Certificate Course

Object-Oriented Programming in C++

Dr. Fritz Solms
Solms Training, Consulting and Development
http://www.solms.co.za
113 Barry Hertzog Ave, Emmarentia, Johannesburg, South Africa
Tel: +27 11 646 6459, EMail: fritz@solms.co.za

May 26, 2009

Contents

Chapter 1

C^{++}-basics

1.1 Structure of a simple program

Similar to C, any C++ program has at least one function, the function `main`. This function is generally referred to as the main program. The body of any function is enclosed within curly brackets ({...}) each statement is terminated by a semicolon. The curly brackets perform the function of the `begin ... end` block in other languages like, for example, Pascal.

Lets have a look at the C++ version of the traditional *"Hello World"* program:

```
using namespace std;

#include <iostream>

int main()
{
  cout << "Hi there, mate!" << endl;

  return 0;
}
```

The first line includes the header file input/output stream class library. Later we shall see that this is a hierarchy of classes for accessing input and output devices. Header files are used to inform the user of the constants, variables, functions and classes defined in the accompanying CPP (implementation) file. You can include standard libraries supplied with your compiler, third party libraries acquired from some vendor (e.g. a database library) or libraries you wrote yourself.

The latest ANSI C+ standard packagess the standard library elements in the standard namespace. Name spaces will be discussed in detail in chapter 6. For now it is sufficient to state that name spaces are C++'s support for packaging.

Next we define the `main` function of the program. Every C++ program must have at least one function (and hence one non-object oriented element). Its name must be `main`. The `void` specifies that the function `main` has no return value. Functions which

do not return a value are often called procedures. The empty bracket after `main` specify that `main` takes no arguments. Later we shall show how we can give arguments to `main` in order to handle command line parameters.

Each function has a body. The body is enclosed within curly brackets. Our function has only one statement. Statements in C++ are terminated by a semicolon. In this statement we simply send the string `"Hello World"` to the standard output device `cout` which usually is the terminal. Data for output are passed to `cout` with the output operator `<<`. To be able to fully understand stream input/output one has to understand classes and operator overloading. This will be covered in detail in the following chapters. At this stage we shall simply use output streams as illustrated in the *"Hello World"* application.

The `endl` stream manipulator starts a new line and flushes the stream. It is very common for C++ programmers to use the end-of-line character '/n' instead, but this does not flush the stream and if the program aborts, some of the stream output might be lost.

1.1.1 Compiling and running the program

The process for compiling and running the program depends on both, the compiler used and thhe target platform. We shall use the GNU C++ compiler. In that case we can compile thhe application via

```
g++ -Wall -ansi HelloWorld.cpp -o HelloWorld
```

This will generate an executable file which is called `HelloWorld` on Unix-type operating systems (e.g. *Linux*) and `HelloWorld.exe` on *Windows*.

On *Unix/Linux* we run the application from thhe directory in which the executable is stored via

```
./HelloWorld<ENTER>
```

On *Windows* you would use

```
HelloWorld<ENTER>
```

instead.

1.1.2 Adding comments to your code

Informative, unambiguous commenting and consistent indentation can make the task of understanding code (even your own code when you come back to it after a while) much more understandable. You must, however, take care to maintain your comments together with your code. Otherwise you can easily end up with comments which are ambiguous or even simply wrong. Such comments are worse than no comments at all.

There are two ways in which you can add comments to your C++ code. Both are illustrated in the commented version of the *"Hello World"* application given below:

```
using namespace std;

#include <iostream>

int main()
{
  cout << "Hi there, mate!" << endl;

  return 0;
}
```

You can use the `/* ... */` comment to insert a comment which possibly spans over several lines. For example, at the top of the file we give the name of the file in which the program is stored and then a description of what the program does.

Alternativerly you use a double-slash (`//`) to specify that the remainder of that line should be treated by as a comment. For example, after `#include <iostream.h>` precompiler directive we explain why we need the library.

Do not comment statements which are obvious from the code anyway. For example adding a comment after the `include` statement that we are including the `iostream` library gives us no further information and just increases the amount of text the reader has to read and the amount of work involved when changing the code (since we have to update the comments too).

1.2 Declarations

There are 5 basic data types and 5 qualifiers in C^{++}. These are shown in figure 1.1.

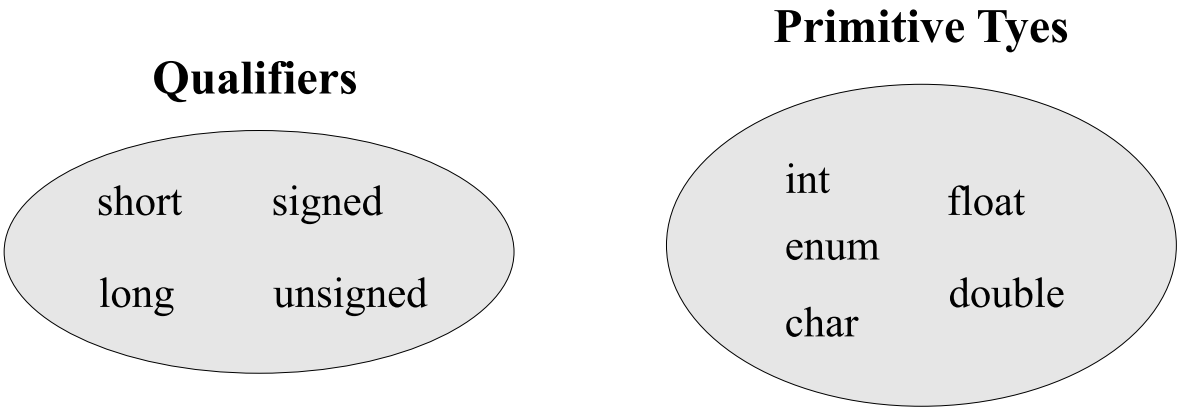

Figure 1.1: The qualifiers can be combined with the primitive types to specify new primitive types.

The data type `char` is a single byte, big enough to hold a single character. The qualifiers and the basic types can be combined. For example, one can have

```
unsigned int i = 120, j;

int k;

long double  x, y, z = 21.2345677;

enum vehicle {bicycle, car, bus, truck, unknown=99} verhicle1, verhicle2;
```

Here `i` and `j` are unsigned integer variables and `i` is initialized to 120. `k` is an integer variable (per default `unsigned short`), and `x, y` and `z` are long double precision variables (on the IBM-PC usually 80 bits long ranging between 3.4×10^{-4932} to $1.1\times^{+4932}$ with an accuracy of about 21 decimal digits). The size of any data type can be obtained with the C^{++}-function `sizeof(datatype)`.

`verhicle1` and `verhicle2` are enumeration variables which can only take one of the values `bicycle`, `car`, `bus` and `truck` or unknown. Enumeration types are really named integer constants. In the above example the compiler assigns `bicycle=0`, `car=1`, `bus=2`, `truck=3` and `unknown=99`. Note that `vehicle` is a new data type and variables of that type can be declared like any other variable type:

```
vehicle verhicle3;
```

Variables of any data type can be declared constant:

```
const long double  pi = 3.1415926535897932;
```

If a certain variable should always remain constant it should be declared so. This enables the compiler to report any violation of the constantness and thus enhances program integrity.

1.2.1 Type-casting

Implicit type-casting

The fundamental data types can be mixed freely in assignments and expressions. In arithmetic expressions the types are converted as to loose as little information as possible. If either of the operands has higher resolution, then the other operand is automatically converted to the higher resolution. For example adding a `float` and a `long double` causes the `float` to be converted to a `long double` and the result is a `long double`. Similarly, integral data types are promoted to the type with the largest range and if mixing integral and floating point data types then the integral data type is promoted to a floating point. Hence, adding an `int` to a `double` causes the `int` to be converted to a `double` and the result is a `double`.

We can also assign any fundamental data type to any other fundamental data type. If we assign a `float` to a `double` the extra resolution is simply padded by zeros. The other way around we will, of course, necessarily loose resolution (the least significant digits will be truncated).

Similarly, assigning an integral data type to a floating point data type is no problem. The decimal digits are simply set to zero. The other way around results in the truncation of the decimal digits.

Explicit type casting

In some cases it is necessary to type-cast explicitly. For example

```
double x = 2/3;
```

sets `x` to zero since division has presedence over assignment and the two data types participating in the division are both integral data types – hence integer division is used, yielding zero. In this case one would explicitely type-cast one of the integers to a `double`:

```
double x = (double)x/3;
```

C++ supports two notations for type-casting. The traditional *C*-cast shown above and the functional notation

```
double x = double(x)/3;
```

The *C* type-casting notation has the advantage that it also works with non-simple type names:

```
long double x = long double(x)/3;   // compiler generates error

long double x = (long double)x/3;   // no problem
```

Similarly, pointers (see later in this chapter) have to type-cast with the *C*-cast.

In later chapters we shall show how you can support implicit and explicit type-casting for your own data types (classes).

Identifier names

The name of an identifier (e.g. a variable, a constant or a function) consists of a sequence of letters, digits and underscore-characters. The first character may not be a digit. C++ does not impose a limit on the name-length. Some compiler may however impose such a limit. A C++ keyword may of course not be used for an identifier name.

Examples of legal identifier names are

```
street_name    x    X    _X    x_    OS_2
```

On the other hand, the following names are not legal identifier names:

```
street-name    1x    OS/2    $system    first.name
```

Reference types

Reference types allow us to access the same object by different names – i.e. a reference is an alias for an object of a specified type. Consider the following code snippet:

```
int  i = 3;
int& ir = i;   // Reference to the variable i
ir = 5;
cout << i;     // Prints 5 on the screen
```

After assigning the reference variable `ir` to `i`, bot `i` and `ir` use the same memory location. Changing the one will change the other.

We can make the reference as such a constant via `const int& ir = i;`. This will fix `ir` such that it will always use the same memory area as the object `i`. Alternatively, we can have a reference to a constant object:

```
const double pi = 3.1415926;
double& ref = pi;   // Reference to the variable i
```

Now, as long as `ref` is a reference to `pi`, neither `pi` nor `ref` may be changed. We can of course reassign the reference variable to be an alias for another `double` object via, for example, `ref = x;`. Finally we can have a constant reference point to a constant object as in the following example

```
const double pi = 3.1415926;
const double& pi2 = pi;   // Reference to the variable i
```

We shall see that reference types are closely related to pointer types. Furthermore, reference variables are very important for function arguments (see section 1.5.1).

1.2.2 The scope and lifespan of variables

In C++ blocks are delimited by curly brackets. For example the function body is a block delimited by curly brackets. Within the function body we can declare further blocks (nested blocks). The scope of a variable is bounded by the block in which it is defined. Consider, for example, the following program listing:

```
#include <iostream>

using namespace std;

const float version = 2.11;

double y;

double f(const double x)
{
  double y = x*x;
  return y;
}

int main()
{
  double a;
  cout << "a = ";  cin >> a;
  y = f(a);

  if (y > 5)
  {
    double y = a*a - a;
```

```
    cout << "y = " << y << endl;
  }
  cout << "y = " << y << endl;

  char k; cin >> k;

  return 0;
}
```

Here `version` is a global floating point constant which has the scope of the file, i.e. it can be used anywhere in the file. Similarly, we define a global variable `y`. In the function `f` we declare a local variable `y` which hides the global `y`. The scope of this variable is within curly brackets – hence within the body of the function.

In the main program we use `cout` to write to the terminal and `cin` to read in the value of `a` from the keyboard. The assignment statement `y = f(a)` assigns the global `y` to the result of the function call.

We then check whether `y` is greater then `5` and if this is the case we perform a collection of statements enclosed in a block (within curly brackets). The `if` statement will be discussed in detail later in this chapter. Within the block we declare a local variable, `y`, which hides the global `y`. The scope of this variable is limited to the `if`-block.

Generally it is a good idea to define variables as close to the point where they are used as is possible. This makes maintenance usually a lot easier. The use of global variables should be minimized. In fact, it is a good idea to define your variables such that they have the minimum required scope. It is even possible to restrict the scope of a variable to a subset of sequential statements by enclosing this subset of statements simply within curly brackets.

Lifespan of static variables

Except for the case where you declare a variable as `static`, it's life-time is the same as its scope. For example, in the previous listing, the variable `y`, whose scope is the `if`-block, is created and initialized with the statement `double y = a*a-a;` and is destroyed at the when the execution thread leaves the `if`-block.

A `static` variable has the life-time of the program, i.e. it is created when the program is started and destroyed only when the program terminates. They are initialized the first time the execution thread passes through the declaration.

For example, say you want to monitor the number of times a certain function `f` has been called. You can define a static local variable `ncalls` as follows:

```
double f(const double x)
{
  static int ncalls = 0;

  ++ncalls;
  cout << "f has been called " << ncalls << " times." << endl;
```

```
  return x*x;
}
```

Here `ncalls` is initialized to zero the first time the function is called. Then `ncalls` is incremented and its value is printed onto the terminal. The next time `f` is called, the function remembers the value of `ncalls` from its previous run, increments it and prints out the correct no of function calls.

1.3 C^{++} operators

Table 1.1 shows a summary of C^{++} operators in order of decreasing precedence. Hence `u = x + y % z;` evaluates first the remainder of $\frac{y}{z}$, then adds this to `x` and then assigns the result to `u` (note that `10%4` returns `2`).

The use of these operators will be illustrated with example programs throughout this text.

All operators can be overloaded (see discussion on operator overloading later in the text) except for

```
.    .*    ::    ?:
```

When overloading operators, one should bare in mind that the order of precedence and the syntax remains the same as that for the built-in data types.

Operator	operator name	example
`::`	scope resolution	*class_name*`::`*member*
`::`	global	`::`*name*
`.`	member selection	*object*`.`*name*
`->`	member selection	*pointer*`->`*member*
`[]`	subscription	*pointer*`[`*expr*`]`
`()`	function call	*expr*`(`*expr_list*`)`
`()`	value construction	*type*`(`*expr_list*`)`
`sizeof`	size of object	`sizeof` *expr*
`sizeof`	size of type	`sizeof(`*type*`}`
`++`	post or pre increment	*lvalue*`++` or `++`*lvalue*
`--`	post or pre decrement	*lvalue*`--` or `--` *lvalue*
`~`	complement	`~`*expr*
`!`	not	`!`*expr*
, -+	unary plus and minus	`-`*expr* and `+`*expr*
`&`	address of	`&`*lvalue*
`*`	dereferencing	`*`*expr*
`new`	create (allocate memory)	`new` *type*
`delete`	deallocate pointer memory	`delete` *pointer*
`delete[]`	free array memory	`delete[]` *pointer*
`()`	cast (type conversion)	`(`*type*`)` *expr*
`.*`	member section	*object*`.*`*pointer-to-member*
`->*`	member section	*pointer*`->*`*pointer-to-member*
`* , / , %`	multiply, divide, modulo (remainder)	*expr* `/` *expr*
`+ , -`	add, subtract	*expr* `+` *expr*
`<< , >>`	shift left , shift right	*expr* `<<` *expr*
`< , <= , > , >=`	relational operators	*expr* `<` *expr*
`==`	equal	*expr* `==` *expr*
`!=`	not equal	*expr* `!=` *expr*
`&`	bitwise AND	*expr* `&` *expr*
`^`	bitwise exclusive OR	*expr* `^` *expr*
`|`	bitwise inclusive OR	*expr* `|` *expr*
`&&`	logical AND	*expr* `&&` *expr*
`||`	logical inclusive OR	*expr* `||` *expr*
`? :`	conditional expression	*expr* `?` *expr* `:` *expr*
`=`	simple assignment	*lvalue* `=` *expr*
`*= , /= , =` , -=+	multiply and assign, ...	*lvalue* `*=` *expr*
`<<= , >>=`	shift left (right) and assign	*lvalue* `<<=` *expr*
`&= , |= , ^=`	AND (OR, XOR) and assign	*lvalue* `&=` *expr*
`throw`	throw exception	`throw` *expr*
`,`	comma (sequencing)	*expr* `,` *expr*

Table 1.1: C^{++} operators in order of decreasing precedence. Each box holds operators of the same level of precedence.

1.4 Control statements

Control statements control the program flow. For example, selection statements such as `if ... else` and `switch` use certain criteria to select a course of action within a program. Iterative control statements (like `for`, `while ... do`, `do ... while` on the other hand see to it that under certain conditions control is passed from the last statement in a block to the first statement.

1.4.1 Selection statements

The if-statement

The following examples illustrate the `if` statement:

```
int isPositive (const int x)
{
  if (x>=0) return 1;
  return 0;
}
```

The condition is defined as **false** if the argument is equal to zero and **true** otherwise. The else statement is similar to that of other programming languages:

```
void isZero (const double x)
{
  if (x)
    cout << "Argument is non-zero." << endl;
  else
   cout << "Argument is zero" << endl;
}
```

The condition `if (x)` is equivalent to the condition `if (x==0)`. Note the difference between the "is equal" operator `==` and the assignment operator `=`.

Instead of executing a single statement conditionally, we can also execute a block of statements conditionally:

```
if ((x>0 && x*x<2) || (x==-1))
  {
    cout << "case 1: x = " << x << endl;
    x *= 3;
  }
else
 {
    cout << "case 2: x = " << x << endl;
    x /= 3;
 }
```

Note that multiplication takes presedence over the relational operator `<` and that the relational operators take presedence over the logical and operator `&&`.

The switch-statement

One can nest `if ... else` statements to test for a number of conditions. In cases where the conditions are integral constants, it is, however, more convenient to use the `switch` statement (the equivalent of `case` statement of Modula-2 or Ada).

```
char c;  cin >> c;

switch(c)
{
  case 'a':  cout << "case a" << endl;
  case 'b':  cout << "case b" << endl;  break;
  case 'c':  cout << "case c" << endl;  break;
  default:  cout << "default code." << endl;
}
```

The switch statement compares a variable of an integral type with an integral constant defined at the `case` labels. Execution jumps to the first match found and will continue sequentially. The case statements act like labeled statements. Hence if the user presses the key `'a'`, the output will be

```
case a
case b
```

Execution continues to the next statement (another `case` statement) until a `break` is reached. The `break` statement can be used only in iteration statements and in `switch` statements. In the case of the `switch` statement it causes execution to jump out of the `switch` block. In the case of iteration statements it causes execution to jump out of the innermost iterative loop.

If the user presses key 'b', or 'c' the screen output will be `case b` and `case c` respectively. Execution jumps to the default label if no match is found. Hence if the user pressses any ther key than `a`, `b` or `c`, the output will be `default code`.

1.4.2 Iterative statements

There are three different iterative statements in C++, namely the `for` statement , the `while` and the `do ... while` statements.

The `for` statement receives an initialization expression, a test expression and an arithmatic expression and performs a statement (or a block of statements) while the test condition evaluates to `true` (non-sero). For example, the following function calculates the faculty of a number iteratively:

```
long int faculty (const long int n)
{
  long int result = 1;
  for (long int i=2; i<=n; i++)
    result *= i;
  return result;
}
```

The variable `i` is initialized to 2 and the statement `result *= i;` is performed while `i<=n` and after each loop `i` is incremented. Note that we declare `i` in the `for` statement itself. Also we could have performed any other arithmetic operation (instead of incrementing). For example, we could have added `3` to the value of `i` after each iteration by replacing the expression `i++` with the expression `i+=3`.

Furthermore, we do not need to supply all three expressions. If we omit any of the expressions we do have to include the semicolons though. For example, the following `for` statement is an infinte loop.

```
for (;;niter++)
{
  ...
  if (x<y) break;
}
```

Here we have no initialization statement and no test statement. After each iteration the variable `niter` is incremented. If `x` remains always smaller then `y` we will be in an infinite loop. Only if `x` becomes greater or equal to `y`, do we break out of the loop.

The `while` loop performs a statement or block of statements while a condition is true. The only difference between the `while` and `do .. while` statements is that the latter performs the test at the end of the loop and hence always goes through at least one iteration. Both of these statements are illustrated in the following two example programs.

1.4.3 Example program: Celsius ⇔ Fahrenheit

Consider first the following program which can be used to convert between degrees celsius and degrees Fahrenheit and which can also print a table of comparison:

```
//                          CELSIUS.CPP
//---------------------------------------------------------------------
//  This program converts degrees celsius to degrees Fahrenheit
//  and vise versa. It can also print a table of comparison
//  for a certain temperature range.

#include <ctype.h>       // for toupper (convert to upper case)
#include <iostream>      // for cin and cout

#include <fstream>

using namespace std;

const double a1 = 1.8;             // Constants required for conversions
const double a0 = 32;              // between 'C and 'F.

double Fahrenheit(const double celsius)    // Function converting 'C to 'F.
{
  return a1*celsius+a0;
}
```

```
double Celsius(const double fahrenheit)    // Function converting 'F to 'C.
{
  return (fahrenheit-a0)/a1;
}

int main()                          // Main program
{
  char choice;
  bool happy;
  do
  {
    cout << "C  ->  convert degrees celsius to degrees fahrenheit"
         << endl;
    cout << "F  ->  convert degrees fahrenheit to degrees celsius"
         << endl;
    cout << endl << "Enter choice (C/F) : ";
    cin  >> choice;

    choice = (char)toupper(choice);  // convert to upper case
    happy  = ((choice == 'C') || (choice == 'F'));

    if (!happy)
      cout << "*** ERROR *** : Illegal choice: ReEnter." << endl;

  } while (!happy);

  char table;
  cout << "Do you want a table of degrees celsius "
       << "versus degrees fahrenheit (y/n)? ";
  cin  >> table;

  if ((table == 'n') || (table == 'N'))
    {
      switch (choice)
      {
        case 'C': double celsius;
                  cout << "Enter degrees celsius : ";
                  cin  >> celsius;
                  cout << celsius << " degrees celsius = "
                       << Fahrenheit(celsius)
                       << " degrees fahrenheit." << endl;
                  break;

        case 'F': double fahrenheit;
                  cout << "Enter degrees fahrenheit : ";
                  cin  >> fahrenheit;
                  cout << fahrenheit << " degrees fahrenheit = "
                       << Celsius(fahrenheit) << " degrees celsius."
```

```
                    << endl;
                break;
      }
    }
  else
    {
      double lower, upper, step;
      cout << "Enter lower limit of range : ";
      cin  >> lower;
      cout << "Enter upper limit of range : ";
      cin  >> upper;
      cout << "Enter increment : ";
      cin  >> step;

      double temp = lower;

      while (temp <= upper)
      {
        switch (choice)
        {
          case 'C': cout << temp << " <-> " << Fahrenheit(temp)
                         << endl;
                    break;

          case 'F': cout << temp << " <-> " << Celsius(temp)
                         << endl;
                    break;
        }
        temp += step;
      }
    }
    char c; cin >> c;
    return 0;
}
```

The two simple functions `Celsius` and `Fahrenheit` perform the conversions between the two temperature scales. Note that the input variables (`fahrenheit` in the case of the function `Celsius`) are declared constant, since the routine does not and should not change this variable. If something should remain constant it is a good to declare it such and the compiler will enforce this constraint.

We defined an enumeration type `boolean` which can take the values `false` and `true` mapped onto zero and one. We declared `happy` as a variable of our boolean data type. It is set to `true` if the upper case of the input character `choice` is equal to either `C` or `F`. The `do ... while` loop continues until `happy` is `true` (nonzero).

`cout` is the standard output stream (usually the screen). The `>>` operator means "push onto output stream". When defining our own classes we shall see how we can overload this operator to push whole matrices or records with a single statement onto any output stream.

1.4.4 Compound Interest

The second example, which calculates the compound interest earned from an investment.

```
//                        INTRST1.CPP
//-----------------------------------------------------------------

/* Calculates the return of an investment after being invested at
   a fixed interest rate (either compounded daily or compounded
   monthly). */

#include <iostream>  // for cin and cout
#include <stdlib.h>    // for exit()

using namespace std;

int main()
{
  double invest, rate;  // declaring 2 doubleing point variables

  enum CompoundingType {daily,monthly,unknown} compounding;

   /* CompoundType is an enumeration type which can take the values
      daily and monthly and compounding is declared as a variable of
      that type. */

  char inputchar;  // declaring a character (byte) variable

  cout << "Enter amount invested: R";
  cin  >> invest;

  if (invest < 0)
  {
    cout << "*** ERROR *** : negative amount invested" << endl;
    exit(0);
      /* exit terminates the process. Before termination all files
         are closed and all buffered output is written. */
  }

  cout << "Enter interest rate (%/year) : ";
  cin  >> rate;

  if (rate < 0)
    cout << "*** Warning *** : entered negative interest rate" << endl;

  compounding = unknown;

  while (compounding == unknown)
  {
```

```
    cout << "Enter the compounding period (d for daily or m for monthly): ";
    cin  >> inputchar;
    switch (inputchar)
    {
      case 'm': compounding = monthly;
                break;
      case 'd': compounding = daily;
                break;
      default:  compounding = unknown;
                cout << "unknown compounding period entered" << endl;
                break;
    }
  }

  if (compounding == monthly)
    {
      int nomonths;  // declared local to this block { ... }

      cout << "Enter number of months invested : ";
      cin  >> nomonths;

      for (int nm=1; nm <= nomonths; nm++)
        invest += invest * rate/(100.0*12.0);
          // typecasting to double (Visual C++ defaults to double)

      cout << "The investment after " << nomonths
           << " months is R" << invest << endl;
    }
  else
    {
      long int nodays;
      cout << "Enter number of days invested : ";
      cin >> nodays;

      for (int nd=1; nd <= nodays; nd++)
        invest += invest * rate/(100.0*365.0);

      cout << "The investment after " << nodays
           << " days is R" << invest << endl;
   }
  char c; cin >> c;

  return 0;
}
```

Note that if the `switch` statement does not find a match, the control is transferred to the statements following the `default` label.

Note that we declare the variables where they are required. For example, the loop

iterators, `nd` and `nm`, for the `for` loops are declared in the loop itself. Similarly, `nomonths` is declared within the loop block and it is local to that loop, i.e. it is not known outside the loop. This is a primitive form of encapsulation.

In case of invalid input the program exits via the `exit()` function defined in `stdlib.h`. This function terminates the calling process after emptying all buffered output streams, closing all files and calling any registers exit routines (defined by `atexit`).

The data is read in from the standard input stream (usually the keyboard) via `cin`+ and pushed into the relevant variables via the `>>` operator.

Note that there is a much easier way to calculate compound interest. Assume the daily interest rate is given by r and that the initial capital is given by $C_)$. After one day the accumulated capital C_1 is given by

$$C_1 = C_0 + rC_0 = (1 + r)C_0$$

After 2 days the accumulated capital is by

$$C_2 = C_1 + rC_1 = (1 + r)^2 C_0$$

Similarly after n days the accumulated capital is given by

$$C_n = (1 + r)^n C_0$$

1.5 Interfacing with functions

A function communicates with other parts of the program via its arguments and its return value.

1.5.1 Passing arguments by value or by reference?

In C^{++} a procedure is treated as a function without a return value (return value `void`). Consider the following trivial program

```
#include <iostream.h>

void f(double x, double& y)
{
  y = x*x;    // y <- 2*2 = 4
  x++;        // x <- x+1 = 3
}

void main()
{
  double x=2, y=0;
  f(x,y);
  cout << "y = " << y << endl;    // writes 4
  cout << "x = " << x << endl;    // writes 2 !!!
}
```

The first argument (`x`) is passed by value. In this case a local copy of the variable is made. Changing this variable within the procedure has no effect on the value of that variable in the calling program. Hence `x` remains equal to `2` after calling `f(x,y)` in `main`. The second argument (`y`) is passed by reference . A reference is an alias to the actual variable passed and hence the actual object passed is used (no local copy is made).

If the function is to return a new value for one of its arguments, this argument should be passed by reference. It is often inefficient to pass large objects (e.g. large matrices or large records) by value. Making a copy of a large object may waste both, computing and memory resources. Instead one can pass them by reference and declare them constant

```
void f (const matrix& A)
{ ... }
```

The compiler will give an error message if the programmer attempts to change any of the array elements of the constant matrix `A`.

Function return values

The return value of a function may also be passed by either value or reference. There is a golden rule you should always keep in mind. Never, ever return a non-static local object (or variable) by reference. Recall that a non-static object is destroyed as soon as it goes out of scope. Hence, a variable declared local within a function is destroyed as soon as you leave the function and youu would be returning a handle to something which no longer exists. For example,

```
double& sqr (const double& x)
{
  double y = x*x;
  return y;
}
```

would yield unpredictable results, since you return a handle to `y` which will no longer exist outside the function. Instead we have to remove the `&` after the`double` so that the result is returned by value (i.e. that a copy is made).

On the other hand, in some cases it might be a good idea or even essential that the functions result is returned by reference. Consider, for example, the following code

```
double& max (const double& x, const double& y)
{
  if (x>=y)
    return x;
  else
    return y;
}
```

In this case, both `x` and `y` exist in the calling routine (they are not local to our function), and it is quite safe to return the greater of the two by reference. In fact, it might be the method of choice because it avoids the overheads of making a copy of that object. This might be especially important if we defined the `max` function for larger data types

or if we defined a generic `max` function (see the section on function templates later in this chapter).

1.5.2 Function overloading

In C++ we can give different functions the same name. The linker resolves the correct function by the function signature. The function signature is the name of the function and the types of its arguments. These are used to define a unique function. Hence

```
float f(int x);

float f(float x);

float f(const float x);

float f(float x, int n);
```

are all unique functions. We can thus have several functions with the same identifier (name). An argument matching process determines which function is to be called. The matching process allows for type conversions between the actual arguments with which the function is called and the formal arguments expected by the function. If more than one function matches the function call, then the compiler complaines about ambiguities between these functions. Certain conversions are considered trivial conversions and these do not define unigue function signatures. These are conversions from pass-by-value to pass-by-reference (e.g. from `float` to `float&`) and from a built-in array type to a pointer (we shall discuss arrays and pointers shortly). Hence `int f(int& a)` would clash with the first of the functions defined above (the compiler would report an ambiguity).

On the other hand, C++ does distinguish between passing an argument by reference and passing it by constant reference. If the function is called with a constant argument the const-reference version is used, since the other version would not guarantee the constantness of the argument.

Note that the return value is not used when matching function calls. For example, the following two functions

```
int f(int x);

float f(int x);
```

would result in an ambiguity, since C++ does not force you to use the return value. C++ would not be able to match the calls

```
int n = 7;
f(n);
```

and

```
int n=7;
cout << f(n);
```

to a unique function.

1.5.3 Functions matching by type conversion

C++ supports type conversions from one integral type to another which uses the same number of bytes or more. Hence there is a sequence of type conversion from `char` to `short int` or `enum` to `long int`. A `int` can be converted to a `long int`, but not vice versa.

Similarly, there exists a series of type conversions for floating point numbers from `float` to `double` and `long double`.

This can be used if one wants to define two versions of a function, one for integral types and one for floating point variables. For example, the gamma function is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \tag{1.1}$$

and when x is an integer it is simply the factorial function offset by one

$$\Gamma(x) = (n-1)! \tag{1.2}$$

It would be natural to define one implementation for integral data types and another for floating point arguments. This could be done by defining

```
long double Gamma (const long double& x);

long int Gamma (const long int x)
```

If we called `Gamma` with a `float` the first function would be used and if we called it with an int, type conversion would result in a call to the second function.

In later chapters we shall see how we can define type conversions for our own data types (e.g. from an array to linked list). C++ would use our type conversions in the same way as it uses the type conversion for built-in data types when trying to match functions.

1.5.4 Default Values: Optional Arguments

C++ has also the facility to assign default values to arguments which have been omitted when the function is called. For example, a function might expect two argument, but the user might call it with only one argument. Usually this would result in a linker error (no function with the correct signature would be found by the linker). If, on the other hand, you give the second argument a default value, C++ will call the function with two argument, using the default value for the second argument.

For example, you might want to write a function `root`, which by default calculates the square root of a floating point number, `x`. If, on the other hand, the user supplies a second integer argument, `n`, then it returns the `n`'th root of `x`. Such a function is shown in the following listing:

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

```
using namespace std;

double root(const double& x, const int n=2)
{
  return pow(x,(1.0/n));
}

int main ()
{
  cout << "root(3.0)    = " <<  root(3.0)   << endl;
  cout << "root(3.0,3) = " << root(3.0,3) << endl;

  char c; cin >> c;

  return 0;
}
```

Here the first call to `root` calculates the square root of 3, while the second call calculates the cubic root. Naturally, you can only omit trailing arguments. In other words, if you give one function argument a default value all the following function arguments must also be given default values.

1.5.5 Functions with Variable Number of Arguments

In adition to supporting optional arguments, C++ also supports functions which receive a variable number of arguments of possibly varying and unspecified types. They are used at times for functions which sum up, calculate the product, or find the minimum or maximum of a varying number of arguments. In most practical cases one would, however, use arrays in preference.

The format of the C++ header of a function which receives a variable number of arguments is one integer argument for the number of remaining arguments followed by a comma and 3 dots:

```
int f(int argCount, ...)
```

The processing of variable arguments functions is facilitated through a header file, `stdarg.h`, which contains the following elements:

- `va_list` represents a pointer to the arguments.
- `va_start` is a mathod used to initialize the pointer to the variable length argument list.
- `va_arg` is a method used to retrieve the next argument.
- `va_end` is a clean-up function which should be called before the function returns.

Below is a little example `sum`-function which sums up a variable number of floating point arguments and returns the result.

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

using namespace std;

double sum(int argCount, ...)  // header for variable argument list
{
  va_list ap;
  double result = 0;
  va_start(ap, argCount);

  while (argCount-- > 0)
    result += va_arg(ap, double);

  va_end(ap);

  return result;
}

int main()
{
  double s = sum(2, 1.2,2.1);
  cout << "sum = " << s << endl;

  s = sum(5, 1.3,2.1,3.3,2.1,2.2);
  cout << "sum = " << s << endl;

  char c; cin >> c;

  return 0;
}
```

The output of the application is

```
sum = 3.3
sum = 11
```

1.5.6 Command-line parameters

Command line parameters are traditionally used when running you program from a command line (e.g. DOS) and supplying arguments to the program. For example, the `copy` program of DOS can take a source and a destination file name as arguments. Entering

```
copy file1.dat file2.dat
```

copies the file **file1.dat** onto the file **file2.dat**, creating the latter if it does not exist. Here **file1.dat** and **file2.dat** are the command-line parameters – command line parameters are seperated by spaces.

You might think that in this day and age of GUI environments (e.g. OS/2, X-Windows or Windows) command-line parameters have become superfluous. This is not the case. For starters, OS/2 and Unix still allow you to work from a command line if you wish. Furthermore, even if you are working only in the GUI-environment, the command line parameters are still used for drag-and-drop or for setting command line parameters in your settings of your program icon.

In the following program we have a function `ToPolar(...)` which receives the rectangular coordinates by value (since they are not altered) and the polar coordinates `r`, `theta` and `phi` by reference.

```
//                    ToPolar.CPP
//------------------------------------------------------------
//       Converts rectangular to polar coordinates

#include <iostream>   // for cout and cin
#include <stdlib.h>     // for exit()
#include <math.h>       // for sin, acos, sqrt and atof

using namespace std;

//------------------------------------------------------------

void ToPolar (const double x, const double y, const double z,
                     double& r, double& theta, double& phi)
{
       // x, y and z do not changed and are sent as a variable
       // (a local copy is made)
       // r, theta and phi are to be changed and are hence sent
       // via reference (the same memory area is used as in the
       // calling program)

  r = sqrt(x*x+y*y+z*z);

  if (r != 0)
    theta = acos(z/r);
  else
    theta = 0;

  if ((theta != 0) && (r != 0))
    phi   = acos(x/(r*sin(theta)));
  else
    phi = 0;
}

//------------------------------------------------------------

void ReadCoordinates(double& x, double& y, double& z)
{
  cout << "Enter x coordinate : ";  cin >> x;
```

```
  cout << "Enter y coordinate : ";  cin >> y;
  cout << "Enter z coordinate : ";  cin >> z;
}

int main(int argc, char* argv[])
{
        // argc holds the no of command line parameters and
        // the array of strings argv[] holds the command line
        // parameters itself. The name of the program is
        // passed in argv[0]

  double x, y, z;

  cout << "::" << argv[0] << "::" << endl;

  switch (argc-1)
  {
    case 0:  ReadCoordinates(x, y, z);  // when no command line
             break;                     // parameters, read x,y,z

    case 3:  x = atof(argv[1]);  // converts first string to double
             y = atof(argv[2]);
             z = atof(argv[3]);
             break;

    default: cout << "*** ERROR *** : invalid number of command "
                  << "line parameters." << endl;
             exit(0);
  }

  double r, theta, phi;

  ToPolar(x,y,z,r,theta,phi);  // calculate polar coordinates

  cout << "(" << x << "," << y << "," << z << ")  => " << r
       << " r  + " << theta << " theta + " << phi << " phi." << endl;

  char c; cin >> c;

  return 0;
}
```

Note also that the function `main` has two arguments, an integer variable `argc` which holds the number of command line parameters and an array of strings `argv[]` which holds the actual command line parameters. If you start the program from the command line by typing

```
ToPolar 1.0 3.0 1.2 <ENTER>
```

`argc` will be set to three and `argv[1] = "1.0"`, `argv[1] = "3.0"` and `argv[1] = "1.2"`. The `argv[0]` will hold the name of the program (`"ToPolar.exe"`). The `switch` state-

ment checks for the number of command-line parameters. If there are 3 command-line parameters it uses these for the rectangular coordinates. If there are no command-line parameters, the rectangular coordinates are read from the standard input stream (the keyboard) and otherwise the program aborts with an error message.

1.6 Pointers

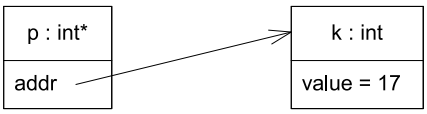

Figure 1.2: A value of a pointer to an int (int*) is the adress of a memory location which has the size and is interpreted as an integer.

Pointers a simply addresses in memory. They are used extensively in C and C++ and a thorough understanding of pointers is essential for good programming skills. Typically pointers in C++ are typed, i.e. they point to a memory location which contains a certain specified data type. For example, in the following listing we define a pointer `p` to an integer and an integer variable `k`.

```
int* p;                 // Defining a pointer to a 2 byte
                        //   (sizeof(int)) memory location.
int k = 17;             // Defining an integer variable.

p = &k;                 // Pointer p now points to address of k.

cout << p;              // Writes the address of p.

cout << (*p);           // Pushes 17 onto the output stream
                        //   (the pointer p is dereferenced).
```

We then use the address operator `&` to set the pointer `p` to the address of `k`. The result is graphically depicted in figure 1.2. We can either print the contents of the pointer variable itself which will be an address or we can dereference the pointer, using the dereferencing operator `*`. The result of the dereferencing operation is the contents of the memory position to which the pointer `p` points.

1.6.1 The NULL pointer

Commonly a NULL pointer is used when the address to which the pointer should point is not yet determined (NULL implies undefined).

```
T* p = NULL;
```

Here we defined a pointer to data type `T` and initialized it to the NULL-pointer. It is good programing practice to initialize all pointers which do not yet point to valid memory locations to the NULL pointer.

1.6.2 Pointers and strong typing

Similarly, if we want to keep the data type to which the pointer points unspecified we can use a `void*`. ANSI C allows pointer of type `void*` to be assigned to any pointer and also any pointer to be assigned to `void*`. In C++ we can still assign a pointer of any type to `void*`, but if we want to assign a `void*` to any other pointer type we must make an explicit type casting. This is illustrated in the following listing

```
void* pvoid;
char* str;

pvoid = str;            // quite legal in C++
str = pvoid;            // illegal in C++
str = (char*)pvoid      // legal with type-casting
```

1.6.3 Constant pointers and pointers to constants

In C++ one can declare either the pointer itself constant (i.e. the memory location it points to may not change), or the item it points to as constant or both. This is illustrated in the following demostration code:

```
int k=17, l=5;

      int*       ip   = &k;
const int*       cip  = &k;
      int* const ipc  = &k;
const int* const cipc = &k;

   ip = &l;     //     legal, pointer value may change.
  *ip = 32;     //     legal, what ip points to may change.

  cip = &l;     //     legal, pointer value may change.
 *cip = 32;     // not legal, what cip points to may not change.

  ipc = &l;     // not legal, pointer value may not change.
 *ips = 32;     //     legal, what ipc points to may change.

 cipc = &l;     // not legal, pointer value may not change.
*cips = 32;     // not legal, what cipc points to may not change.
```

1.6.4 Pointer arithmetic

Pointer arithmetic calculates memory addresses. One can increment, decrement, add an integer to a pointer, subtract an integer from a pointer or subtract one pointer from another pointer.

Pointers in C++ are generally typed. For example, a `double*` points to a memory location of the size of a double precision variable. Adding an integer `n` to the pointer adds `n` times the size of the data type it points to to the address it points to.

```
int *ip = new int[10];

int* ip2 = ip = ip+2;

cout << " ip2 = " <<  ip2 << endl;  // prints address of ip[2];
cout << "*ip2 = " << *ip2 << endl;  // print ip[2];
```

Similarly we can subtract a pointer, increment a pointer (adding 1 times the size of its data type to the address it points to) and decrement a pointer. Subtracting two pointers of the same type from one another returns the distance in memory between the two pointers in multiples of the size of the data type they point to.

1.6.5 Pointers to functions: passing a function as a function argument

For science and engineering applications it is quite common that one wants to pass a function to another function as an argument. Functions in C++ are passed as a and executed from a pointer. For example

```
double (*f)(const double&)
```

defines a pointer variable `f` of type function-pointer. We can assign this function pointer to any function which is type-compatible with the above signature. Consider the following listing:

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

double sqr (double x)
{return x*x;}

void main()
{
  double x = 1.7;
  double (*f)(double) = NULL;
  f = sin;
  cout << "sin(x) = " << f(x) << endl;
  f = sqr;
  cout << "sqr(x) = " << f(x) << endl;
}
```

We declare and initialize a double precision variable `x` and a pointer variable `f` which points to a function which receives a `double` as argument and returns a `double`. We first assign `f` to the function `sin` defined in the standard C++ library `math.h`. Calling `f(x)` returns `sin(x)`. We then assign the function pointer `f` to our own function `sqr`. Now `f(x)` returns `x*x`.

In a similar way we pass a fuction to another function as an argument. Consider, for example, that you want to write a numerical integration routine which evaluates the integral of any given function between given integration boundaries. In other words,we want to evaluate

$$I = \int_a^b f(x)dx \tag{1.3}$$

for any given $f(x)$, a and b. The function header could look like this:

```
double Integrate (double (*func)(double), const double a,
                  const double b, const double eps);
```

where `eps` is the accuracy with which we want to approximate the exact integral.

A simple, yet quite robust integration rule is Simpson's integration rule:

$$\begin{aligned} I \approx & \frac{h}{3}[f(a)+f(b)+4(f(a+h)+f(a+3h)+\ldots+f(b-h)) \\ & +2(f(a+2h)+f(a+4h)+\ldots+f(b-2h))] \end{aligned}$$

Here h is the stepsize on which the function is evaluated, i.e. the integration interval $[a,b]$ is subdevided into N equally sized subintervals with width

$$h = \frac{b-1}{N}$$

Below we give a simple implementation of the Simpson integration rule:

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

using namespace std;

double f(const double& x) {return exp(x*x);}

double Simpson (double (*func)(const double&),
                const double& a, const double& b, const int nintvl)
{
  double dx = (b-a)/nintvl;
  double x  = a+dx;

  double sumeven = 0;
  double sumodd  = func(x);

  for (int n=2; n<nintvl; n=n+2)
  {
    x += dx;
    sumeven += func(x);
    x += dx;
    sumodd += func(x);
  }
```

```
  return (func(a) + func(b) + 2*sumeven + 4*sumodd)*dx/3;
}

int main()
{
  int nintvl;
  double a, b;

  cout << "Enter no of intervals = ";            cin  >> nintvl;
  cout << "Enter integration range a b : ";    cin  >> a >> b;

  cout << "Simpson: "   << Simpson(f,a,b,nintvl) << endl;

  char k; cin>>k;

  return 0;
}
```

If we run the example program which integrates e^x with 100 intervals integrating over the range $[0, 1]$, we obtain the following result:

```
Enter no of intervals = 40
Enter integration range a b : 0 1
Simpson: 1.46265
```

1.7 Arrays

An array is a contiguous region of storage, large enough to hold all its elements. The array elements are thus ordered in memory and can be accessed via numeric subscripts.

1.7.1 Arrays in static memory

We can declare an array of a fixed size by using the element access operator `[]`. For example, `int iArray[5]` reserves a memory for 5 integer variables which are accessed via `iArray[0]`, ..., `iArray[4]`. Arrays can be multi-dimensional. For example, in the listing below we define a (3x3) array `M` of double precision variables. The elements are accessed via `M[i][j]` where `i` and `j` run from 0 to 2. The function `sizeof(M)` returns the size of the entire memory block of 9 double precision floating point numbers. Note that `nrows` and `ncols` are defined as constants. This is essential, because the size of an array in static memory must be known at compile time.

```
//                    StaticArray.cpp
//-------------------------------------------------------
//    Creating arrays whose size is known at compile time.

#include <iostream>

using namespace std;
```

```
int main()
{
  const int nrows = 3;
  const int ncols = 3;

  double M[nrows][ncols];

  cout << "Enter (" << nrows << "x" << ncols << ") matrix M:" << endl;

  for (int nr=0; nr<nrows; nr++)
  {
    for (int nc=0; nc<ncols; nc++)
    {
      cout << "M[" << (nr+1) << "," << (nc+1) << "] = ";
      cin >> M[nr][nc];
    }
  }
  cout << "sizeof(M) = " << sizeof(M) << endl;
      // returns size of 9 double precision
      // floating point variables.

  char k; cin >> k;

  return 0;
}
```

An array can also be declared and initialized in a single statement. For example

```
int intArray[] = {1, 78, 12, 5};
```

allocates memory for 4 integer variables, initializes these memory positions the relevant integer values and sets the pointer `intArray` to the start of that memory block. Similarly

```
double matrix[3][3] = { {1.0, 1.7, 2.3},
                        {3.2, 7.9, 0.4},
                        {0.1, 9.9, 4.2} };
```

declares and initializes a 3x3 array o floating point variables. while

```
double matrix[3][3] = { {1.0},
                        {3.2},
                        {0.1} };
```

declares a 3x3 array and initializes the first column. Note that unlike FORTRAN, C++ stores its arrays in row-major order.

1.7.2 Arrays in dynamic memory

Static allocation of arrays has, however, a few severe disadvantages. If one, for example, writes a function for least squares fitting of a set of data points to a function, the number

of data points are mot generally known beforehand. One could make the assumption that the user will never use more than say 200 data points. Then the user could never supply more than 200 data points. Furthermore, if he supplies only 5 data points the program would still use memory for 200 points. Finally, the meory cannot be released while the array is in scope, even if it is no longer needed.

An alternative approach is to reserve memory at run-time. This is done in C^{++} via the `new` operator. If we want to reserve memory for an array of `n` integer variables we can do this by

```
int *vector = new int[n];
```

This statement declares an integer pointer `vector`, reserves a block of contiguous memory space, large enough for `n` integer variables, and sets the address of the pointer variable `vector` to the start of the memory block. The representation in memory is illustrated in figure 1.3 (we replaced the name of the pointer variable by `v` to achieve a more compact notation).

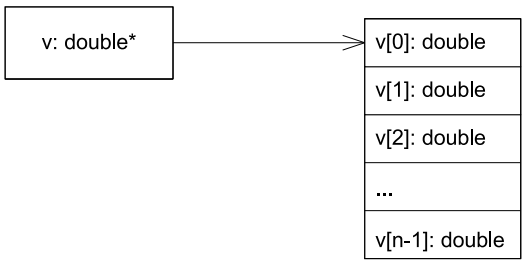

Figure 1.3: Memory structure for a one-dimensional dynamic array.

Note that `n` need not be constant and can be read in from the terminal. When we no longer require the vector we can free the memory and thereby conserve our hardware resources. This is done by the following statement

```
delete[] vector;
```

The following program shows how one could dynamically allocate a two-dimensional array (which can be used to represent the data of a matrix).

```
//                    DynamicArray.cpp
//-----------------------------------------------------------
//                Dynamic Memory Arrays

#include <iostream>

using namespace std;

int main()
{
  int nrows, ncols;
```

```
  cout << "Enter number of rows : ";
  cin >> nrows;
  cout << "Enter number of columns : ";
  cin >> ncols;

  double **M = NULL;  // it is usually a good idea to initialize
                      // pointers to NULL.

  int nr, nc;

  M = new double*[nrows];
  for (nr=0; nr<nrows; nr++)
    M[nr] = new double[ncols];

  cout << "Now enter the elements of the (" << nrows << "x"
       << ncols << ") matrix M :" << endl;

  for (nr=0; nr<nrows; nr++)
  {
    for (nc=0; nc<ncols; nc++)
    {
      cout << "M[" << (nr+1) << "," << (nc+1) << "] = ";
      cin >> M[nr][nc];
    }
  }

  cout << "sizeof(M)    = " << sizeof(M)
       << endl;  // returns size of a pointer variable
  cout << "sizeof(M[1]) = " << sizeof(M[1])
       << endl;  // returns size of a pointer variable
  cout << "sizeof(M[1][1]) = " << sizeof(M[1][1])
       << endl;  // returns size of a double precision
                 // floating point variable

  for (nr=0; nr<nrows; nr++)
    delete[] M[nr];

  delete[] M;

  char k; cin >> k;

  return 0;
}
```

An example output of the program is shown below:

```
Enter number of rows : 2
Enter number of columns : 3
Now enter the elements of the (2x3) matrix M :
```

```
M[1,1] = 1
M[1,2] = 2
M[1,3] = 3
M[2,1] = 2
M[2,2] = 3
M[2,3] = 1
sizeof(M)    = 4
sizeof(M[1]) = 4
sizeof(M[1][1]) = 8
```

`M` is defined as a pointer to a pointer to a double precision variable. We first reserve memory for `nrows` pointers and set the pointer `M` to the start of that memory block. For each of the `nrows` pointers we reserve enough memory for `ncols` double precision variables. The memory arrangement is illustrated in figure 1.4.

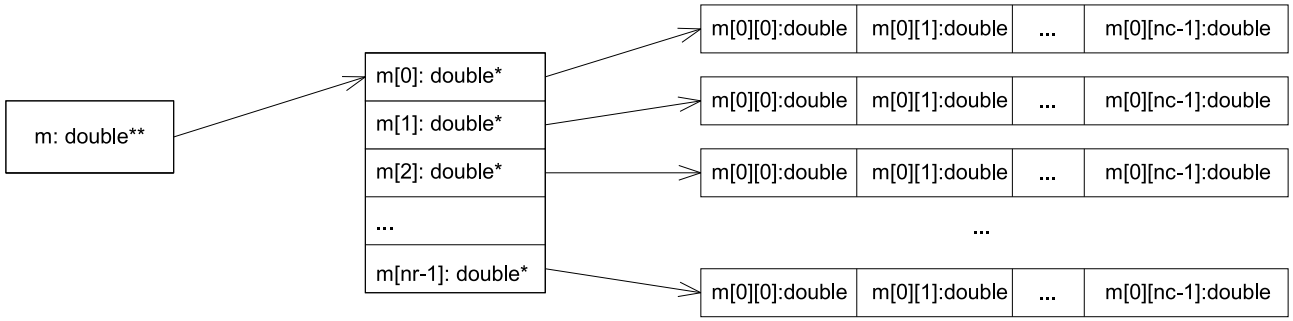

Figure 1.4: Memory structure for a two-dimensional dynamic array.

Note that `sizeof(M)` and `sizeof(M[1])` both return the memory required for a pointer variable while `sizeof(M[1][1])` returns the memory required for a double precision variable.

When using dynamic memory allocation, one should be careful that there are no memory leaks (i.e. that all memory which is allocated during run-time is deallocated again — preferably as soon as it is no longer required) and that the integrity of pointers is ensured (i.e. that pointers do not end up pointing to memory areas which are reserved for other purposes). It is a good programming practice to set all pointers for which there is no memory reserved equal to NULL (known as the null pointer).

Example program: Linear regression

We demonstrate the usefulness of dynamic memory with a general linear regression program. In this case an experimentalist has measured a set of data points and he wants to fit the "best" straight line through these data points.

Consider a set of data points $\{(x_i, y_i),\ i = 1 \ldots n\}$. By minimizing the sum of the squares of the error (least-squares fit) one obtains the following expressions for the slope

of the straight line and its y-intercept:

$$\begin{aligned} \text{slope} &= \frac{(\sum_{i=1}^{n} x_i) \cdot (\sum_{i=1}^{n} y_i) - n \sum_{i=1}^{n} x_i y_i}{(\sum_{i=1}^{n} x_i)^2 - n \sum_{i=1}^{n} x_i^2} \\ y - \text{intercept} &= \frac{\sum_{i=1}^{n} y_i - \text{slope} \cdot \sum_{i=1}^{n} x_i}{n} \end{aligned}$$

The following little program will ask for the number of data points and it will determine the best straight-line fit to these points:

```
//                 LinearRegression.cpp
//-------------------------------------------------------------------
// Asks for a collection of 2-dimensional data points
// and then fits the "best" straight line through these
// data points using the linear regression algorithm

#include <iostream>

using namespace std;

void readDataPoints(double*& x, double*& y, int& numDataPoints)
{
  cout << "Enter number of data points: ";  cin >> numDataPoints;

  x = new double[numDataPoints];
  y = new double[numDataPoints];

  for (int i=0; i<numDataPoints; ++i)
  {
    cout << "x[" << (i+1) << "] y[" << (i+1) << "] = ";
    cin >> x[i] >> y[i];
  }
}

//-------------------------------------------------------------------

void linearRegression (double x[], double y[], const int numDataPoints,
                       double& slope, double& y_intercept)
{
  double sum_x=0, sum_x2=0, sum_y=0, sum_xy=0;

  for (int i=0; i<numDataPoints; ++i)
  {
    sum_x  += x[i];
    sum_y  += y[i];
    sum_x2 += x[i]*x[i];
    sum_xy += x[i]*y[i];
  }

  slope = (sum_x*sum_y - numDataPoints*sum_xy)
```

```
            / (sum_x*sum_x - numDataPoints*sum_x2);

  y_intercept = (sum_y - slope*sum_x) / numDataPoints;
}

//---------------------------------------------------------------

int main()
{
  int numDataPoints;

  double *x, *y;

  readDataPoints(x, y, numDataPoints);

  double slope, y_intercept;

  linearRegression(x, y, numDataPoints, slope, y_intercept);

  char* sign = " + ";
  if (y_intercept < 0)
    sign = " - ";

  cout << "Straight line fit obtained from linear regression: "
       << " y = " << slope << "x " << sign << y_intercept << endl;

  delete[] x;
  delete[] y;

  char k; cin >> k;

  return 0;
}
```

Note that we declare the pointer variables in the main program and that we reserve memory for the arrays and assign the pointer variables to the starting address of these arrays in the function `readDataPoints`. We thus have to pass the pointers by reference. Note also that `double* x` and `double x[]` are type compatible. Once we now longer require the arrays, the memory is released via the array-delete operator, `delete[]`.

As an example we ran the program with the five data points indicated by asterisks in figure 1.5 and have plotted the linear regression result onto the same figure.

1.7.3 Array versus vectors and matrices

We will differentiate between C and C++ arrays (static or dynamic) and vectors and matrices. Arrays can represent the data structure of a vector or a matrix, but vectors and matrices have associated with them certain operations like the vector addition or the vector dot product or matrix multiplication. Also, when using C-type arrays (via

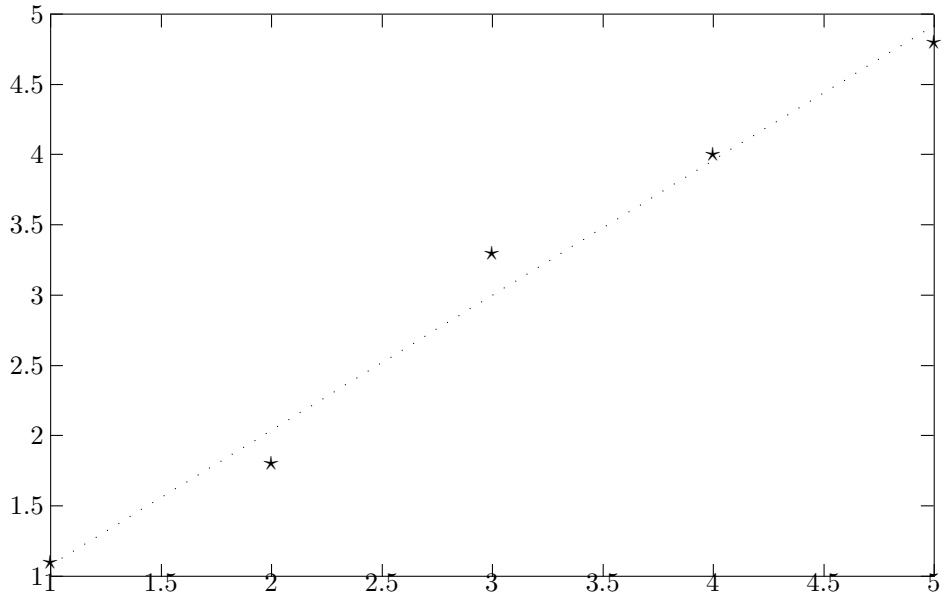

Figure 1.5: Linear regression result.

pointers) there is no automatic range checking.

Later in this book we shall discuss array and vector and matrix classes which do support controlled element access and data-type specific functionality. This will allow us to write code which is very similar to Matlab or Mathematica code in that we will be able to, for example, directly multiply matrices in very simple and very powerful code like the code extract shown below:

```
matrix<double> A(n,n), B(n,n), C(n,n)
A = B*C;
cout << "matrix B x matrix C = " << A;
```

1.7.4 Character arrays as strings

C++ has no predefined string data type. Instead one uses arrays to characters, which behave many, but not in all ways identical to any other array type. There are, however, some important differences which are illustrated in the following code:

```
#include <iostream.h>

void main()
{
  char str[20] = "This is a string";
  char* pchar = str;
  cout << str    << endl;         // prints "This is a string"
  cout << pchar << endl;          // prints "This is a string"

  int ivec[5] = {1, 2, 3, 4, 5};
```

```
  int* pvec = ivec;
  cout << pvec << endl;           // prints the address of ivec[0]
  cout << (*pvec) << endl;        // prints its contents, 1

  ++ pvec;
  cout << (*pvec) << endl         // prints the second element, 2
}
```

In the above listing, both `str` and `p` are effectively pointers to characters. The first statement does a lot of things in the background. It first defines `str` as a pointer to `char`, reserves 20 bytes (for 19 characters and the terminating `\0` character) in memory, initializes these memory positions with the string characters and the terminating NULL character and finally it sets the pointer `str` to the starting position of this memory area. The second statement simply defines a pointer `pchar` to `char` and initializes this pointer to the memory position of `str`. Pushing `pchar` onto the output stream prints not only the first character `'T'`, but the entire string. This behavior is not usual for pointers. In fact, it only works like this for pointers to the data type `char`. The reason for this is historical – it simulates some functionality of a string data type. If we do the same for an array of integers then things look quite differently. We first define a vector of 5 integers which are initialized to 1–5. We then define a pointer `pvec` and set it to the starting address of this vector. Now printing out `pvec` results in the more standard behavior, i.e. the address of the first element of the vector is printed. Dereferencing the pointer results in the contents of this memory position which is the first element in the array. Since the vector is typed (of type `int*`) we can increment the vector using the standard pre- or post-fix incrementation operator. The result is that the pointer now points to the next memory area of size `sizeof(int)` and dereferencing the pointer now yields the second element of the vector.

Since C^{++} supports most aspects of C one can define a string on a `char`-pointer and use the ANSI-C string manipulation functions like `strcpy` for string copy, `strcat` for string concatenation, `strcmp` for string comparison and so forth.

Note that strings are really data types with their own functionality (e.g. addition would imply string concatenation). It is hence usually a good idea to either write your own string class or to use a string class supplied with your compiler or by an independant vendor.

1.8 Function templates

Templates are the way in which C++ implements generic functions. Traditionally, if you define a function you would have to define it for every data type with which you would want to be able to call that function. If you would write a mathematical library containing Bessel functions you would have to define the Bessel functions for `int`, `float`, `double`, `long double` and `complex`. If you want to improve your algorithm at a later stage you would have to search for all definitions of the Bessel function and make the relevant changes. This is cumbersome and prone to errors. Similarly, you would typically have to write a Simpson integrator function for functions of type `float`,

`double` and `long double` and possibly also for your own data types like, for example, `Rational`. By defining the Simpson integrator on a template

```
template <class T>
T Simpson (T (*f)(const T&), const T& a, const T& b, const int nintvl);
```

the compiler will automatically generate the code for the data types required in the code. The phrase `template <class T>` in front of the function definition specifies that the function is defined on a template. Here `T` is a type-variable (you can use your own variable name, if you like. At least one of the arguments of the function must refer to type `T`. Since the return value of a function does not participate in defining a unique signature (since the user need not use the return value), it is not sufficient that the template type is referred to only in the return value of the function.

Now, if you call `Simpson` twice, once with a function of the form

```
double func(const double& x);
```

and once with a function of the form

```
Rational fr(const Rational& r)
```

then the compiler will write for you two functions, one with every occurance of `T` replaced by `double` and one with every occurance of `T` replaced by `Rational`.

It is interesting to note that the template argument `T` can refer to object types, function names, constant expressions or character strings.

1.8.1 Example program: Bubble-sort

We illustrate function templates via a simple bubble-sort program. Since the program will be defined on a template, it will be able to sort arrays of any data type, as long as there is a greater-as operator, `>`, and an assignment operator, `=`, defined for the data type (or the class). In our example program we sort an array of `float` and an array of `char` with the same Bubble sort routine.

```
//                        BubbleSort.cpp
//-------------------------------------------------------------------
// A Bubble sort algorithm defined on a template. It can hence sort
// an array of any class, e.g. integers, floating point numbers, characters,
// or any user-defined class (e.g. a data base of employees)

#include <iostream>

using namespace std;

template <class T>                              // T is determined by the
void BubbleSort (T vec[], const int length)     // calling program
{
  int exchanged, nleft = length;
```

```
  do
  {
    exchanged = 0;
    nleft--;
    for (int n=0; n<nleft; n++)
    {
      if (vec[n] > vec[n+1])  // if next element smaller: exchange
      {
        T dummy  = vec[n+1];
        vec[n+1] = vec[n];
        vec[n]   = dummy;
        exchanged++;
      }
    }
  } while (exchanged);  // if no exchanges in previous loop: done
}

int main()
{
  int lngth=5;
  float* fvec = new float[lngth];    // create an array of floats

  fvec[0]=8.1; fvec[1]=3.7;  fvec[2]=4.2; fvec[3]=9.0; fvec[4]=7.2;

  BubbleSort (fvec, lngth);      // bubble sort this vector

  for (int i=0; i<lngth; i++)    // show result on screen
    cout << "fvec[" << i << "] = " << fvec[i] << endl;

  delete[] fvec;   // vector no longer needed => discard

  cout << endl;

  lngth=6;                       // the same for vector of chars
  char* cvec = new char[lngth];

  cvec[0]='s'; cvec[1]='d'; cvec[2]='r'; cvec[3]='a'; cvec[4]='z';
  cvec[5]='m';

  BubbleSort (cvec, lngth);

  for (int i=0; i<lngth; i++)
    cout << "cvec[" << i << "] = " << cvec[i] << endl;

  delete[] cvec;

  char k; cin >> k;

  return 0;
```

```
}
```

The applicability of this sorting routine is, however, not limited to predefined data types. It can also be used for user defined data types (e.g. an array of rational numbers or an array of employee records).

1.8.2 Multiple templates

We can also define a function (or a class) on multiple templates. For example, you might want to sort an associative array (also called a dictionary or a map). An associative array keeps for each element a key. Given the key, we can access the value (this is an abstraction of accessing the array elements via integers).

A simple, non-object-oriented way of defining a sorting algorithm for an associative array would be to use two template types:

```
template <class K, class V>
void quickSort (K keys[], V values[], const int length);
```

1.8.3 Overloading template functions

Template functions can be overloaded just like any other function. Consider, for example, the following program listing

```
#include <iostream>

using namespace std;

template <class T>
T Max(const T& x, const T& y)
{
  if (x > y)
    return x;
  else
    return y;
}

template <class T>
T Max(const T* const vec, const int length)
{
  T largest = vec[0];
  for (int i=1; i<length; i++)
    if (vec[i] > largest)
      largest = vec[i];
  return largest;
}

int main()
{
  double a=5.1, b=1.23;
```

```
  cout << "a, b = " << a << ", " << b << endl;
  cout << "Max(a,b) = " << Max(a,b) << endl << endl;

  int v[5];
  v[0]=9; v[1]=2; v[2]=11; v[3]=7; v[4]=8;
  cout << "v = [";
  for (int i=0; i<4; ++i)
    cout << v[i] << ", ";
  cout << v[4] << "]" << endl;

  cout << "Max(v,4) = " << Max(v,4) << endl;

  char k; cin >>k;

  return 0;
}
```

The output of the above program is shown below:

```
a, b = 5.1, 1.23
Max(a,b) = 5.1

v = [9, 2, 11, 7, 8]
Max(v,4) = 11
```

Here we defined two version of the function `Max`. Both versions are defined on a template. The first version takes two scalar arguments (for example two floating point numbers) and returns the larger of the two. The second version, which carries the same name as the first takes, a pointer variable (for an array) as the first argument and an integer argument specifying the number of elemnts in the array. This function returns the largest of the array elements. Note that the first argument, `vec`, is defined as

```
const T* const vec
```

In this case both, the pointer variable `vec`, and the memory locations it points to (the array elements) are treated as `const`.

1.9 Recursion

Recursion occurs when a function calls itself, either directly or indirectly. Many mathematical problems can be defined very elagently using recursion. We have direct recursion when a function calls itself. Indirect recursion involves, for example, a function `f1` calling another function `f2` which calls again `f1`. For example the right-hand side calling hierarchy in figure **??** contains both direct (`f2` calling itself) and indirect recursion (for example, `f6` calls `f8` calls `f9`).

Often one finds simple functions, like the factorial function, implemented recursively:

```
long int factorial(const long int n)
{
```

```
  long int local = n;
  if (n>=2)
    local *= factorial(n-1);
  return local;
}
```

Note, however, that if we calculate $n!$ the function, `factorial`, is called n–1 times and hence n–1 stack frames are created, each frame containing a copy of the local variable `local`, the function argument `n` and the calling address. Finally the stack has to be unwound before the final result is passed to the function calling `factorial`. A good proportion of these overheads is avoided by mapping the recursive algorithm onto a non-recursive one:

```
long int factorial(const long int n)
{
  long int local = n;

  for (int i=n-1; i>=2; i--)
    local *= i;

  return local;
}
```

Furthermore, recursive cycles in a function calling diagram complicate the analysis, design and testing significantly. In Figure **??** a function calling lattice (a lattice is a tree for which branches can share descendents) is compared to a function calling diagram which contains direct and indirect recursion. The lattice structure is not only simpler to comprehend, but it has the additional advantage that the leaves of the tree (f6, f9 and f10) can be tested independetly, and once it is established that these work correctly, the next level of functions can be tested. These functions (f5,f7 and f8) call only functions which have been tested already. On the other hand, if the function diagram contains indirect recursion (f4–f6 and f6–f8–f9), then the function making up the recursive cycle cannot be tested independetly.

Hence, both direct and indirect recursion havesignificant disadvantages and it is generally a good idea to try and map recursive dependancy cycles onto lattice structures. This may, of course not always be a reasonable option.

1.10 Simple file I/O and simple strings

The following program illustrates simple file input and output as well as simple strings. As is the case for vectors and matrices, strings in C^{++} are best handled by defining for them an abstract data type (a class). This will be done in the coming chapters. Here we use the standard C^{++} strings which are defined as arrays of characters with a terminating `NULL` character (`\0`). Hence the string variable `inputfilename` is defined as an array of 60 characters (the string can be no longer than that length with the actual length determined by the position of the `NULL` character in that array of characters).

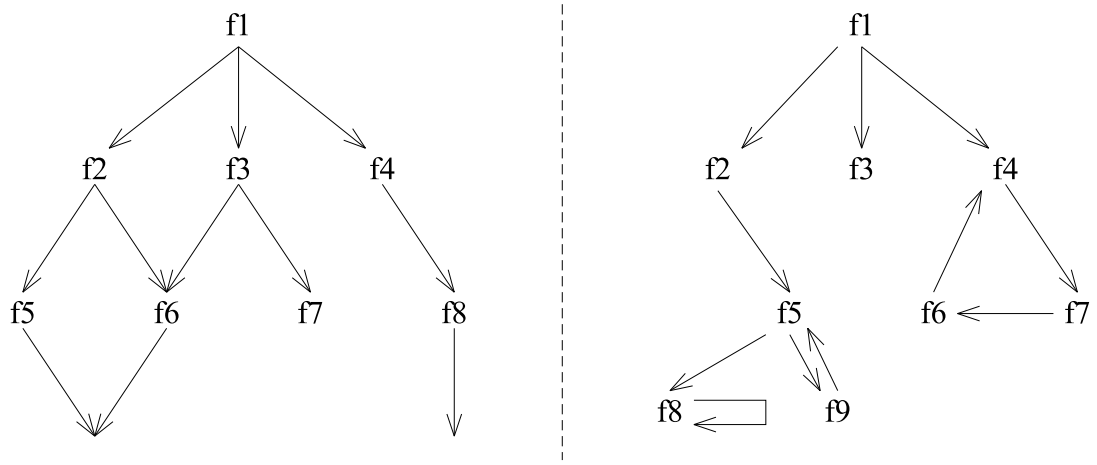

Figure 1.6: A lattice function calling hierarchy (left) compared to a more general function calling diagram containing direct and indirect recursion.

The function `strlen()` returns the length of the string by determining the position of the `NULL` character.

A string can also be declared and initialized in a single statement

```
char* myname = "Peter Smith";
```

In this case the compiler allocates enough memory to hold the relevant string (including the terminating `'/0'`-character) and sets the pointer to the start of that memory space.

1.10.1 Example program: Replacing all occurances of a string in a file

```
//                    Replace.CPP
//-----------------------------------------------------------
// Search the contents of a file and replace all occurances
// of oldstr with the string newstr writing the result into
// a new file.

#include <stdlib.h>            // for exit()
#include <string>             // for strlen()
#include <fstream>            // for fileIO
#include <iostream>           // for console IO

using namespace std;

void replace(ifstream& infile, ofstream& outfile,
             char oldStr[], char newStr[], long int& numSubstitutions)
{
  char c;
```

```
  unsigned int k=0;
  numSubstitutions = 0;
  while (!infile.eof())   // while not end of input file
  {
     infile.get(c);  // read next character from input file

     if (c==oldStr[k])    // if character conforms to next
     {                    // character of string to be replaced
       ++k;               // increment pointer which points to
                          // number of chars which aggree.
       if (k==strlen(oldStr))  // If entire string aggrees with
       {                       // string to be replaced, write
         outfile << newStr;    // replacement string to output
         k = 0;                // file & reset substring pointer.
         ++numSubstitutions;   // Counting no of replacements.
       }
     }
     else                  // if next character does not aggree:
     {
       for (unsigned int kk=0; kk<k; ++kk) // First write substring which
       outfile << oldStr[kk];     // did aggree, then
       outfile << c;              // the char which didnt aggree
       k=0;                       // and reset substring pointer
     }
  };
}


int main()
{
  char inputfilename[60];
  cout << "Enter name of input file: "; cin  >> inputfilename;

  ifstream infile (inputfilename);  // opening input stream
  if (!infile)                      // fromfile inputfilename
  {
     cout << "*** ERROR *** :  cannnot open " << inputfilename
          << endl;
     exit(1);  // Flushes output buffers, closes files,
  };           // aborts program

  char outputfilename[60];
  cout << "Enter name of output file: "; cin  >> outputfilename;
  ofstream outfile (outputfilename); // open output file stream

  char oldStr[80], newStr[80];

  cout << "Enter string to be replaced throughout file: ";
  cin  >> oldStr;
```

```
  cout << "Enter replacement string: ";
  cin  >> newStr;

  long int numSubstitutions = 0;
  replace(infile, outfile, oldStr, newStr, numSubstitutions);

  infile.close();
  outfile.close();

  cout << "The string <" << oldStr << "> has been replaced by <"
       << newStr << "> " << numSubstitutions << " times." << endl;

  char k; cin >> k;

  return 0;
}
```

An input file is opened by creating an object of class `ifstream` with the file name as parameter. The statement

```
if (!infile) ...
```

checks whether the file `infile` was opened successfully. We use the stream method `get()` instead of the `>>` operator, since the latter filters out spaces, carriage return and line feed characters. When the files are no longer required they are closed via the stream method `close()`.

When running the program on its own source code, replacing `oldStr`, we obtain the following output:

```
Enter name of input file: Replace.cpp
Enter name of output file: Replace.out
Enter string to be replaced throughout file: oldStr
Enter replacement string: THE_VERY_OLD_STRING
The string <oldStr> has been replaced by <THE_VERY_OLD_STRING> 8 times.
```

1.11 Constants and inline functions instead of macros

For *C*-programmers it was common to define constants via the macros . For example, one would define

```
#define eps 1.0e-8
```

In this case the *C* or C^{++} preprocessor replaces every occurance of `eps` with `1.0e-8` before the source code is forwarded to the compiler. This has several disadvantages. Firstly any compiler error involving `eps` will refer to `1.0e-8` and not to `eps`. If the code is long it might take the programmer a very long time before he finds the error. The same problem is encountered when one uses a symbolic debugger, since `eps` is never entered into the programs symbol table. Furthermore, the constant `eps` cannot be given an explicit type.

All these problems can be solved in C^{++} by replacing the above macro by

```
const double 1.0e-8
```

The constant can be defined either locally or globally (preferably locally).

An even bigger sin is to use macros in order to define inline functions. Consider the following commonly used macro

```
#define SQR(x) x*x
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

One needs the parenthesis around `x` and `y` in order to allow the user to use the macro with expressions. The macros look harmless enough. Consider, however, the following pitfalls:

```
y = SQR(1+2);     // result: 5 instead of 9
z = MAX(x++,y);
```

In the first case the macro expansion results in

```
y = 1+2 * 1+2;
```

which evaluates to 5 (since multiplication has higher precedence than addition). In the second case `x` will be incremented either once or twice, depending on whether `(x+1)` is greater than `y` or not. It should be clear that macros like the above are not only bad style but downright dangerous. Again there is an elegant alternative available in C^{++}:

```
//                NoMacro.cpp
//--------------------------------------
#include <iostream>

using namespace std;

template <class T>
inline T sqr(const T& x) {return x*x;}

template <class T>
inline T& max(T& x, T& y)
{
  if (x > y)
    return x;
  else
    return y;
}

int main()
{
  cout << "sqr(2+1) = " << sqr(2+1) << endl;

  double a=2.1, b=1.7;
  cout << "max(a,b) = " << max(a,b) << endl;

  char k; cin >> k;
```

```
  return 0;
}
```

The output of the program is:

```
sqr(2+1) = 9
max(a,b) = 2.1
```

Declaring these functions as `inline` avoids the function call overheads. Note, however, that `inline` is a request to the compiler which might be ignored if the compiler feels it is not a good idea (e.g. for example for recursive functions and complex functions).

Defining the functions on a template ensures that `sqr` and `max` can be used for several data types (e.g. for integers, floating point numbers, or your own classes — say a class of rational numbers). In contrast to the macro definitions, the inline functions are type-safe.

E:1.1 The rules for leap years are a little convoluted: every 4'th year is a leap year except every century which isn't except every 4'th century which is. Write a function `leapyear` which takes an integer as argument and returns 1 if its is a leap year and zero otherwise. In you main program, read in years in a loop until the user enters a zero for the year. For each year enetered, the program should report whether it is a leap year or not. (Hint: use the remainder operator `%` in your leap year function).

E:1.2 Write a program which reads in the parameters a, b and c of a parabola $y(x) = ax^2 + bx + c$ and gives as output the roots of the parabola as well as its turning point. The roots of a parabola are given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

and its turning point is given by

$$x_0 = -\frac{b}{2a}; \qquad y_0 = y(x_0) \tag{1.4}$$

(there is a `sqrt` function in the standard C++ library **math.h**).

E:1.3 Write a program which reads in a series of floating point numbers (use dynamic memory allocation), calculates their mean and their standard deviation from that mean (declare functions `mean` and `stddeviation`). The mean and the standard deviation are defined as follows:

$$\begin{aligned} \text{mean} &= \frac{1}{N}\sum_{i=1}^{N} x_i \\ \text{stddeviation} &= = \sqrt{\sum_{i=1}^{N}(x_i - \text{mean})^2} \end{aligned}$$

E:1.4 Write a program which reads in a string of up to 80 characters and which gives as output the upper case version of that string together with the number of characters in that string. For the case where the input string is upper case already, let it print a message to that effect. (The standard *C* functions `int toupper(int chr)` and `int isupper(int chr)` are defined in the library `<ctype.h>`. `isupper` returns nonzero if `chr` is an upper case character (`A` to `Z`).

E:1.5 Write a procedure `swap` which swaps any two objects `x` and `y` (Hint: use function templates). Write a short main program which illustrates how this procedure can be used to swap two floating point numbers, two characters, two strings and two arrays of floating point numbers (declare any two strings and arrays). Explain what happens in the latter two cases (what happens to the pointer variables and to the memory they point to?).

E:1.6 Write a program which reads the source code of a C^{++} program and removes all the comments from it.

E:1.7 Write a series of functions calculating the circumference of various geometrical shapes. Each of the functions should be called `circumference` (function overloading). If it is called with a single parameter it is to assume that the shape whose circumference you want to calculate is a circle and the parameter is the radius. If you call it with two parameters it should assume that the shape is a rectangular with the two parameters being the width and height of the rectangular. If you call it with 3 parameters it should assume the shape is a triangle with the parameters being the three side lengths.

Chapter 2

Introduction to Object-Oriented Programming in C++

2.1 Objects and Classes

In object-oriented programming the central concept is obviously an object. Let us first clarify the concepts of objects and classes, before going into the C++ language syntax for defining classes adn creating and using objects.

2.1.1 What is an Object?

The central concept of object-oriented programming is an object. An object is a unit which

- has identity,
- has attributes and
- can perform operations.

We think object-oriented. When we use a nouns in a sentence structure we generally refer to objects. For example,

"The green car drives along the lane."

Here the nouns, *car* and *lane*, are objects. The objects have attributes, i.e. the car is *green*, and can perform operations, ı.e. the car *drives*. Hence, object orientation is not an unfamiliar concept in human thought – it is our natural way of thinking.

2.1.2 Identifying Objects

We have already mentioned one way to identify objects. That is by extracting the nouns from a linguistic description of the system we are modeling. Otherwise we can try and identify units which have a clear conceptual meaning and which can be assigned an identity which makes them unique. Typically these units would have attributes and

typically they can conceptually perform certain operations, i.e. it is natural to give thjem responsibilities for certain tasks.

Another approach is to think of a system and identify its components. Each of the components would be an object in their own right. In this way you can go on to deeper and deeper levels of components.

Alternatively you can look at an object and identify any messages it sends in order to request certain services (i.e. identify the recipients of the function calls). The recipients of these messages would themselves be objects.

2.1.3 Classes as Abstractions of Objects

A class is a template from which objects are created. It encapsulates the commonalities of all instances (objects) of that class. Objects are instances of classes.

A class defines the attribute types as well as the operations (services) which its instances have. For example, an account may have an account number, a reference to an owner, a balance and may offer services for crediting, debiting and querying the balance. A particular account is an instance of the class. The class itself is a more abstract concept. One can discuss features of the class of accounts which apply to all account instances.

2.2 Defining Classes

A class is a template from which objects are generated. Conceptually one defines in a class the commonalities (common attributes and services) of the instances of the class.

2.2.1 A Simple Account Class

Below we show a simple account class which specifies that instances of that class can be credited and debited and that the balance can be queried:

```
class Account
{
  public:
    void credit(double amount) {_balance += amount;}
    void debit(double amount) {_balance -= amount;}
    double balance() {return _balance;}

  private:
    double _balance;
};

#include <iostream>

using namespace std;

int main()
```

```
{
  Account acc1;
  cout << "balance = " << acc1.balance() << endl;

  acc1.credit(10000);
  cout << "balance = " << acc1.balance() << endl;

  char c; cin >> c;

  return 0;
}
```

2.2.2 Methods/Services

A Method interface or header specifies how users can interface with the method, i.e. how to call the method. The interface of a method is defined by the name of the method, the argument types and the type of the return value if any. The syntax for a method interface definition is

```
<ReturnType> <methodName>(Argument1Type arg1Name, Argumet2Type arg2Name, ...)
```

Method names

The method name should start with a lower case letter with word boundaries capitalized. Otherwise the same rules hold as those for all other identifier names (see section **??**).

2.2.3 Access levels

For the time being we shall look at only two access levels, `public` and `private`. The former specifies that a class member is publically accessible from anywhere. The latter specifies that a member can only be accessed from within the class itself. We shall return to the issue of access levels in section **??**.

In C++ an access level is specified for a blocks. The access level applies for all consecutive elements until it is changed.

```
class Account
{
  public:
     void debit(double amount);
     void credit(double amount);
     ...

  private:
    double balance;
};
```

Encapsulation

The `public` elements of a class are those which are meant for general use. They typically include the public services offered by instances of the class.

All the implementation details should be encapsulated within the class, i.e. should not be visible from outside the class. This ensures that users of the class don't develop dependencies on these implementation details.

Consider the following scenario: Assume you wrote a `Date` class with 3 public data fields:

```
class Date
{
  public:
    increment();
    addDays(int numDays);
    ...

  public:
    int day, month, year;
};
```

Of course, the `increment()` and `addDays(numDays)` methods are non-trivial, having to check for end-of-year, end-of-month and potentially for leap years and after using significant amounts of code of the form

```
for (Date d = d1; d<d2; d.increment())
   ...;
```

it may seem an excessive burden to do all this work for every increment. At some stage you might come to the conclusion that your for-father's idea of using the earth's rotation around its own axis, the moon and the earth's rotation around the sun as reference was not such a good idea after all and you may decide to work with days in some new units. Keeping still the earth's rotation around it's own axis, you may decide to store a date as an absolute date, choosing your favourite date (e.g. the first time you came home after midnight and youre parents actually accepted it) as day number 1 and counting days sequentially. After 2 years you'll reach day 730 and you were born on day -5475. Now, incrementing and decrementing dates or adding a number of days to a date and many other functions become trivial and the only time you are faced with any of the old complexity is when you have to convert between your snazy internal units and the archaic day, month, year units of your for-fathers.

If your data fields (day, month, year) had been declared private to start off with, you could simply make the implementation changes to your class and that would be it. If, on the other hand, they were declared public, they may be accessed from any other code distributed throughout your organization and you would have to search for sny such code and make the corresponding changes there.

Furthermore, you cannot guarantee the integrity of instances of your class if you give public access to your data fields. Anybody could go ahead and set the month to 27, bypassing any form of sanity checking you may havein your set-methods.

2.2.4 Constructors

Constructors are used to construct (create) instances of classes – i.e. objects. A constructor has the name of the class and no return value – not even `void`. You can give a constructor as many arguments as you like and you can write multiple constructors, each with different arguments. Below we added two constructors to our account class. The first one takes no arguments (the default constructor) and the second one takes the initial balance as argument.

```
class Account
{
  public:
    Account(): _balance(50) {}

    Account(double balance)
    {
      _balance = balance;
    }

    void credit(double amount) {_balance += amount;}
    void debit(double amount) {_balance -= amount;}
    double balance() {return _balance;}

  private:
    double _balance;
};

#include <iostream>

using namespace std;

int main()
{
  Account acc1;
  Account acc2(200);

  cout << "acc1 balance = " << acc1.balance() << endl;
  cout << "acc2 balance = " << acc2.balance() << endl;

  acc1.credit(400);
  acc1.debit(123.45);

  acc2.debit(456);

  cout << "balance = " << acc1.balance() << endl;
  cout << "acc2 balance = " << acc2.balance() << endl;

  char c; cin >> c;
```

```
  return 0;
}
```

Member Initialization via the Constructor's Parameter List

In the second constructor the private data field, `_balance`, was initialized within the body of the constructor. Alternatively we could have initioalized the private data field from the argument in the parameter list like this:

```
class Account
{
  public:
    ...

    Account(double balance): _balance(balance) {}
    ...
}
```

This is effectively done in the first, the default constructor. If a new account is created it is automatically credited by 50 whatevers.

Compiler-Generated Default Constructor

Recall that in our earlier `Account` class example (see section 2.2.1) we did not define a constructor at all. Yest, we were able to create accounts via

```
Account acc1;
```

How was this possible? The compiler wrote a default constructor – one which takes no arguments and has an empty body – for us. However, if we had added a constructor which takes arguments (e.g. the one which takes the initial balance as argument) without adding a default constructor, we can no longer create an account in this way:

```
class Account
{
  public:
    Account(double balance)
    {
      _balance = balance;
    }

    void credit(double amount) {_balance += amount;}
    void debit(double amount) {_balance -= amount;}
    double balance() {return _balance;}

  private:
    double _balance;
};
```

```
#include <iostream>

using namespace std;

int main()
{
  Account acc1;  // CAUSES COMPILER ERROR:
                 // Could not find Account::Account()
  return 0;
}
```

The reason for this is that C++ (like Java) creates a default constructor for you if and only if you did not define any constructor whatsoever for your class. In general you should write your own default constructor if you want a default constructor and disable it if you don't. If you want a class without any constructor whatsoever, you can define a `private` default constructor.

2.2.5 The OO Naming Convention

We have adhered to the OO naming convention which is very simple, clean and effective. It is used throughout the C++ community (except in the Microsoft community), in Java, Smalltalk and UML (the Unified Modeling Language) and simply states:

- Class names start with capital letter. Word boundaries are capitalized.
- Everything else (variable names, method names, function names and object names) start with lower case letter. Word boundaries are still capitalized.

2.3 Requesting services from objects

In section 2.2.4 we requested several services from different accpount instances (objects). In object-orientation one requests a service from an object by sending a message to it. One does not call a function – we shall see later that the client often does not know which actual function is called.

To send a message to an object one uses the member selection operator, the dot. For example, if we want to send a message to an account asking for the balance we can do it as follows:

```
double bal = acc1.balance();
```

Similarly, if we want to debit that same account, we can send a debit message:

```
double bal = acc1.debit(500);
```

2.4 The Life-Span of an Object on the Stack

So far we have created objects on the stack, i.e. they life in the same stack frame as other local variables. An object on the stack then also has a life span similar to that of local variables. It exists from where it is declared until the end of the block (the closing curly bracket) in which it is declared.

2.4.1 Destructors

Destructors are called when the object is deleted. Destructors are responsible for releasing any memory which the class grabbed from the heap as well as releasing other resources like closing files or network sockets. We shall see that when objects are created on the heap, the destructor has to be called manually. However, for objects which have been created on the stack, the destructor is called automatically.

Like constructors, the destructor also carries the name of the class, but it is preceded by a tilda and it may not have any arguments. The following example shows a destructor which simply laments the death of an object and illustrates how the destructor is called automatically as the object leaves its scope.

```
#include <iostream>

using namespace std;

class Account
{
  public:
    Account(): _balance(50) {}

    Account(double balance)
    {
      _balance = balance;
    }

    ~Account()
    {
      cout << "I, " << this << ", am destroyed." << endl;
    }

    void credit(double amount) {_balance += amount;}
    void debit(double amount) {_balance -= amount;}
    double balance() {return _balance;}

  private:
    double _balance;
};

void f()
{
```

```
  cout << "Entered f()." << endl;
  Account account;
  account.credit(1e6);
  cout << "About to leave f()" << endl;
}

int main()
{
  f();

  Account acc1;
  cout << "Created acc1, entering block." << endl;
  {
    Account acc2;
    cout << "Created acc2 in block,leaving block." << endl;
  }

  char c; cin >> c;

  return 0;
}
```

The output of the application looks something like this:

```
Entered f().
About to leave f()
I, 0012FF44, am destroyed.
Created acc1, entering block.
Created acc2 in block,leaving block.
I, 0012FF7C, am destroyed.
```

2.5 Splitting Headers and Implementation

Naturally one does not want to define all code (classes and functions) within a single source file. Furthermore, one does not want to recompile the entire source if one makes a modification to a specific area of the source.

To this end C++ enables you to define the header of a class which contains the method headers and the data fields of a class separate from the implementation code of these methods. For example, we can define the header of the verb+Account+ class in a file `Account.h` and its method implementations in a file called `Account.cpp`.

2.5.1 The Header File

The header file defines the method headers without the method bodies. Below we list a C++ header file for our account class:

```
#ifndef AccountH
#define AccountH
```

```
class Account
{
  public:
    Account();
    Account(double initialBalance);

    ~Account();

    void credit(double amount);
    void debit(double amount);

    double balance();

  private:
    double _balance;
};

#endif
```

Note that macros are used to avoid duplicate inclusion of the header file. A macro variable, `ACCOUNT_H` is defined the first time the file is read. The contents of the header file is only included if the variable has not yet been defined within the compilation process.

2.5.2 The Implementation File

The implementation file defines the method bodies of the methods whose header is specified in the header file. Since header and implementation files may contain multiple classes, the scope of the function must be specified. For example

```
Account::debit(double amount) {_balance -= amount;}
```

defines the body of the `debit` method of the `Account` class. Similarly,

```
Account::Account(): _balance(50) {}
```

defines the implementation of the default constructor. The complete implementation file is listed below:

```
#include "Account.h"
#include <iostream>

using namespace std;

Account::Account() : _balance(50) {}

Account::Account(double initialBalance)
  : _balance(initialBalance) {}
```

```
Account::~Account()
{
  cout << "I, " << this << ", am destroyed." << endl;
}

void Account::credit(double amount)
{
  _balance += amount;
}

void Account::debit(double amount)
{
  _balance -= amount;
}

double Account::balance() {return _balance;}
```

2.5.3 Including Header Files

C++ supports tw notations for including header files – they may be either specified within tag delimiters or within quotes. The former refers to header files which can be located along the system path while the latter notation is used for files which are located relative to the current directory (using relative paths) or at specified locations (using absolute paths).

Our main program includes the `Account.h` header file as well as some system header files:

```
#include <iostream>
#include "Account.h"

using namespace std;

int main()
{
  Account acc1;
  Account acc2(1000);

  acc1.credit(250);
  acc2.debit(100);

  cout << "balance of acc1: " << acc1.balance() << endl;
  cout << "balance of acc2: " << acc2.balance() << endl;

  char c; cin >> c;

  return 0;
}
```

2.6 Creating Objects on the Heap

So far we created objects on the stack. The objects were scoped to within a block, often a function body (e.g. `main`) and are stored within the stack frame of that object. Objects which have been declared on the stack are automatically deleted when they go out of scope.

However, the stack is a limited resource. Furthermore, one often requires objects to survive the scope in which they have been created. More often than not, one would want to create objects on the heap. This requires, however, that the developer has to control the memory management of the object and that introduces considerable risks in terms of potential memory losses as well as dangling pointers. The latter happens if an object is deleted while another part of the application still has a pointer to it. At a later stage this pointer could be used resulting in system corruption or system crash.

This problem is non-trivial – so much so that many commercial C++ applications end up with memory leaks. In fact, there is a market for C++ memory leak detector tools. From a more purist perspective the memory management is best tackled through a solid design (perhaps using UML) whereeach object has ultimately one owner who takes over the memory and pointer management for that object.

Objects are created on the heap via the new operator. They are deleted via the delete operator which ultimately calls the destructor. Below is an application which creates a collection of accounts on the heap, sends them through to a function and finally deletes them:

```
#include <iostream>
#include "Account.h"

using namespace std;

void debitServiceFees(Account* acc)
{
  acc->debit(45);
}

void run()
{
  int numAccounts = 5;
  Account** accounts = new Account*[numAccounts];

  for(int i=0; i<numAccounts; ++i)
    accounts[i] = new Account();

  for(int i=0; i<numAccounts; ++i)
    debitServiceFees(accounts[i]);

  // Because the accounts were created on the heap,
  // they have to be deleted manually:
  for(int i=0; i<numAccounts; ++i)
```

```
    delete accounts[i];
  delete[] accounts;
}

int main()
{
  run();

  char c; cin >> c;

  return 0;
}
```

2.7 Class Members

So far most of the members we added to our classes (methods and data fields) were instance members. For example, each instance of the account class had its own `_balance` and when we requested the `debit(double)`, `credit(double)` or the `balance()` query service, we sent the message to a specific instance (object):

```
account1.credit(550);

cout << account1.balance();
```

The only members which were class members thus far were the constructors. They had the name of the class and where implicitely class services. When we create an account we requested the service from the `Account` class and not a specific instance of the class – a particular account.

At times one wants to assign also other responsibilities to the class itself. For example, we may want to keep track of the number of instances of the account class. This responsibility does not fit naturally within a specific instance of the class. Instead, we might want to assign this responsibility to the class itself – after all, the class is responsible for creating instances (if one uses a factory pattern, the responsibility is naturally hosted by the factory).

In C++, as in Java, one uses the keyword `static` to specify that a specific element is a class member. For example, we may want to assign a datafield `_numInstances` to the class as well as a query service, `numInstances()`. We would declare both these members `static`, i.e. class members. Below we show the header file of a `Client` class which keeps track of the number of instances of that class:

```
// Client.h
//
#ifndef Client_H
#define Client_H
class Client
{
  public:
```

```
    Client(char* name);

    ~Client();

    static int numInstances();

  private:
    char* _name;
    static int _numInstances;
};
#endif
```

Nte that the class itself hosts a data field, `_numInstances`, which can be accessed from within the class scope or from the within the scope of any particular instance of the `Cient` class.

2.7.1 Constructors are Class Services

One uses the keyword `static` to specify class services and datafields of the class. However, constructors themselves are also class services – one asks the `Client` class for an instance, not a particular account (there may not even be an account instance around yet). So constructors are implicitly static methods.

In our constructor we added a line which increments the instance counter:

```
Client::Client(char* name)
{
  ++_numInstances;
  _name = name;
}
```

2.7.2 Destructors are Instance Services

Destructors are not class services. Their resposibility is to provide some finalization code (e.g. clean-up code) when an object (a particular instance) is deleted. The message is sent to a specific `Client`. Still, the instance counter should be decremented:

```
Client::~Client()
{
  --_numInstances;
}
```

Note that class members can be directly accessed from within instance members (like the destructor), but not vice versa.

2.7.3 Specifying Implementation Details of Class Members

The body of a static method is defined in the same way as the body of an instance method. Naturally, one cannot access instance members from a class service. After all

the scope is the class itself – which instance should it refer to. Below we show the trivial implementation of the `numInstances()` query method:

```
Client::numInstances()
{
  return _numInstances;
}
```

We still have to initialize the class member, `_numInstances` to zero. But were should we do that? We cannot initialize the data field in the constructor because then the field would be initialized every time an instance is created. We want to do once off initialization when the application is loaded. This initializtion statement is done at global scope via:

```
int Client::_numInstances = 0;
```

The full implementation file, `Client.cpp` is listed below:

```
// Client.cpp
//
#include "Client.h"

Client::Client(char* name)
{
  ++_numInstances;
  _name = name;
}

Client::~Client()
{
  --_numInstances;
}

int Client::numInstances()
{
  return _numInstances;
}

int Client::_numInstances = 0;
```

2.7.4 Using Class Members

Off course, we have used some of them already – the cnstructors. Below we show an example application which creates a few accounts, uses them and then deletes them. In between we ask the class now and then to report the number of instances which currently exist for the class:

```
// ClientTest.cpp.h
//
#include <iostream>
```

```
#include "Client.h"

using namespace std;

int main()
{
  Client* c1 = new Client("Peter");
  Client* c2 = new Client("Jill");

  cout << "numClients = " << Client::numInstances() << endl;

  delete c1;

  cout << "numClients = " << Client::numInstances() << endl;

  delete c2;

  cout << "numClients = " << Client::numInstances() << endl;

  char c; cin >> c;

  return 0;
}
```

2.8 Exercises

E:2.1 Write a Parabola class whose instances have specific values for the 2'nd, 1'st and 0'th order coefficients, a, b and c. The parabola should offer services for calculating the turning point and roots of the class as well as for calculating the function value for any given x-value. Separate the header from the implementation file.

E:2.2 Write a simple stack class (you may want to use a singly-linked list as underlying collection algorithm) which holds a collection of Parabolas. You should be able to push Parabolas onto the stack and be able to pop the off the stack. All objects should be created n the heap and you shuld take care that the application is save from memory leaks.

Chapter 3

Abstract Data Types: Rational Numbers

In this chapter we going to use C++ classes to build a complete abstract data type for rational numbers. An abstract data type (a class in C++) defines not only the data representation (e.g. that a rational number consists of two integral numbers, one of which is the numerator, the other the denominator), but also the operations which can be performed on this data type (e.g. how to add two rational numbers, how a rational number is to be displayed on an output stream or how it is incremented). By defining a class `Rational` we want to be able to extend the C++ programming environment in such a way that rational numbers can be used seemlessly – as if they were built in data types like `int` or `double`. We want to be able to write programs like

```
#include <iostream>
#include "Rational.h"

using namespace std;

int main()
{
  Rational r1, r2;

  cout << "Enter rational number (numerator denominator): r1 = ";
  cin  >> r1;
  cout << "r1 = ";
  cin  >> r2;

  Rational r3 = r1 + r2;
  Rational r4 = r1 * r2;

  cout << "r1 + r2 = " << r3 << endl;
  cout << "r1 * r2 = " << r4 << endl;

  return 0;
}
```

Adding two rational numbers is of course quite different from adding two integers or two floating point variables. For example, $\frac{2}{3} + \frac{1}{5}$ should yield $\frac{13}{15}$.

To achieve the above we must define a class with its data members (the numerator and the denominator). We must be able to create instances of the class `Rational` — i.e. to define variables of type `Rational`. This is achieved by the constructors of the class. Furthermore, we have to define the various operators like `+` or `*` as well as the output stream operator `<<` and the stream extraction operator `>>`. We can even define what the assignment operator `=` should do and how type conversion between rational numbers and say floating point numbers should be done.

3.1 Private and public data members

A class defines both, the data members of an abstract data type and the operations (methods) which can be performed on the data type. Following the encapsulation objective of object-oriented programming, we want to hide the underlying data structures from the user. This has several advantages. Firstly, the data fields are protected from direct manipulation by restricting the access to the predefined methods (and operators) of the class. This can ensure, among other things, data integrity. For example, if a user could manipulate the data fields of rational numbers directly, he would be able to set the denominator equal to zero, creating an invalid rational number. Data hiding is achieved by declaring the relevant data elements as private members of the class:

```
class Rational
{
    ...
  private:
    int _numer, _denom;
}
```

This ensures that the variables, `_numer` and `_denom`, can only be accessed from within the class — from within one of the methods (functions) of the class (we use the convention of trailing the variable name with an underscore for private data members of a class). Note that a class is defined by the keyword `class` followed by the name of the class and then, within curly brackets, we define the data elements and the functionality (class methods). In the above example we only defined the two data elements.

The second advantage of data hiding (encapsulation) is that at any stage the underlying data structures can be altered without affecting the program which use the class. For example, we would soon realize that the above definition would often result in the denominator overflowing. You might then want to change `int` to `long` or possibly even to a user-defined data type `VeryLongInt`. We can make these changes without altering the user-interface and hence the programs which use this class will not be affected (avoiding the traditional maintenance nightmare of searching through huge amount of code for occurrences of numerators and denominators and changing their data type).

Public data members can be accessed directly by the user and should hardly ever be used. If we had declared the numerator and the denominator public data members

```
class Rational
{
    ...
  public:
    int _numer, _denom;
}
```

then the user could manipulate them directly. He could, for example, set the denominator equal to zero:

```
void main()
{
  Rational<int> r(1,3);
  ...
  r._denom = 0;
}
```

3.2 Template classes

In chapter 1 we used function templates (generic or parametrized functions) to define a whole family of related functions (one corresponding to each required template type).

Analogously we can define classes on a template to create parametrized or generic data types. Vectors and matrices would typically be defined on a template so that we can use the same code to define a vector of integers, double precision numbers or even of a user defined type like a vector of rational numbers.

```
void main()
{
  Vector<double> vd1, vd2;
  Vector<Rational> vr1, vr2, vr3;
  ...
  Vector<double> vd3 = vd1 - vd2;
  ...
  vr1 = vr2 + vr3;
}
```

Similarly, we define rational numbers themselves on a template so that the user can create rational numbers where the numerators and denominators are stored as say either `long` or as the user-defined data type `VeryLongInt`. This is done as follows:

```
template <class T> class Rational
{
    ...
  private:
    T _numer, _denom;
}
...
void main()
{
  Rational<long> r1;
  Rational<VeryLongInt> r2;
  ...
}
```

3.3 Constructors

Constructors enable the user to create objects (instances) of a certain class. Constructors are methods (functions) which carry the name of the class and have no return value. They can be overloaded like any other function, i.e. we can have several constructors with the same function name, but with different number and/or types of arguments.

3.3.1 Specifying Constructors

Consider the following extract of the definition of the `Rational` class:

```
template <class T> class Rational
{
  public:
    Rational ();
    Rational (const Rational<T>& r);
    Rational (const T& numer, const T& denom);
    ...
  private:
    T _numer, _denom;
}
```

Note that all three constructors are defined in the public declaration block of the class. This ensures that they are all accessible from outside the class. The three constructors would be used in the following code:

```
void main()
{
  Rational<int>  r1, r2;     // using default constructor
  Rational<int>  r3(r1);     // using copy constructor
  Rational<long> r4(17,23);  // creating and initializing a rational number
  ...
}
```

The first of the three constructors has no arguments. It is called the *default constructor*. Should you omit to define any constructor, your friendly C++ compiler will write one for you. The compiler-supplied version will have an empty function body (it will do nothing except reserve space for the data members of the class). As soon as you do define any constructor, the compiler will no longer write a default constructor for you. If you want one (and, as we shall see later, you will need one if you want to allow the user to define arrays of your class) you will have to write it yourself.

3.3.2 Copy Constructors

The second constructor is called the *copy constructor*. It takes an instance of the same class (in our case a rational number) as argument and creates a copy of it. Should you omit to define a copy constructor, the compiler will write one for you. The compiler version creates a byte for byte copy of all the data members of the class. In our case that would achieve what we want to achieve and we could simply use the compiler supplied version (instead of defining our own). This is, however, not always the case. As soon as the class uses dynamic memory allocations we are forced to write our own version for the copy constructor (or face disaster). The vector and matrix classes will use dynamic memory allocation and will illustrate this point.

One should, however, be aware that the copy constructor is not only used in explicit variable declarations as above. Every time we pass a rational number by value (or when a function returns a rational number) the copy constructor is called implicitly. Consider, for example, the following function:

```
template<class T>
Rational<T> Square (const Rational<T> r)
{
  Rational<T> result = r*r;
  return result;
}
```

Here the copy constructor is used firstly to make a local copy of the variable, `r` and secondly, when the local variable `result` is returned by value (Note: never return a local variable by reference — you are returning a handle to an object which no longer exists outside the function).

3.3.3 Other Constructors

The third constructor takes two arguments, the numerator and the denominator for the rational number to be created. It allows us to create and initialize a rational number to a certain value with a single function call.

In the class definition given above we have only defined the function headers for the constructors. This part would typically be stored in a header file (e.g. RATIONAL.H). The implementation details (the function bodies) would then be stored in the accompanying CPP-file (RATIONAL.CPP). In this file we would define what the constructors (and the other member functions) actually do. Below we give the implementation details of the default constructor:

```
template<class T>
Rational<T>::Rational ()
{
  _numer = 0;
  _denom = 1;
}
```

Here the name of the function is `Rational()` and it has no arguments and no return value. The first line specifies that this function is defined on a template. We still have to specify that the function `Rational()` belongs to the class `Rational<T>`. This is done by preceding the function name with the class name followed by 2 colons. The function body is no different to any other function body. In this case it initializes the numerator and denominator to certain values. Note that the class variables are accessible by all class methods (functions).

3.3.4 Implementing Constructors

Instead of initializing class variables via assignment statements it is usually a good idea to initialize them via a parameter list. This is done as follows:

```
template<class T>
Rational<T>::Rational(): _numer(0), _denom(1) {}
```

After the function header we have a semicolon followed by the class member initializations. In this case both, the numerator and the denominator are initialized. The function body (between the curly brackets) is now empty. The advantage of this method

is that if the numerator and denominator are objects (instances of other classes) then the objects are created and initialized with a single function call to the copy constructor. If we look at the implementation of the previous paragraph (where the class variables were initialized via assignment statements), we see that we have two function calls, one to the default constructor (before the body of the function is executed) and a second to the assignment operator. This of course is less efficient then the parameter-list initialization. Furthermore, as we shall see later, class constants and reference variables have to be initialized in the parameter list.

The copy constructor should be defined such that it creates an exact copy of an object:

```
template<class T>
Rational<T>::Rational (const Rational& r):
    _numer(r._numer), _denom(r._denom) {}
```

Recall that should you omit to define a copy constructor, the compiler will write one for you which makes a byte-for-byte copy of the data members. In rare cases one wants to prevent the user from making a copy of instances of a certain class. An example could be a task scheduler. At any moment there should only exist a single task scheduler. In this case one can define a function header for the copy constructor, without supplying any implementation details (function body). This will cause a compile-time error if the user attempts to make a copy of an object of that particular class (e.g. task scheduler). Note that it is always desirable to have compile-time errors rather than run-time errors.

The implementation details of the third constructor are given below:

```
template <class T>
Rational<T>::Rational (const T& numer, const T& denom)
                 : _numer(numer), _denom(denom)
{
  #if EXCEPTIONHANDLING
    if (denom == 0)
      throw DivideByZero("Rational(numer,denom) => denom=0");
  #endif
  if (_denom < 0)
    {_numer = -_numer;  _denom = -_denom;}
  T common = gcd(_numer, _denom);
  if (common > 1)
    {_numer /= common;  _denom /= common;}
};
```

We first check that the user does not try to create a rational number with zero denominator. If that is the case we throw a `DivideByZero` exception. In chapter 7 we develop a unified exception-handling class hierarchy which is used throughout the entire C++ class library supplied with this book. We use a compiler directive to control conditional compilation. In the exception-handling library defined in the file `ErrHandl.h`, we define a global constant `EXCEPTIONHANDLING` which is set by the user to either 1 or 0 depending on whether he wants to use exception handling or not. Using exception-handling is of course safer, but it is also slower. Typically one would use exception handling during the development and debugging stages of software

development and one would switch it off once the code is known to be error-free. We use this exception handling mechanism here and the reader is not expected to understand it at this stage. It will be covered in detail in chapter 7.

We make certain that the denominator is always positive and we check whether the numerator and denominator have a common divisor which is greater than 1. If this is the case, both are divided by this common divisor. Hence, if we create a rational number $\frac{2}{-6}$, then it will be immediately simplified to $\frac{-1}{3}$. The common divisor is calculated by a private member function `gcd()` which is discussed in the following section.

3.4 Member functions

Class methods (member functions) describe the activities which can be performed by instances (objects) of that class. Furthermore, usually they allow controlled access to the data fields of the object. In C++ one usually distinguishes between member functions and class operators.

One further distinguishes between constant member functions which leave the members of an object unaltered, and non-constant member functions which alter members of an object. For constant member functions one appends the keyword `const` to the function header. This tells the compiler (and the user) that this particular member function leaves the state of the object unaltered and any attempt to change its state (change one of its data members) results in a compile-time error. Consider the following extract of the definition for the class `Rational`

```
template <class T> class Rational
{
  public:
    Rational ();
    Rational (const Rational<T>& r);
    Rational (const T& numer, const T& denom);

    T numerator   () const;
    T denominator () const;

    Rational<T>   power (const long)   const;
    double power (const double) const;
    ...
  private:
    T _numer, _denom;
}
```

Here we defined four constant member functions, two of which are query functions (they query the state of the object). The implementation section for the two query functions is trivial. For example

```
template <class T>
inline T Rational<T>::numerator () const {return _numer;};
```

The private data member is simply returned by value. Note that if we would return a private data member by reference we would give the user a handle to directly manipulate it, thereby destroying encapsulation. The function is very short and it would be wasteful

to incur the overheads of a function call. Hence we declare the function `inline`. Note that the `inline` specification trails the function header in the implementation section — this implementation detail is not specified in the class interface.

The implementation of the member function power taking a `double` as an argument is given below

```
template <class T>
inline double Rational<T>::power (const double y) const
{
  double x = (double)_numer/(double)_denom;
  return pow(x,y);
}
```

Note that this function leaves the data members of the class, `_numer` and `_denom`, unaltered. It converts the rational number into a `double` and then uses the `pow(double, double)` function of the C++ math library.

The implementation of the power method taking a `long` as an argument is a lot more involved and will be discussed in section 3.5.7.

3.4.1 Public versus private member functions

Analogous to data members, one can define member functions (and for that matter class operators) as either public or private. Those member functions which the user should be able to use to manipulate an object or to prompt an object to perform a certain action should be declared `public`. Those member functions which are only for internal usage should be defined in the private block. The member functions we have defined so far were all public member functions.

For internal usage we need a member function, `gcd`, which calculates the greatest common divisor between two integral numbers of the template type `T`. We declare this member function private:

```
template <class T> class Rational
{
  public:
    ...
  private:
    T _numer, _denom;
    T gcd (T a, T b) const;
}
```

The implementation details for this private class method are given below

```
template <class T>
T Rational<T>::gcd (T a, T b) const
{
  if (a < 0) a = -a;
  if (b < 0) b = -b;

  while (b > 0)
  {
    T m = a % b;
    a = b;
```

```
    b = m;
  }
  return a;
}
```

3.5 Class operators

In order to mimic the behavior of the built in data types like `int` or `double`, C++ allows one to define (overload) the built in operators for ones own data types. For example, if we have two rational numbers we would like to be able to use them in statements like

```
r3 = r2 + r1;
```

where `r1`, `r2` and `r3` are all of type `Rational<T>` where `T` could be, for example, `long`. Of course we have to define what both, the addition operator and the assignment operator do.

There are, however, some quite severe restrictions on operator overloading. Consider table 1.1 which lists the C++ operators in order of decreasing precedence. The precedence levels remains unaltered for the overloaded operators. Hence `*` always has precedence above `+`, irrespective of the data type (built-in or user-defined class). Table 1.1 also shows the syntax of the C++ operators. One cannot alter the syntax for overloaded operators.

3.5.1 Arithmetic operators

The result of the addition of two rational numbers is obtained by

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + cb}{bd} \tag{3.1}$$

followed by a division by the greatest common divisor of the numerator and the denominator of the result.

Operators can be overloaded like any other function — we can define various versions of an operator, each taking different types and/or different numbers of arguments. Below we define two addition operators for the class of rational numbers:

```
template <class T> class Rational
{
  public:
    ...
    Rational<T>  operator+  (const Rational<T>& r) const;
    Rational<T>  operator+  (const T& k)           const;
    ...
}
```

Note that the operators are really defined as functions with function name (`operator+`), arguments and a return value. From this one can see that C++ translates operators to function calls, i.e. `r1+r2` is implemented as the function call `r1.operator+(r2)`, i.e. the addition operator for object `r1` is called with argument `r2`. An analysis like this is very important if the objects on the two sides of a binary operator (e.g. assignment operator) are not of the same type. Consider for example

```
void main()
{
  int m=17;
  Rational<int> r1(1,3), r2;

  r2 = r1 + m;
  r2 = m + r1;
}
```

In the first of the two addition statements is interpreted as `r1.operator+(m)` and since `r1` is an instance of the class `Rational`, the function

```
Rational<int>::operator+(const T&)
```

is called. In the second addition statements the compiler searches for

```
int::operator+(Rational<int>)
```

But `int` is a built in data type (not even really a class) and no such operator is defined. In order to allow the user to form statements of the form of the second addition statement we have to define a global operator which takes as first argument a template type (say `T`) and as second argument a rational number `Rational<T>`. We discuss globally defined operators in the following section.

The function body of the two addition operator defined above is given by

```
template <class T>
inline Rational<T> Rational<T>::operator +
             (const Rational<T>& r) const
{
  return Rational<T>(_numer*r._denom
           + r._numer*_denom, _denom*r._denom);
}

template <class T>
inline Rational<T> Rational<T>::operator +
             (const T& k) const
{
  return Rational<T>(_numer+k*_denom,_denom);
}
```

Note that we make an explicit call to the constructor which takes the numerator and the denominator as an argument, creates a new rational number accordingly. The constructor, which is discussed in section 3.3 simplifies the resultant rational number (by dividing numerator and denominator with a greatest common divisor). The rational number created with this constructor call is returned by value.

3.5.2 Friends of a class and globally defined operators

In order to allow the user to use constructs like

```
r2 = 17 + r1;
```

we either have to define the addition operator

```
Rational<int> int::operator+(Rational<int>);
```

(which we cannot do because `int` is a built-in data type) or we have to define a global addition operator which takes as first argument an integer and as second argument a rational number. Again we define the operator on a template so that the user can use either `int`, `long`, or a user-defined type like `verylong`. C++ has built in automatic type conversion between `int` and `long` and the developer of the class `verylong` should supply automatic conversion between his class and the built-in data types, `int` and `long`. We shall show in section 3.5.4 how this is done.

Consider the following code extract

```
template <class T> class Rational
{
  public:
    ...
    friend Rational<T> operator+ (const T& k, const Rational<T>& r);
    friend Rational<T> operator/ (const T& k, const Rational<T>& r);
    ...
}

template <class T>
Rational<T> operator/ (const T& k, const Rational<T>& r)
{return Rational<T>(r._denom*k,r._numer);}

inline Rational<T> operator+ (const T& k, const Rational<T>& r)
{return r+k;}
```

Your friends know your private matters. Similarly friends of a class are functions or other classes which are not members of the class, but which have access to the private (and as we shall see later protected) members of the class.

In the above example we define two global operators as a friends of the class. Note that the operators are not members of the class and hence they cannot be declared class constants.

Consider now the implementations of the two operators. Since they are global functions — not members of the class — we do not have the class name followed by the scope resolution operator `::` in front of the operator name. The division operator uses directly the private data members `_numer` and `_denom` — he can do so since he is a friend of the class. Again we call the constructor to create a local rational number which is initialized to the inverse of `r` multiplied with `k` and we return this locally defined rational number (which has no name) by value.

Since addition is commutative, we simply reverse the statement and call the addition operator of the class which takes a variable of the template type as argument (see previous section). Both operators functions are declared `inline` so that we do not incur the overheads of a function call.

The addition operator does not use any of the private members of the class — we need not have defined it a friend of the class. We do so in order to have a neat interface for the `Rational` class, i.e. that the user who studies the class interface is aware of all operators relevant to the class.

3.5.3 Unary Operators and the this pointer

Unary operators are usually defined as class members without arguments — acting directly on the particular instance of the class. Consider for example the pre- and post-increment operators. Again, one should take care that the incrementing operators for the class `Rational` mimic the corresponding operators for the built-in data types. This is achieved as follows:

```
template <class T> class Rational
{
  public:
    ...
    Rational<T>& operator++ ();
    Rational<T>  operator++ (int);
    ...
}

template <class T>
inline Rational<T>& Rational<T>::operator++ ()
{*this+=1;  return *this;}

template <class T>
inline Rational<T> Rational<T>::operator++ (int)
{Rational<T> old(*this); *this+=1;  return old;}
```

Note that we have used a new keyword, `this`, which is a self-reference pointer — a pointer to the object itself. Note that although there is a unique copy of member variables for each instance of a class, all instances (objects) of that class share a single set of member functions. To indicate the specific instance whose variables should be used for a call to one of the member functions, the compiler adds an additional argument named `this` to the argument list of each member function. The second alteration made by the C++ compiler is to a `this->` prefix to all member variables and member functions. where the operator `->` is the member selection operator. Consequently the statement

```
_numer = r._numer;
```

in the code above is written as

```
this->_numer = r._numer;
```

Recall further that the unary operator `*` is the dereferencing operator. Since `this` is a pointer to the object itself, dereferencing this pointer via `*this` results in the object itself.

Now consider again the body of the pre-incrementing operator and in particular the statement `*this+=1;`. Here we add one to the object (the rational number) itself. Then we return the object itself by reference. This is perfectly legal since it is of the return type `Rational<T>` and it is not a local object. Note that we have to return the object itself in order to allow statements like `r2 = ++r1;`.

The post-increment operator makes a local copy of the number to be incremented, increments the number and returns the original rational number by value. If `r1`=$\frac{1}{3}$ and we perform `r2=++r1` then both `r1` and `r2` are equal to $\frac{4}{3}$. On the other hand, if `r1`=$\frac{1}{3}$ and we perform `r2=r1++` then `r1` is equal to $\frac{4}{3}$, but `r2` is equal to $\frac{1}{3}$.

The decrementing operators are defined analogously. We have also defined the unary + and unary - operators. These are very simple and the implementation details can be found in the class listing at the end of this chapter.

3.5.4 Type-Conversion Operators

We also want to be able to explicit and implicit type conversion between rational numbers and built-in types like `double`. For example, we would like to allow the user to make constructs like

```
void main()
{
  Rational<long> r(234,167);

  double x = r;

  y = sqrt((double)r);
}
```

In the assignment statement we make an implicit type conversion from `Rational<long>` to `double`. When calling the sqrt we made this type conversion explicitly. In both cases the following type-conversion operator was used:

```
template <class T> class Rational
{
  public:
    ...
    operator double() const;
}

template <class T>
inline Rational<T>::operator double () const
{ return (double)_numer/(double)_denom; }
```

We can define type-conversion operators to both, built-in data types and to other user-defined data types.

3.5.5 Relational Operators

In order to mimic the built in data types as closely as possible we also have to define the relational operators. All these operators leave the members of the class unaltered and hence they are all declared class constants.

```
template <class T> class Rational
{
  public:
    ...
    int operator== (const Rational<T>& r) const;
    int operator<  (const Rational<T>& r) const;

    int operator<  (const T& k) const;
}
```

```
template <class T>
inline int Rational<T>::operator==
        (const Rational<T>& r) const
{ return ((_numer == r._numer) && (_denom == r._denom));}

template <class T>
inline int Rational<T>::operator>
        (const Rational<T>& r) const
{ return ((double)*this > (double)r);}
```

The implementation of the `==` operator is pretty standard. For the `>` operator we first typecast the object itself to a `double` using our own type-conversion operator defined in the previous section and then we compare it to the argument — again type-converted to a double.

3.5.6 The Assignment Operator

As for the copy constructor, should you omit to define an assignment operator, your obliging C++ compiler will write one for you which, once again, makes a byte for byte copy of the data fields. This might or might not be what you want for your class. Usually, if you use dynamic memory allocation in your class you will be forced to write your own assignment operator. We shall use dynamic memory allocation for our vector and matrix classes.

Consider the following extract of the class interface where we define two assignment operators:

```
template <class T> class Rational
{
  public:
    ...
    Rational<T>& operator= (const Rational<T>& r);
    Rational<T>& operator= (const T& k);
    ...
}
```

The first of these operators takes a rational number as argument. It would be used in statements like

```
r1 = r2;
```

The implementation details are given below:

```
template <class T>
inline Rational<T>& Rational<T>::operator =
               (const Rational<T> r)
{
  _numer = r._numer; _denom = r._denom;
  return *this;
}
```

The function `operator=` returns the object itself by reference, not by value. Why would we want to do this. In the statement

```
r1 = r2;
```

we perform the function call `r1.operator=(r2)`, but we do not use the return value. Recall, however, that it is legal in C++ to concatenate assignments like `r1=r2=r3;` and that the assignment operator is the only binary operator which is right-associative. Hence the above statement is performed as `r1=(r2=r3)`. Now, if our assignment operator returned `void` (or anything else except the object itself), then we would assign `r1=void`. Returning the object itself solves this problem. It is of course more efficient to return the object by reference instead of by value (saving the overhead of making a copy) and it is also perfectly legal since we are not returning a reference to a local object.

The implementation of the second assignment operator is very similar and can be found in the class listing (see section 3.9).

3.5.7 The Power Method

One of the more interesting implementations is that of the `power` methods::

```
template <class T> class Rational
{
  public:
    ...
    Rational<T>   power (const long)   const;
    double power (const double) const;
}

template <class T>
Rational<T> Rational<T>::power (const long n) const
{
  if (n==0)
    return Rational<T>(1,1);
  else
    {
      Rational<T> result(*this);
      for (int i=2; i<=labs(n); i++)
        result *= *this;
      if (n<0)
      {
        T dummy = result._denom;
        result._denom = result._numer;
        result._numer = dummy;
      }
      return result;
    }
}

template <class T>
inline double Rational<T>::power (const double y) const
{return pow(*this,y);}
```

The first method which allows the user to evaluate an integral power (n'th power) of a rational number. For the case where n=0 we simply return the rational number $\frac{1}{1}$.

Otherwise we use the copy constructor to create a local copy called `result` and multiply the result $|n|-1$ times by the object itself. Finally, if n is negative we simply invert the result.

The second method calculates the real-valued power of a rational number. Here we use the `pow` function from the `math` library which takes two variables of type `double` as arguments. Hence, implicitly the type-conversion operator (which we defined in section 3.5.4) is used. The result is a local dummy variable which is returned by value.

3.6 Input/Output Stream Access

We want to define stream access for our abstract data type. This is achieved by overloading the stream-extraction and stream-output operators for our class. The stream classes are built in classes in C++ and we have to define these operators as global operators. In order that these methods have direct access to the private data members of our class (avoiding the overheads of calling the query functions of the class) we declare both operators as `friend`s of the class:

```
template <class T> class Rational
{
  public:
    ...
    friend ostream& operator<< (ostream& os,
                                const Rational<T>& r);
    friend istream& operator>> (istream& is,
                                      Rational<T>& r);
}

template <class T>
ostream& operator<< (ostream& os, const Rational<T>& r)
{
  T non_frac = r._numer/r._denom;
  T frac_numer = r._numer - non_frac*r._denom;
  if (non_frac != 0) os << non_frac;
  if (frac_numer != 0)
    os << " " << frac_numer << "/" << r._denom;
  return os;
}

template <class T>
istream& operator>> (istream& is, Rational<T>& r)
{
  T numer, denom;
  is >> numer >> denom;
  r = Rational<T>(numer,denom);
  return is;
}
```

The stream extraction operator is very simple. The user enters a rational number by entering the numerator and the denominator separated by standard C white spaces (blank, tab, newline, formfeed and carriage return). A rational numer is created accordingly (using the constructor) and `r` is set equal to this rational number. Note that we return the stream by reference in order to allow for concatenation of stream extractions:

```
cin >> r1 >> r2 >> r3;
```

For the stream output operator we first simplify the rational number by extracting the highest possible integral factor (e.g. $\frac{9}{2} \longrightarrow 4\frac{1}{2}$).

3.7 User's Guide to the Rational Class

In the following section we give a listing of the header file `RATIONAL.H` which defines the class interface. One should always consult the header file in order to see which operations are available to the user. You will find that all the standard mathematical operators (e.g. addition, multiplication, incrementing, assignment, ...) which are defined for built in data types (e.g. `float`) are also defined for rational numbers. In addition we support mixed-type arithmetic. For example, we allow the user to add an integer or a floating point number to a rational number, yielding a rational number or a floating point number respectively.

Furthermore, all the relational operators like `==` or `>=` are defined for rational numbers. Again we mode mixed-type comparisons, i.e. we can check whether a rational number is less than or equal to a floating point number.

Finally we supply two power methods. The one allows the user to evaluate an integral power of a rational number, yielding a rational number. The second allows the user t take a floating point power of a rational number, yielding a floating point number.

On the accompanying disk there is the following small demonstration program, `TRATIONL.CPP`, which illustrates the usage of the class `Rational`:

```
// FILE: TestRationl.CPP

#include <iostream>

#include "ErrHandl.h"

#include "Rational.h"
#include "Rational.cpp"

using namespace std;

int main()
{
  Rational<long> r1(3,9);       cout << "r1 = " << r1 << endl;
  Rational<long> r2(13,2);      cout << "r2 = " << r2 << endl;

  // Now using default constructor, addition and assigment operators
  Rational<long> r3 = r1 + r2;
  cout << "r3 = r1+r2 = " << r3 << endl; // output stream operator
  cout << "r1*r2 = " << r1*r2 << endl; // multiplying 2 rational nos
```

```
  cout << "r1/r2 = " << r1/r2 << endl;

  cout << "-r1 = " << -r1 << endl;               // unary minus
  cout << "r1>r2  -> " << (r1>r2)  << endl;      // relational operators
  cout << "r1<=r2 -> " << (r1<=r2) << endl;

  double x = r1;     // using type conversion operator (implicitely)
  cout.precision(15);
  cout << "x = " << x << endl;

  cout << "r1 + 2 = " << (r1 + 2.1) << endl; // using member operator
  cout << "2 + r1 = " << ((long)2 + r1) << endl; // using global operator
  cout << "r1++ = " << (r1++) << endl;     // post-incrementing
  cout << "r1 = " << r1 << endl;
  cout << "++r1 = " << (++r1) << endl;     // pre-incrementing
  cout << "r1^5  = " << r1.power((long)3) << endl;  // power method
  cout << "r1^-0.5 = " << r1.power((double)-0.5) << endl;

  cout << "r1 < 2.3 = " << (r1<2.3) << endl;

  Rational<long> r4 = -17;
  cout << "fabs(" << r4 << ") = " << fabs(r4) << endl;

  cout << "Type in rational number r=n/m: n m = ";
  try
  {
    cin >> r1;
    cout << "Read in rational number, r1 = " << r1 << endl;
  }
  catch (DivideByZero error)
  {
    cout << "*** ERROR ***: " << error.source << endl;
  }
  char k; cin >> k;

  return 0;
}
```

In the first two lines of the body of `main` we create, initialize and send to the standard output stream 2 rational numbers `r1`=$\frac{1}{3}$ and `r2`=$\frac{13}{2}$.

In the following statement we create a rational number `r3` (via the default constructor), use the addition operator to add two rational numbers and the assignment operator to assign `r3` to the result of the addition. We then demonstrate multiplication and division of two rational numbers, followed by the demonstration of the unary minus

and some relational operators.

The statement

```
double x = r1;
```

makes implicitly use of the type conversion operator discussed in section 3.5.4.

We continue to show when the member operator for addition and when the global operator is used. The difference between the post- and pre-incrementing operators is demonstrated in the following few statements followed by a demonstration of the two `power` methods. Finally we read in a rational number from the keyboard.

We recommend that you run this program in order to have a look at its output.

3.8 Listing of the Rational Class

In this section we give a complete listing of the class `Rational`. The header file `RATIONAL.H` defines the class interface. This is often the only readable file available to the user of your class.

```
// FILE: RATIONAL.H

#ifndef __RATIONAL_H
#define __RATIONAL_H

#include <math.h>
#include <stdexcept>
#include <iostream>

#include "ErrHandl.h"

using namespace std;

template <class T>
class Rational
{
  public:
    Rational ();
    Rational (const T& numer, const T& denom);
    Rational (const Rational<T>& r);
    Rational (const T& k);

    inline T numerator   () const;
    inline T denominator () const;

    Rational<T>   power (const long)   const;
    double power (const double) const;

    Rational<T>& operator= (const Rational<T> r);
    Rational<T>& operator= (const T& k);

    Rational<T>  operator+  (const Rational<T>& r) const;
    Rational<T>  operator-  (const Rational<T>& r) const;
```

```
  Rational<T>  operator*  (const Rational<T>& r) const;
  Rational<T>  operator/  (const Rational<T>& r) const;
  Rational<T>& operator+= (const Rational<T>& r);
  Rational<T>& operator-= (const Rational<T>& r);
  Rational<T>& operator*= (const Rational<T>& r);
  Rational<T>& operator/= (const Rational<T>& r);

  Rational<T>  operator+  (const T& k) const;
  Rational<T>  operator-  (const T& k) const;
  Rational<T>  operator*  (const T& k) const;
  Rational<T>  operator/  (const T& k) const;
  double       operator+  (const double& x) const;
  double       operator-  (const double& x) const;
  double       operator*  (const double& x) const;
  double       operator/  (const double& x) const;
  Rational<T>& operator+= (const T& k);
  Rational<T>& operator-= (const T& k);
  Rational<T>& operator*= (const T& k);
  Rational<T>& operator/= (const T& k);

  int operator== (const Rational<T>& r) const;
  int operator!= (const Rational<T>& r) const;
  int operator<  (const Rational<T>& r) const;
  int operator>  (const Rational<T>& r) const;
  int operator<= (const Rational<T>& r) const;
  int operator>= (const Rational<T>& r) const;

  int operator== (const double& x) const;
  int operator!= (const double& x) const;
  int operator<  (const double& x) const;
  int operator>  (const double& x) const;
  int operator<= (const double& x) const;
  int operator>= (const double& x) const;

  int operator== (const T& k) const;
  int operator!= (const T& k) const;
  int operator<  (const T& k) const;
  int operator>  (const T& k) const;
  int operator<= (const T& k) const;
  int operator>= (const T& k) const;

  Rational<T> operator+ () const;
  Rational<T> operator- () const;

  Rational<T>& operator++ ();
  Rational<T>& operator-- ();
  Rational<T>  operator++ (int);
  Rational<T>  operator-- (int); // decrement

  operator double () const;      // type conversion to double

private:
  T _numer, _denom;
```

```
    T gcd (T a, T b) const;
};


template <class T>
ostream& operator<< (ostream& os, const Rational<T>& r)
{
  T numer = r.numerator();
  T denom = r.denominator();
  T non_frac = numer/denom;
  T frac_numer = numer - non_frac*denom;
  if (non_frac != 0) os << non_frac;
  if (frac_numer != 0)
    if (frac_numer > 0)
      os << " " << frac_numer << "/" << denom;
    else
      os << " " << -frac_numer << "/" << denom;
  return os;
}

template <class T>
istream& operator>> (istream& is, Rational<T>& r)
{
  T numer, denom;
  is >> numer >> denom;
  r = Rational<T>(numer,denom);
  return is;
}

template <class T>
Rational<T> operator+ (const T& k, const Rational<T>& r)
{return r+k;}

template <class T>
Rational<T> operator- (const T& k, const Rational<T>& r)
{return (-r)+k;}

template <class T>
Rational<T> operator* (const T& k, const Rational<T>& r)
{return r*k;}

template <class T>
Rational<T> operator/ (const T& k, const Rational<T>& r)
{return Rational<T>(r.denominator()*k,r.numerator());}

template <class T>
Rational<T> fabs (const Rational<T>& r)
{return Rational<T>(abs(r.numerator()),r.denominator());}

#endif
```

The implementation details are defined in a separate file `RATIONAL.CPP` which is generally not available to users of your class.

```
// FILE: Rational.CPP
#include "Rational.h"
// CONSTRUCTORS
// ============

template <class T>
Rational<T>::Rational (): _numer(0), _denom(1) {};

template <class T>
Rational<T>::Rational (const T& numer,
                       const T& denom)
  : _numer(numer), _denom(denom)
{
#if EXCEPTIONHANDLING
  if (denom == 0)
    throw DivideByZero("Rational(numer,denom) => denom=0");
#endif
  if (_denom < 0)
    {_numer = -_numer;  _denom = -_denom;}

  T common = gcd(_numer, _denom);

  if (common > 1)
    {_numer /= common;  _denom /= common;}
};

template <class T>
Rational<T>::Rational (const Rational<T>& r)
  : _numer(r._numer), _denom(r._denom) {};

template <class T>
Rational<T>::Rational (const T& k)
  : _numer(k), _denom(1) {};

// QUERY FUNCTIONS
// ===============

template <class T>
inline T Rational<T>::numerator   () const
  {return _numer;};

template <class T>
inline T Rational<T>::denominator () const
  {return _denom;};

// OTHER CONSTANT MEMBER FUNCTIONS
// ===============================

template <class T>
Rational<T> Rational<T>::power (const long n) const
{
  if (n==0)
    return Rational<T>(1,1);
  else
```

```
    {
      Rational<T> result(*this);
      for (int i=2; i<=labs(n); i++)
        result *= *this;
      if (n<0)
      {
        T dummy = result._denom;
        result._denom = result._numer;
        result._numer = dummy;
      }
      return result;
    }
}

template <class T>
inline double Rational<T>::power (const double y) const
{return pow(*this,y);}

// ASSIGNMENT OPERATORS
// ====================

template <class T>
inline Rational<T>& Rational<T>::operator=  (const Rational<T> r)
{
  _numer = r._numer; _denom = r._denom;
  return *this;
}

template <class T>
inline Rational<T>& Rational<T>::operator= (const T& k)
{
  _numer = k; _denom = (T)1;
  return *this;
}

// ARITHMETIC OPERATORS
// ====================

template <class T>
inline Rational<T> Rational<T>::operator+
             (const Rational<T>& r) const
{
  return Rational<T>(_numer*r._denom
           + r._numer*_denom, _denom*r._denom);
}

template <class T>
inline Rational<T> Rational<T>::operator+
             (const T& k) const
{
  return Rational<T>(_numer+k*_denom,_denom);
}

template <class T>
```

```
inline Rational<T> Rational<T>::operator-
             (const Rational<T>& r) const
{ return *this + (-r); }

template <class T>
inline Rational<T> Rational<T>::operator-
             (const T& k) const
{
  return Rational<T>(_numer-k*_denom,_denom);
}

template <class T>
inline Rational<T> Rational<T>::operator*
        (const Rational<T>& r) const
{ return Rational<T>(_numer*r._numer, _denom*r._denom);}

template <class T>
inline Rational<T> Rational<T>::operator*
             (const T& k) const
{
  return Rational<T>(_numer*k,_denom);
}

template <class T>
inline Rational<T> Rational<T>::operator/
        (const Rational<T>& r) const
{ return Rational<T>(_numer*r._denom, _denom*r._numer);}

template <class T>
inline Rational<T>& Rational<T>::operator+=
     (const Rational<T>& r)
       {*this = *this + r; return *this;}

template <class T>
inline Rational<T>& Rational<T>::operator-=
     (const Rational<T>& r)
       {*this = *this - r; return *this;}

template <class T>
inline Rational<T>& Rational<T>::operator*=
     (const Rational<T>& r)
       {*this = *this * r; return *this;}

template <class T>
inline Rational<T>& Rational<T>::operator/=
     (const Rational<T>& r)
       {*this = *this / r; return *this;}

template <class T>
inline Rational<T>& Rational<T>::operator+=
     (const T& k)
       {*this = *this + k; return *this;}

template <class T>
```

```
inline Rational<T>& Rational<T>::operator-=
     (const T& k)
       {*this = *this - k; return *this;}

template <class T>
inline Rational<T>& Rational<T>::operator*=
     (const T& k)
       {*this = *this * k; return *this;}

template <class T>
inline Rational<T>& Rational<T>::operator/=
     (const T& k)
       {*this = *this / k; return *this;}

template <class T>
inline double Rational<T>::operator+  (const double& x) const
{return (double)*this+x;}

template <class T>
inline double Rational<T>::operator-  (const double& x) const
{return (double)*this-x;}
template <class T>

inline double Rational<T>::operator*  (const double& x) const
{return (double)*this * x;}
template <class T>

inline double Rational<T>::operator/  (const double& x) const
{return (double)*this/x;}


// UNARY OPERATORS
// ===============

template <class T>
inline Rational<T>& Rational<T>::operator++ ()
{*this+=1;  return *this;}

template <class T>
inline Rational<T> Rational<T>::operator++ (int)
{Rational<T> old(*this); *this+=1;  return old;}

template <class T>
inline Rational<T>& Rational<T>::operator-- ()
{*this-=1;  return *this;}

template <class T>
inline Rational<T> Rational<T>::operator-- (int)
{Rational<T> old(*this); *this-=1;  return old;}

template <class T>
inline Rational<T> Rational<T>::operator- () const
{
  Rational<T> r(*this);
```

```
  r._numer = -r._numer;
  return r;
}

template <class T>
inline Rational<T> Rational<T>::operator+ () const
{ return *this; }

// RELATIONAL OPERATORS
// ====================

template <class T>
inline int Rational<T>::operator==
        (const Rational<T>& r) const
{ return ((_numer == r._numer) && (_denom == r._denom));}

template <class T>
inline int Rational<T>::operator!=
        (const Rational<T>& r) const
{ return (!(*this == r)); }

template <class T>
inline int Rational<T>::operator>
        (const Rational<T>& r) const
{ return ((double)*this > (double)r);}

template <class T>
inline int Rational<T>::operator<
        (const Rational<T>& r) const
{ return ((double)*this < (double)r);}

template <class T>
inline int Rational<T>::operator>=
        (const Rational<T>& r) const
{ return ((double)*this >= (double)r);}

template <class T>
inline int Rational<T>::operator<=
        (const Rational<T>& r) const
{ return ((double)*this <= (double)r);}

template <class T>
inline int Rational<T>::operator==
        (const T& k) const
{ return (_numer == k);}

template <class T>
inline int Rational<T>::operator!=
        (const T& k) const
{ return (_numer != k);}

template <class T>
inline int Rational<T>::operator>
        (const T& k) const
```

```
{ return ((double)*this > k);}

template <class T>
inline int Rational<T>::operator<
        (const T& k) const
{ return ((double)*this < k);}

template <class T>
inline int Rational<T>::operator>=
        (const T& k) const
{ return ((double)*this >= k);}

template <class T>
inline int Rational<T>::operator<=
        (const T& k) const
{ return ((double)*this <= k);}

template <class T>
inline int Rational<T>::operator==
        (const double& x) const
{ return ((double)*this == x);}

template <class T>
inline int Rational<T>::operator!=
        (const double& x) const
{ return ((double)*this != x);}

template <class T>
inline int Rational<T>::operator>
        (const double& x) const
{ return ((double)*this > x);}

template <class T>
inline int Rational<T>::operator<
        (const double& x) const
{ return ((double)*this < x);}

template <class T>
inline int Rational<T>::operator>=
        (const double& x) const
{ return ((double)*this >= x);}

template <class T>
inline int Rational<T>::operator<=
        (const double& x) const
{ return ((double)*this <= x);}

// TYPE CONVERSIONS
// ================
template <class T>
inline Rational<T>::operator double () const
{ return (double)_numer/(double)_denom; }

// PRIVATE METHODS
```

```
// ===============

template <class T>
T Rational<T>::gcd (T a, T b) const
{
  if (a < 0) a = -a;
  if (b < 0) b = -b;

  while (b > 0)
  {
    T m = a % b;
    a = b;
    b = m;
  }
  return a;
}
```

3.9 Listing of the Exception Classes

```
#ifndef __ERRHANDL_H
#define __ERRHANDL_H

#define EXCEPTIONHANDLING 1

#include <stdexcept>

using namespace std;

// BASE CLASS FOR ALL EXCEPTION HANDLING
// =====================================
class Exception
{
  public:
    const char* source;
    Exception(): source(NULL) {};
    Exception(const char* src): source(src) {};
};

// BASE CLASS FOR ALL MATH ERRORS
// ==============================
class MathError: public Exception
{
  public:
    MathError() {};
    MathError(const char* src): Exception(src) {};
};

// NOW ALL THE SPECIAL INSTANCES OF MATH ERRORS
// ============================================
class DivideByZero: public MathError
{
  public:
    DivideByZero() {};
```

```
    DivideByZero(const char* src)
      : MathError(src) {};
};

class Overflow: public MathError
{
  public:
    Overflow() {};
    Overflow(const char* src)
      : MathError(src) {};
};

class IllegalOperation: public MathError
{
  public:
    IllegalOperation() {};
    IllegalOperation(const char* src)
      : MathError(src) {};
};

// BASE CLASS FOR ALL MEMORY ERRORS
// ================================
class MemoryError: public Exception
{
  public:
    MemoryError() {};
    MemoryError(const char* src): Exception(src) {};
};

// NOW ALL THE SPECIAL INSTANCES OF MEMORY ERRORS
// ==============================================
class OutOfMemory: public MemoryError
{
  public:
    OutOfMemory() {};
    OutOfMemory(const char* src)
      : MemoryError(src) {};
};

class Range: public MemoryError
{
  public:
    Range() {};
    Range(const char* src)
      : MemoryError(src) {};
};

// SOME OTHER EXCEPTIONS
// =====================
class IllegalArguments: public Exception
{
  public:
    IllegalArguments() {};
    IllegalArguments(const char* src): Exception(src) {};
```

```
};

class IllegalCall: public Exception
{
  public:
    IllegalCall() {};
    IllegalCall(const char* src): Exception(src) {};
};

#endif
```

3.10 Exercises

E:3.1 Write a function `mean` which uses the class `Rational` and which calculates the arithmetic mean of two rational numbers.

E:3.2 Write a complete class for complex numbers supporting the same functionality as the **Rational** class discussed in this chapter.

Chapter 4

Working at Different Levels of Abstraction

4.1 Introduction

In many ways the power of object-orientation comes from the support for abstraction, enabling developers to work at very abstract and general levels and facilitating the development of very generic solutions. Of course, at times one has to work at a very specific level and that too is cleanly facilitated through object-orientation.

There are mainly two mechanisms through which abstraction is achieved, super-classes and interfaces. The former has more an implementation focus while the latter is driven more by client needs.

4.2 Abstraction via Super-Classes

4.2.1 Thinking at Different Levels of Abstraction (Part 1)

We think quite naturally at different levels of abstraction. Consider the sentence Consider the following statements which become more and more concrete:

A person deposited money into the account.

A client deposited money into the account.

Mr P.J. Smith deposited money into the account.

It might be sufficient for you to know that somebody deposited money in the account. Alternatvely, you might want to know whether it is a cient or a friend who deposited the money and for bookkeeping purposes you would want to know which client deposited the money into the account. Here client is a special type of person, and P.J. Smith may be a special kind of client.

So we naturally introduce abstractions into our everyday language and it is really in the same spirit that object-oriented languages support abstraction too.

4.2.2 Specialization through Subclassing

Java allows you to define objects at various levels of abstraction. For example, You might view a particular manager as a `Manager`, an `Employee`, a `Person` or simply as an `Object`. This is a process from very concrete to more and more abstract.

The reverse direction can be viewed as specialization. A `Manager` *is a* special type of `Employee` which *is a* special type of `Person` which *is a* special type of `Object`.

Note that we can identify a *specialization relation* ship as a *is a special kind of* relationship and this should **always** be the criterion for deciding on subclassing.

Subclassing is achieved in via the following syntax:

```
class Manager: public Employee {...};

class Employee: public Person {...};

class Person {...};
```

Here `Manager` is a *subclass* of `Employee` which is a subclass of `Person`. Conversely, `Person` is the superclass of `Employee` which is the superclass of `Manager`. Note that we did not have to specify that `Person` extends Object.

I want to stress that the language relies on you to design your class hierarchies in such a way that the subclass *is a* specialization of the superclass. This is best illustrated by the following simple statements:

```
Employee* employee;

employee = new Manager();
```

We first declare a pointer to an `Employee`. This pointer can refer to any object which *is an* `Employee`. In the second statement we create a `Manager` and assign the `Employee`-pointer to refer to the manager. The compiler is quite happy because `Manager` is a subclass of `Employee` and he/she trusts you in having applied the *is a* criterion for subclassing.

Quite generally the following rule should hold. Everywhere where an `Employee` is required you should be able to supply any `Employee`, whether it is a vanilla `Employee` or a specialized `Employee` like `Manager` or `Programmer`. We can thus work with employees at various levels of abstraction. Let us have a look at the intestines of a simple `Person` class.

The Person Header

The `Person` class has no surprises with the exception of a `toString()` method which has been declared virtual. We shall discuss this method in section 4.2.7.

```
#ifndef Person_H
#define Person_H

#include <iostream>
```

```
#include <string>

using namespace std;

class Person
{
  public:
    Person(const string& name, const string& idNo);

    string name() const;

    string idNo() const;

    virtual string toString() const;

  //friends:
    friend ostream& operator<<(ostream& os, const Person&);

  private:
    string _name;
    string _idNo;
};
#endif
```

The Person Implementation

The implementation class is similarly straight-forward:

```
#include "Person.h"

#include <string.h>

Person::Person(const string& name, const string& idNo)
         : _name(name), _idNo(idNo) {}

string Person::name() const {return _name;}

string Person::idNo() const {return _idNo;}

string Person::toString() const
{
  return _name + ": " + _idNo;
}

// Implementation of friend functions:

ostream& operator<< (ostream& os, const Person& p)
{
  os << p._name << ": " << p._idNo;
  return os;
```

```
}
```

The Header for the Employee Sub-Class

The `Employee` class is declared a subclass of the `Person` class via

```
class Employee: public Person {...};
```

The subclass inherits all instance members – not the class (`static`) members from the superclass. Hence, we do not define data fields for the name and id number. Note that these fields were declared `private` in the `Person` class and that they thus cannot be directly accessed from the subclass. They are still inherited, though. If we want to acces them from the subclass, we have to use the same public interface as everybody else.

With the name and id number inherited, we only have to define a data field with corresponding query and set methods for the additional `salary` field:

```
#ifndef Employee1_H
#define Employee1_H

#include <iostream>

#include "Person.h"

using namespace std;

class Employee1: public Person
{
  public:
    Employee1(const string& name, const string& idNo,
              const double& salary);

    double salary() const;

    void salary(const double& newSalary);

    string toString() const;

  //friends:
    friend ostream& operator<< (ostream& os, const Employee1& e);

  private:
    double _salary;
};
#endif
```

The Implementation File for the Employee Sub-Class

When instantiating a class, one always instantiates all its superclasses. For example, everytime an instance of the `Employee` class is created, an instance of the `Person` class

which lives inside the `Employee` class is created too. Otherwise, where would a person get its name and id number from.

Objects are created via constructors. So, within the constructors of the subclass we have to somehow specify how the superclass is instantiated. This is done in C++ via the parameter list:

```
Employee::Employee(const string& name, const string& idNo,
                   const double& salary)
  :Person(name, idNo), _salary(salary){}
```

Should we omit to specify which constructor should be used to instantiate the superclass, the compiler will do the default this – i.e. try and instantiate the superclass via the default constructor. The complete listing of the `Employee` implementation is shown below:

```
#include "Employee1.h"
#include <stdlib.h>

Employee1::Employee1(const string& name, const string& idNo,
                     const double& salary)
  :Person(name, idNo), _salary(salary){}

double Employee1::salary() const {return _salary;}

void Employee1::salary(const double& newSalary)
{
  _salary = newSalary;
}

string Employee1::toString() const
{
  char* s_salary = new char[24];
  sprintf(s_salary, "%f", _salary);
  return Person::toString() + " (salary = " + s_salary + ")";
}

ostream& operator<< (ostream& os, const Employee1& e)
{
  os << e.name() << ": " << e.idNo() << "(salary = "
     << e._salary << ")";
  return os;
}
```

4.2.3 Inheritance

Instances of the subclass inherit the services offered by instances of the superclass. Hence, we can directly ask an employee for his/her name, even though the `Employee` class itself does not define a `name()` service:

```
#include <iostream>

#include "Person.h"
#include "Employee1.h"

using namespace std;

//--------------------------------------------------------------------------
int main()
{
  Person* p1 = new Person("Jack", "6811125657102");
  Employee1 * e1 = new Employee1("Jill", "6910825655105", 210000);

  cout << "p1's name: " << p1->name() << endl;
  cout << "e1's name: " << e1->name() << endl;
  cout << "e1's salary: " << e1->salary() << endl;

  Person* p2 = e1;

  cout << "p2's name: " << p2->name() << endl;

  char c; cin >> c;

  return 0;
}
```

The output of the program is quite predictably:

```
p1's name: Jack
e1's name: Jill
e1's salary: 210000
p2's name: Jill
```

4.2.4 Access Levels

C++ defines 3 access levels:

private: Elements which have been declared `private` can be accessed from within any instance of the class. Note that access is not restricted to within an object but to within any mamber of the same class.

public: Public members are generally accessible from anywhere where there is a handle (pointer or reference) available to the object hosting the member.

protected: Protected members can be accessed from within the class in which they are defined as well as from within subclasses of the class.

Should you Declare Data Fields Protected?

Declaring data fields protected gives subclasses direct access to them. This, is argued, is often desirable – after all, an instance of the subclass *is* also an instance of the superclass. It also provides the performance benefit that subclasses do not have to use the public access methods to access the data fields.

Howeer, `protected` data fields aren't (protected). Try and resist any warm feelings which the word, `protected`, may arouse in you. They are not protected from corruption by code defined in any of the subclasses because they bypass the access methods which were designed to provide controlled access to them. Furthermore, if a less respectable member of society ends up subclassing your class, they could make your so-called protected data field public. This is illustrated in the following block of code:

```
#include <iostream>

using namespace std;

class A
{
  public:
    A() : secretCode(1234) {}

  protected:
    int secretCode;

  friend ostream& operator<< (ostream& os, const A& a);
};

ostream& operator<< (ostream& os, const A& a)
{
  os << "I am an A. My secret is " << a.secretCode;

  return os;
}

class B: public A
{
  public:
    int& publishAsSecret()
    {
      return secretCode;
    }
};

int main()
{
  B* b = new B();
  A* a = b;
```

```
  cout << "a = " << (*a) << endl;

  int& theSecret = b->publishAsSecret();
  theSecret = 666;

  cout << "a = " << (*a) << endl;

  char c; cin >> c;

  return 0;
}
```

```
a = I am an A. My secret is 1234
a = I am an A. My secret is 666
```

Finally, declaring data dields protected may also introduce high maintenance costs. When the implementation of a class is changed (e.g. the data fields which store internally the state information of instances of that class), then one has to search for all subclasses in order to check if they require corresponding changes.

4.2.5 Public, Protected and Private Subclassing

The `Employee` class was publicly derived from the `Person` class:

```
class Employee: public Person {...};
```

During public subclassing the access levels of the superclass members are not modified:

- `public` members of the superclass are also `public` members of the subclass.
- `protected` members of the superclass are also `protected` members of the subclass.
- `private` members of the superclass remain private to that class.

C++ provides no way of making `private` or `protected` members of the superclass `public` members of the subclass – i.e. it is not possible to increase access to the superclass members during subclasses.

On the other hand, C++ does provide mechanisms to reduce the access level during subclassing. This is achieved via `protected` or `private` subclassing. The former reduces `public` members of the superclass to `protected` members of the subclass. The latter reduces the access level of all superclass members to `private` within the subclass.

For example, had we used `private` or `protected` subclassing when deriving the `Employee` class from the `Person` class, e.g.

```
class Employee: protected Person {...};
```

then the following code would result in compilation errors:

```
Employee employee("Peter", "7011115353100");

cout << employee.name(); // name() not accessible for employees.
```

Should you Actually use Private or Protected Subclassing?

Private and protected subclassing actually violate the core principal behind subclassing, *substitutability*: If anybody requires, say, a `Person` you should be able to provide them with any object which *is a* `Person`, i.e. with any instance of any subclass.

For example, I may use a primitive authentication routine which authenticates a `Person` by matching name and id number. The method header could look something like this:

```
void authenticate(const Person& person);
```

The method could use the `name()` and `idNo()` services to do the simple authentication. But `authenticate()` can be called providing anything which *is a* `Person` as argument, for example, an `Employee`. This can only work if the public members of `Person` are also public members of `Employee` – i.e. if `Employee` is publicly derives from `Person`.

Java and UML only cater for public subclassing –it is the only form of subclassing which is compatible with the logical framework of object-orientation.

4.2.6 Overriding Methods

Note that the `Person` class has a method, `toString`, allowing the user to obtain a string representation of the class. The `Employee` class would inherit this functionality. The developers have, however, chosen to supply a separate `toString()` method for the `Employee` class which overrides the `Person`'s `toString()` method. In its function body it first calls `Person`'s `toString()` method via `Person::toString()` and adds its own information to the string representation of the `Person` class.

4.2.7 Polymorphism and Virtual Methods

Polymorphism can be seen as message abstraction. You send an abstract message to an object and the recipient of the message interprets the message within its own context.

A very simple illustration of polymorphism at work is shown below:

```
Person* person = NULL;  // Initializing reference to null reference

person = new Person("Peter Smith", "631112 5225 087");
string str = person->toString();
cout << str << endl;

person = new Employee("Tandi Ndlovu", "732232 1121 087", 8000.00);
str = person->toString();
cout << str << endl;
```

In both cases we say `person->toString()`, but in the first case the recipient of the `toString()` message is a vanilla `Person`, while in the second case the recipient is a `Employee`. The recipient of the message interprets within its own context.

Thus the `Person::toString()` method will be called in the first instance, while the `Employee::toString()` method will be called in the second.

Note that a language which supports polymorphism must allow for dynamic binding (run-time linking). Take the above example. We could have asked the user at run-time whether he/she wants to create a `Person` or a `Employee`. Hence, only at run-time would it be known whether the `toString()` method of `Person` or `Employee` should be called. Run-time linking is requested in C++ by declaring a method `virtual`:

```
class Person
{
  public:
    ...

    virtual string toString() const;

    ...
};
```

This implies that any request for the `toString()` service offered by any type of person (e.g. an instance of the `Person` or the `Employee` class) which is sent through a pointer will be resolved at run-time:

```
#include <iostream.h>

#include "Person.h"
#include "Employee1.h"

#include <condefs.h>
USEUNIT("Person.cpp");
USEUNIT("Employee1.cpp");

void printPersonA(Person* person)
{
  cout << "printPersonA: " << person->toString() << endl;
}

void printPersonB(Person person)
{
  cout << "printPersonB: " << person.toString() << endl;
}

void main()
{
  Person* p1 = new Person("Jack", "6811125657102");
  Person* p2 = new Employee1("Jill", "6910825655105", 210000);

  printPersonA(p1);    printPersonA(p2);
  printPersonB(*p1);   printPersonB(*p2);
```

```
  cout << (*p1) << endl;
  cout << (*p2) << endl;

  char c; cin >> c;
}
```

4.2.8 Polymorphic Collections

Let us further illustrate polymorphism via a simple polymorphic collections. We shall use a primitive array for this. Collection classes are supplied with the *Standard Template Library (STL)* which is part of the ANSI/C++ specification. We shall discuss these classes in chapter 8.

```
#include <iostream>

#include "Person.h"
#include "Employee.h"

using namespace std;

//-------------------------------------------------------------------------
int main()
{
  const int numPersons = 3;
  Person** persons = new Person*[numPersons];
  persons[0] = new Person("Alfred", "588753287");
  persons[1] = new Employee("Pieter", "618732987", 196000);
  persons[2] = new Person("Petra", "8109894542424");

  for (int i=0; i<numPersons; ++i)
    cout << persons[i]->toString() << endl;

  for (int i=0; i<numPersons; ++i)
    delete persons[i];
  delete[] persons;

  char c; cin >> c;

  return 0;
}
```

In the above program we add persons and employees to our polymorphic collection of persons. Then we print out the string representation of each person. The `toString()` method is resolved polymorphically (via dynamic binding at run-time), i.e. at run-time the message is sent to the object and the object uses its own `toString()` method. If it does not have a `toString()` method the corresponding method of its superclass is used. The output of the program is listed below:

```
Alfred: 588753287
```

```
Pieter: 618732987 (salary = 196000.000000)
Petra: 8109894542424
```

4.3 Multiple Inheritance

At times a class can be naturally viewed as a specialization of more than one class. For example, en `EmployedClient` class could be viewed as a specialization of both, the `Employee` and the `Client` class (see figure 4.1).

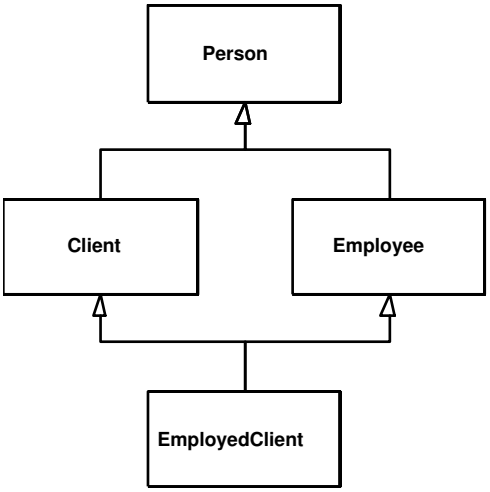

Figure 4.1: An employed client is a client and an employee.

Indeed, the *is a* test is satisfied along both legs, an `EmployedClient` is an `Employee` and *is a* client. Hence, if somebody expects an `Employee`

```
double restructureSalary(Employee& employee)
{
  ...
}
```

you should be able to pass any `Person` which *is an* `Employee`, i.e. an instance of `Employee` or an instance of the `EmployedClient` class.

Similarly, if somebody expects a `Client`

```
void debitMonthlyServiceFees(Client& client)
{
  ...
}
```

Unlike Java, C++ does support multiple inheritance of classes, i.e. a class can be a subclass of more than one class.

4.3.1 Simplistic Multiple Inheritance and its Problems

Let us simplistically define a `Client` class as follows:

```
#ifndef Client_H
#define Client_H

#include "Person.h"
#include "Account.h"

class Client1: public Person
{
  public:
    Client1(const string& name, const string& idNo, Account& account);

    Account& account() const;

  private:
    Account& _account;
};
#endif
```

We provide a trivial implementation of this class in:

```
#include "Client1.h"

Client1::Client1(const string& name, const string& idNo, Account& account)
         : Person(name, idNo), _account(account) {}

Account& Client1::account() const {return _account;}
```

Now we are in a position to define an `EmployedClient` class. Let us simply define a join class which adds no new services above those inherited from the `Employee` and `Client` superclasses:

```
#ifndef EmployedClient1_H
#define EmployedClient1_H

#include "Client1.h"
#include "Employee1.h"

class EmployedClient1: public Employee1, public Client1
{
  public:
    EmployedClient1(const string& name, const string& idNo,
                    const double& salary, Account& account);
};
#endif
```

We provide a trivial implementation of this class in:

```
#include "EmployedClient1.h"

EmployedClient1::EmployedClient1(const string& name, const string& idNo,
                                 const double& salary, Account& account)
  : Employee1(name, idNo, salary), Client1(name, idNo, account) {}
```

Now, when we use instances of the the `EmployedClient` class, we can directly ask them for their salary or their account:

```
#include <iostream>

#include "Account.h"
#include "Person.h"
#include "Client1.h"
#include "Employee1.h"
#include "Manager.h"
#include "EmployedClient1.h"

using namespace std;

//------------------------------------------------------------------------
int main()
{
  Account account;
  EmployedClient1* employedClient
    = new EmployedClient1("Peter", "546464389672", 190000, account);

  cout << employedClient->salary() << endl;
  Account& acc = employedClient->account();
  acc.credit(700);
  cout << acc.balance() << endl;

  Employee1* e = employedClient;

/*
  Person* p = employedClient;  // Compiler error: cannot convert
                               // EmployedClient* to Person*
*/

/*
  cout << employedClient->name(); // Compiler error: Person::name
                                  // and Person::name ambiguous
*/

  Manager* manager = new Manager("Charles", "5753223678", 300000, "BMW 740");

  Person* p = manager;

  cout << manager->name() << endl;
```

```
  char c; cin >> c;

  return 0;
}
```

There are, however, a number of severe problems:

- Firstly, our employed client is an `Employee`, but somehow he/she is not a Person??? This manifests itself in that we are able to have an `Employee` pointer point to an employed client (he/she is, after all, an `Employee`), but when we try and assign a `Person` pointer to the employed client we are told that it is not a `Person`.
- Secondly, though we can make use of the inherited `salary()` and `account()` services we somehow do not inherit `name()` and `idNo()`.

But things get even stranger. Let us define anothe subclass of `Employee`, a `Manager` class:

```
#ifndef Manager_H
#define Manager_H

#include "Employee1.h"

class Manager: public Employee1
{
  public:
    Manager(const string& name, const string& idNo, const double& salary,
            const string& statusSymbol);

    string statusSymbol();

  private:
    string _statusSymbol;
};
#endif
```

with implementation

```
#include "Manager.h"

Manager::Manager(const string& name, const string& idNo, const double& salary,
                 const string& statusSymbol)
  : Employee1(name, idNo, salary), _statusSymbol(statusSymbol) {}

string Manager::statusSymbol() {return _statusSymbol;}
```

If we create an instance of this class then we can assign a `Person` pointer to it and we can request the `name()` and `idNo()` services. So why could we not do it for `Employees`.

The answer lies in the way we derived `Employee1` and `Client1` from `Person`. The default thing C++ does is, from an object-oriented perspective, incorrect.

Recall that when we instantiate a class, all its superclasses are instantiated too. So, when we create an `Employee` we create a `Person`, and similarly, creating a `Manager_`creates an `Employee` which creates a `Person`. So far, no problem – and indeed, the `Manager` class behaved correctly.

Lets look whaĥappens in the `EmployedClient` case. Creating an `EmployedClient` creates both, an `Employee` as well as a `Client`. Each of these, in turn, creates a `Person`. Our `EmployedClient` class *is* thus two persons and not a single person. That is why we could not assign a `Person` pointer to the `EmployedClient`. Also, when we requested the `name()` service the compiler did not know which person's name he/she/it should choose.

This is, of course, nonsensical. An employed client is of course an employee, as well as a client and a single person. Any other scenario does not make logical sense. This is, however, not the way it is seen by default in C++. So how do we fix it?

4.3.2 Virtual Specialization

We want to specify that every client and every employee *is a single* `Person`. This is specified by inserting the `virtual` keyword in the inheritance link. This has to be done when deriving both, `Client` class

```
#ifndef Client_H
#define Client_H

#include "Person.h"
#include "Account.h"

class Client: public virtual Person
{
  public:
    Client(const string& name, const string& idNo, Account& account);

    Account& account() const;

  private:
    Account& _account;
};
#endif
```

and the employee class

```
#ifndef Employee_H
#define Employee_H

#include <iostream>

#include "Person.h"

using namespace std;
```

```
class Employee: public virtual Person
{
  public:
    Employee(const string& name, const string& idNo,
             const double& salary);

    double salary() const;

    void salary(const double& newSalary);

    string toString() const;

  //friends:
    friend ostream& operator<< (ostream& os, const Employee& e);

  private:
    double _salary;
};
#endif
```

from the verb+Person+ class. The implementations of these classes are identical to the implementations of the `Employee1` and `Client` classes shown above.

The header of the `EmployedClient` class could remain as above, though we do change the inheritance links to virtual for reasons discussed below.

```
#ifndef EmployedClient_H
#define EmployedClient_H

#include "Client.h"
#include "Employee.h"

class EmployedClient: public virtual Employee, public virtual Client
{
  public:
    EmployedClient(const string& name, const string& idNo,
                   const double& salary, Account& account);
};

#endif
```

We have to modify the implementation of the constructor, though. Not only do we have to specify how the direct subclasses are isntantiated, but we also have to specify how the one and only instance of the `Person` class is created:

```
#include "EmployedClient.h"

EmployedClient::EmployedClient(const string& name, const string& idNo,
                               const double& salary, Account& account)
  : Employee(name, idNo, salary), Client(name, idNo, account),
    Person(name, idNo) {}
```

Now things do behave correctly as illustrated in the code below:

```
#include <iostream>

#include "Account.h"
#include "Person.h"
#include "Client.h"
#include "Employee.h"
#include "EmployedClient.h"

using namespace std;

int main()
{
  Account account;
  EmployedClient* employedClient
    = new EmployedClient("Peter", "546464389672", 190000, account);

  cout << employedClient->salary() << endl;
  Account& acc = employedClient->account();
  acc.credit(700);
  cout << acc.balance() << endl;

  cout << employedClient->name() << endl;

  Person* p = employedClient;
  cout << p->idNo() << endl;

  char c; cin >> c;

  return 0;
}
```

Note that we had to realize the potential problem early, i.e. when we defined the `Client` and `Employee` subclasses – not when we did our multiple inheritance thing. At that stage it may not have dawned on us that we will one day have employed clients which are both employees and clients. One ust has to have this mystical feeling.

No, more seriously, specialization should, from an object-oriented perspective, allways be `virtual`. If you define a non-virtual subclass, then you are specifying what is known in UML as a `{disjoint}` constraint – i.e. that all subclasses of that class may not multply inherit from any classes within that particular class hierarchy.

4.4 Abstract Classes

So far all our classes were *concrete*, i.e. we were able to create instances of them. At times one wants to introduce classes without having the intention of actually instantiating them. The reasoning may be that one still wants to encapsulate some commonalities

accross classes within a single superclass, but that the class which encapsulates these commonalities by itself. Such a class would be an *abstract* class.

For example, all assets have, say an id. Specialized assets like properties, vehicles, government or corporate bonds, ..., can be created, but one would never create just simply an asset. Still, one would want to define a `Asset` class encapsulating all the commonalities among all assets (the asset id in our case). The asset class would be, conceptually, an abstract class.

4.4.1 How to Declare a Class with Concrete Methods Abstract

C++, unlike Java, does not have a mechanism for declaring a class directly abstract. What we need to achieve is that the class can only be instantiated from within the context of one of its subclasses. But this can be done simply by declaring its constructors with `protected` access level:

```
class Asset
{
  public:
    string id() const {return _id;}

  protected:
    Asset(const string& id) _id(id) {}

  private:
    string _id;
};
```

Abstract classes usually have at least one concrete subclass, i.e. one which can be instantiated. For example

```
class Bond: public Asset
{
  public:
    Bond(string id, Date maturity, double notional, ...);
    ...
}
```

4.4.2 Abstract Methods for Interface Definition

One of the main advantages of abstract classes is that they can incorporate interface specifications via abstract methods. Assume, for example, that all assets can be queried for their value on a specific date. How the value of an asset is calculated depends on the type of asset. We want to specify, however, that all assets have a service for querying the value, i.e. that they all must be able to process a `getValue(Date)` message. This is donme in C++ by adding an abstract method – i.e. a method for which there is no implementation – to the class:

```
virtual Tender value(const Date& date) = 0;
```

Of course, we do not know how to value an abstract asset. An abstract method is one which has no method body. A class which has one or more abstract methods is implicitly an abstract class and can hence not be instantiated.

4.4.3 SubClassing Abstract Classes

Subclasses of abstract classes must either comply to the specifications laid down in the abstract superclass (i.e. provide implementations for the abstract methods of the superclass) or they must be declared abstract themselves.

For example, subclasses of the abstract `Asset` class must provide an implementation for the `value(Date)` service. If they don't, the compiler will regard the subclass itself still as abstract. The reason for this is that the subclass does not fulfill the requirements laid down in the superclass. Since an abstract class cannot be instantiated, the compiler ensures that there will be no `Asset` objects (instances of the `Asset` class) which cannot be valued.

4.4.4 Abstract, Virtual and Non-Virtual Methods

There are three types of service specifications you can define in a class which may be subclassed:

- Pure specifications for services which must be supplied by concrete subclasses.
- Services with default implementations which may be overridden.
- Services with a fixed implementation which you should NOT override.

Each of these is dicussed seperately below.

Abstract or pure virtual methods

Abstract methods lay down specifications for services which must be supplied by concrete subclasses. No implementation (body) is supplied for abstract methods – the implementation must be supplied by the concrete subclasses. In this case one inherits only a requirements specification.

An abstract method must be declared `virtual` and is specified by an `= 0` behind the message header:

```
virtual Tender value(const Date& date) = 0;
```

Abstract or pure virtual methods are allways hosted by abstract classes.

Concrete virtual methods

Non-abstract virtual methods still require subclasses to provide the specified service, but if they do not supply their own implementation, a default implementation is provided by the superclass. This default implementation may, but need not, be overridden in the subclass.

A concrete virtual method is thus specified with an implementation. For example, an account class may have a virtual `debit` method specifying that all accounts must supply such a service and providing a default implementation in which the balance is simply adjusted with the debit amount.

Specialized subclasses like `CreditCard`, `ChequeAccount` or `HomeLoan` may override this method with their specialized implementation (for example, subtracting a transaction fee), but they may also choose to simply inherit the default implementation.

A concrete virtual method is simply a `virtual` method with a supplied function body:

```
virtual void debit(double amount) { _balance -= amount;}
```

Non-virtual methods

Non-virtual methods should be regarded as methods with a fixed implementation which subclasses should NOT override. C++ does not prevent the method from being overridden, but if you do, you may get unexpected behaviour. They are specified as methods without the `virtual` keyword.

For example, we might want to specify that the `authenticate` method should not be overridden in specialized `User` classes, i.e. no matter how special the user, when he is authenticated the block of code specified in the `User` class should be executed:

```
class User
{
  public:
    ...
    void authenticate() { ... }
    ...
};
```

Non-virtual methods are linked statically (i.e. at compile time). They do not support polymorphism and should only be used if you want to specify that that particular method may not be overridden. By default, you should define your methods `virtual`.

So, why is it so bad to override non-virtual methods. Simply because your system may exhibit strange, unexpected behaviour. The problem is illustrated in the following little program where we have an instance of a class `A` providing a virtual method `f()` and a non-virtual method, `g()`. Both methods are overridden in a subclass, `B`.

```
#include <iostream.h>

class A
{
  public:
    void virtual f()
    {
      cout << "Requested virtual service f() from an instance of A."
           << endl;
    }
```

```
    void g()
    {
      cout << "Requested non-virtual service g() from an instance of A."
           << endl;
    }
};

class B: public virtual A
{
  public:
    void virtual f()
    {
      cout << "Requested virtual service f() from an instance of B."
           << endl;
    }
    void g()
    {
      cout << "Requested non-virtual service g() from an instance of B."
           << endl;
    }
};

void main()
{
  B* b = new B();

  A* a = b;   // legal because a B is an A.
  // We have 2 pointers to the same object, an instance of B.

  cout << "Requesting virtual service f():" << endl;
  cout << "  We request the same service from the same object through" << endl;
  cout << "  two different message paths (two different pointers."
       << endl << endl;

  a->f();
  b->f();

  cout << endl << "Behaved as expected. Now requesting non-virtual service g():"
       << endl;
  cout << "  Once again we request the same service from the same object" << endl
       << "  through two different message paths (two different pointers."
       << endl << endl;
          date
  a->g();
  b->g();

  cout << endl << "This time the object behaved differently, depending on the"
       << endl
       << "message path (pointer) used to deliver the service request message."
```

```
      << endl;

  char c; cin >> c;
}
```

The poutput of the application is

```
Requesting virtual service f():
  We request the same service from the same object through
  two different message paths (two different pointers.

Requested virtual service f() from an instance of B.
Requested virtual service f() from an instance of B.

Behaved as expected. Now requesting non-virtual service g():
  Once again we request the same service from the same object
  through two different message paths (two different pointers.

Requested non-virtual service g() from an instance of A.
Requested non-virtual service g() from an instance of B.

This time the object behaved differently, depending on the
message path (pointer) used to deliver the service request message.
```

Note that we create an instance of `B` and have an `B*` and and `A*` pointing to this one and only object. We then request the service `f()` through both of these pointers, and, as expected, `B::f()` is called in both cases. After all, we only have a `B`.

We then request the non-virtual service `g()` through both of these pointers, but now `B`'s service `g()` is supplied when we request the service through the `B*` and A's `g()` when the service is requested through the `A*`. We never requested the service from an `A` though – there isn't even an `A`. We requested the same service from the same object and it does different things. This is not behaviour we want to see in clean object-oriented systems, and hence a non-virtual method should not be overridden.

4.5 C++ and Interfaces

Interfaces have become a very important concept in the software world. Not only do the Unified Modeling Language (UML) and Java directly support the concept of an interface, but a lot of the progress in the software development industry has been around interfaces.

For example, the entire CORBA specification published by the OMG is only an interface specification. So is the EJB framework published by Sun. Most new class libraries are designed around interfaces. These include the Standard Template Library of C++, the Java Messaging, Transaction and Services (JMS and JTS) and many others.

An interface simply specifies a set of services which classes which implement the interface must supply. As such an interface is like an abstract class with only abstract methods – and, yes, this is the way you implement interfaces in C++.

For example, you may specify that any class which claims to be a source of interest rates should implement an `InterestRateSource` interface which requires the implementing classes to supply two services:

```
class InterestRateSource
{
  public:

    virtual InterestRate getRate(const Date& date1, const Date& date2) const = 0;

    virtual double getDiscountFactor(const Date& date1, const Date& date2) const = 0;
};
```

A wide range of classes could potentially supply these services. Here we list a few in order to illustrate how totally different their implementations can be:

ReutersRateSource which reports the current market rates as perceived by Reuters.

ZeroCurve which stores spot interest rates for investements of different durations made at the spot date. This class would calculate interest rates over periods covered by the curve from the spot rates.

DealerWindow which pops up on the terminal of a dealer on the interest rate desk. He/she is expected to know the market and would be in a position to supply the interest rate for a supplied period.

BondPortfolio which contains a collection of priced Bonds. Interest rates for a given period can be calculated from the bond prices.

These different implementation have virtually nothing in common, except, that they all can provide the service. But why should you lock within your code into a specific service provider. If your code couples to the interface instead, you will be able to plug in any implementation of an interest rate source. You are thus decoupling from specific implementations and have the basics of plug-and-play programming.

4.6 Exercises

E:4.1 Define an interface `Asset` as an abstract class. One should be able to query the value of an asset for a supplied `Date`. Define the concrete assets, `Vehicle` and `Property`. When creating a `Vehicle` one should specify the purchase price and date as well as the write-off period in years. Assume the vehicle is written off linearly over the write-off period. Properties also have a purchase price and date. Assume, for sake of simplicity, that properties neither loose nor gain value. Define further a portfolio class to which you can add assets. The portfolio itself is, also an asset. Now create two portfolios, each with a property and a vehicle and add the one portfolio to the other (after all, a portfolio is an asset too). Query todays value of the portfolio. To help you work with dates look at the following functions defined in `time.h`:

difftime which calculates the time in milliseconds between two times.

mktime which makes a `time_t` from a `tm` structure.

localtime which creates a `time_t` representing the current date/time.

For simplicity, assume each year has 365 days.

Chapter 5

Classes using Dynamic Memory: Vectors and Matrices

5.1 Introduction

Among the most important data types in science and engineering are vectors and matrices. Traditionally one defined an array of floating point or complex numbers and separately a set of subroutines or functions which manipulate these arrays. In this chapter we build complete abstract data types for vectors and matrices which allow us to simplify our programs considerably. Consider, for example, the following code extract:

```
void main()
{
  Matrix<double> A(3,4), B(4,3), C(3,3);  // defining 3 matrices of differing dimensions
  Vector<double> v1(3);
  ...
  Vector<double> v2 = (A*B-3*C)*v1;
  cout << "resultant vector = " << v2 << endl;
}
```

will be legal code once we have defined our vector and matrix classes. We shall also see how the various data types interlink. For example, we shall use our vector class to define vectors of rational numbers which we can add, multiply, ...

Naturally, we would like our vectors and matrices to use dynamic memory. We want to be able to define vectors and matrices of any size (within the hardware limits) and we want to be able to release this memory as soon as we no longer need these data structures.

5.2 The Vector Class

5.2.1 Introduction

We shall develop a vector class which supports (among other things) all the standard vector operations like vector addition, dot products, Kronecker and Hadamard products,

normalization and the calculation of various vector norms.

5.2.2 Static Members of a Class

Usually we want to encapsulate variables within an object. For example, each instance of the class `Rational` has a numerator and a denominator. These data members are encapsulated within an object by declaring them private data members.

Sometimes we want to have all instances of a certain class share a single class constant or class variable. We want to encapsulate such data members within a class — we do not want to use global variables or constants.

For example, one might want to know how many instances of a certain class (e.g. the number of clients a bank) exist at certain instant during program execution. In such a case one could define a static class variable:

```
class Client
{
  public:
    ...
    static long numberofinstances() const;

  private:
    static long _numberofinstances;
}
```

Here `_numberofinstances` is a static class variable — there exists only one such variable shared by all instances of the class. Of course we cannot initialize static variables in the constructors of the class — the variable would be initialized every time we create a new instance of that class. The initialization can be in the implementation file, say VECTOR.CPP, as a free-standing declaration

```
template<class T> const int Vector<T>::_defaultprecision = 4;
template<class T> const int Vector<T>::_defaultwidth     = 7;
```

If we had declared the static variable public, we could have accessed it either via an instance of the class or directly, i.e. the following code would be legal:

```
void main()
{
  cout << "Currently there are " << Vector<double>::_numberofinstances
       << "instances of the Vector<double> class." << endl;

  Vector<double> v1;

  cout << "Currently there are " << v1._numberofinstances
       << "instances of the Vector<double> class." << endl;
}
```

For obvious reasons of encapsulation we have declared this static variable private instead — otherwise we could not really be sure that it holds the value of the number of instances of that class (user-code could have modified it directly). Instead we give access via a public static function `numberofinstances()`. Again, we can call this function even if there does not exist an instance of the class as is shown in the code below

```
void main()
{
  cout << "Currently there are " << Vector<double>::numberofinstances()
       << "instances of the Vector<double> class." << endl;

  Vector<double> v1;

  cout << "Currently there are " << v1.numberofinstances()
       << "instances of the Vector<double> class." << endl;
}
```

In the case of the vector and especially the matrix class, we want the user to be able to control the precision and width with which the elements are written onto an output stream. We thus include methods which allow the user to query or set the output-precision and output-width. We define class constants `_defaultprecision` and `_defaultwidth` for the default output format. If we had not declared these class constants as static members of the class, every instance of that class would have a copy of these constants and we would initialize these class constants via the parameter list. It is, however, very wasteful that each instance of the class contains a copy of the identical constants. Instead we declare them as static class constants

```
template <class T> class Vector
{
  public:
    ...
    int  outputprecision    () const;  // returns output precision
    int  outputwidth        () const;  // returns output width
    void setoutputprecision (const int precisn);
    void setoutputwidth     (const int wdth);

  private:
    ...
    T* _v;
    long _length;
    int _precision, _width;
    static const int _defaultprecision;
    static const int _defaultwidth;
}
```

and we initialize them with free-standing declarations in the file VECTOR.CPP:

```
//=========================================================
// Initializing Static Class Constants
//---------------------------------
template<class T> const int Vector<T>::_defaultprecision = 4;
template<class T> const int Vector<T>::_defaultwidth     = 7;
//=========================================================
// CONSTRUCTORS and DESTRUCTOR
//---------------------------
template <class T> Vector<T>::Vector ()
  : _v(NULL), _length(0),
    _precision(_defaultprecision), _width(_defaultwidth) {}
...
```

Note that each objects own output precision and width (which can be altered by the user) are initialized to their default values in the parameter list of the constructors.

Any member of a class can be declared a static member, i.e. we can have static data members, static member functions and even static member objects.

5.2.3 Constructors, Destructor, Assignment Operator

For classes using dynamic memory one must be particularly careful when developing constructors and assignment operators. One can no longer rely on the compiler to write these functions — the compiler supplied versions will lead to catastrophe.

Furthermore, once an instance of such a class goes out of scope, the memory which has been claimed by the constructors and assignment operators must be released. This is done by defining a destructor for the class. Again, the compiler supplied version will only lead to grief.

In the previous section we saw that the default constructor of the vector class initializes the

Constructors

From the excerpt of the class interface shown in section 5.2.2 we see that the vector class has a pointer `v` to the template type and a long integer `_length` as private variables of the class. We also showed in the previous section that the default constructor initializes the pointer variable to the `NULL` pointer and `_length` to zero. Below we give the implementation details for the remaining two constructors of the vector class:

```
template <class T> Vector<T>::Vector (const Vector<T>& v)
  : _v(NULL), _length(0),_precision(_defaultprecision),
    _width(_defaultwidth)  {*this=v;}

template <class T> Vector<T>::Vector (const long lngth)
  : _precision(_defaultprecision), _width(_defaultwidth)
{
  #if EXCEPTIONHANDLING
    if (lngth<0) throw Exception("Vector(lngth) => lngth<0");
  #endif
  _v = new T[lngth];
  _length=lngth;
}
```

Note that the copy constructor is used every time you pass or return a vector by value. Should you omit to define one yourself, your C++ compiler will write one for you. The compiler-supplied version makes a byte-for-byte copy of the data members of the class. Thus the value of the pointer variable would simply be copied, without reserving separate memory for the new vector. Both vectors would use the same memory area with obvious catastrophical consequences. The reservation of new memory and the copying of the vector elements is done correctly in the assignment operator defined in the following section. The copy constructor simply calls our assignment operator.

In the final constructor we first check that it is called with legal arguments, i.e. that the requested length of the vector is equal to or greater than zero. If that is not the case we throw an exception. This sanity checking is only done if the global variable

EXCEPTIONHANDLING is on (=1). We use conditional compilation for this purpose. (See chapter 7 for a detailed discussion of exception handling).

We then reserve enough memory of lngth variables of the template type T and set the private variable _length to the length of the vector.

The Assignment Operator

As for the copy constructor, if you omit to define an assignment operator for your class, your compiler will write one for you. The compiler-version will simply make a byte-for-byte copy of the vector fields and thus of the pointers _v. Consequently the two vectors would point to the same memory area and setting an element of the one vector would also change the relevant element of the second vector. If one vector goes out of scope, its memory is released, and the second vector will use illegal memory.

Our vector class allows assigning one vector to another of different length. In this case we first release the memory of the first vector, grab enough memory from the heap for a copy of the second vector and only then we copy the vector elements. The resizing of the vector is done by a private member function resize

```
template <class T> void Vector<T>::resize (const long lngth)
{
  #if EXCEPTIONHANDLING
    if (lngth<0)
      throw IllegalArguments("Vector::resize(lngth) => lngth<0");
  #endif
  delete[] _v;
  if (lngth>0)
   {
     _v = new T[lngth];
     _length = lngth;
   }
  else
  {
   _v = NULL;
   _length = 0;
  }
}
```

If the requested vector length is negative we throw and exception. Otherwise we release the heap-memory currently occupied by the vector and reserve enough memory for a vector of length lngth.

Below we give the implementation details of our assignment operator:

```
template <class T>      // assignment operator
Vector<T>& Vector<T>::operator= (const Vector<T>& v)
{
  if (this != &M)
  {
    if (_length != v._length)
      resize(v._length);
    for (long i=0; i<_length; i++)
      _v[i] = v._v[i];
```

```
  }
  return *this;
}
```

It is critical to check for self-assignment. Self-assignment can be more subtle than simply `v1=v1`. For example, `v2` could be a reference variable which has been set equal to `v1` and at a later stage one could have a camouflaged self-assignment of the for `v1=v2`. The check for self-assignment is not only recommended for reasons of efficiency. If we wouldn't check whether the two vectors are of the same length, we would first release the memory, of the vector, then grab new memory and finally copy the value of the elements from memory which no longer belongs to the vector and which quite possibly is used elsewhere. In the case of self-assignment we simply return the object itself by reference (allowing concatenations of assignment statements). If the lengths of the two vectors differ, we resize the first vector and then we copy the actual vector elements.

The Destructor

The destructor of a class is called automatically as soon as an instance of that class goes out of scope. For example, if one declares a vector locally in a function, then as soon as one returns from the function, the destructor is called for that object. The function of a destructor is to perform any clean-up operation like releasing memory which was grabbed by the object from the heap.

In the constructors and the assignment operator we reserve memory from the heap for our vector. As soon as an instance of `Vector` goes out of scope this memory should be released again. If one does not define a destructor, your overly-eager C++ compiler will go ahead and write one with an empty body for you. This is of course no good if your class allocates memory dynamically from the heap. In this case you will have to supply your own constructor. In the body for the destructor for the vector class we simply release the memory allocated in the constructor or the assignment operator

```
inline template <class T> Vector<T>::~Vector ()
{delete[] _v;}
```

Note that destructors are methods which have no arguments and no return values and which carry the name of the class preceded by a tilda.

Destructors can be called manually:

```
Vector<double> v(100);
...
delete v;
```

5.2.4 Element Access

In standard C and C++ arrays, elements of an array are accessed via the subscription operator `[]`. We define the subscription operator for the vector class to do the same:

```
template <class T>
inline T& Vector<T>::operator[] (const long i) const
{
  #if EXCEPTIONHANDLING
```

```
    if ((i<0) || (i>=_length))
      throw Range("Vector::operator[]:");
  #endif
  return _v[i];
}
```

If exception handling is switched on (see chapter 7 for a detailed discussion on exception handling) we perform range checking and throw the relevant exception if the check fails. Otherwise we return the relevant element by reference. This is a very important point. If we had returned the element by value we would not be able to assign an element to a value, i.e. the following code would not compile

```
Vector<double> v(3);
...
double x = v[0];    // no problem
v[0] = 1.3;         // compiler error if not returned by reference
```

If the template type is a huge data type (we could, for example, have a vector of matrices) returning the element by reference has the further advantage of efficiency.

5.2.5 User's Guide to the Vector Class

Below we discuss some useful methods we included in the vector class. For most of these methods we do not discuss the implementation details. These can be found in section 5.2.7 where we give a complete listing of the vector class.

Arithmetic Operations

Vector addition and subtraction, together with the related operators `+=`, `-=`, are defined in a standard fashion. We use the multiplication operator `*` to denote the vector dot product (scalar or inner product)

$$\texttt{a*b} \Longrightarrow \mathbf{a} \cdot \mathbf{b} := \sum_i a_i b_i \tag{5.1}$$

and the modulo or remainder operator `%` to denote the Kronecker product

$$\texttt{a\%b} \Longrightarrow \mathbf{A} = \mathbf{a} \otimes \mathbf{b} := \{a_i b_j\} \tag{5.2}$$

— the result of the Kronecker product between two vectors is a matrix whose ij'th element is equal to $a_i b_j$. Recall that C++ restricts us to overloading only the built-in operators (used for the standard data types like `float` or `int`) and that we cannot alter the precedence level or the syntax of these operators. This does not leave us much choice when searching for an operator for the Kronecker product. The only operator available with the correct syntax and precedence level (same precedence level as multiplication) is the modulo operator — see table 1.1.

For element-for-element multiplication of two vectors we use the function `hadamardproduct`

$$\texttt{a.hadamardproduct(b)} \Longrightarrow \mathbf{c} = \mathbf{a} \otimes \mathbf{b} := \{a_i b_i\} \tag{5.3}$$

We have no suitable operators left to choose from and hence we use a member function instead. We have defined the arithmetic operations such that we maximize consistence between the vector and matrix classes.

We also support mixed-type arithmetic, allowing the user to add a scalar to a vector or a vector to a scalar. The same holds for subtraction, multiplication or division of a vector by a scalar.

Vector functions

Our class supports vector functions. For example `v1.func(exp)` returns a vector where each element is the exponent of the relevant element in `v1`. we define two version of the func method

```
template <class T> class Vector
{
  public:
    ...
    Vector<T> func (T (*f)(const T&)) const;  // returns f(vec)
    Vector<T> func (T (*f)(T)) const;  // returns f(vec)
}
```

one of which the argument of the function is declared constant. The implementation details are quite simple:

```
template <class T>            // returns f(vec)
Vector<T> Vector<T>::func (T (*f)(const T&)) const
{
  Vector<T> result(_length);
  for (long i=0; i<_length; i++)
    result._v[i] = f(_v[i]);
  return result;
}
```

5.2.6 Vector Norms, Normalization, Mean, ...

We define two methods `norm`, one for the euclidean norm

$$||\mathbf{x}|| := \left(\sum_i |x_i|^2 \right)^{\frac{1}{2}} \tag{5.4}$$

(which returns the length of a vector in euclidean space) and a generalized norm called the p-norm

$$||\mathbf{x}||_p := \left(\sum_i |x_i|^p \right)^{\frac{1}{p}} \tag{5.5}$$

Naturally the euclidean norm is a special case of the p-norm. The reason for supplying two methods, instead of a single method with default value 2 for p is that we can implement the euclidean norm with more efficiency than the p-norm with $p = 2$. Below we give the signatures of the two `norm` methods

```
template <class T> class Vector
{
  public:
    ...
    T    norm      () const;       // Euclidean norm
    T    norm      (const double& p) const;  // p-norm => (sum(|v_i|^p))^(1/p)
}
```

The implementations are very simple and can be found in the program listing in section 5.2.7. The method `normalize()` divides the vector elements by the euclidean norm of the vector resulting in a normalized vector — a vector with unit length.

The arithmetic mean of the elements of a vector

$$\overline{\mathbf{x}} := \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5.6}$$

is returned by the method `mean()` and the sum of all vector elements by the method `sumels()`. The largest, smallest, largest absolute and smallest absolute element is returned by the methods `maxel()`, `minel()`, `maxabsel()` and `minabsel()` respectively. Alternatively one can obtain the index of the largest, smallest, largest absolute and smallest absolute element via the methods `maxindx()`, `minindx()`, `maxabsindx()` and `minabsindx()` respectively.

Stream Access

The input and output stream operators which take a stream as first argument and a vector as second argument are declared friends of the class — hence they have direct access to the private members of the vector class. The function uses a chaotic mapping with uniform density.

The method `insert(const Vector<T>& v, const long before)` returns a vector where the vector `v` is inserted before element no `before`. Consider the statement

```
Vector<double> v3 = v1.insert(v2,n);
```

If `n=0` then v2 is inserted in front of v2. If `n=v1.length()`, then `v2` is appended to `v1`. The member function

```
Vector<T> Vector<T>::slice (const long istart, const long noelements,
                            const long stride=1);
```

extracts `noelements` elements from the vector, starting from element no `istart`, selecting every `stride` elements. If the argument `stride` is not supplied, it is set equal to its default value 1. Consider for example the vector $\mathbf{v} = [0123\ldots1920]$. Then `v.slice(5,6)` returns the vector [5678910] and `v.slice(3,4,2)` returns the vector [3579].

As we discussed before, the member function `resize(const long lngth)` resizes the vector to the required length. Note that all elements will be lost. The main purpose for `resize()` was to allow the user to create an array of vectors using the default constructor and then to resize each vector to the required size. This will be discussed further in the matrix class. If the user wants to resize a vector without losing the elements he could for example use the following statement

```
v1 = v1.insert(Vector<double>(3),v1.length());
```

This will increase the size of `v1`by 3 elements, keeping the the old elements of `v1` unchanged.

Finally we supply a function `swap(const long i, const long j)` which simply swaps elements `i` and `j`.

Vectors of user-defined data types

Note that the vector class is defined on a template. Hence we define one class for any underlying data type and the compiler will generate the relevant type-specific classes himself. For example, in the following listing we define 3 vectors

```
Vector<int> vi(5);
Vector<double> vd;
Vector<Rational<long> > vr1, vr2;
Vector<Rational<long> > vr3 = vr1 + vr2;
```

Note that one of the vectors is a vector of rational numbers — a user defined data type (see previous chapter). When adding two vectors of rational numbers the appropriate addition operator of the `Rational` class is used when adding elements. Similarly,when sending a vector of rational numbers to an output stream, the relevant output stream operator for rational numbers is automatically used for sending the individual elements to the output stream.

The Demonstration Program

On the disk accompanying this book we supply the following demonstration program TVECTOR.CPP which illustrates the usage and power of the vector class:

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

#include "Vector.h"
#include "Rational.h"

#include "Vector.cpp"
#include "Rational.cpp"

using namespace std;

int main()
{
  Vector<double> v1(4);
  v1[0]=-1;  v1[1]=-4;  v1[2]=3; v1[3]=0.1;
          // using element access operator
  cout << "v1 = " << v1 << endl;  // using output stream operator

  Vector<double> v2 = v1.func(exp);
  v2.setoutputprecision(8); v2.setoutputwidth(10);
  cout << "v2 = exp(v1) = " << v2 << endl;
```

```
Vector<double> v3 = v1 + v2;     // vector addition and assignment
cout << "v1+v2 = " << v3 << endl;
cout << "v1*v2 = " << v1*v2 << endl;   // dot product
cout << "v1*2  = " << v1*2 << endl;     // using member function
cout << "2.0-v1  = " << 2.0-v1 << endl;     // using global function

cout << "v1.insert(v2,2)" << v1.insert(v2,2) << endl;

cout << v1.maxel() << " " << v1.minel() << " "
     << v1.maxabsel() << " " << v1.minabsel() << endl;

v1.normalize();
cout << "v1 = " << v1 << " => " << v1.norm() << endl;

v1.unit(2);
cout << "v1.unit(2) = e3 = " << v1 << endl;

Vector<int> v4(20);
for (int i=0; i<20; i++)
  v4[i] = i;
cout << "v4 = " << v4 << endl;
cout << "v4.slice(5,6) = " << v4.slice(5,6) << endl;
cout << "v4.slice(3,4,2) = " << v4.slice(3,4,2) << endl;

cout << "-v4 = " << -v4 << endl;  // unary minus

Vector<Rational<long> > vr1(3), vr2(3);
vr1[0] = Rational<long>(1,1);    vr2[0] = Rational<long>(17,2);
vr1[1] = Rational<long>(1,2);    vr2[1] = Rational<long>(-21,4);
vr1[2] = Rational<long>(1,3);    vr2[2] = Rational<long>(3,7);

vr1.setoutputwidth(4);  vr2.setoutputwidth(4);
cout << "vr1 =" << vr1 << "     vr2=" << vr2 << endl;
cout << "(2*vr1+vr2)*vr1 = " << (*(new Rational<long>(2))*vr1+vr2)*vr1
     << "  (dot product)" << endl;

try
{
  Vector<int> v5(-1);
}
catch (Exception error)
{
  cout << "*** ERROR ***: " << error.source << endl;
}

v1.random(29);
cout << "v1 = " << v1 << endl;
cout << "Enter vector v1: ";  cin >> v1;
cout << "v1 = " << v1 << endl;

Vector<double> vlong(100);
vlong.random();
cout << "long vector before sorting: " << vlong << endl;
vlong.quickSort();
```

```
  cout << "long vector after sorting: " << vlong << endl;

  return 0;
}
```

5.2.7 Listing of the Vector Class

The interface for the vector class is defined in the header file VECTOR.H:

```
#include <iostream>
#include <cstdlib>  // for srand()
#include <assert.h>
#include <math.h>

#include "ErrHandl.h"

using namespace std;

#ifndef __VECTOR_H
#define __VECTOR_H


template <class T> class Vector
{
  public:
    Vector ();
    Vector (const Vector<T>& v);
    Vector (const long lngth);
   ~Vector ();

    void resize (const long lngth);

    T& operator[] (const long i) const;  // element access

    long length    () const;  // returns length of vector

    int  outputprecision    () const;  // returns output precision
    int  outputwidth        () const;  // returns output width
    void setoutputprecision (const int precisn);
    void setoutputwidth     (const int wdth);

    void fill      (const T& x);   // set all v_i=x
    void unit      (const long i); // set to unit vector e_i

    void random (const T& upper=1, const T& lower=0);

    Vector<T> insert (const Vector<T>& v, const long before) const;
    Vector<T> slice  (const long istart, const long noelements,
                      const long stride=1) const;

    Vector<T> func (T (*f)(const T&)) const;  // returns f(vec)
    Vector<T> func (T (*f)(T)) const;  // returns f(vec)

    T    sumels    () const;       // sum of all elements
```

```
    T    mean      () const;       // arithmetic
    T    norm      () const;       // Euclidean norm
    T    norm      (const double& p) const;  // p-norm => (sum(|v_i|^p))^(1/p)
    T    maxel      () const;      // largest element
    T    minel      () const;      // smallest element
    T    maxabsel   () const;      // largest abs(element)
    T    minabsel   () const;      // smallest abs(element)
    long maxindx    () const;      // index of largest element
    long minindx    () const;      // index of smallest element
    long maxabsindx () const;      // index of largest abs
    long minabsindx () const;      // index of smallest abs
    void normalize  ();            // so that vec*vec=1
    void swap       (const long i, const long j);
    void quickSort  (long low=0, long high=-1);

    Vector<T> hadamardproduct (const Vector<T>& v) const;
                           // element for element multiplication

    Vector<T>& operator= (const Vector<T>& v);
    Vector<T>  operator+ (const Vector<T>& v) const;
    Vector<T>  operator- (const Vector<T>& v) const;
    T          operator* (const Vector<T>& v) const; // dot product
    Vector<T>  operator+ (const T& x) const;
    Vector<T>  operator- (const T& x) const;
    Vector<T>  operator* (const T& x) const;
    Vector<T>  operator/ (const T& x) const;

    void operator+= (const Vector<T>& v);
    void operator-= (const Vector<T>& v);
    void operator+= (const T& x);
    void operator-= (const T& x);
    void operator*= (const T& x);
    void operator/= (const T& x);

    Vector<T> operator+ () const;   // unary +
    Vector<T> operator- ();         // unary -
/*
    friend ostream&  operator<< (ostream& os, const Vector<T>& v);
    friend istream&  operator>> (istream& is,       Vector<T>& v);
    friend Vector<T> operator*  (const T& x,  const Vector<T>& v);
    friend Vector<T> operator+  (const T& x,  const Vector<T>& v);
    friend Vector<T> operator-  (const T& x,  const Vector<T>& v);
*/
  private:
    T* _v;
    long _length;
    int _precision, _width;
    static const int _defaultprecision;
    static const int _defaultwidth;

    long partitionSmallerLarger (long low, long high, long ipivot);
};

template <class T>
```

```
ostream& operator<< (ostream& os, const Vector<T>& v);

template <class T>
istream& operator>> (istream& is, Vector<T>& v);

template <class T>
Vector<T> operator* (const T& x, const Vector<T>& v);

template <class T>
Vector<T> operator+ (const T& x, const Vector<T>& v);

template <class T>
Vector<T> operator- (const T& x, const Vector<T>& v);


// With the Matrix class we define a global operator
/*     Matrix<T> operator% (const Vector<T>&, const Vector<T>&);  */
// which computes the Kronecker product between 2 vectors
// and returns the result as a matrix.

#endif
```

The implementation details are given in the file VECTOR.CPP

```
#include "Vector.h"

//=========================================================
// Initializing Static Class Constants
//---------------------------------
template<class T> const int Vector<T>::_defaultprecision = 4;
template<class T> const int Vector<T>::_defaultwidth     = 7;
//=========================================================
// CONSTRUCTORS and DESTRUCTOR
//---------------------------
template <class T> Vector<T>::Vector ()
  : _v(NULL), _length(0),
    _precision(_defaultprecision), _width(_defaultwidth) {}
//---------------------------------------------------------
template <class T> Vector<T>::Vector (const Vector<T>& v)
  : _v(NULL), _length(0),_precision(_defaultprecision),
    _width(_defaultwidth)  {*this=v;}
//---------------------------------------------------------
template <class T> Vector<T>::Vector (const long lngth)
  : _precision(_defaultprecision), _width(_defaultwidth)
{
  #if EXCEPTIONHANDLING
    if (lngth<0) throw Exception("Vector(lngth) => lngth<0");
  #endif
  _v = new T[lngth];
  _length=lngth;
}
//---------------------------------------------------------
template <class T> inline Vector<T>::~Vector ()
{delete[] _v;}
//---------------------------------------------------------
```

```
template <class T> inline long Vector<T>::length () const
{return _length;}
//-------------------------------------------------------
template <class T> inline int Vector<T>::outputprecision () const
{return _precision;}
//-------------------------------------------------------
template <class T> inline int Vector<T>::outputwidth () const
{return _width;}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::setoutputwidth (const int wdth)
{
  #if EXCEPTIONHANDLING
    if (wdth<=0)
      throw IllegalArguments("Vector::setoutputwidth(wdth)");
  #endif
  _width = wdth;
}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::setoutputprecision (const int precisn)
{
  #if EXCEPTIONHANDLING
    if (precisn<=0)
      throw IllegalArguments("Vector::setoutputwidth(precisn)");
  #endif
  _precision = precisn;
}
//=======================================================
// ELEMENT ACCESS
//---------------
template <class T>
inline T& Vector<T>::operator[] (const long i) const
{
  #if EXCEPTIONHANDLING
    if ((i<0) || (i>=_length))
      throw Range("Vector::operator[]:");
  #endif
  return _v[i];
}
//=======================================================
// USEFUL METHODS
//---------------
template <class T>
void Vector<T>::fill (const T& x)
{for (long i=0; i<_length; i++) _v[i]=x;}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::unit (const long i)
{
  #if EXCEPTIONHANDLING
    if ((i<0) || (i>=_length))
      throw Range("Vector::unit(i)");
  #endif
```

```
  fill(0);  _v[i]=1;
}
//-------------------------------------------------------

template <class T>
void Vector<T>::random (const T& upper, const T& lower)
{
  for (long i=0; i<_length; i++)
    _v[i] = (T)rand();
}

//-------------------------------------------------------
template <class T>    // inserts vector v into *this before element no before
Vector<T> Vector<T>::insert (const Vector<T>& v,
                             const long before) const
{
  #if EXCEPTIONHANDLING
    if ((before<0) || (before>_length))
      throw Range("Vector::insert(v,before)");
  #endif

  Vector<T> result(_length+v._length);

  for (long i=0; i<before; i++)
    result._v[i] = _v[i];
  for (long i=0; i<v._length; i++)
    result._v[before+i] = v._v[i];
  for (long i=before; i<_length; i++)
    result._v[v._length+i] = _v[i];
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> Vector<T>::slice  (const long istart,
     const long noelements, const long stride) const
{
  long last = istart+(noelements-1)*stride;
  if ((istart<0) || (istart>=_length) || (last<0) ||
      (last>=_length)) throw Range("Vector::slice");

  Vector<T> result(noelements);
  long i=istart;
  for (long j=0; j<noelements; j++)
  {
    result._v[j] = _v[i];
    i += stride;
  }
  return result;
}
//-------------------------------------------------------
template <class T>            // returns f(vec)
Vector<T> Vector<T>::func (T (*f)(const T&)) const
{
  Vector<T> result(_length);
```

```
  for (long i=0; i<_length; i++)
    result._v[i] = f(_v[i]);
  return result;
}
//-------------------------------------------------------------
template <class T>            // returns f(vec)
Vector<T> Vector<T>::func (T (*f)(T)) const
{
  Vector<T> result(_length);
  for (long i=0; i<_length; i++)
    result._v[i] = f(_v[i]);
  return result;
}
//-------------------------------------------------------------
template <class T>            // sum of all elements
T Vector<T>::sumels () const
{
  T sum = 0;
  for (long i=0; i<_length; i++)
    sum += _v[i];
  return sum;
}
//-------------------------------------------------------------
template <class T>           // arithmetic mean of elements
inline T Vector<T>::mean () const
{return sumels()/_length;}
//-------------------------------------------------------------
template <class T>           // Euclidean norm
T Vector<T>::norm () const
{
  T sum=0;
  for (long i=0; i<_length; i++)
  {
    T x = _v[i];
    sum += x*x;
  }
  return sqrt(sum);
}
//-------------------------------------------------------------
template <class T>          // so that vec*vec=1;
void Vector<T>::normalize ()
{
  T fact = norm();
  if (fact==(double)0)
    throw DivideByZero("Vector::normalize");
  for (long i=0; i<_length; i++)
    _v[i] /= fact;
}
//-------------------------------------------------------------
template <class T>        // (sum(|v_i|^p))^(1/p)
T Vector<T>::norm (const double& p) const
{
  T sum=0;
  for (long i=0; i<_length; i++)
```

```
  {
    sum += pow(fabs(_v[i]),p);
  }
  return pow(sum,(double)1/p);
}
//-------------------------------------------------------
template <class T>     // returns index of largest element
long Vector<T>::maxindx () const
{
  #if EXCEPTIONHANDLING
    if (_v == NULL)
      throw IllegalCall("Vector::maxindx => NULL vector");
  #endif
  long indx=0;
  T max = _v[0];
  for (int i=1; i<_length; i++)
  {
    if (max < _v[i])
    {
      indx = i;
      max  = _v[i];
    }
  }
  return indx;
}
//-------------------------------------------------------
template <class T>     // returns index of smallest element
long Vector<T>::minindx () const
{
  #if EXCEPTIONHANDLING
    if (_v == NULL)
      throw IllegalCall("Vector::minindx => NULL vector");
  #endif
  long indx=0;
  T min = _v[0];
  for (int i=1; i<_length; i++)
  {
    if (min > _v[i])
    {
      indx = i;
      min  = _v[i];
    }
  }
  return indx;
}
//-------------------------------------------------------
template <class T>   // returns index of largest abs element
long Vector<T>::maxabsindx () const
{
  #if EXCEPTIONHANDLING
    if (_v == NULL)
      throw IllegalCall("Vector::maxabsindx => NULL vector");
  #endif
  long indx=0;
```

```
  T max = fabs(_v[0]), x;
  for (int i=1; i<_length; i++)
  {
    if (max < (x=fabs(_v[i])))
    {
      indx = i;
      max  = x;
    }
  }
  return indx;
}
//-------------------------------------------------------------
template <class T> // returns index of smallest abs element
long Vector<T>::minabsindx () const
{
  #if EXCEPTIONHANDLING
    if (_v == NULL)
      throw IllegalCall("Vector::minabsindx => NULL vector");
  #endif
  long indx=0;
  T min = fabs(_v[0]), x;
  for (int i=1; i<_length; i++)
  {
    if (min > (x=fabs(_v[i])))
    {
      indx = i;
      min  = x;
    }
  }
  return indx;
}
//-------------------------------------------------------------
template <class T>        // returns largest element
inline T Vector<T>::maxel () const
{return _v[maxindx()];}
//-------------------------------------------------------------
template <class T>        // returns smallest element
inline T Vector<T>::minel () const
{return _v[minindx()];}
//-------------------------------------------------------------
template <class T>        // returns largest abs element
inline T Vector<T>::maxabsel () const
{return fabs(_v[maxabsindx()]);}
//-------------------------------------------------------------
template <class T>        // returns smallest abs element
inline T Vector<T>::minabsel () const
{return fabs(_v[minabsindx()]);}
//-------------------------------------------------------------
template <class T>        // swaps elements i and j
inline void Vector<T>::swap (const long i, const long j)
{
  if (i!=j)
  {
    #if EXCEPTIONHANDLING
```

```
      if ((i<0) || (j<0) || (i>=_length) || (j>=_length))
        throw Range("Vector::swap");
    #endif
    T dummy = _v[i];
    _v[i]   = _v[j];
    _v[j]   = dummy;
  }
}
//-------------------------------------------------------
template <class T>
void Vector<T>::quickSort (long low, long high)
{
  if (high==-1)
    high=_length-1;
  if (low >= high)
    return;  // nothing to sort

  long ipivot = (low+high)/2;
  ipivot = partitionSmallerLarger (low,high,ipivot);
  if (low < ipivot)
    quickSort(low,ipivot-1);
  if (ipivot < high)
    quickSort(ipivot+1,high);
}
//-------------------------------------------------------
template <class T>
long Vector<T>::partitionSmallerLarger (long low, long high,
long ipivot)
{
  if (ipivot != low)
    swap(low,ipivot);
  ipivot = low;
  low++;

  while (low <= high)
  {
    if (_v[low] <= _v[ipivot])
      ++low;
    else if (_v[high] > _v[ipivot])
           --high;
         else
           swap(low,high);
  }

  if (high != ipivot)
    swap(ipivot,high);
  return high;
}
//-------------------------------------------------------
template <class T>    // element for element multiplication
Vector<T> Vector<T>::hadamardproduct (const Vector<T>& v) const
{
  #if EXCEPTIONHANDLING
    if (_length != v._length)
```

```
      throw IllegalOperation("Vector::hadamardproduct");
  #endif
  Vector<T> result(v);
  for (long i=0; i<_length; i++)
    result._v[i] *= _v[i];

  return result;
}
//=========================================================
// Class Operators
//----------------
template <class T>      // assignment operator
Vector<T>& Vector<T>::operator= (const Vector<T>& v)
{
  if (this != &v)
   {
       if (_length != v._length)
         resize(v._length);
       for (long i=0; i<_length; i++)
        _v[i] = v._v[i];
   }
  return *this;
}
//---------------------------------------------------------
template <class T>      // vector addition
Vector<T> Vector<T>::operator+ (const Vector<T>& v) const
{
  #if EXCEPTIONHANDLING
    if (_length != v._length)
      throw IllegalOperation("Vector::operator+");
  #endif
  Vector<T> result(*this);
  for (long i=0; i<_length; i++)
    result._v[i] += v._v[i];
  return result;
}
//---------------------------------------------------------
template <class T>      // vector subtraction
Vector<T> Vector<T>::operator- (const Vector<T>& v) const
{
  #if EXCEPTIONHANDLING
    if (_length != v._length)
      throw IllegalOperation("Vector::operator-");
  #endif
  Vector<T> result(*this);
  for (long i=0; i<_length; i++)
    result._v[i] -= v._v[i];
  return result;
}
//---------------------------------------------------------
template <class T>      // Dot product
T Vector<T>::operator* (const Vector<T>& v) const
{
  #if EXCEPTIONHANDLING
```

```
    if (_length != v._length)
      throw IllegalOperation("Vector::operator*");
  #endif
  T result=0;
  for (long i=0; i<_length; i++)
    result += _v[i]*v._v[i];
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> Vector<T>::operator+ (const T& x) const
{
  Vector<T> result(*this);
  for (long i=0; i<_length; i++)
    result._v[i] += x;
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> Vector<T>::operator- (const T& x) const
{
  Vector<T> result(*this);
  for (long i=0; i<_length; i++)
    result._v[i] -= x;
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> Vector<T>::operator* (const T& x) const
{
  Vector<T> result(*this);
  for (long i=0; i<_length; i++)
    result._v[i] *= x;
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> Vector<T>::operator/ (const T& x) const
{
  Vector<T> result(*this);
  for (long i=0; i<_length; i++)
    result._v[i] /= x;
  return result;
}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::operator+= (const Vector<T>& v)
{*this = *this + v;}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::operator-= (const Vector<T>& v)
{*this = *this - v;}
//-------------------------------------------------------
template <class T>
```

```
inline void Vector<T>::operator+= (const T& x)
{*this = *this + x;}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::operator-= (const T& x)
{*this = *this - x;}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::operator*= (const T& x)
{*this = *this * x;}
//-------------------------------------------------------
template <class T>
inline void Vector<T>::operator/= (const T& x)
{*this = *this / x;}
//-------------------------------------------------------
template <class T>       // unary +
inline Vector<T> Vector<T>::operator+ () const
{return (*this);}
//-------------------------------------------------------
template <class T>       // unary -
Vector<T> Vector<T>::operator- ()
{
  Vector<T> result(_length);
  for (long i=0; i<_length; i++)
    result._v[i] = -_v[i];
  return result;
}
//=======================================================
// PRIVATE MEMBER FUNCTIONS
//-------------------------
template <class T> void Vector<T>::resize (const long lngth)
{
  #if EXCEPTIONHANDLING
    if (lngth<0)
      throw IllegalArguments("Vector::resize(lngth) => lngth<0");
  #endif
  delete[] _v;
  if (lngth>0)
   {
     _v = new T[lngth];
     _length = lngth;
   }
  else
   {
     _v = NULL;
     _length = 0;
   }
}
/*
//=======================================================
// FRIENDS OF VECTOR
//------------------
template <class T>
ostream& operator<< (ostream& os, const Vector<T>& v)
```

```
{
  os << "[ ";
  for (long i=0; i<v._length; i++)
  {
    os.precision(v._precision); os.width(v._width);
    os << v._v[i] << ' ';
  }
  return os << "]";
}
//-------------------------------------------------------
template <class T>
istream& operator>> (istream& is, Vector<T>& v)
{
  for (long i=0; i<v._length; i++)
    is >> v._v[i];
  return is;
}
//=======================================================
// Global Operators
//-----------------
template <class T>
Vector<T> operator* (const T& x, const Vector<T>& v)
{
  Vector<T>  result(v._length);
  for (long i=0; i<v._length; i++)
    result._v[i] = x*v._v[i];
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> operator+ (const T& x, const Vector<T>& v)
{
  Vector<T>  result(v._length);
  for (long i=0; i<v._length; i++)
    result._v[i] = x+v._v[i];
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> operator- (const T& x, const Vector<T>& v)
{
  Vector<T>  result(v._length);
  for (long i=0; i<v._length; i++)
    result._v[i] = x-v._v[i];
  return result;
}
*/
//=======================================================

template <class T>
ostream& operator<< (ostream& os, const Vector<T>& v)
{
  os << "[ ";
  for (long i=0; i<v.length(); i++)
```

```
  {
    os.precision(v.outputprecision()); os.width(v.outputwidth());
    os << v[i] << ' ';
  }
  return os << "]";
}
//-------------------------------------------------------
template <class T>
istream& operator>> (istream& is, Vector<T>& v)
{
  for (long i=0; i<v.length(); i++)
    is >> v[i];
  return is;
}
//=======================================================
// Global Operators
//-----------------
template <class T>
Vector<T> operator* (const T& x, const Vector<T>& v)
{
  Vector<T>  result(v.length());
  for (long i=0; i<v.length(); i++)
    result[i] = x*v[i];
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> operator+ (const T& x, const Vector<T>& v)
{
  Vector<T>  result(v.length());
  for (long i=0; i<v.length(); i++)
    result[i] = x+v[i];
  return result;
}
//-------------------------------------------------------
template <class T>
Vector<T> operator- (const T& x, const Vector<T>& v)
{
  Vector<T>  result(v.length());
  for (long i=0; i<v.length(); i++)
    result[i] = x-v[i];
  return result;
}
```

5.3 Matrix Class

5.3.1 Introduction

5.3.2 Arrays of Objects

The underlying data structure for the matrix is an array of vectors. Below we show the private data members of the class:

```
template <class T> class Matrix
{
    ...
  private:
    Vector<T>*       _M;
    long             _nrows, _ncols;
    int              _precision, _width;
    static const int _defaultprecision;
    static const int _defaultwidth;
};
```

`_M` is the array of vectors which holds the actual matrix elements, `_nrows` and `_ncols` hold the dimension of the matrix and the remaining data members are identical to those used in the vector class for setting the output format of a matrix object.

When declaring an array of objects C++ always uses the default constructor of the class — it is thus recommended that you always define a default constructor for your class except if you have good reasons to disallow the user to define an array of a certain class.

We define three constructor for the matrix class, the default constructor, the copy constructor and a constructor allowing the user to declare a matrix of a certain size. The first two of these are virtually identical to their counterparts in the vector class and the reader can look at their implementation in the class listing given in section 5.3.6. The implementation details of the third constructor are given below:

```
template <class T>
Matrix<T>::Matrix (const long nrows, const long ncols)
  : _precision(_defaultprecision), _width(_defaultwidth)
{
  #if EXCEPTIONHANDLING
    if ((nrows<0) || (ncols<0))
      throw InvalidArguments("Matrix(nrows,ncols) => nrows<0 or ncols<0");
  #endf
  _M = new Vector<T>[nrows];
  for (long nr=0; nr<nrows; nr++)
    _M[nr].resize(ncols);
  _nrows = nrows;  _ncols = ncols;
}
```

First we perform sanity-checking on the required size of the matrix. Then we reserve memory for `nrows` vectors of the template type via the `new` operator and we set the pointer variable `_M` equal to the start of this memory area. It is at this point where C++ calls the default constructor of the vector class for each of the array elements. Consequently the vectors (the rows of the matrix) have all zero length. To set the size of these vectors equal to the number of columns, we use the `resize` method of the vector class. Finally we set the private data fields for the number of rows and number of columns of the matrix.

5.3.3 Using the Functionality of the Underlying Vector Class

Many methods of the matrix class look particularly simple because we use the functionality of the underlying vector class. Consider, for example, matrix addition:

```
template <class T>
Matrix<T> Matrix<T>::operator+ (const Matrix<T>& M) const
{
  #if EXCEPTIONHANDLING
    if ((_nrows!=M._nrows) || (_ncols!=M._ncols))
      throw InvalidOperation("Matrix::operator+(Matrix)");
  #endif
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    result._M[nr] += M._M[nr];
  return result;
}
```

If the matrices are of different sizes, we throw an `InvalidOperation` exception. Otherwise we make a local copy of the matrix itself (using the copy constructor) and use vector addition to add the matrices row-for-row.

Similarly, the method random which fills the matrix with random numbers chosen between `upper` and `lower` limits requests each of its rows (each of its vectors) to fill itself with random numbers:

```
template <class T>
void Matrix<T>::random (const T& upper=1, const T& lower=0)
{
  for (long nr=0; nr<_nrows; nr++)
    _M[nr].random(lower,upper);
}
```

5.3.4 Matrix Multiplication, Kronecker and Hadamard Products

As for vectors, we define three different types of products for matrices. For standard matrix multiplication we use the multiplication operator `*`. The Kronecker product (also known as direct product) is defined by

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix} \tag{5.7}$$

It will be used extensively in the sections on neural networks and on non-linear programming.

Unlike for normal matrix multiplication where we have a reversal of order upon taking the transpose of a an ordinary product

$$(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}} \tag{5.8}$$

we have no such reversal when taking the transpose of a Kronecker product of matrices

$$(\mathbf{A} \otimes \mathbf{B})^{\mathrm{T}} = \mathbf{A}^{\mathrm{T}} \otimes \mathbf{B}^{\mathrm{T}} \tag{5.9}$$

It can also be shown (see exercise ??) that the Kronecker product is distributive

$$\mathbf{A} \otimes (\mathbf{B} + \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} + \mathbf{A} \otimes \mathbf{B} \tag{5.10}$$

and associative

$$\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) = (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} \tag{5.11}$$

Neither the ordinary nor the Kronecker product are commutative

$$\mathbf{AB} \neq \mathbf{BA} \qquad \mathbf{A} \otimes \mathbf{B} \neq \mathbf{B} \otimes \mathbf{A}$$

The implementation of the Kronecker product (5.7) is given below

```
template <class T>     // Kroneckerproduct
Matrix<T> Matrix<T>::operator% (const Matrix<T>& M) const
{
  Matrix<T> result(_nrows*M._nrows,_ncols*M._ncols);

  for (long nr1=0; nr1<_nrows; nr1++)
    for (long nc1=0; nc1<_ncols; nc1++)
      for (long nr2=0; nr2<M._nrows; nr2++)
        for (long nc2=0; nc2<M._ncols; nc2++)
          result._M[nr1*M._nrows+nr2][nc1*M._ncols+nc2]
            = _M[nr1][nc1]*M._M[nr2][nc2];
  return result;
}
```

The second special product which is particularly useful in statistics is the Hadamard or Schur product. If two matrices, $\mathbf{A}$ and $\mathbf{B}$, both have dimension $(m \times n)$, then $\mathbf{A} \circ \mathbf{B}$ yields a $(m \times n)$ matrix whose elements are simply the product of the corresponding elements in $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{A} \circ \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{bmatrix} \tag{5.12}$$

The Hadamard product is distributive and associative. The implementation of the Hadamard product is trivial and can be found in the program listing (see section 5.3.6).

5.3.5 Users Guide to the Matrix Class

Besides the assignment operator, the destructor, default constructor, copy constructor and a constructor allowing the user to create a matrix of specific dimensions via for example

```
Matrix<double> M(3,4));
```

we supply a wide range of operators, query and manipulation functions with the matrix class.

Element, Row and Column Access

Row access is provided via the subscription operator `[]`. Hence `M[0]` returns the first row of the matrix as a vector. The access is very fast (especially when exception handling is switched off) — this inline function simply returns the relevant row by reference:

```
template <class T>
inline Vector<T>& Matrix<T>::operator[] (const long& nrow) const
{
  #if EXCEPTIONHANDLING
    if ((nrow<0) || (nrow >= _nrows))
      throw Range("Matrix::operator[] (nrow)");
  #endif
  return _M[nrow];
}
```

Concatenating subscription operators gives matrix elements access. Hence `M[0][0]` returns the 1-1-element of the matrix. The first subscription operator acts on the matrix `M` returning a vector. The second subscription operator now acts on a vector object (the subscription operator of the vector class is called) returning the 0'th element of the vector representing the 0'th row.

Column access carries significantly higher computational overheads. We use the function call operator `()`. Hence `M(1)` returns the second column as a vector. The implementation details are given below:

```
Vector<T> Matrix<T>::operator() (const long ncol) const
{
  #if EXCEPTIONHANDLING
    if ((ncol<0) || (ncol >= _ncols))
      throw Range("Matrix::operator() (ncol)");
  #endif
  Vector<T> result(_nrows);
  for (long nr=0; nr<_nrows; nr++)
     result[nr] = _M[nr][ncol];
  return result;
}
```

After range checking we create a local vector `result` into which we copy the matrix elements of the relevant column. The vector is a local variable and has to be returned by reference.

Query Functions

The member functions `rows()` and `cols()` return the number of rows and columns of the matrix, `diag()` returns the diagonal elements as a vector and `trace()` returns the trace of the matrix — the sum of the diagonal elements. The query functions `outputwidth()` and `outputprecision()` return the current choice of output format for the matrix elements.

Arithmetic Operators

Support for matrix addition, subtraction and multiplication is provided by the relevant operators, `+`, `-` and `*`. Hence all of the following statements are legal

```
Matrix<double> A(4,4), B(4,4), C(4,4);
A = B+C;     // matrix addition
A = B-C;     // matrix subtraction
A = B*C;     // matrix product
A = B%C;     // Kronecker product
```

We use the modulo operator `%` for the Kronecker product and we define a member function `hadamardproduct(const Matrix<T>& M)` for the Hadamard-product between two matrices. The different matrix products are discussed in more detail in section 5.3.4.

Our matrix class supports mixed type arithmetic. The user can add/subtract a vector to/from a matrix or a matrix to/from a vector (in the second case the friend operator is used). The operators are implemented such that a diagonal matrix with the vector elements on the diagonal is added or subtracted from the matrix.

$$\texttt{M+v;} \quad \Longrightarrow \quad \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{bmatrix} + \begin{bmatrix} v_1 & 0 & \cdots & 0 \\ 0 & v_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & v_n \end{bmatrix} \tag{5.13}$$

We also allow the user to add a scalar to a matrix. In this case a diagonal matrix with the scalar value n the diagonal is added to the matrix.

$$\texttt{M+x;} \quad \Longrightarrow \quad \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{bmatrix} + \begin{bmatrix} x & 0 & \cdots & 0 \\ 0 & x & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x \end{bmatrix} \tag{5.14}$$

The same holds for subtraction, multiplication and division of a matrix by a scalar.

The corresponding assignment operators, `+=`, `-=`, ... are also defined for the matrix class.

Matrix Manipulation Functions

The method `unit()` sets a square matrix to a unit matrix. The method `fill(const T& x)` sets all elements of the matrix equal to x. The method `random(const T& upper=1, const T& lower=0)` fills the matrix with random numbers between `lower` and `upper`. The method is very similar to the corresponding method of the vector class. The following two methods

```
template <class T> class Matrix
{
  public:
    ...
    void setcol    (const long ncol, const Vector<T>& v);
    void setrow    (const long nrow, const Vector<T>& v);
}
```

allow the user to set a certain row or column to the values contained in the vector `v`. The method `setdiag(const Vector<T>& v)` sets the diagonal elements equal to

the elements of the vector, while the method `setdiag(const T& x)` sets all diagonal elements equal to `x`.

The method `transpose()` returns a matrix which is the transpose of the matrix itself. Hence `A = B.transpose` sets the elements of `A` such that `A[i][j]` is equal to `B[j][i]`.

Inserting and Removing Rows and Columns

The methods `removerow(const long nrow)` and `removecol(const long ncol)` allow the user to remove a specific row or column of a matrix. For example `A.removerow(1)` removes the second row from the matrix `a`.

The following two methods allow the user to insert a row or column before a certain row or column:

```
template <class T> class Matrix
{
  public:
    ...
    void insertrow (const long beforerow, const Vector<T>& v);
    void insertcol (const long beforecol, const Vector<T>& v);
}
```

For example

```
A.insertcol(A.cols(),v);
```

appends a column vector `v` to the matrix `a`.

Stream Access

We provide input and output stream access via the standard stream extraction and stream output operators, `>>` and `<<`. The two operators can be used as follows:

```
void main()
{
  Matrix<int> M(2,3);
  cin >> M;
  M.setoutputprecision(2);
  M.setoutputwidth(3);
  cout << "You have just read in the following matrix:" << endl << M;
}
```

After declaring a (2×3) matrix of integers, `M` we extract the matrix from the standard input stream. The stream extraction operator simply extract 6 integers from the stream and reads them row-for-row into the matrix.

We then set the output format for the matrix elements and send the matrix to the standard output stream via the operator `>>`. Both, the stream extraction and stream output operators are declared friends of the class and hence they do have access to the private data members of the class. The implementation details for the output stream operator are given below:

```
template <class T>
ostream& operator<< (ostream& os, const Matrix<T>& M)
{
  for (long nr=0; nr<M._nrows; nr++)
  {
    os << "| ";
    for (long nc=0; nc<M._ncols; nc++)
    {
      os.precision(M._precision); os.width(M._width);
      os << M._M[nr][nc] << ' ';
    }
    os << "|" << endl;
  }
  return os;
}
```

and the resultant output for the matrix `M` could be

```
|  1  2  3 |
|  4  5  6 |
```

The Demonstration Program

On the disk accompanying this book we supply the following demonstration program TMATRIX.CPP which illustrates the usage and power of the matrix class:

```
#include <iostream>

#include "Matrix.h"
#include "Vector.h"
#include "Rational.h"

#include "Vector.cpp"
#include "Matrix.cpp"
#include "Rational.cpp"

using namespace std;

int main()
{
  try
  {
    Matrix<double> A(4,4);
    A.random();
    Matrix<double> B = A;
    B.setoutputwidth(6); B.setoutputprecision(2);
    cout << "B = " << endl << B;

    Vector<double> v1(4); v1.fill(1.7);
    B.insertrow(1,v1);
    cout << "Inserted row 1: B = " << endl << B;
    B.removecol(2);
    cout << "Removed column 2: B = " << endl << B;
    B.swaprows(0,3);
```

```
    cout << "swapped rows 0 and 3: B = " << endl << B;
    A.fill(2);
    A=B.transpose();
    Matrix<double> I1(2,2); I1.setoutputwidth(3); I1.setoutputprecision(2);
    I1.random(8,-2);
    cout << "I1 = " << endl << I1;
    Matrix<double> I2=200.0 + 2.0*I1 / 10;
    cout << "I2 = " << endl << I2 << endl;

    Vector<double> v2(7);
    v1.random(); v2.random();
    cout << "v1 = " << v1 << endl;
    cout << "v2 = " << v2 << endl;
    cout << "Kronecker product: v1%v2 = " << endl << v1%v2;

    Rational<long> r(1,3);
    Matrix<Rational<long> > Mr1(3,3);
    for (long nr=0; nr<3; nr++)
      for (long nc=0; nc<3; nc++)
        Mr1[nr][nc] = r+(nr+1L)*Rational<long>(5,nc+1);
    cout << "Mr1 = " << endl << Mr1;
    Matrix<Rational<long> > Mr2 = Mr1*Mr1-Mr1/2;
    cout << "Mr2 = Mr1*Mr1-Mr1/2 = " << endl << Mr2;

  }
  catch (Exception error)
  {
    cout << "*** ERROR ***: " << error.source << endl;
  }
  return 0;
}
```

Again, we recommend that the reader runs this program.

5.3.6 Listing of the Matrix Class

The interface for the matrix class is defined in the header file MATRIX.H:

```
#ifndef __MATRIX_H
#define __MATRIX_H

#include <math.h>
#include <iostream>
#include <cstdlib>

#include "Vector.h"
#include "ErrHandl.h"

using namespace std;

template <class T> class Matrix
{
  public:
    Matrix ();
```

```
 Matrix (const Matrix<T>& M);
 Matrix (const long nrows, const long ncols);
~Matrix ();

 long rows         () const;
 long cols         () const;

 int  outputwidth        () const;
 int  outputprecision    () const;
 void setoutputwidth     (const int precisn);
 void setoutputprecision (const int wdth);

 void random (const T& upper=1, const T& lower=0);
 void fill   (const T& x);
 void unit   ();

 void setcol    (const long ncol, const Vector<T>& v);
 void setrow    (const long nrow, const Vector<T>& v);
 void insertrow (const long beforerow, const Vector<T>& v);
 void insertcol (const long beforecol, const Vector<T>& v);
 void removerow (const long nrow);
 void removecol (const long ncol);
 void swaprows  (const long r1, const long r2);
 void swapcols  (const long c1, const long c2);
 void setdiag   (const Vector<T>& v);
 void setdiag   (const T& x);

 Vector<T> diag  () const;
 T         trace () const;

 Matrix<T> transpose () const;
 Matrix<T> hadamamardproduct (const Matrix<T>& M) const;

 Matrix<T>& operator=  (const Matrix<T>& M);
 Vector<T>& operator[] (const long nrow)    const;
 Vector<T>  operator() (const long ncol)    const;

 Matrix<T>  operator+  (const Matrix<T>& M) const;
 Matrix<T>  operator-  (const Matrix<T>& M) const;
 Matrix<T>  operator*  (const Matrix<T>& M) const;
 Matrix<T>  operator%  (const Matrix<T>& M) const;
 Matrix<T>  operator+  (const Vector<T>& v) const;
 Matrix<T>  operator-  (const Vector<T>& v) const;
 Matrix<T>  operator+  (const T& x)         const;
 Matrix<T>  operator-  (const T& x)         const;
 Matrix<T>  operator*  (const T& x)         const;
 Matrix<T>  operator/  (const T& x)         const;
 Vector<T>  operator*  (const Vector<T>& v) const;
 void       operator+= (const Matrix<T>& M);
 void       operator-= (const Matrix<T>& M);
 void       operator*= (const Matrix<T>& M);
 void       operator%= (const Matrix<T>& M);
 void       operator+= (const Vector<T>& v);
 void       operator-= (const Vector<T>& v);
```

```
    void       operator+= (const T& x)        ;
    void       operator-= (const T& x)        ;
    void       operator*= (const T& x)        ;
    void       operator/= (const T& x)        ;
/*
    friend Matrix<T> operator% (const Vector<T>&,const Vector<T>&);
    friend Matrix<T> operator+ (const T& x,const Matrix<T>& M);
    friend Matrix<T> operator- (const T& x,const Matrix<T>& M);
    friend Matrix<T> operator* (const T& x,const Matrix<T>& M);
    friend Vector<T> operator* (const Vector<T>& v, Matrix<T> M);

    friend istream& operator>> (istream& is,       Matrix<T>& M);
    friend ostream& operator<< (ostream& os, const Matrix<T>& M);
*/
  private:
    Vector<T>*       _M;
    long             _nrows, _ncols;
    int              _precision, _width;
    static const int _defaultprecision;
    static const int _defaultwidth;

};

// Initializing Static Class Constants
template<class T> const int Matrix<T>::_defaultprecision = 3;
template<class T> const int Matrix<T>::_defaultwidth     = 9;

template <class T>
Matrix<T> operator% (const Vector<T>& v1,const Vector<T>& v2);

template <class T>
inline Matrix<T> operator+ (const T& x,const Matrix<T>& M);

template <class T>
Matrix<T> operator- (const T& x,const Matrix<T>& M);

template <class T>
inline Matrix<T> operator* (const T& x,const Matrix<T>& M);

template <class T>
Vector<T> operator* (const Vector<T>& v,const Matrix<T>& M);

template <class T>
istream& operator>> (istream& is, Matrix<T>& M);

template <class T>
ostream& operator<< (ostream& os, const Matrix<T>& M);

#endif
```

The implementation details are given in the file MATRIX.CPP

```
#include "Matrix.h"

//=========================================================
```

```
// CONSTRUCTORS and DESTRUCTOR
// ---------------------------
template <class T>
Matrix<T>::Matrix (): _M(NULL), _nrows(0), _ncols(0),
  _precision(_defaultprecision), _width(_defaultwidth) {}
//-------------------------------------------------------
template <class T>
Matrix<T>::Matrix (const Matrix<T>& M): _M(NULL),
    _nrows(0), _ncols(0), _precision(_defaultprecision),
    _width(_defaultwidth)  {*this=M;}
//-------------------------------------------------------
template <class T>
Matrix<T>::Matrix (const long nrows, const long ncols)
  : _precision(_defaultprecision), _width(_defaultwidth)
{
  #if EXCEPTIONHANDLING
    if ((nrows<0) || (ncols<0))
      throw IllegalArguments("Matrix(nrows,ncols) => nrows<0 or ncols<0");
  #endif
  _M = new Vector<T>[nrows];
  for (long nr=0; nr<nrows; nr++)
    _M[nr].resize(ncols);
  _nrows = nrows;  _ncols = ncols;
}
//-------------------------------------------------------
template <class T>
Matrix<T>::~Matrix () {delete[] _M;}
//=======================================================
// QUERY FUNCTIONS
// ---------------
template <class T>
long Matrix<T>::rows () const {return _nrows;}
//-------------------------------------------------------
template <class T>
long Matrix<T>::cols () const {return _ncols;}
//-------------------------------------------------------
template <class T>
int Matrix<T>::outputwidth () const {return _width;}
//-------------------------------------------------------
template <class T>
int Matrix<T>::outputprecision () const {return _precision;}
//=======================================================
// OTHER MEMBER FUNCTIONS
// ----------------------
template <class T>
void Matrix<T>::setoutputwidth (const int wdth)
{_width=wdth;}
//-------------------------------------------------------
template <class T>
void Matrix<T>::setoutputprecision (const int precisn)
{_precision = precisn;}
//-------------------------------------------------------
template <class T>
void Matrix<T>::random (const T& upper, const T& lower)
```

```
{
  for (long nr=0; nr<_nrows; nr++)
    _M[nr].random(lower,upper);
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::fill (const T& x)
{
  for (long nr=0; nr<_nrows; nr++)
    _M[nr].fill(x);
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::unit ()
{
  #if EXCEPTIONHANDLING
    if (_nrows != _ncols)
      throw IllegalOperation("Matrix::unit() => Matrix not square");
  #endif
  for (long nr=0; nr<_nrows; nr++)
    _M[nr].unit(nr);
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::setcol (const long ncol, const Vector<T>& v)
{
  #if EXCEPTIONHANDLING
    if ((ncol<0) || (ncol>=_ncols))
      throw Range("Matrix::setcol(ncol)");
  #endif
  for (long nr=0; nr<_nrows; nr++)
    _M[nr][ncol] = v[nr];
}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::setrow  (const long nrow, const Vector<T>& v)
{
  #if EXCEPTIONHANDLING
    if ((nrow<0) || (nrow>=_nrows))
      throw Range("Matrix::setrow(nrow)");
  #endif
  _M[nrow]=v;
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::insertrow (const long beforerow, const Vector<T>& v)
{
  #if EXCEPTIONHANDLING
    if ((beforerow<0) || (beforerow>_nrows))
      throw Range("Matrix::insertrow(beforerow,v)");
  #endif
  Matrix<T> dummy(_nrows+1,_ncols);
  for (long nr=0; nr<beforerow; nr++)
    dummy._M[nr] = _M[nr];
```

```
  dummy._M[beforerow] = v;
  for (long nr=beforerow+1; nr<=_nrows; nr++)
    dummy._M[nr] = _M[nr-1];
  *this = dummy;
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::insertcol (const long beforecol, const Vector<T>& v)
{
  #if EXCEPTIONHANDLING
    if ((beforecol<0) || (beforecol>_ncols))
      throw Range("Matrix::insertcol(beforecol,v)");
  #endif
  Matrix<T> dummy(_nrows,_ncols+1);
  for (long nc=0; nc<beforecol; nc++)
    for (long nr=0; nr<_nrows; nr++)
      dummy._M[nr][nc] = _M[nr][nc];
  for (long nr=0; nr<_nrows; nr++)
    dummy._M[nr][beforecol] = v[nr];
  for (long nc=beforecol+1; nc<=_ncols; nc++)
    for (long nr=0; nr<_nrows; nr++)
      dummy._M[nr][nc] = _M[nr][nc-1];
  *this = dummy;
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::removerow (const long nrow)
{
  #if EXCEPTIONHANDLING
    if ((nrow<0) || (nrow>=_nrows))
      throw Range("Matrix::removerow(nrow)");
  #endif
  Matrix<T> dummy(_nrows-1,_ncols);
  for (long nr=0; nr<nrow; nr++)
    dummy._M[nr] = _M[nr];
  for (long nr=nrow+1; nr<_nrows; nr++)
    dummy._M[nr-1] = _M[nr];
  *this = dummy;
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::removecol (const long ncol)
{
  #if EXCEPTIONHANDLING
    if ((ncol<0) || (ncol>_ncols))
      throw Range("Matrix::removecol(ncol)");
  #endif
  Matrix<T> dummy(_nrows,_ncols-1);
  for (long nc=0; nc<ncol; nc++)
    for (long nr=0; nr<_nrows; nr++)
      dummy._M[nr][nc] = _M[nr][nc];
  for (long nc=ncol+1; nc<_ncols; nc++)
    for (long nr=0; nr<_nrows; nr++)
      dummy._M[nr][nc-1] = _M[nr][nc];
```

```
  *this = dummy;
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::setdiag (const Vector<T>& v)
{
  #if EXCEPTIONHANDLING
    if (_nrows != _ncols)
      throw IllegalOperation("Matrix::setdiag() => Matrix not square");
  #endif
  for (long nr=0; nr<_nrows; nr++)
    _M[nr][nr] = v[nr];
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::setdiag (const T& x)
{
  #if EXCEPTIONHANDLING
    if (_nrows != _ncols)
      throw IllegalOperation("Matrix::setdiag() => Matrix not square");
  #endif
  for (long nr=0; nr<_nrows; nr++)
    _M[nr][nr] = x;
}
//-------------------------------------------------------
template <class T>
Vector<T> Matrix<T>::diag () const
{
  #if EXCEPTIONHANDLING
    if (_nrows != _ncols)
      throw IllegalOperation("Matrix::diag() => Matrix not square");
  #endif
  Vector<T> vdiag(_nrows);
  for (long nr=0; nr<_nrows; nr++)
    vdiag[nr] = _M[nr][nr];
  return vdiag;
}
//-------------------------------------------------------
template <class T>
T Matrix<T>::trace () const
{
  #if EXCEPTIONHANDLING
    if (_nrows != _ncols)
      throw IllegalOperation("Matrix::diag() => Matrix not square");
  #endif
  T sum=0;
  for (long nr=0; nr<_nrows; nr++)
    sum += _M[nr][nr];
  return sum;
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::swaprows (const long r1, const long r2)
{
```

```
  if (r1 != r2)
  {
    #if EXCEPTIONHANDLING
      if ((r1<0) || (r1>=_nrows) || (r2<0) || (r2>=_nrows))
        throw Range("Matrix::swaprows(r1,r2)");
    #endif
    Vector<T> dummy = _M[r1];
    _M[r1] = _M[r2];
    _M[r2] = dummy;
  }
}
//-------------------------------------------------------
template <class T>
void Matrix<T>::swapcols (const long c1, const long c2)
{
  if (c1 != c2)
  {
    #if EXCEPTIONHANDLING
      if ((c1<0) || (c1>=_ncols) || (c2<0) || (c2>=_ncols))
        throw Range("Matrix::swapcols(c1,c2)");
    #endif
    for (long nr=0; nr<_nrows; nr++)
    {
      T dummy = _M[nr][c1];
      _M[nr][c1] = _M[nr][c2];
      _M[nr][c2] = dummy;
    }
  }
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::transpose () const
{
  Matrix<T> result(_ncols,_nrows);
  for (long nr=0; nr<_nrows; nr++)
    for (long nc=0; nc<_ncols; nc++)
      result._M[nc][nr] = _M[nr][nc];
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::hadamamardproduct (const Matrix<T>& M) const
{
  Matrix<T> result(M);
  for (long nr=0; nr<_nrows; nr++)
    for (long nc=0; nc<_ncols; nc++)
      result._M[nr][nc] *= _M[nr][nc];
  return result;
}
//-------------------------------------------------------
template <class T>
inline Vector<T>& Matrix<T>::operator[] (const long nrow) const
{
  #if EXCEPTIONHANDLING
```

```
    if ((nrow<0) || (nrow >= _nrows))
      throw Range("Matrix::operator[] (nrow)");
  #endif
  return _M[nrow];
}
//-------------------------------------------------------
template <class T>
Vector<T> Matrix<T>::operator() (const long ncol) const
{
  #if EXCEPTIONHANDLING
    if ((ncol<0) || (ncol >= _ncols))
      throw Range("Matrix::operator() (ncol)");
  #endif
  Vector<T> result(_nrows);
  for (long nr=0; nr<_nrows; nr++)
     result[nr] = _M[nr][ncol];
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T>& Matrix<T>::operator= (const Matrix<T>& M)
{
  if (this != &M)
  {
    if (_nrows!=M._nrows)
    {
      delete[] _M;
      _M = new Vector<T>[M._nrows];
      _nrows = M._nrows;
    }
    for (long nr=0; nr<M._nrows; nr++)
      _M[nr] = M._M[nr];
    _ncols = M._ncols;
  }
  return *this;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator+ (const Matrix<T>& M) const
{
  #if EXCEPTIONHANDLING
    if ((_nrows!=M._nrows) || (_ncols!=M._ncols))
      throw IllegalOperation("Matrix::operator+(Matrix)");
  #endif
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    result._M[nr] += M._M[nr];
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator- (const Matrix<T>& M) const
{
  #if EXCEPTIONHANDLING
```

```
    if ((_nrows!=M._nrows) || (_ncols!=M._ncols))
      throw IllegalOperation("Matrix::operator-(Matrix)");
  #endif
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    result._M[nr] -= M._M[nr];
  return result;
}
//-------------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator* (const Matrix<T>& M) const
{
  #if EXCEPTIONHANDLING
    if (_ncols!=M._nrows)
      throw IllegalOperation("Matrix::operator*(Matrix)");
  #endif
  Matrix<T> result(_nrows,M._ncols);
  for (long nr1=0; nr1<_nrows; nr1++)
    for (long nc2=0; nc2<M._ncols; nc2++)
    {
      T sum=0;
      for (long nc1=0; nc1<_ncols; nc1++)
        sum += _M[nr1][nc1]*M._M[nc1][nc2];
      result._M[nr1][nc2]=sum;
    }
  return result;
}
//-------------------------------------------------------------
template <class T>
Vector<T> Matrix<T>::operator* (const Vector<T>& v) const
{
  #if EXCEPTIONHANDLING
    if (_ncols!=v.length())
      throw IllegalOperation("Matrix::operator*(Vector)");
  #endif
  Vector<T> result(_nrows);
  for (long nr=0; nr<_nrows; nr++)
  {
    T sum=0;
    for (long nc=0; nc<_ncols; nc++)
      sum += _M[nr][nc]*v[nc];
    result[nr]=sum;
  }
  return result;
}
//-------------------------------------------------------------
template <class T>     // Kroneckerproduct
Matrix<T> Matrix<T>::operator% (const Matrix<T>& M) const
{
  Matrix<T> result(_nrows*M._nrows,_ncols*M._ncols);

  for (long nr1=0; nr1<_nrows; nr1++)
    for (long nc1=0; nc1<_ncols; nc1++)
      for (long nr2=0; nr2<M._nrows; nr2++)
```

```
        for (long nc2=0; nc2<M._ncols; nc2++)
          result._M[nr1*M._nrows+nr2][nc1*M._ncols+nc2]
            = _M[nr1][nc1]*M._M[nr2][nc2];
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator+ (const Vector<T>& v) const
{
  #if EXCEPTIONHANDLING
    if ((_nrows==_ncols) && (_nrows==v.length()))
      throw IllegalOperation("Matrix::operator+(Vector)");
  #endif
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    result._M[nr][nr] += v[nr];
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator- (const Vector<T>& v) const
{
  #if EXCEPTIONHANDLING
    if ((_nrows==_ncols) && (_nrows==v.length()))
      throw IllegalOperation("Matrix::operator-(Vector)");
  #endif
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    result._M[nr][nr] -= v[nr];
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator+ (const T& x) const
{
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    result._M[nr][nr] += x;
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator- (const T& x) const
{
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    result._M[nr][nr] -= x;
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator* (const T& x) const
{
  Matrix<T> result(*this);
```

```
  for (long nr=0; nr<_nrows; nr++)
    for (long nc=0; nc<_ncols; nc++)
      result._M[nr][nc] *= x;
  return result;
}
//-------------------------------------------------------
template <class T>
Matrix<T> Matrix<T>::operator/ (const T& x) const
{
  Matrix<T> result(*this);
  for (long nr=0; nr<_nrows; nr++)
    for (long nc=0; nc<_ncols; nc++)
      result._M[nr][nc] /= x;
  return result;
}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator+= (const Matrix<T>& M)
{*this = *this+M;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator-= (const Matrix<T>& M)
{*this = *this-M;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator*= (const Matrix<T>& M)
{*this = *this*M;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator%= (const Matrix<T>& M)
{*this = *this%M;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator+= (const Vector<T>& v)
{*this = *this+v;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator-= (const Vector<T>& v)
{*this = *this-v;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator+= (const T& x)
{*this = *this+x;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator-= (const T& x)
{*this = *this+x;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator*= (const T& x)
{*this = *this+x;}
//-------------------------------------------------------
template <class T>
inline void Matrix<T>::operator/= (const T& x)
```

```
{*this = *this+x;}
//-------------------------------------------------------
// Friend operators
/*
template <class T>
Matrix<T> operator% (const Vector<T>& v1,const Vector<T>& v2)
{
  long lv1=v1.length();
  long lv2=v2.length();

  Matrix<T> result(lv1,lv2);
  for (long nr=0; nr<lv1; nr++)
    for (long nc=0; nc<lv2; nc++)
      result._M[nr][nc] = v1[nr]*v2[nc];
  return result;
}
//-------------------------------------------------------
template <class T>
inline Matrix<T> operator+ (const T& x,const Matrix<T>& M)
{return M+x;}
//-------------------------------------------------------
template <class T>
Matrix<T> operator- (const T& x,const Matrix<T>& M)
{
  Matrix<T> result(M._nrows,M._ncols);
  for (long nr=0; nr<M._nrows; nr++)
    for (long nc=0; nc<M._ncols; nc++)
      result.M[nr][nc] = x - M._M[nr][nc];
  return result;
}
//-------------------------------------------------------
template <class T>
inline Matrix<T> operator* (const T& x,const Matrix<T>& M)
{return M*x;}
//-------------------------------------------------------
template <class T>
Vector<T> operator* (const Vector<T>& v,const Matrix<T>& M)
{
  #if EXCEPTIONHANDLING
    if (M._nrows!=v.length())
      throw IllegalOperation("operator*(Vector,Matrix)");
  #endif
  Vector<T> result(_ncols);
  for (long nc=0; nc<_ncols; nc++)
  {
    T sum=0;
    for (long nr=0; nr<_nrows; nr++)
      sum += v[nr]*M[nr][nc];
    result[nc]=sum;
  }
  return result;
}
//-------------------------------------------------------
template <class T>
```

```
istream& operator>> (istream& is, Matrix<T>& M)
{
  for  (long nr=0; nr<M._nrows; nr++)
    for (long nc=0; nc<M._ncols; nc++)
       is >> M[nr][nc];
  return is;
}
//-------------------------------------------------------
template <class T>
ostream& operator<< (ostream& os, const Matrix<T>& M)
{
  for (long nr=0; nr<M._nrows; nr++)
  {
    os << "| ";
    for (long nc=0; nc<M._ncols; nc++)
    {
      os.precision(M._precision); os.width(M._width);
      os << M._M[nr][nc] << ' ';
    }
    os << "|" << endl;
  }
  return os;
}
*/
template <class T>
Matrix<T> operator% (const Vector<T>& v1,const Vector<T>& v2)
{
  long lv1=v1.length();
  long lv2=v2.length();

  Matrix<T> result(lv1,lv2);
  for (long nr=0; nr<lv1; nr++)
    for (long nc=0; nc<lv2; nc++)
      result[nr][nc] = v1[nr]*v2[nc];
  return result;
}
//-------------------------------------------------------
template <class T>
inline Matrix<T> operator+ (const T& x,const Matrix<T>& M)
{return M+x;}
//-------------------------------------------------------
template <class T>
Matrix<T> operator- (const T& x,const Matrix<T>& M)
{
  Matrix<T> result(M._nrows,M._ncols);
  for (long nr=0; nr<M._nrows; nr++)
    for (long nc=0; nc<M._ncols; nc++)
      result[nr][nc] = x - M[nr][nc];
  return result;
}
//-------------------------------------------------------
template <class T>
inline Matrix<T> operator* (const T& x,const Matrix<T>& M)
{return M*x;}
```

```
//-------------------------------------------------------
template <class T>
Vector<T> operator* (const Vector<T>& v,const Matrix<T>& M)
{
  #if EXCEPTIONHANDLING
    if (M._nrows!=v.length())
      throw IllegalOperation("operator*(Vector,Matrix)");
  #endif
  Vector<T> result(_ncols);
  for (long nc=0; nc<_ncols; nc++)
  {
    T sum=0;
    for (long nr=0; nr<_nrows; nr++)
      sum += v[nr]*M[nr][nc];
    result[nc]=sum;
  }
  return result;
}
//-------------------------------------------------------
template <class T>
istream& operator>> (istream& is, Matrix<T>& M)
{
  for  (long nr=0; nr<M._nrows; nr++)
    for (long nc=0; nc<M._ncols; nc++)
       is >> M[nr][nc];
  return is;
}
//-------------------------------------------------------
template <class T>
ostream& operator<< (ostream& os, const Matrix<T>& M)
{
  for (long nr=0; nr<M.rows(); nr++)
  {
    os << "| ";
    for (long nc=0; nc<M.cols(); nc++)
    {
      os.precision(M.outputprecision()); os.width(M.outputwidth());
      os << M[nr][nc] << ' ';
    }
    os << "|" << endl;
  }
  return os;
}
```

5.4 Exercises

E:5.1 Write a linear regression program (linear least squares fit) which fits a straight line through a set of data points. Use the vector class where appropriate.

E:5.2 Show that the Kronecker product (5.7) obeys equations (5.9), (5.10) and (5.11).

Show further that

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{Ac} \otimes \mathbf{BD} \tag{5.15}$$

by comparing both sides of the equation. What are the required relationships between the dimensions of the matrices?

E:5.3 Add a method `kroneckerpower(n)` to the matrix class which returns the n'th Kronecker power of the matrix. The 2'nd and 3'rd Kronecker powers are for example given by

$$\mathbf{A}^{[2]} = \mathbf{A} \otimes \mathbf{A} \qquad \mathbf{A}^{[3]} = \mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A} \tag{5.16}$$

and prove that $(\mathbf{AB})^{[k]} = \mathbf{A}^{[k]}\mathbf{B}^{[k]}$.

Chapter 6

Packaging via Namespaces

Namespaces in C++ perrform the same function as packages in UML or Java and modules in CORBA. They provide a mechanism for hierachical naming. But why do we need this?

6.1 NameSpace Pollution

So far we declared all our classes and functions within global scope. One can also define objects and variables at global scope, though we have refrained from doing this.

One of the problems with declaring elements at global scope is that one has to somehow ensure that eac name is globally unique. If one used two libraries which both define a verb+Date+ class at global scope – this would not be that unlikely – then the compiler could not distinguish between the two `Date` classes if both libraries were used simultaneously.

6.1.1 Unique Naming

Prior to the ANSI/C++ standard which introduced the concept of namespaces, the approach was to insert a prefix (or append a postfix) into each name, hoping that this aproach would result in unique names and avoid the polution of the global namespace. For example, if we defined a `Date` class, we could have called the class

```
class SolmsTraining_Date
{
  ...
};
```

But this may not be enough to avoid name clashes. Particularly in large organizations, where there are multiple development teams, one might want to define further nested scopes for the individual development teams.

For example, both, the front-end and back-end development teams may define their own date/time class, `Date`. In this case one may ant to introduce a further level in the naming:

```
class OrganizationName_FrontEnd_Date
{
  ...
};


class OrganizationName_BackEnd_SolmsTraining_Date
{
  ...
};
```

The resultant names become very long and the code becomes less and less readable.

6.2 Defining NameSpaces

A namespace is in C++ a conceptual organization of the global namespace into a hierachical structure which is conceptually very similar to UML or Java packages. As with Java, one should always put ones classes and functions into namespaces.

To put a elements into a namespace one uses the `namespace` keyword:

```
namespace SolmsTraining
{
  const double PI = 3.14159265359;

template <class T> bubbleSort(T array[], const int length) { ... }

template <class T> class Matrix
{
   ...
};
}
```

6.2.1 Namespaces are Cumulative

If we define the same namespace at different locations, the contents of the namespace will be the sum total of all the elements defined at the various locations for that namespace. Thus

```
namespace SolmsTraining
{
  const double PI = 3.14159265359;
}

namespace TimLewis
{
  void convertGifToJPeg(istraem& gifStream, ostream& jpegStream) {...};
}

namespace SolmsTraining
```

```
{
template <class T> bubbleSort(T array[], const int length) { ... }

template <class T> class Matrix
{
   ...
};
}
```

would assign the same 3 elements to the `SolmsTraining` namespace as the previous example.

6.2.2 Namespaces Span Accross Files

As with Java packages, namespace elements are typically defined in multiple files. However, unlike Java's package mechanism, C++ implies no mapping between the namespace structure and the directory hierarchy.

Conversely, a file may contain elements assigned to multiple namespaces.

6.3 Using Elements Defined in Other NameSpaces

You can either access an element from another name space through a fully qualified name or by importing it via a `using` clause. For example, if we wanted to refer to a matrix class generated from the `Matrix` template defined in the `SolmsTraining` namespace, we could do sovia

```
SolmsTraining::Matrix<double> m(20,20);
```

or we could import the name into the current name spece via the following using statement:

```
using SolmsTraining::Matrix;
using SolmsTraining::PI;

Matrix<double> m(20,20);

m[0][0]= 2.1*PI;
```

6.3.1 Importing Entire Namespaces

At times you want to have direct access to all elements defined in a namespace without importing the elements individually. This is particularly useful when reworking existing code which did not make use of namespaces to use elements which have now been packaged within namespaces. To import an entire namespace – i.e. all elements from that namespace, one adds the `namespace` keyword after the `using` keyword:

```
using namespace SolmsTraining;

Matrix<double> m(20,20);

m[0][0]= 2.1*PI;
```

6.4 Hierarchical Naming via Nested NameSpaces

One would typically want to use a hierarchical naming along development team boundaries or, more sensibly, along application domain boundaries. Thus, to encourage re-use accross projects and accross team boundaries, the packaging should not be done along either of these criteria. Instead you would want to have a packaging hierarchy which packages related objects together, irrespective of the project they have been developed for or the team which developed them.

An example of elements from a packaging hierarchy which follows the along the lines of a clean conceptual structure is shown below:

```
SolmsTraining::Maths::LinearAlgebra::Vector<T>;
SolmsTraining::Maths::LinearAlgebra::Matrix<T>;
...
SolmsTraining::Maths::Numeric::Integrate::SimpsonIntegrator;
SolmsTraining::Maths::Numeric::Integrate::RombergIntegrator;
...
SolmsTraining::Utils::DateTime::Date;
SolmsTraining::Utils::DateTime::TimePeriod;
SolmsTraining::Utils::DateTime::AnchoredTimePeriod;
```

Such a hierarchy is conceptually clean and with the support of a decent documentation generation tool it can simplify the search for components – it facilitates a natural search guided by once requirements.

For globally unique naming one may want to prefix the namespace hierarchy with the inverted domain name of the organization, e.g. `za::co::SolmsTraining::Maths`.

6.5 Splitting Implementation from Header

When splitting the implementation from the header by writing seperate `cpp` and `h` files, one should keep in mind that within the implementation file one has to do one of the following:

- Import the names for the elements whose implementation is specified in the file.
- Import the entire namespace(s).
- Use fully qualified names.

6.6 Anonymous Namespaces

At times one wants to hide certain elements within a single file. Such elements are used by other elements in the file but should are not generally useful. Furthermore, we want to ensure that the name used for these elements is save from name-clashes with similarly names elements defined in other files.

In this case one may want to introduce an anonymous namespace. This is in some ways similar to Java's `package` access-level specification, just that here the access is restricted to a particular file:

For example, when writing a collection of sorting routines we may want to define a `swap` function and we may want to ensure that the this function is distinguished from any other swap function which may defines in another library we are using. Furthermore, we do not want to publish this function for general use because we want to be at liberty to modify it as required by the sorting algorithms it serves.

In this case we may want to put our `swap` function into an anonymous namespace as follows:

```
namespace
{
  template <class T> void swap (T& x, T& y);
}

template <class T> void bubbleSort(T* array);
template <class T> void mergeSort(T* array);
template <class T> void quickSort(T* array);
```

6.7 An Example

Let us look at an example where we put an account class and a client class into seperate namespaces and at a little example application which uses these two classes.

Account.h

The account class is put into a nested namespace, `SolmsTraining::finance`:

```
#ifndef AccountH
#define AccountH
namespace SolmsTraining
{
 namespace finance
 {
  class Account
  {
    public:
      Account();
      Account(double initialBalance);
```

```
      ~Account();

      void credit(double amount);
      void debit(double amount);

      double balance();

    private:
      double _balance;
  };
 }
}

#endif
```

Account.cpp

In the implementation file we use fully qualified names to resolve the packaged items:

```
#include "Account.h"
#include <iostream>

using namespace std;

SolmsTraining::finance::Account::Account()
  : _balance(50) {}

SolmsTraining::finance::Account::Account(double initialBalance)
  : _balance(initialBalance) {}

SolmsTraining::finance::Account::~Account()
{
  cout << "I, " << this << ", am destroyed." << endl;
}

void SolmsTraining::finance::Account::credit(double amount)
{
  _balance += amount;
}

void SolmsTraining::finance::Account::debit(double amount)
{
  _balance -= amount;
}

double SolmsTraining::finance::Account::balance() {return _balance;}
```

Client.h

Similarly, the client class is put into a nested namespace, `SolmsTraining::clients`. It uses the `Account` class and avoids having to use fully qualified naming for it by importing it via a `using` satement:

```
#ifndef Client_H
#define Client_H

#include "Person.h"
#include "Account.h"

namespace SolmsTraining
{
 namespace clients
 {
  using SolmsTraining::finance::Account;

  class Client: public virtual Person
  {
    public:
      Client(const string& name, const string& idNo,
             Account& account);

      Account& account() const;

    private:
      Account& _account;
  };
 }
}
#endif
```

Client.cpp

In this implementation file we totally avoid using fully qualified names by importing both, the `Account` class from the `SolmsTraining::finance` namespace and the `Client` class from the `SolmsTraining::clients` namespace:

```
#include "Client.h"

using SolmsTraining::finance::Account;
using SolmsTraining::clients::Client;

Client::Client (const string& name, const string& idNo, Account& account)
          : Person(name, idNo), _account(account) {}

Account& Client::account() const
{
  return _account;
```

```
}
```

TestNameSpace.cpp

Finally we use these classes in the following example application:

```
#include <iostream>

#include "Client.h"
#include "Account.h"

using namespace std;

int main()
{
  using SolmsTraining::clients::Client;

  SolmsTraining::finance::Account* acc
    = new SolmsTraining::finance::Account();

  acc->credit(380);

  Client* client1 = new Client("Abduhl", "6754236754", *acc);

  cout << client1->name() << " has a balance of R"
       << client1->account().balance() << endl;

  char c; cin >> c;

  return 0;
}
```

6.8 Finding Classes

Packaging can help significantly in finding classes. One should make some effort in simplifying the search process for existing functionality, not only because one can avoid the cost of re-developing the functionality, but also because it will help reduce the code bulk which has to be maintained over time (as well as the size of executables). Even tasks like testing and performance tuning are simpler with less code bulk.

Packaging can help significantly in simplifying the search for existing functionality, especially if the package hierarchy is along the lines of functionality and responsibility.

Documentation tools may present class documentation along in a hierarchical form which mirrors the package structure.

One may also want to use Java's idea of mapping the package (namespace) hierarchy onto a directory hierarchy. This may also help to simply the search for components.

6.9 Exercises

Decide on a sensible package hierarchy for the elements of the exercise of chapter 4 and insert the components into the hierarchy. Test that your application still compiles and runs.

Chapter 7

Error Handling Techniques

7.1 Introduction

It is a good idea to separate a program into distinct subsystems that either complete successfully or fail. Thus, a limited form of local error checking should be implemented throughout the system in such a way that the overheads in development time, execution time, memory requirements and overall complexity do not become too high. A fault-tolerant system should be designed hierarchically, with each level coping with as many errors as it can (without making the system too contorted) and leaving the remaining errors to be handled by higher levels of the system.

One generally has however the problem that the error cannot be handled locally. Note that the user of your class library knows what to do when an error occurs, but he does not know how to detect an error in your classes. The author of the class library knows how to detect errors, but he does not know how the error should be handled. We thus need a global error communication system. Typically this would be one of the following:

- Error state = return value of functions and object services.
- Classes with error state variables.
- Exception handling.

7.2 Error Communication via Return Values

In the past it was commmon to communicate that the requested service could not be supplied by a return value returned by the service. Typically this was an integer which may have been zero if the service was supplied as requested and some other value otherwise. The value would supply some information about the source of the problem.

This scenario may be ok if your direct client (i.e. the object from which the service was requested) is the one who always takes over the responsibility of handling the

situation. Often this is, however, not the case. In many cases the error information has to be propagated up several layers in the calling hierarchy.

In such situations the return-value mechanism becomes very difficult to manage. Any of these services may generate its own code and to keep track of where the error originated from becomes virtually impossible when error codes are returned.

This can be partially addressed by returning errfor reporting objects instead of simple error codes. These objects could contain the information about the source of the problem and any other relevant information.

But even this can become a little burdensome. The error reporting objects have to be passed manually up the calling hierarchy and error checking must be done manually after each service request.

7.3 Error Communication via Error States

Sometimes the information about the problem is stored within the service provider object itself. This is typically done via instance members which can be queried by the client after the return from the service. If an error occurs the error state variable set to a value indicating error type.

For example, the standard IOstream library of C++ has integer state variable **state**. The different bits of this state variable indicate different error states. The error state is queried via the class methods **bad()**, **eof()**, **fail()**. The user can query the error state and implement his own error handling mechanism.

This mechanism does not scale well at all. Firstly, the same object may provide services to a wide range of clients and typically it would only keep track of the last error state. Personally I prefer even the return value mechanism to the error state mechanism.

7.4 Error Communication via Exceptions

A more sophisticated method facilitating the split in responsibilities between error detection and error handling is provided by C++'s support for exception handling. Exceptions handling mechanisms are non-local. They provide a means of communicating the reason for not supplying a service oor the fact that a problem has been encountered while providing the service to the client who requested the service.

7.4.1 What is an Exceptional Situation?

A server encouters an exceptional situation when it does not know when it cannot supply the requested service. For example, if you want to withdraw funds from an account and your account cannot provide the service because there are insufficient funds in the account or because it has been frozen due to some legal dispute, then the account object would throw an exception, notifying you as client that it cannot supply the service.

7.4.2 What does a client do with an Exception?

Assume you requested a withdrawel from your account and that the account did not oblige, but threw an exception instead.

If you ignore the exception, the person who asked you for the cash (you client) will be notified. In other words exceptions are passed up the client-server hierarchy (or the call-hierarchy).

On the other hand you could decide to catch the exeption. When you catch an exception you have to provide the exception handling code. Here you could specify that if your withdrawel from account A failed that you will try and withdraw from account B or that it is time to visit your father in law or whatever.

7.4.3 Throwing and Catching Exceptions

We shall first look at the problem detection which occors at the server side and then we shall focus on the exception handling on the client side.

7.4.4 Creating Exception Classes

In *C++* you can throw an instance of any class when an exceptional situation arises. Typically you would define a simple class exception class which may, at times, carry no information beyond the information conveyed through its identity. Such a class would simply be defined as an empty class (of course the compiler will provide default and copy constructors as well as an assignment operator and a `this` pointer).

For example, we could define an `InsufficientFunds` exception simply as follows:

```
class InsufficientFunds {};
```

Exceptions which carry additional information

Often one would like to add further information to an exception class. Since any class can be used for exceptions, one can add any attributes as well as any services. One should however refrain from adding any attributes or services which are unrelated to the core purpose of exceptions, that of conveying information about the reasons why the requested service could not be supplied.

For example, we could define an insufficient funds exception which carries information about the account which raised the exception as well as the funds available in that account:

```
class InsufficientFunds
{
  public:
    InsufficientFunds(Account* const source,
                      const double availableFunds)
       : _source(source), _availableFunds(availableFunds) {}

    Account* const getSource() const {return _source;}
```

```
    double availableFunds() {return _availableFunds;}

  private:
    Account* const _source;
    const double _availableFunds;
};
```

7.4.5 The Server Side: Throwing Exceptions

The server detects a situation where it cannot complete the requested service and throws an exception. For example, in the `debit()` method of our `Account` class we check for sufficient funds.

```
class Account
{
  public:
  ...
  void debit(double amount)
  {
    if (amount > balance)
      throw InsufficientFunds(this, _balance);
    else
      balance -= amount;
  }
  ...
}
```

When the problem is detected we instantiate the relevant Exception class and throw the exception via java's `throw` keyword. Note that the function announces that it may throw an `IllegalArgumentException`. This is done with a `throws` clause in the method header.

This is all the server does. What could he do more anyway? The control is transferred at the `throw` clause from the server to the client – the remainder of the method body is skipped.

Note that when the exception is thrown the server code is exited, i.e. the remaining statements are skipped.

Exception Notification

C++ also has support for exception notification. For example, if you want to notify the clients of the `Account` class that the debit service is not provided unconditionally, i.e. that under certain circumstances it throws an `InsufficientFunds` exception, then you add a `throw` clause to the method header:

```
class Account
{
  public:
```

```
   ...
    void debit(double amount) throw (InsufficientFunds);
  ...
}
```

In the above listing we declare that the `Account` class may raise an `InsufficientFunds` exception. In other words, that the `debit` service is not supplied unconditionally. We are also saying that this service will NOT raise any other exception.

If a service raises multiple exceptions, these are simply inserted into the brackets. For example, if we constrain the amount which may be withdrawn on a single day from an account, we may specify the `throw` clause of the `debit` service as follows:

```
class Account
{
  public:
   ...
    void debit(double amount) throw (InsufficientFunds, DailyLimitExceeded);
  ...
}
```

We may also want to explicitly promise that a particular service is provided unconditionally. In this case the `throw` clause is followed by an empty bracket:

```
class Account
{
  public:
   ...
    void credit(double amount) throw ();
  ...
}
```

Exception Notification Violations

Assume a method or function promises in a `throw` clause not to throw any exceptions or only to throw certain exceptions. What happens if it in practice does throw another exception. This will only be known at run-time. The function `unexpected()` defined in the C++ standard library which, by default, will terminate the application.

7.4.6 The Client Side: Catching Exceptions

The client makes use of the services offered by the server by sending messages to it. The service may complete successfully or the server may throw an exception.

The client may or may not be able to handle the exception. If the client is not able to handle the exception it will be passed up the client server hierarchy (calling hierarchy) until, hopefully, there is a point where there is enough information to handle the problem.

If a client is willing to catch an exception, it puts the service request message within a `try` block:

```
class TheClientClass
{
  public;
    ...
    void aClientMethod()
  {
    ...
    try
    {
      account.debit(amount);  // this service may throw an exception
      ...
    }
    catch (InsufficientFunds e)
    {
      /* Exception handling code comes here.
         Sorry, this is still your baby. */
    }
    ...
  }
};
```

If an exception is thrown control is transferred from the service-request statement to the corresponding catch clause and, depending on the content of the catch clause, execution continuous with the statements following the catch clause.

Note that if the client makes use of services which may through an exception, and if the client does not catch the exception, the client's method must notify users that it may thow that exception, even though it does not do so explicitly:

```
class TheClientClass
{
  public:
    ...
    void aClientMethod() throw (InsufficientFunds)
  {
    ...
    account.debit(amount);  // this service may throw the exception
    ...
  }
}
```

7.4.7 A Complete Example

Let us now look at the complete code for the `Account` example:

Account.h

```
#ifndef AccountH
#define AccountH
```

```
namespace SolmsTraining
{
 namespace finance
 {
  class Account; // Forward declaration of Account

  class InsufficientFunds
  {
    public:
      InsufficientFunds(Account* const source,
                        const double availableFunds)
         : _source(source), _availableFunds(availableFunds) {}

      Account* const getSource() const {return _source;}

      double availableFunds() {return _availableFunds;}

    private:
      Account* const _source;
      const double _availableFunds;
  };

  class Account
  {
    public:
      Account();
      Account(double initialBalance);

      ~Account();

      void credit(double amount);
      void debit(double amount) throw (InsufficientFunds);

      double balance();

    private:
      double _balance;
  };
 }
}

#endif
```

Account.cpp

```
#include "Account.h"
#include <iostream>

using namespace SolmsTraining::finance;
```

```
using namespace std;

Account::Account(): _balance(50) {}

Account::Account(double initialBalance): _balance(initialBalance) {}

Account::~Account()
{
  cout << "I, " << this << ", am destroyed." << endl;
}

void Account::credit(double amount)
{
  _balance += amount;
}

void Account::debit(double amount)
          throw (InsufficientFunds)
{
  if (amount > _balance)
    throw InsufficientFunds(this, _balance);
  _balance -= amount;
}

double Account::balance() {return _balance;}
```

TestException1.cpp

```
#include <iostream>
#include "Account.h"

using namespace SolmsTraining::finance;

using namespace std;

int main()
{
  Account* myAccount = new Account();
  myAccount->credit(2000);

  Account* myUnclesAccount = new Account(3000);

  while (true)
  {
    try
    {
      myAccount->debit(350);
      cout << "Debited my account. Balance: " << myAccount->balance() << endl;
```

```
    }
    catch (InsufficientFunds exception)
    {
      cout << "my Account only has R"
           << exception.availableFunds() << " left."
           << " Debited uncle instead." << endl;
      try
      {
        myUnclesAccount->debit(350);
      }
      catch (InsufficientFunds e)
      {
        cout << "Out of luck. Uncle broke too." << endl;
        return -1;
      }
    }
  }
}
```

Running the application yields the following ouput:

```
Debited my account. Balance: 1700
Debited my account. Balance: 1350
Debited my account. Balance: 1000
Debited my account. Balance: 650
Debited my account. Balance: 300
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
my Account only has R300 left. Debited uncle instead.
Out of luck. Uncle broke too.
```

7.4.8 Catching Exceptions at various levels of Abstraction

In order to be able to handle exceptions at different levels of abstraction we have to introduce a class hierarchy for exceptions. For example, we may feel want to introduce the concept of an exception through a base class `Exception`. We could then define a specialized `TransactionFailed` exception with further specializetions like `InsufficientFunds` and `DailyLimitExceeded` exceptions:

```
class Exception {};

namespace finance
{
  class TransactionFailed: public Exception {};
```

```
  class InsufficientFunds: public TransactionFailed {};

  class DailyLimitExceeded: public TransactionFailed {};

  class AuthorizationFailed: public TransactionFailed {};
}
```

Once we have done this we can handle exceptions at different levels of abstraction. Consider, for example, the following code:

```
try
{
  account.debit(amount);
}
catch (InsufficientFunds)
{
  /* handle this problem one way */
}
catch (TransactionFailed)
{
  /* Handle all other TransactionFailed exceptions this way */
}
catch (Exception)
{
  /* Handle all other exceptions yet another way. */
}
```

Catching all Exceptions

C++ also supports a `catch` clause which catches all exceptions irrespective of whether they are part of the same class hierarchy or not. To achieve this one inserts 3 dots into the arguments brackets of the catch clause:

```
try
{
  account.debit(amount);
}
catch (InsufficientFunds)
{
  /* Handle this problem one way. */
}
catch (...)
{
  /* Catch anything else which is thrown in this way. */
}
```

7.4.9 Partial Handling of an Exception

Assume that we want to perform some action upon an exception being thrown by a service we requested, but that we cannot handle it completey (i.e. cannot decide how to resolve the problem completely). In such a situation we can catch the exception, perform some action in the `catch` block and either rethrow the same exception or throw another exception from within the catch block. In this way we notify the users of our method that we still have not resolved the problem completely.

C++ has a special syntax for rethrowing the same exception:

```
try
{
  /* do some IO */
}
catch (EndOfFile)
{
  /* Process data. */
}
catch (...)
{
  /* Close files and rethrow. */

  ifsteam.close();

  throw;
}
```

7.5 Throwing Type Declarations

C++ also allows you to throw primitives like enumerated integers:

```
class Account
{
  private:
    enum State {success, noFunds, authorizationFailed, limitExceeded};
    State state = success;

  public:
    void debit(amount)
    {
      ...
      if (amount > _balance)
        state = noFunds;
      ...
      if (state != success)
        throw state;
      ...
   }
};
```

7.5.1 Defining Exception Classes as Nested Classes

It is often a good idea to package the exception class together with the class which may raise the exception.

For example for a **Rational** class developed may raise a **DivideByZero** exception if the denominator of a **Rational** object becomes zero and a `OutOfRange` exception if the numerator or denominator falls outside the range of the `long` data type:

```
class Rational
{
  public:
    Rational (const long int Numer, const long int Denom) ;
    ...
    long int numerator();
    long int denominator();
    ...
    Rational operator + (const Rational& r)
    ...
    class DivideByZero {};     // exception class for /0 error
    class OutOfRange   {};     // exception class for range error
  private:
    long int numer, denom;
}
```

The calling program could catch these exceptions in the following way

```
void main()
{
  ...
  try
  {
    long int a, b, c, d;
    ...
    calc1(a,b);
    calc2(c,d);
    ...
    Rational r1(a,b), r2(c,d);
    Rational r3 = r1+r2;
  }
  catch (Rational::DevideByZero)
  {
     ...       // error handler for /0 errors
  }
  catch (Rational::OutOfRange)
  {
     ...       // error handler for range errors
  }
  ...
}
```

Chapter 8

The Standard Template Library

8.1 Introduction and Overview

The Standard Template Library (STL) is C++'s collection class library. It supports most classical data structures like linked lists, maps, queues and automatically resizing arrays.

The Java 2 Collection Framework itself is largely based on the STL. Though it is in some ways more powerful, it is in other ways less general than the STL.

The library consists of a collection of heavily parametrized classes and functions. It is built around 3 core pillars:

Containers stores the actual elements and may provide methods for accessing elements.

Iterators are generalizations of pointers. They can be used to step through a collection and to retrieve, modify or insert an element at the position they are currently pointing to,

Algorithms are general operations which may be performed on containers. This includes searching, sorting, copying, filling and more.

8.1.1 Some Core Design Decisions

A lot of effort has been put into the design of the STL and many of these design ideas have since been directly taken over by other collection class libraries like the Java 2 Collection Framework.

The design is centered around interfaces

Like the Java 2 Collection Framework, the STL is designed around interfaces and not around the implementation classes.

Algorithms are defined seperately from the container classes

The STL follows an un-usual design from an object-oriented perspective and this design has been mirrored in the Java 2 Collection Framework in that the algorithms are supplied as stand-alone functions and not as instance methods of the container classes.

Iterators are modelled as specializations of pointers

One of the design decisions made for the STL is to define iterators as specializations of pointers, i.e. iterators support dereferencing as well as pointer arithmetic. This makes it possible that the algorithms are equally applicable to the collection classes and to primitive pointer-based arrays.

8.2 Containers

The Standard Template Library supplies the following container implementations:

vector – a random access container based on an automatically resizing array.

list – a random access container based on a doubly-linked circular list implementation.

hashset – a sorted collection guaranteeing that each element is only contained once. It is typically based on a balanced binary tree implementation.

set – a sorted collection guaranteeing that each element is conly contained once. It is typically based on a balanced binary tree implementation.

map – a container which maps keys onto values. A map keeps the keys in sorted order.

hash-map – a container which maps keys onto values. The keys are not kept in sorted order.

The STL container hierarchy is quite elaborate. However, looking at the hierarchy for collections (vectors, lists and sets) and maps one can clearly see how the Java 2 Collection Framework is a simplification of the STL.

All these collection classes are heavily parametrized. We shall discuss a few of them below.

8.2.1 The vector container type

The vector template class supplied by the STL is not really a true vector in the mathematical sense but a automatically resizing array structure.

It takes one compulsory template argument, the data type, and one optional template arguments, an allocator which provides a facility to assign your own low-level mechanism for memory allocation and de-allocation.

```
template <class T, class Alloc> vector
```

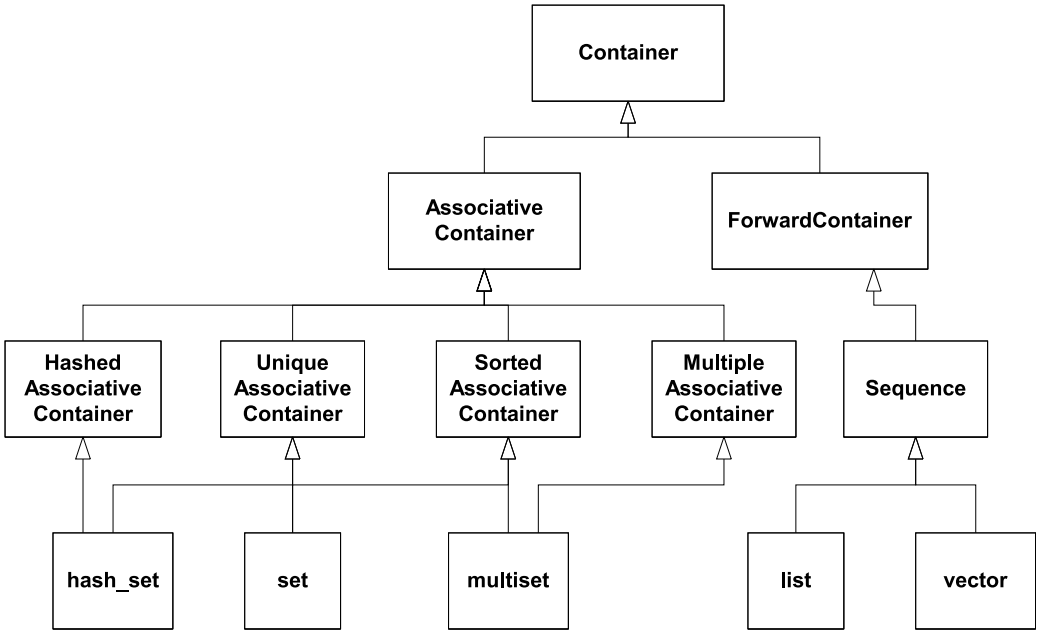

Figure 8.1: The class/interface hierarchy leading to the STL collections (sets, list and vector).

In nearly all practical cases the default allocator is used and the second template argument is ommited. This to create an empty vector of floating point numbers on the stack you could write the following statement:

```
vector<double> vec1;
```

If you want to specify an initial size of 20 elements you could write the following statement:

```
vector<double> vec2(20);
```

8.2.2 An example program using a vector

```
#include <vector>
#include <iostream>

#include "Person.h"

using namespace std;

void print(vector<double>& v)
{
```

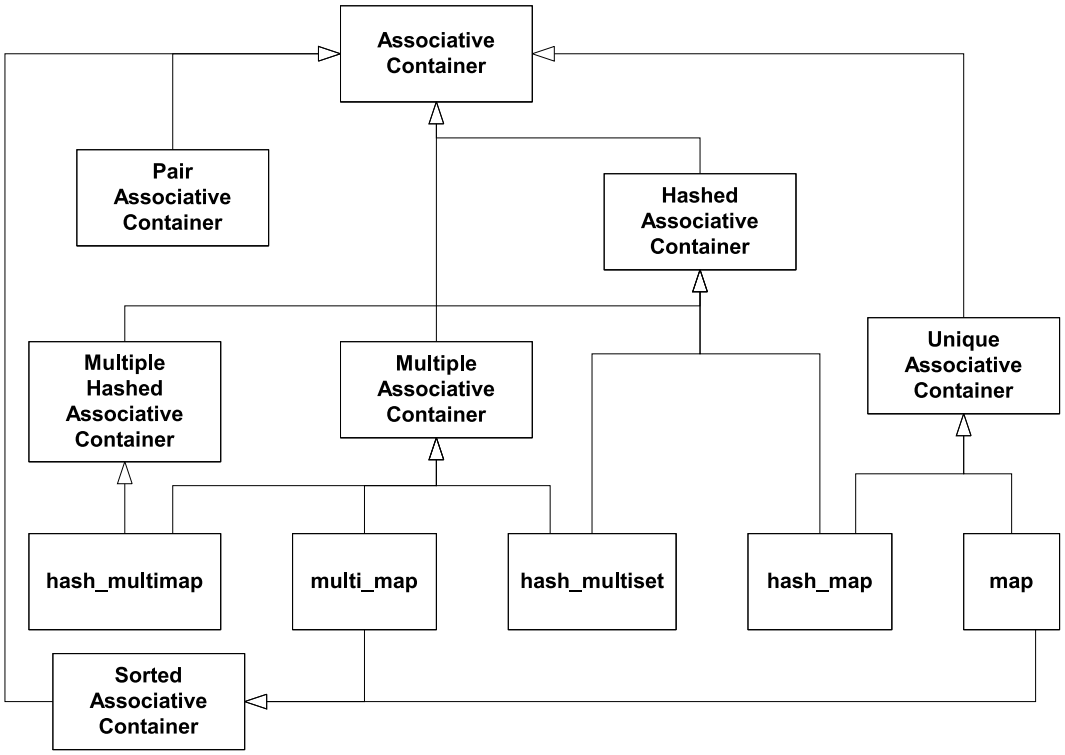

Figure 8.2: The class/interface hierarchy leading to the STL maps.

```
  cout << "v = [ ";
  for (vector<double>::iterator iter = v.begin();
       iter != v.end(); ++iter)
           cout << (*iter) << " ";
  cout << "]" << endl;
}

int main()
{
  vector<double> v(20);
  v[0] = 11.1;   v[10] = 1.2;
  v[1] = 11.5;   v[11] = 2.2;
  v[2] = 8.1;    v[12] = 2.1;
  v[3] = 9.11;   v[13] = 1.3;
  v[4] = 6.35;   v[14] = 4.2;
  v[5] = 11.1;   v[15] = 1.3;
  v[6] = 8.45;   v[16] = 4.1;
  v[7] = 9.67;   v[17] = 1.2;
  v[8] = 9.76;   v[18] = 1.7;
  v[9] = 9.45;   v[19] = 6.3;
```

```
  cout << "Before sorting: " << endl;
  print(v);

  sort(v.begin(), v.end());

  cout << "After sorting: " << endl;
  print(v);

  v.insert(v.begin(), -999);

  v.insert(v.end(), 999);

  cout << "After inserting: " << endl;
  print(v);

  vector<Person*> persons(3);
  persons[0] = new Person("Jack Hill", "6541654");
  persons[1] = new Person("Jill Crack", "6641654");
  persons[2] = new Person("Jack Jillian", "6741654");
  for (vector<Person*>::iterator iter = persons.begin();
       iter != persons.end(); ++iter)
           cout << (**iter) << " ";
  cout << endl;

  double* dvec = new double[3];
  dvec[0] = 2.1;
  dvec[1] = 1.2;
  dvec[2] = 3.1;

  double* end = &dvec[2];

  sort(dvec, end);

  cout << dvec[0] << " " << dvec[1] << " " << dvec[2] << endl;

  char c; cin >> c;

  return 0;
}
```

8.2.3 An example program using a set and a map

```
#include <iostream>
#include <map>
#include <set>
#include <string>

#include <stdlib.h>
```

```
#include "Account.h"
#include "Person.h"

using SolmsTraining::finance::Account;

using namespace std;

struct PersonComparator
{
  bool operator()(const Person * const p1, const Person * p2)
  {
    return p1->name() < p2->name();
  }
};

int main()
{
  Person* p = new Person("Jacky de Lill", "6142354");

  set<Person*, PersonComparator> persons;
  persons.insert(new Person("Jack Hill", "6541654"));
  persons.insert(p);
  persons.insert(new Person("Jaco Mill", "6323254"));

  map<Person*, Account*> clientsAccounts;

  int accNo = 101;
  for (set<Person*, PersonComparator>::iterator iter = persons.begin();
       iter != persons.end(); ++iter)
  {
    const string accNo("Acc:");
    cout << "X" << accNo << endl;
    clientsAccounts[*iter] = new Account(accNo, 1000);
  }

  clientsAccounts[p]->credit(2000);

  for (set<Person*, PersonComparator>::iterator iter = persons.begin();
       iter != persons.end(); ++iter)
          cout << "bal: " << clientsAccounts[*iter]->balance() << endl;

  for (map<Person*, Account*>::iterator iter = clientsAccounts.begin();
       iter != clientsAccounts.end(); ++iter)
  {
    Person* p = iter->first;
    Account* acc = iter->second;
    cout << *p << "  =>  " << *acc <<  endl;
  }
```

```
  return 0;
}
```

8.3 Iterators

InputIterator s provide read access to the container elements via `x = *iter;`

OutputIterator s provide write access to the container elements via `*iter = x;`

ForwardIterator supports traversal of the container in the forward direction via the pointer increment operator, `++`.

ReverseIterator supports traversal of the container in the reverse direction via the pointer decrement operator, `--`.

RandomAccessIterator s provide random access to containers which support random access (e.g. `list` and `vector`) via the element access operator, e.g. `x = iter[index];`

8.4 Algorithms

The STL defines generic algorithms seperately from the container classes. These algorithms are applicable to

- the STL container classes themselves,
- any other container implementation of the STL interfaces,
- primitive pointer-based arrays.

Some of the commonly used algorithms include

- `copy(InputIterator first, InputIterator last, OutputIterator result)`
- `copy(InputIterator first, InputIterator last, OutputIterator result)`
- `fill(InputIterator first, InputIterator last, const T& value)`
- `find(InputIterator first, const EqualityComparable& value)`
- `sort(RandomAccessIterator first, RandomAccessIterator last)`
- `stable_sort(RandomAccessIterator first, RandomAccessIterator last)`
- `binarySerach(ForwardIterator first, const LessThanComparable& value)`