then be set down again with *drop*.

If a function is given a string to which it has only reading access, the string should be marked with "const". (Otherwise no strings of the type "const char" can be passed on.)

```
// LEARN25.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
```

```
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();  // Hide turtle to draw text on background
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();  // lift-drop pair
}


void gmain ()
{
   ginit( "Learn25" );
   Turtle::speed( 100 );

   TurtleKid laura;         // Laura is a TurtleKid now
   laura.turtleColor( YELLOW );
   laura.learnName( "Laura" );   // She learns her name
   laura.forward( 100 );
   laura.left( 45 );
   laura.sayName();        // She knows how to write it
   laura.shape();        // She still knows how to draw

   getch();
   gend();
}
```

Step 26: Overriding

Aim: Getting to know the technique of overriding member functions.

Procedure: Define two classes *TurtleGirl* and *TurtleBoy* derived from *TurtleKid*. Instances of *TurtleGirl* should draw circles when *shape* is called, those of *TurtleBoy* triangles.

Remarks: In the derived classes, the member function *shape* is redefined. Even though the derived class inherits the function *shape* from the base class *ToyTurtle*, this function is hidden as soon as a different function with the same name and the same parameter list is defined in the derived class.

```
// LEARN26.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ------------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
```

```
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      void shape ();
};

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}


void gmain ()
{
   TurtleKid * pKid;
   TurtleGirl * pGirl;
   TurtleBoy * pBoy;

   ginit( "Learn26" );
   Turtle::speed( 100 );

   pGirl = new TurtleGirl;
   pGirl->turtleColor( YELLOW );
   pGirl->learnName( "Kathy" );
   pGirl->left( 45 );
   pGirl->forward( 100 );
   pGirl->shape();
   pGirl->sayName();

   pBoy = new TurtleBoy;
   pBoy->turtleColor( RED );
   pBoy->learnName( "Jimmy" );
   pBoy->right( 45 );
   pBoy->forward( 100 );
   pBoy->shape();
   pBoy->sayName();

   pKid = new TurtleKid;
   pKid->turtleColor( GREEN );
   pKid->learnName( "John" );
   pKid->back( 100 );
   pKid->left( 45 );
   pKid->shape();
   pKid->sayName();

   CP::msgBox("Learn26", MB_OK, "Delete turtles");
   delete pGirl;
   delete pBoy;
```

```
   delete pKid;

   getch();
   gend();
}
```

Step 27: Overloading and default parameters

Aim: Getting to know the technique of overloading functions.

Procedure: Define a second function *shape* in the class *TurtleBoy* that draws a filled-in shape. The filling color is to be chosen with a parameter value.

Remarks: Functions even within the same scope can have the same name, under the condition that their parameter lists are different. It is the compiler's task to choose the right function, based on the parameter list.

Supplements: It is possible to give a parameter a default value. This value is assigned to the parameter whenever the parameter is not used in the function call. Default parameters must be listed in the parameter list in an uninterrupted sequence from right to left.

Often overloading can be avoided by using default parameters, which usually results in a better style of programming. Change the preceding program, so that overloading is not necessary.

```
// LEARN27.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();
```

```
   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl -----------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      void shape ();
      void shape ( int fillcolor );     // This is new
};

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void TurtleBoy::shape ( int fillcolor )
{
   fill( fillcolor );
   repeat ( 3 )
   {
      forward(50);
```

```
      left(120);
   }
   fillOff();
}

void gmain ()
{
   TurtleBoy * pBoy;
   Turtle::speed( 100 );

   ginit( "Learn27" );

   pBoy = new TurtleBoy;
   pBoy->turtleColor( RED );
   pBoy->learnName( "Jimmy" );
   pBoy->forward( 60 );
   pBoy->shape();
   pBoy->forward( 60 );
   pBoy->shape( YELLOW );
   pBoy->sayName();
   delete pBoy;

   getch();
   gend();


}// LEARN27A.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
```

```
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ----------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      void shape ( int fillcolor = LIGHTCYAN );     //
This is new
};

void TurtleBoy::shape ( int fillcolor )
{
   fill( fillcolor );
   repeat ( 3 )
   {
      forward(50);
      left(120);
   }
   fillOff();
}

void gmain ()
{
   TurtleBoy * pBoy;
   Turtle::speed( 100 );

   ginit( "Learn27a" );

   pBoy = new TurtleBoy;
   pBoy->turtleColor( RED );
   pBoy->learnName( "Jimmy" );
   pBoy->forward( 60 );
```

```
   pBoy->shape();
   pBoy->forward( 60 );
   pBoy->shape( YELLOW );
   pBoy->sayName();
   delete pBoy;

   getch();
   gend();
}
```

Step 28: Constructors

Aim: Getting to know constructors.

Procedure: Change the class definition of *TurtleBoy*, so that the color of the turtle can be set in the instantiation.

Remarks: Often, in the instantiation of an object, certain operations must automatically be executed, e.g. initializing data elements. For this purpose there are special member functions called constructors that carry the same name as their class. Like any other function, constructors can be overloaded.

Constructors cannot return values. For this reason, no return value, not even void, can be defined.

Every class possesses a so-called default constructor that the compiler automatically creates. It does not have any parameters and merely reserves storage room for the object. However, if a constructor is defined in the class declaration, the default constructor created by the compiler is not used any longer and therefore has to be defined explicitly if needed.

Supplements: In a class hierarchy, constructors are automatically called in order from the base class to the derived class. If they need to be given parameter values, the constructor of the base class must be listed in the definition of the one of the derived class, separated by a colon.

Usually, it is dangerous to call constructors explicitly; in particular it is often wrong to explicitly call a constructor of the base class within the body of a constructor of a derived class.

The class Turtle already has a constructor that determines the color of the turtle. Introduce such constructors in the whole class hierarchy.

```
// LEARN28.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
```

```
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ------------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      TurtleBoy ( int color ); // Ctor declaration
```

```
      void shape ();
};

TurtleBoy::TurtleBoy ( int color ) // Ctor definition
{
   turtleColor( color );
}

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void gmain ()
{
   ginit( "Learn28" );
   Turtle::speed( 100 );

   TurtleBoy aBoy( LIGHTRED );
   aBoy.forward( 60 );
   aBoy.shape();

   getch();
   gend();


}// LEARN28A.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      ToyTurtle ( int color );
      void shape ();
};

ToyTurtle::ToyTurtle ( int color ) : Turtle(color)
{}

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
```

```
{
   public:
      TurtleKid ( int color );
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

TurtleKid::TurtleKid ( int color ) : ToyTurtle( color )
{}

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl -----------------------
class TurtleGirl : public TurtleKid
{
   public:
      TurtleGirl ( int color );
      void shape ();
};

TurtleGirl::TurtleGirl ( int color ) : TurtleKid( color
)
{}

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      TurtleBoy ( int color );
      void shape ();
};

TurtleBoy::TurtleBoy ( int color ) : TurtleKid( color )
{}
```

```
void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void gmain ()
{
   ginit( "Learn28a" );
   Turtle::speed( 100 );

   TurtleBoy aBoy( LIGHTRED );
   aBoy.forward( 60 );
   aBoy.shape();

   getch();
   gend();
}
```

Step 29: Destructors

Aim: Getting to know destructors.

Procedure: Change the class definition of TurtleBoy so that the instances do a farewell pirouette.

Remarks: Destructors are used when the disappearing of an object (caused by a scope exit or a delete operation) makes it necessary to have some cleaning-up automatically done. Destructors carry the same name as their class, preceded by a tilde.

Destructors cannot return any values. For this reason no return value, not even void, can be defined. Destructors may not possess parameters.

```
// LEARN29.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
```

```
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid -----------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl -----------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy -----------------------
class TurtleBoy : public TurtleKid
{
   public:
      TurtleBoy ( int color );
      ~TurtleBoy ();        // Destructor declaration
      void shape ();
};

TurtleBoy::TurtleBoy ( int color )
{
```

```
   turtleColor( color );
}

TurtleBoy::~TurtleBoy ()    // Destructor definition
{
   home();
   repeat ( 8 )
      right(45);
}

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void gmain ()
{
   ginit( "Learn29" );
   Turtle::speed( 40 );

   {
      TurtleBoy aBoy( LIGHTRED );

      aBoy.forward( 60 );
      aBoy.shape();
   } // Destructor is called automatically here

   getch();
   gend();
}
```

Step 30: Virtual functions

Aim: Getting to know virtual functions.

Procedure: Define a function *flower* that gets a reference to *TurtleGirl* or *TurtleBoy* and draws a flower. Depending on which of the two references is given during execution time, the flower is drawn with the *shape* of the one object or the other.

Remarks: At compilation time the compiler cannot decide which member function *shape* to call. Because this can only be determined at execution time, the technique is called *late* or *dynamic binding*.

Investigate and explain the behavior when "virtual" is omitted.

Supplement: Instead of passing a reference to the function *flower*, use a pointer.

```
// LEARN30.CPP
```

```
#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      virtual void shape ();       // Now virtual!
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ------------------------
class TurtleGirl : public TurtleKid
{
   public:
      virtual void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
```

```
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      virtual void shape ();
};

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void flower ( ToyTurtle & aToy );

void gmain ()
{
   ginit( "Learn30" );
   Turtle::speed( 100 );

   char sex;
   CPInputChar( "Select sex", "Boy or Girl (b,g)?",
                sex, 'b' ).showModal();

   if ( sex == 'b' )
   {
      TurtleBoy john;
      flower( john );     // John is also a ToyTurtle
   }
   else
   {
      TurtleGirl laura;
      flower( laura );    // Laura is also a ToyTurtle
   }

   getch();
   gend();
}

void flower ( ToyTurtle & aToy )
{
   aToy.forward( 100 );
   aToy.right( 40 );

   repeat ( 3 )
   {
      // Draw shape of the actual object
      // shape() is virtual
      aToy.shape();
      aToy.left( 120 );
   }
}
```

```
// LEARN30A.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      virtual void shape ();       // Now virtual!
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ------------------------
class TurtleGirl : public TurtleKid
{
   public:
      virtual void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
```

```
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      virtual void shape ();
};

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void flower ( ToyTurtle* pToy );

void gmain ()
{
   ginit( "Learn30a" );
   Turtle::speed( 100 );

   ToyTurtle * pToy;
   char sex;

   CPInputChar( "Select sex", "Boy or Girl (b,g)?",
                sex, 'b' ).showModal();

   if ( sex == 'b' )
      pToy = new TurtleBoy;
   else
      pToy = new TurtleGirl;

   flower( pToy );

   getch();
   delete pToy;
   gend();
}

void flower ( ToyTurtle* pToy )
{
   pToy->forward( 100 );
   pToy->right( 40 );

   repeat ( 3 )
   {
      pToy->shape();
      pToy->left( 120 );
   }
}
```

Step 31: Static class members

Aim: Getting to know to use static data members and static member functions.

Procedure: Starting with program TURTLE29.CPP add a constructor in the class *ToyTurtle* that clears the screen and paints it to green color.when the first ToyTurtle instance is created.

Remarks: Because we clear the screeen we must do this only when the first object is instantiated. That's why we use a boolean flag *haveToPrepare* which cannot be a data member but must be unique for all instances of the class. Thus we declare it **static** and must reserve storage for it by defining it outside the class definition.

Supplements: The clearing operation should take place before the turtle is shown. Do do this, just add a static data member of the same class to the private section and the constuctor is called before *gmain* (or *main*) is started! Because the clearing operation is a graphics function the graphics system must be initialized at this point now.

It is a good programming style to isolate the initializing operation in a **static** member function *prepare* that is called by the constructor. In contrast to normal member functions, static member functions belong to the class and not to each individual instance. (To save storage space the code of a normal member function is not stored for each instance of the class, but the function gets the hidden parameter *this*, that identifies the instance calling it.)

```
// LEARN31.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      ToyTurtle ();   // New (default) constructor
      void shape ();

   private:
      static bool haveToPrepare;
};

// Static data member
bool ToyTurtle::haveToPrepare = true;
// Reserve storage space and initialize

ToyTurtle::ToyTurtle ()
{
   if ( haveToPrepare )
   {
      gclear( LIGHTGREEN );
```

```
      haveToPrepare = false;
   }
}

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ------------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      TurtleBoy ( int color );
```

```
      ~TurtleBoy ();
      void shape ();
};

TurtleBoy::TurtleBoy ( int color )
{
   turtleColor( color );
}

TurtleBoy::~TurtleBoy ()
{
   home();
   repeat ( 8 )
      right(45);
}

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void gmain ()
{
   ginit( "Learn31" );
   Turtle::speed( 100 );

   TurtleBoy aBoy( LIGHTRED );
   aBoy.forward( 60 );
   aBoy.shape();

   TurtleGirl aGirl;
   aGirl.left( 90 );
   aGirl.forward( 60 );
   aGirl.shape();

   getch();
   gend();
}


// LEARN31A.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      ToyTurtle ();
      void shape ();

   private:
```

```
      static bool haveToPrepare;
      static ToyTurtle aToyTurtle;
};

// Static data members
bool ToyTurtle::haveToPrepare = true;
ToyTurtle ToyTurtle::aToyTurtle; // Ctor is called here

ToyTurtle::ToyTurtle ()
{
   if ( haveToPrepare )
   {
      ginit( "Learn31a" );
      gclear( LIGHTGREEN );
      speed( 100 );
      haveToPrepare = false;
   }
}

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ------------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
```

```
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy ------------------------
class TurtleBoy : public TurtleKid
{
   public:
      TurtleBoy ( int color );
      ~TurtleBoy ();
      void shape ();
};

TurtleBoy::TurtleBoy ( int color )
{
   turtleColor( color );
}

TurtleBoy::~TurtleBoy ()
{
   home();
   repeat ( 8 )
      right(45);
}

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}

void gmain ()
{
   TurtleBoy aBoy( LIGHTRED );
   aBoy.forward( 60 );
   aBoy.shape();

   TurtleGirl aGirl;
   aGirl.left( 90 );
   aGirl.forward( 60 );
   aGirl.shape();

   getch();
   gend();
}


// LEARN31B.CPP
```

```
#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      ToyTurtle ();
      void shape ();

   private:
      static void prepare ();
      static bool haveToPrepare;
      static ToyTurtle aToyTurtle;
};

// Static data members
bool ToyTurtle::haveToPrepare = true;
ToyTurtle ToyTurtle::aToyTurtle; // Ctor will be called

// Static function
void ToyTurtle::prepare ()
{
      ginit( "Learn31b" );
      gclear( LIGHTGREEN );
      speed( 100 );
}

ToyTurtle::ToyTurtle ()
{
   if ( haveToPrepare )
   {
      prepare();
      haveToPrepare = false;
   }
}

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char name[] );
      void sayName ();

   private:
      char myName[20];
};

void TurtleKid::learnName( const char name[] )
{
```

```
  strcpy( myName, name );
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

// ----------- Class TurtleGirl ----------------------
class TurtleGirl : public TurtleKid
{
   public:
      void shape ();
};

void TurtleGirl::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 180 )
   {
      forward( 1 );
      left( 2 );
   }
   fillOff();
}

// ----------- Class TurtleBoy -----------------------
class TurtleBoy : public TurtleKid
{
   public:
      TurtleBoy ( int color );
      ~TurtleBoy ();
      void shape ();
};

TurtleBoy::TurtleBoy ( int color )
{
   turtleColor( color );
}

TurtleBoy::~TurtleBoy ()
{
   home();
   repeat ( 8 )
      right(45);
}

void TurtleBoy::shape ()
{
   fill( LIGHTCYAN );
   repeat ( 3 )
   {
      forward( 50 );
      left( 120 );
   }
   fillOff();
}
```

```
void gmain ()
{
   TurtleBoy aBoy( LIGHTRED );
   aBoy.forward( 60 );
   aBoy.shape();

   TurtleGirl aGirl;
   aGirl.left( 90 );
   aGirl.forward( 60 );
   aGirl.shape();

   getch();
   gend();
}
```

Step 32: "This" pointer

Aim: Getting to know the "this" pointer.

Procedure: Modify the class *ToyTurtle* so that *shape* returns a reference to a ToyTurtle. Thus, programs become more elegant because calls can be concatenated.

Remarks: Since "this" is a pointer to the current instance, it must be dereferenced with a * in order to obtain a reference.

In order to save space the member functions are coded once only for all instances of a class. When the member function is called, its "this" pointer, i.e. the pointer to the current object, is added as a hidden parameter. This way, the object's data members are accessible from within the member function.

```
// LEARN32.CPP

#define OBJECT_TURTLE
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      // Return reference to ToyTurtle
      ToyTurtle & shape ();
};

ToyTurtle & ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
   return *this;
}
```

```
void gmain ()
{
   ginit( "Learn32" );
   Turtle::speed( 100 );

   Turtle john;
   john.forward( 100 ).right( 90 ).forward( 100 );

   ToyTurtle laura;
   laura.shape().left( 45 ).forward( 100 );

   getch();
   gend();
}
```

Step 33: Overloading operators; friend functions

Aim: Getting to know the technique of overloading operators and realizing the importance of friend functions.

Procedure: Overload the operator << (extractor) so that the name of a ToyTurtle can be printed out with *cout*.

Remarks: It is best to understand the << operator as a usual function with the special name (operator<<) and the parameter list (left, right), left and right meaning the arguments to the left resp. to the right of the operator in the call.

In order for the operator function (in this case not a member function) to have access to the private data in the class ToyTurtle, it must be declared as a friend in that class.

The operator returns a reference to "ostream", so that several calls with *cout* can be concatenated.

It is also possible to declare a class or a function as a friend. It then has access to all the private data members and member functions. However, since this means breaking through the protection barrier, it should be done as rarely as possible.

Instead of regarding an operator as a global (friend) function, it can also be defined as a member function. Like every other instance, it can then obtain access to the current instance with the "this" pointer.

Supplements: Introduce a class NameBox, the members of which can print out a name in a Windows messagebox. Then overload the << operator so that the code

```
NameBox blackBoard;
blackBoard << kid;
```

prints out the name of the TurtleKid.

```
// LEARN33.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName ( const char * name );
      friend ostream & operator<< ( ostream & out,
                                    TurtleKid & aKid );

   private:
      char myName[20];
};

void TurtleKid::learnName ( const char * name )
{
  strcpy( myName, name );
}

ostream & operator<< ( ostream & out,
                       TurtleKid & aKid )
{
   out << aKid.myName;
   return out;
}


void gmain ()
{
   ginit( "Learn33" );
   cinit( "Learn33", 10, 40, CPPosition( 300, 200 ) );

   TurtleKid kid;
   kid.speed( 100 );
   kid.turtleColor( YELLOW );
   kid.learnName( "Laura" );
   kid.forward( 100 );
   kid.left( 45 );
   kid.shape();
```

```
   cout << "My name is " << kid << " !";
   // What a nice syntax!

   getch();
   cend();
   gend();
}


// LEARN33A.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class NameBox -------------------------
class NameBox
{
   public:
      void show ( const char * msg );
};

void NameBox::show ( const char * msg )
{
   CP::msgBox( "Welcome, my name is:", MB_OK, msg );
}


// ----------- Class ToyTurtle -----------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid -----------------------
class TurtleKid : public ToyTurtle
{
   public:
      void learnName( const char * name );
      friend void operator<< ( NameBox & aBox,
                               TurtleKid & aKid );

   private:
      char myName[20];
};

void TurtleKid::learnName ( const char * name )
{
  strcpy( myName, name );
}
```

```
void operator<< ( NameBox & aBox, TurtleKid & aKid )
{
   aBox.show( aKid.myName );
}

void gmain ()
{
   ginit( "Learn33a" );
   NameBox blackBoard;

   TurtleKid kid;
   kid.speed( 100 );
   kid.turtleColor( YELLOW );
   kid.learnName( "Laura" );
   kid.forward( 100 );
   kid.left( 45 );
   kid.shape();
   blackBoard << kid;   // Nice, isn't?
   gend();
}
```

Step 34: Copy constructor, assignment operator, addition operator

Aim: Getting to know when the copy constructor is called and how to redefine the assignment and addition operaters.

Procedure: Define a copy constructor in the class *TurtleKid* to enable initializations during instantiation. The name of the turtle is to be stored in dynamic storage space (on the heap).

Remarks: The compiler uses the copy constructor when instantiating an object and at the same time initializing it. The copy constructor is also needed when a call-by-value is used, and when a function returns an object, because a temporary object must be created (on the stack).

Supplements: Overload the + and the = operator to enable the "addition" and the "assignment" of instances of the class TurtleKid.

Remarks: For the observation of the processes, a text is written into the text window at important points in the program. In particular, it can be noted that, before the return of the function *operator+*, a new stack object is created on the stack, the initialization of which is executed by the copy constructor. Then, the "temp" object, defined locally within the function, is removed. Next, the stack object is copied into the element to the left of the equality sign by the assignment operator, and finally, the stack element is removed (of several superposed turtles, only the top one is visible).

```
// LEARN34.CPP

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>
```

```
// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}


// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      TurtleKid ();                       // Default ctor
      TurtleKid ( char * name );    // Initializer ctor
      TurtleKid ( TurtleKid & aKid );      // Copy ctor
      ~TurtleKid();                        // Destructor

      void sayName();

   private:
      char * myName;
};

TurtleKid::TurtleKid ()
{
   myName = new char[1];
   strcpy( myName, "" );
}

TurtleKid::TurtleKid ( char * name )
{
   myName = new char[strlen( name ) + 1];
   strcpy( myName, name );
}

TurtleKid::TurtleKid ( TurtleKid & aKid )
{
    setPos( aKid.xCor(), aKid.yCor() );
    setHeading( aKid.heading() );
    myName = new char[strlen( aKid.myName ) + 1];
    strcpy( myName, aKid.myName );
    cout << "Copy contructor executing...\n";
    getch();
}

TurtleKid::~TurtleKid ()
{
   delete [] myName;
   cout << "Destructor executing...\n";
   getch();
}
```

```
void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor()+10 );
   gtext( myName );
   drop();
}

void drawTree ( TurtleKid aKid );
// Try instead with
// void drawTree ( TurtleKid & aKid );

void gmain ()
{
   ginit( "Learn34" );
   cinit( "Learn34", 10, 40, CPPosition( 300, 200 ) );
   Turtle::speed( 100 );

   {
      TurtleKid kid1( "John" );
      kid1.forward( 100 );
      {
         TurtleKid kid2 = kid1;
         kid2.left( 90 );
         kid2.forward( 100 );
         kid2.sayName();
         cout << "Clone of kid2 will draw a tree..."
              << endl;
         getch();
         drawTree( kid2 );
         cout << "kid2 will fall out of scope..."
              << endl;
         getch();
      }
      cout << "kid1 will fall out of scope..."
           << endl;
      getch();
   }
   getch();
   cend();
   gend();
}

void drawTree ( TurtleKid aKid )
{
   aKid.right( 90 );
   aKid.forward( 50 );
   aKid.left( 45 );
   repeat ( 3 )
   {
      aKid.forward( 50 );
      aKid.back( 50 );
      aKid.right( 45 );
   }
   cout << "drawSquare will return..." << endl;
   getch();
}
```

```
// LEARN34A.C

#define OBJECT_TURTLE
#include <string.h>
#include <champ.h>

// ----------- Class ToyTurtle ------------------------
class ToyTurtle : public Turtle
{
   public:
      void shape ();
};

void ToyTurtle::shape ()
{
   repeat ( 4 )
   {
      forward( 50 );
      left( 90 );
   }
}

// ----------- Class TurtleKid ------------------------
class TurtleKid : public ToyTurtle
{
   public:
      TurtleKid ();
      TurtleKid ( char* name );
      TurtleKid ( TurtleKid & aKid );
      ~TurtleKid ();

      // Overload assignment
      TurtleKid & operator= ( TurtleKid & aKid );

      // Declare operator +
      friend TurtleKid operator+ ( TurtleKid & aKid1,
                                   TurtleKid & aKid2 );

      void sayName ();

   private:
      char * myName;
};

TurtleKid::TurtleKid ()
{
   myName = new char[1];
   strcpy( myName, "" );
}

TurtleKid::TurtleKid ( char * name )
{
   myName = new char[strlen( name ) + 1];
   strcpy( myName, name );
}

TurtleKid::TurtleKid ( TurtleKid & aKid )
{
    setPos( aKid.xCor(), aKid.yCor() );
    setHeading( aKid.heading() );
```

```
    myName = new char[strlen( aKid.myName ) + 1];
    strcpy( myName, aKid.myName );
    cout << "Copy contructor executing for "
         << myName << "..." << endl;
    getch();
}

TurtleKid::~TurtleKid()
{
   cout << "Destructor executing for "
        << myName << "..." << endl;
   getch();
   delete [] myName;
}

TurtleKid & TurtleKid::operator= ( TurtleKid & aKid )
{
   // Make sure you don't destroy target,
   // when using aKid = aKid
   if ( this == &aKid )
      return *this;

   delete [] myName;     // Destroy old name

   setPos( aKid.xCor(), aKid.yCor() );
   setHeading( aKid.heading() );
   myName = new char[strlen( aKid.myName ) + 1];
   strcpy( myName, aKid.myName );

   cout << "Assignment op executing for " << myName <<
"..." << endl;
   getch();

   return *this;
}

void TurtleKid::sayName ()
{
   lift();
   gpos( xCor()+10, yCor() );
   gtext( myName );
   drop();
}

TurtleKid operator+ ( TurtleKid & aKid1, TurtleKid &
aKid2 )
{
   cout << "Now creating temp object..." << endl;
   getch();
   TurtleKid temp( "Temp" );
   // Temperary turtle will appear
   temp.turtleColor( RED );
   // To observe it, we paint and move it away
   temp.sayName();
   temp.back( 50 );
   getch();
   temp.setPos( ( aKid1.xCor() + aKid2.xCor() )/2.0,
                ( aKid1.yCor() + aKid2.yCor() )/2.0 );
   temp.setHeading( aKid1.heading()+aKid2.heading() );
   temp.myName = new char[strlen( aKid1.myName )
```

```
                              + strlen( aKid2.myName ) + 1];
   strcpy( temp.myName, aKid1.myName );
   strcat( temp.myName, aKid2.myName );
   cout << "operator+() returning now..." << endl;
   getch();
   return temp;
   // Copy ctor executes when temp is copied to stack
}


void gmain ()
{
   ginit( "Learn34a" );
   cinit( "Learn34a", 20, 40, CPPosition( 300, 200 ) );
   Turtle::speed( 100 );

   {
      TurtleKid kid1( "John" );
      kid1.right( 45 );
      kid1.forward( 100 );
      kid1.sayName();

      TurtleKid kid2( "Laura" );
      kid2.left( 45 );
      kid2.forward( 100 );
      kid2.sayName();

      cout << "Now creating kid3..." << endl;
      getch();
      TurtleKid kid3;
      cout << "Now adding kid1 + kid2..." << endl;
      getch();
      kid3 = kid1 + kid2;

      kid3.forward( 100 );
      kid3.sayName();

      cout << "Turtles will fall out of scope..."
           << endl;
      getch();
   }
   cout << "All done.";
   getch();
   cend();
   gend();
}
```

Step 35: Recursions

Aim: Getting to know recursive procedures.

Procedure: Draw a binary tree.

Remarks: A tree of the order n is based on the combination of a branch and a tree of the order n-1. Make sure the turtle returns to the point of departure.

The recursion must be determined, i.e. for simple n, the function must

return without a recursive call. It is customary to use the unstructured *return* instruction, since this helps make the program clearer.

Recursive functions are usually called through parameters that describe a "recursion depth"; this is why recursive programming is closely related to functional programming.

Since the stack is required for every function call, deeply nested recursions can easily cause stack overflows that lead to fatal execution time errors. The size of the stack is defined in the .DEF file of the project.

```
// LEARN34.CPP

#define OBJECT_TURTLE
#include <champ.h>

void binTree ( int s );
Turtle su;

void gmain ()
{
  CPWindow wnd( "Learn35" );
  Turtle::speed( 50 );
  su.setPos( wnd, 0, -100 );
  binTree( 128 );
  getch();
}

void binTree ( int s )
{
   if ( s < 16 )
      return;
   su.forward( s );
   su.left( 45 );
   binTree( s / 2 );
   su.right( 90 );
   binTree( s / 2);
   su.left( 45 );
   su.back( s );
}
```

6 On the history of C++

The concept of classes and objects showed up for the first time in the programming language SIMULA-67 in 1967. In the early 1970ies, based on SIMULA, Xerox developed the programming language Smalltalk, today's most homogenous and complete object oriented system in the version of Smalltalk-80. Unfortunately, commercial success failed to come for two reasons: a version for PCs was not available until a short time ago; in addition, Smalltalk requires programmers used to classical programming languages to undergo a total change in the way of thinking.

In 1980, a project called "C with classes" was started in the Bell Laboratories. Bjarne Stroustrup succeeded in introducing elements of SIMULA into the programming language C [3]. In July 1983, the new programming language was for the first time presented outside the Bell Laboratories. Rick Mascatti named it C++, after the incremental operator in C.

C++ is a superset of C and, as a hybrid language, elegantly reunites the world of classical procedural programming with that of object oriented programming. Thus, it is possible to continue to use C programs and knowledge of C in C++. The language is believed to have considerable potential for the future, since many modern operating systems are written in C/C++.

7 Literature

[1] Böhm, Jacopini, Flow Diagrams, Turing Machines and Languages with only two Formation Rules, Communications of the ACM, May 1966

[2] Papert, Mindstorms - Children, Computers and Powerful Ideas, Basic Books (1980)

[3] Booch, Object orientied design, Benjamin 1991

[4] Stroustrup, The C++ programming language, Addison-Wesley 1987

[5] Champ Documentation, Salvisberg Software & Consulting, Bern 1995

Appendix 1: Features of Champ

Champ is well-suited for writing graphical and real-time applications, for simulations and for teaching programming. It does not require any knowledge of Windows API calls or message processing and can be used even by programmers that do not have previous experience with C++. System requirements: MS-Windows 3.1x, Borland BC++ 4.5 or TC++ 4.5, 486 DX with 8 MB RAM recommended.

Features:

- Multiple Windows, with redraw capability
- Graphics primitives including line drawing, styles, colors, filling, full font support, pixel and floating point user coordinates
- High-resolution printer output, simply by replacing ginit() by pinit()
- Turtle graphics with full LOGO command set, multiple turtles
- Fully-integrated complete documentation on-line, like Borland's run-time library
- Windows menus

- Windows modal and modeless dialogs, compatible with Rosource Workshop; ready-to-use input dialogs with validation for all standard data types
- VBX 1.0 custem control support
- Text window, supporting character-based I/O (cin/cout/getch/etc.)
- Low-level classes for easy access to the hardware, COMx, LPTx, and timers
- Support for bitmaps and sprite animation
- Simple container classes: list, queue, etc.
- Complete printed documentation
- Integrated Champ Project Manager for automatic creation of projects

Appendix 2: Syntax conventions

All program examples keep to the following conventions (proposition of Elemtel[] and others):

Names of variables begin with lowercase letters, names of types with uppercase letters. Every section of a composite name begins with an uppercase letter.

Pointers are introduced with the letter p, which is interpreted as the first letter of a name.

No type information is included in the name. (This goes against the habits of Windows programmers, but to a great extent corresponds to the habits of scientific programmers.)

The conventions for the organization of the screen (indents, spaces etc.) can be seen in the example programs.

Appendix 3: Basic terminology of object oriented programming

3.1 Encapsulation, inheritance and polymorphism

Central to object oriented programming are **objects** that can be created and destroyed. Objects usually possess certain **attributes** or **properties** (data members) and certain **abilities** or **behavior** (member functions). Thus, the data and the functions operating on this data form a unit. Figuratively spoken, the program, or actually intuitively the programmer, sends **messages** to the objects which perform an action (as an illustration: when the cat is sent the message "come", it jumps onto your lap) or change the attributes (after the message "eat ", the cat is not hungry any more). In C++, sending a message corresponds to calling an member function.

The programmer using the objects usually need not know what happens "inside" the object, i.e. he need not know the details of the member functions and the data members. In a certain way this information is hidden from the outside (**information hiding**), which at the same time grants **information protection**: internal data is protected against

unintentional changes from the outside. This way of hiding the information is called **data encapsulation**.

Contrary to this, in classical, not object oriented programming, sequential calls and structures such as "if...then...", "repeat..." are central. In OOP the object, which on the one hand contains data and on the other hand offers the functions that operate on those data, is central. These functions will generally perform actions, but the actions are never independent of the corresponding object: they are object oriented. According to Booch, object oriented programming can generally be seen as "a message exchange between objects and not as an application of functions to passive data" [3].

Objects are described through their belonging to a **class**. In other words, a class defines an object type in which objects share common structure and behavior. A class can also be called an **abstract data type**. While classical programming focuses on the sequence of commands, object oriented programming begins by defining useful classes that possess the requested abilities. The class definition contains both the internal data and the methods (functions or procedures) which handle the data. This corresponds to the human working technique of dividing complex problems into partial problems that can be solved as independently as possible and the details of which need not be known in order to handle the original complex problem, according to the principle of **"divide and conquer"**.

Crucial for object oriented programming is the possibility to introduce a **class hierarchy**. This way, the objects of a new class can take over certain properties from the hierarchically superior classes. This process is called **inheritance**. The new class is called a **derived class**, the superior a **base class**. (Illustration: the class of the cats can be derived from that of mammals, which can be derived from that of animals, which in turn can be derived from that of living creatures.) We distinguish between a **direct** base class, which immediately precedes a class, and an **indirect** base class, which is more than one step higher in the hierarchic structure.

In object oriented programming, inheritance is used exclusively as a **"kind-of"** or **"is-a"**, but never as a **"part-of"** or **"has-a"** relation. For example, a cat is a kind of mammal and that a kind of living creature. But it would be wrong to derive the class of cats from the classes of hearts, lungs, livers, heads etc., because they are parts of a cat. Instead, parts of an object should be implemented as data members of the corresponding class.

Objects of different classes can contain member functions of the same name (illustration: the message "come" can be sent both to cats and to parrots). However, the inner mechanism performed by that function in each of the objects can be totally different (the message "come" triggers the action "walk" in a cat, but the action "fly" in a parrot). All the same, different functions of the same name are unequivocally distinguishable, because the **scope** of a member function is the class in which it is defined.

In a class hierarchy, member functions can be redefined in derived classes. In that case, the old definition is **overridden**, i.e. instances of the derived class execute the redefined function. This way, objects inside a class hierarchy are given changed or additional abilities.

It is in the nature of class hierarchies that an object of a derived class can also be seen as an object of the base class (illustration: a cat is, at the same time, an animal). This is why

a function expecting an object of the base class for a parameter can also be given an object of a derived class (illustration: a function "come and eat" expecting an animal can also be given a cat or a parrot). However, the message "come" has a different definition for animals, cats or parrots (cats walk, parrots fly).
When we give the function "come and eat" a parrot as an animal, then "come and eat" unfortunately executes the "come" of the animals and not that of the parrots, which is usually not very welcome. To obtain the desired result, we declare the member function "come" as **virtual** (in this case the parameter has to be a reference or a pointer).

The combination of overriding/overloading and virtual functions is called **polymorphism**.

A programming language with the three properties **encapsulation, inheritance and polymorphism** is called object oriented.

3.2 Creation/destruction and visibility of objects

A software object takes up a certain amount of **space** in a computer's memory and exists during a certain amount of **time** inside the program's execution time. For certain program parts it is **visible**, for others it is invisible or **hidden**. Objects are described by their belonging to a class. Through the definition of the class, however, no objects are yet created.

The creation of an object, also called **instantiation**, can basically take place in two different ways: either by means of a **variable definition** or of a **creating operation**. A variable can not only be defined with entire program or file scope or in the head of a function, but anywhere inside a program block. The lifetime of the object is restricted to the program block, in which it was declared, i.e. it is automatically destroyed at the end of the block and the memory space is made available. In contrast to this, a creating operation can create an object during execution time. Such an object is called **dynamic**. It is referred to by means of a pointer. Dynamic objects must be destroyed explicitly.

When the object is created, certain **initializations** might be necessary. Because of this, when defining a class, we usually indicate a function that is automatically called when an object is created. Such a function is called a **constructor**. When an object is destroyed, a similar automatically called function, the **destructor**, can do the "cleaning up".

3.3 The parts of an object

An object consists of **data members** and of **member functions**. Basically, the data members can be read and changed by the member functions. However, visibility towards the outside should be carefully regulated, i.e. data should be encapsulated. This is why there are three different types of access rights: parts of the object, i.e. data members and member functions, can be declared **private**, **protected**, or **public**. Private parts are not accessible from without the object (except by objects or functions that are specifically declared as a **friend**). Protected parts are accessible from within derived classes. The public parts are generally accessible.

In accordance with our habits of perceiving objects around us, software objects possess the following five important properties: A **state**, a **behavior**, an **identity**, a **lifetime**, a **visibility**, and a **type**.

state The state is defined by the values of the **data elements** which are determined through initializing or assignment. During an object's lifetime, the value of its data elements can change.

behavior The behavior of an object is determined through its **member functions**. They have reading and writing access to the data members (withing the access rights), but they can also return function values or change outside data through reference parameters or pointers.

identity An object has an identity that makes it unequivocally distinguishable from all other objects. Every object occupies a certain segment of the memory; its name can also be understood as the name of that segment.

An object can also be referred to by means of a variable, the value of which is the internal reference (memory address) of the object. Such a variable is called a **pointer** to the object. This way, the object is referred to **indirectly** through the pointer: this process is called **dereferencing**.

A **reference variable** can be considered as a pointer which is automatically dereferenced.

Dynamic objects must be referred to exclusively in this way; this is why pointers and references are very important in object oriented programming.

lifetime Usually, objects exist only during part of the program's execution time, perhaps only within a block of program or a function. Part of a good programming style is to minimize objects' lifetime.

visibility Generally, objects can be used only within certain parts of the program, whereas in others they are "invisible" or "hidden". Visibility should be carefully regulated in the interest of information protection, which takes place on different **scope** levels (program, file, function, class).

type The object also unequivocally belongs to a class. Due to that, C++ is called a **typed language**.