A2X - C++ Course 1998

C++ Course 1998

Thomas Papanikolaou

e-mail: papanik@math.u-bordeaux.fr

WWW: http://www.math.u-bordeaux.fr/~papanik

Last change: November 27, 1998

A2X - C++ Course 1998

Contents

Introduction 4

I Programming Paradigms 5

I-1 Unstructured Programming . . . 5
I-2 Procedural Programming . . . 5
I-2.1 Examples of procedural programming . . . 6
I-2.2 Languages supporting Procedural Programming . . . 7
I-3 Modular Programming . . . 7
I-3.1 Examples of Modular Programming . . . 7
I-3.2 Languages supporting Modular Programming . . . 9
I-4 Object-Based Programming . . . 10
I-4.1 Examples of Object-Based Programming . . . 12
I-5 Object-Oriented Programming . . . 13
I-6 Summary . . . 17
I-7 Exercises . . . 17

II Support for Object-Based Programming I 19

II-1 Basics . . . 19
II-1.1 Defining types . . . 19
II-1.2 Access control . . . 20
II-1.3 A schematic class definition . . . 20
II-2 Initialization and Cleanup . . . 21
II-2.1 The scope operator . . . 22
II-3 Assignment and Initialization . . . 23
II-4 Accessors and Modifiers . . . 24
II-4.1 The **const** keyword . . . 24
II-5 Operators . . . 26
II-5.1 References to objects . . . 27
II-5.2 Passing values to functions . . . 28
II-6 Friends . . . 29
II-6.1 Type conversions . . . 30

A2X - C++ Course 1998

II-6.2 Explicit type conversions . . . 33
II-7 Statics . . . 34
II-7.1 Static data members . . . 34
II-7.2 Static member functions . . . 35
II-8 Summary . . . 36
II-9 Exercises . . . 37

III Support for Object-Based Programming II 38
III-1 Parameterized types . . . 38
III-1.1 Template instantiation . . . 39
III-2 Parameterized functions . . . 40
III-3 Parameterized functions vs. macros . . . 40
III-3.1 Inlining . . . 41
III-4 Template specialization . . . 42
III-5 Exercises . . . 44

IV Object-Oriented Concepts 45
IV-1 Classes and objects . . . 45
IV-2 Relationships among classes and objects . . . 45
IV-2.1 Physical . . . 45
IV-2.2 Conceptual . . . 46
IV-3 Inheritance . . . 47
IV-3.1 Single Inheritance . . . 47
IV-3.2 Multiple Inheritance . . . 52
IV-4 Exercises . . . 54

V Exception handling 55
V-1 Handling exceptions in C . . . 55
V-2 The C++ method . . . 56
V-2.1 Syntax . . . 56
V-2.2 Example: Safe division of integers . . . 56
V-2.3 Multiple throws and catches . . . 58
V-2.4 Handling uncatched exceptions . . . 59
V-3 Exercises . . . 59

A The C++ Language 60
A-1 Built-in types . . . 60
A-1.1 Booleans . . . 60
A-1.2 Integers . . . 60
A-1.3 Reals . . . 61

A-1.4 Exercises 61
A-2 Constants 61
A-3 Input and Output 62
A-4 Overloading operators 63
A-5 Free-store management 63

B Implementation of a parameterized type 65
B-1 Specification 65
B-1.1 Definition 65
B-1.2 Prequisites 65
B-1.3 Public interface 66
B-1.4 Private interface 67
B-1.5 Further relevant utilities 67
B-2 Implementation 68
B-2.1 Constructors & destructor 68
B-2.2 Modifiers 70
B-2.3 Assignment 70
B-2.4 Index operator 71
B-2.5 I/O 72
B-3 Usage 73

Bibliography 75

Readings 76

Index 77

A2X - C++ Course 1998

A2X - C++ Course 1998

Introduction

This script contains the lectures given in the A2X C++ course in 1998 [1]. This course is an introduction to C++ for *programmers*, that is people who have already worked with some kind of programming language like PASCAL, C, etc. However, learning C++ is done *by example* rather than *by definition*, so it is also possible that non-programmers will be able to use this script as a tutorial.
The whole course is based in Bjarne Stroustrup's famous paper on OOP [3]. I have kept the same organization (and copy pasted much of the text) but also added examples, reorganized sections and provided more information, since Bjarne Stroustrup's paper is not aimed to teach C++ (although you cannot resist learning C++ after you have read this). In any case, I am the one to receive your criticism for any errors you find in this script but *please* do this in a constructive way.

Enjoy!

Thomas Papanikolaou

How to use this text

This text is neither a complete resource for C++ nor it covers the whole C++ language. The main goal is to introduce C++ smoothly, and help you learn *how* to use it correctly without requiring too many prequisites. You will learn (thinking) C++ by reading, doing, imitating, just as you learn a foreign language. At the beginning you may not understand the notation, but, frankly, did you know anything about grammar as you begun to speak?
A warning: teaching a programming language is a rather complicated task and it is likely that I occasionally miss some important issue. Therefore I highly recommend that beside reading this text, you should use a standard book like the one of B. Stroustrup [2] and S. Lippman [1]. To my personal opinion, you should have these books anyway.

[1] The author is supported by the french C.N.R.S.

A2X - C++ Course 1998

Programming Paradigms

> Object-oriented programming (OOP) is a technique for programming – a paradigm for writing *good* programs for a set of problems.
>
> Bjarne Stroustrup

This sentence points out the most common misunderstanding among programmers concerning OOP: using an object-oriented language to program, does not automatically imply that the resulting programs are object-oriented. A mathematician would say that using an OOP language is a *necessary*, but it is not a *sufficient* condition for OOP.
In this lecture we will illustrate the different programming paradigms. Following Stroustrup [3], we will also point out the distinction between languages which *enable* OOP and languages which *support* OOP.

I-1 Unstructured Programming

This is the programming style usually applied for writing small applications or utilities. Such applications consist of a single program containing a sequence of statements for manipulating data. This data is globally visible throughout the whole program. Unstructured programs are sometimes believed to be efficient and short-living.

I-2 Procedural Programming

Procedural Programming (PP) is the original (and probably still widely used) programming paradigm. In this paradigm, programs are organized mainly according to the algorithms they use: each of these algorithms is the implemented as a separate *procedure*. In this way, common sequences of processing steps are grouped together. Therefore, programming using the PP paradigm can be defined as

A2X - C++ Course 1998

1. identifying which algorithms you need
2. implementing them in procedures
3. writing a main program which defines the order in which procedures will be executed

I-2.1 Examples of procedural programming

One typical example of the PP style is the `sqrt` routine. Given a real argument `x` it returns the square root of `x` using the well known Newton iteration. A possible implementation in C would be:

```
double sqrt(double x)
  {
    double sqrt_x;
    // code to perform the Newton iteration ...
    return sqrt_x;
  }
```

We may then use the `sqrt` routine in some *procedure* in our program:

```
void foobar()
  {
     double y = sqrt(2);
     ...
     // no return statement
  }
```

A more complicated usage example is the *recursive* routine `factorial`. Given an integer `n`, it returns `n!`:

```
int factorial(int n)
  {
    if (n == 1)
      return 1;
    else
      return n * factorial(n - 1);
  }
```

A2X - C++ Course 1998

I-2.2 Languages supporting Procedural Programming

The original procedural language is Fortran. Algol, C and Pascal are later inventions in the same tradition. All these languages provide methods for passing arguments to functions and returning values from functions, distinguishing different kinds of arguments, different kinds of functions (procedures, routines, macros, ...) etc. In this sense all these languages are said to *support* the PP paradigm, since they provide sufficient language facilities which make it convenient (easy, safe and efficient) to use this programming style.

I-3 Modular Programming

As computers became more powerful and programs more complex, the emphasis of the programming paradigm moved over from the design of procedures to the organization of data. The new programming paradigm evolved in this way views data and the procedures manipulating this data as a unit (a so called *module*).
Programming with the Modular Programming paradigm (MP) involves the following steps:

1. partition the program into modules
2. for each module provide a *user interface*; ensure that the module data can only be accessed through this user interface (this is often called the *data hiding* principle).
3. ensure that the module is initialized before its first use.

I-3.1 Examples of Modular Programming

The most common example is a definition of a character `stack` module. We begin by providing the user interface for this module. Following the C convention, we will store this definition into the *header file* `stack.h`:

Definition of the `stack` user interface

```
// File: stack.h

// maximum number of elements that can be stored
const int stack_size = 100;
```

```
// returns true is the stack is empty, false otherwise
int stack_is_empty();

// returns the top element of the stack and removes it
char pop();

// puts a new element on the top of the stack
void push(char);
```

Implementation of the `stack` module

Now we implement the *hidden* part of the module. Again, Following the C convention, we will put the implementation into the *source file* `stack.cc`:

```
#include "stack.h" // include the stack definition file

// we represent the stack by an array of size stack_size
static char the_stack[stack_size];

// the_top will always point at the top of the stack
static char* the_top;

int stack_is_empty()
  {
    return the_top == the_stack;
  }

// check for underflow and pop
char pop()
  {
    if (p < the_stack)
       error("the character stack underflows!");
    else
      {
        char c = *the_top;
        the_top--;
        return c;
      }
  }
```

A2X - C++ Course 1998

A2X - C++ Course 1998

```
// check for overflow and push
void push(char c)
  {
    // ...
  }
```

Note that the functions provided as interface for the stack give no information about the actual *representation* we have just used for storing the stack into memory. Thus, we could replace the array by a singly linked list or by any other appropriate data structure without changing anything in the programs using the stack.
Why is it impossible for the user to access this representation? The answer is simple: by declaring `the_stack` and `the_top` to be **static**, we make these variables *local* to the file / module in which they *live*. Such a stack may be used like this:

```
#include "stack.h"

void some_function()
  {
    if (stack_is_empty()) push('+');
    char c = pop();
    if (c != '+') error("impossible");
  }
```

I-3.2 Languages supporting Modular Programming

Pascal (as originally defined) does not provide any satisfactory facilities for hiding representations for the rest of the program: the only way for hiding a name is to make it local to a procedure. However, this leads to strange procedure nestings and over-reliance on global data.
As we have seen, in C you are able to use the MP paradigm by implementing one module per source file, and to hide the representation data by declaring them **static**. Thus, the programmer has better possibilities to achieve a higher degree of modularity. Nevertheless, in C there is no explicit support for modules such as, for example, given by Modula-2 (a successor of Pascal). In Modula-2 the concept of the module is a central language construct. Modula-2 provides well defined module declarations, explicit control of the scopes of names (import/export), a module initialization mechanism, and a set of generally known and accepted styles of usage.
In this sense, C and Pascal *enable* modular programming, whereas Modula-2 *supports* it.

A2X - C++ Course 1998

I-4 Object-Based Programming

In the previous section we have seen how we can implement a stack of characters in C, using the MP paradigm. Our implementation provided a *single* stack of a *fixed* size. How can we extend our implementation to handle *multiple* stacks of *arbitrary* size? These are two questions at one time, so let us start with the easiest. Consider again our stack definition on section I-3.1. We will extend it in such a way, that each interface function *knows* on which stack it operates:

```
// File: multiple_stack.h

// stack_id will reference a stack
// its definition is left out for simplicity

// returns true is stack_id is empty, false otherwise
int stack_is_empty(stack_id);

// returns the top element of stack_id and removes it
char pop(stack_id);

// puts a new element on the top of stack_id
void push(stack_id, char);
```

Now for being able to create and destroy a stack of a given size, we introduce two functions:

```
// create a stack_id of a give size
stack_id create_stack(int);

// destroy the stack_id
void destroy_stack(stack_id);
```

Our new definition allows us now to use multiple stacks of arbitrary size in the way illustrated by the following function:

```
void some_function()
  {
    stack_id s;
    s = create_stack(200);
    push(s, 'a');
```

```
    char c = pop(s);
    if (c != 'a') error("impossible");
    destroy_stack(s);
}
```

This multiple stack implementation is definitely a great improvement comparing to our first effort. Nevertheless it has two major disadvantages: at first, before using a stack, we have to initialize it explicitly with the number of elements we want to handle. This is done by calling `create_stack(200)`. What happens if we forget to call this function? Probably, the following `push` call will result in a program abort. Similarly, we have to free the memory allocated *by hand* or we get a memory-leak in our program. In this sense, `stack_id` behaves in no way like a built-in type and enjoys support inferior to the support provided for built-in types. The compiler has no chance in detecting the (severe) errors mentioned above.
This problem is solved by languages like Ada or C++ which allow the user to define types that behave in (nearly) the same way as built-in types. Such a type is often called an *Abstract Data Type* (ADT) or a *User-Defined Type* (UDT) .
For example, in C++, the `stack` type could be implemented like this:

```
class stack
  {
     char *the_stack;
     char *the_top;

     public:

     // construct a stack of size n
     stack(int n) { the_top = the_stack = new char[n]; }

     // destruct a stack
     ~stack() { delete[] the_stack; }

     int is_empty() const
       {
          return the_top == the_stack;
       }
     void push(char c) { ... }
     char pop() { ... }
  };
```

A2X - C++ Course 1998

A2X - C++ Course 1998

Using this type would be much more easier and safe:

```
void some_other_function()
  {
     stack s(100); // stack(int n) is called here

     s.push('+');
     char c = s.pop();
     if (c != 'a') error("impossible");

     // ~stack() is called here automatically
  }
```

Programming with ADTs is often called *Object-based Programming* (OBP) or programming using the *Data Hiding* paradigm (DHP). Using the DHP paradigm involves the following steps:

1. decide which types you want
2. specify the representation for each type
3. provide a full set of operations for each type

This looks quite similar to the MP paradigm, and this is natural, since ADTs are generalizations of modules. A class can easily simulate a module, by providing a mechanism which ensures the existence of only one object (*instance*) of this class. Consequently, where there is no need for more that one object of a type the Data Hiding programming style using modules suffices.

I-4.1 Examples of Object-Based Programming

Arithmetic types such as `rational` and complex numbers are common examples of user-defined types:

```
class rational
  {
     private:
     int num, den;

     public:
     rational()              { num = 0; den = 1; }
```

```
    rational(int n)        { num = n; den = 1; }
    rational(int n, int d) { ... }
    ~rational()            { } // yes, it is empty

    friend rational operator + (rational x, rational y);
    friend rational operator - (rational x, rational y);
    friend rational operator * (rational x, rational y);
    friend rational operator / (rational x, rational y);
    ...
  };
```

The declaration of **class** (that is, user-defined type) `rational` specifies the representation of a rational number (`num`, `den`) and the set of operations on a rational number (construction, destruction, arithmetic, etc). The representation fo `rational` is **private**; that is, `num` and `den` are accessible only to the functions specified in the declaration of the rational class[1]. Such functions can be defined like this:

```
rational operator + ( rational x, rational y )
  {
    return rational ( x.num * y.den + x.den * y.num,
                   // -----------------------------
                                x.den * y.den);
  }
```

and used like this:

```
rational x = 3;
rational y = 1 / x;
y += rational(2, 3); // means y = y + 2/3;
...
```

I-5 Object-Oriented Programming

An ADT is a sort of black box. Once it has been defined it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. Consider defining a type `shape` for use in a graphics system. Assume for the moment that your system has to

[1] In C one uses the **struct** keyword to group types together to new types. A **struct** allow allows accessing of its elements. It can be therefore viewed as a **class** having only **public** elements.

A2X - C++ Course 1998

A2X - C++ Course 1998

support circles, triangles and squares. Assume also that you have two classes `point` and `color`:

```
class point{ /* ... */ };
class color{ /* ... */ };
```

You might define a class `shape` like this:

```
// enumerate the different types of shapes
enum kind { CIRCLE, TRIANGLE, SQUARE };

class shape
  {
    private:

    point center;
    color col;
    kind k; // representation of shape

    public:

    point where()  { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations ...
  };
```

The type field `k` is necessary to allow operations such as `draw()` and `rotate()` to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag `k`). The function `draw()` might be defined like this:

```
void shape::draw()
  {
    switch (k)
      {
        case CIRCLE:   // draw a circle
          break;
        case TRIANGLE: // draw a triangle
          break;
```

A2X - C++ Course 1998

```
        case SQUARE:   // draw a square
          break;
      }
  }
```

Do you see the problems of this approach? First of all functions such as `draw()` must *know about* all the kinds of shapes there are. Therefore the code for any such function grows each time a new shape is added to the system. If you define a new shape, every operation on a shape must be examined and (possibly) modified. You are not able to add a new shape to a system unless you have access to the source code for every operation. Since adding a new shape involves *touching* the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type shape.

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. A language with constructs that allows this distinction to be expressed and used supports object-oriented programming. Other languages don't.

The Simula and C++ inheritance mechanism provides a solution. First, specify a class that defines the general properties of all `shapes`:

```
class shape
  {
     private:

     point center;
     color col;
     // ...

     public:

     point where() { return center; }
     void move(point to) { center = to; draw(); }
     virtual void draw();
     virtual void rotate(int);
     // ...
```

A2X - C++ Course 1998

```
  };
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked **virtual** (the Simula and C++ term for *may be redefined later in a class derived from this one*). Given this definition, we can write general functions manipulating shapes. For example the following function rotates all members of a vector `v` of size `size`, `angle` degrees:

```
void rotate_all(shape* v, int size, int angle)
  {
    for (int i = 0; i < size; i++) v[i].rotate(angle);
  }
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the **virtual** functions):

```
class circle : public shape
  {
     private:

     int radius;

     public:

     void draw() { /* ... */ };
     void rotate(int) {}  // yes, the null function
  };
```

In C++, class `circle` is said to be *derived* from class `shape`, and class `shape` is said to be a *base class* of class `circle`. An alternative terminology calls `circle` and `shape` *subclass* and *superclass*, respectively.
The Object-Oriented Programming paradigm (OOP) can be now defined as follows:

1. decide which classes you want
2. provide a full set of operations for each class
3. make commonality explicit by using inheritance

A2X - C++ Course 1998

Note that where there is no commonality, the OBP paradigm suffices. If one should use the OBP or the OOP paradigm depends on the amount of commonality that can be exploited between types used in an application area. Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When designing a system, commonality must be actively sought, both by designing classes specifically as building blocks for other types, and by examining classes to see if they exhibit similarities that can be exploited in a common base class.

I-6 Summary

In this chapter we presented the paradigms for procedural, modular, object-based and object-oriented programming. Each of these paradigms is a natural evolution of another in the sense of *knowledge grouping*. In the procedural paradigm we use procedures to group together common sequences of statements. In modular programming we regard data *and* the procedures manipulating this data as a unit. Object-based programming introduces the requirement of used-defined types behaving like built-in types. Finally, the object-oriented paradigm introduces the distinction between the common and the specific properties of types.
From a philosophical/designer's point of view each new paradigm adds more *intelligence* to the language, thus requiring a more powerful syntax. Whereas a procedure just groups statements, activated by a procedure call, a module *knows* about the data it manipulates and therefore requires some sort of initialization. An ADT also knows which functions are allowed to manipulate data, thus requires access control and automatic initialization and destruction for allowing built-in-type behavior. Finally, the (most intelligent) OO-type inherits some of its knowledge from a base class and needs only to (re)define its specific operations. Both actions (inheritance, redefinition) have to be supported.
In the following chapters we will describe the language mechanisms provided by C++ to support object-oriented programming.

I-7 Exercises

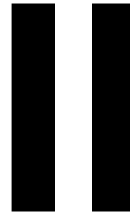
1. Complete the implementation of `stack` and `multiple_stack`. Try out what happens if one pushes a character into a stack which has not been yet initialized.

2. What are the conceptual differences between the various programming paradigms?

3. Take an example from your life (i.e. making coffee) and try to express it using the different programming paradigms. For example the procedural version of making coffee could look like this: get coffee, get sugar, get water, mix, cook.

A2X - C++ Course 1998

A2X - C++ Course 1998



Support for Object-Based Programming I

In the previous chapter we defined Object-Based Programming by the following three steps:

1. decide which types you want
2. specify the representation for each type
3. provide a full set of operations for each type.

The basic support for programming with data abstraction consists therefore of language facilities for defining types, defining a set of operations (functions and operators[1]) for these types, and for restricting access to objects of these types, to that set of operations. In this chapter we will introduce the language mechanisms of C++ for supporting data abstraction.

II-1 Basics

II-1.1 Defining types

User defined types are introduced in C++ by the **class** keyword. For example, the simplest type one can build is the `empty` one:

```
class empty
{
  // empty
};
```

[1]Remember that we require types to behave like built-in types.

A2X - C++ Course 1998

II-1.2 Access control

As required by the object-based paradigm, C++ provides explicit access control for restricting access to parts of a class. This is done via the **private** and **public** keywords[2]. For example the definition of a new type `foo` having a hidden `part1` and a visible `part2` may be implemented like this[3]:

```
class X  // introduces new type X
{
  private: // hidden part1 begins here
           // hidden part1 ends here
  public:  // visible part2 begins here
           // visible part2 ends here
};
```

II-1.3 A schematic class definition

In general, the definition of a C++ **class** hides the representation of the type and internal utilities which are of no interest to the user. The visible part includes functions to create and destruct an object and functions to read or modify the properties of an object. Further, it includes implementations of operators, and probably other functions. Summarizing, the general outline will be

```
class prototype
{
  public:
    // initializing
    // clean-up
    // assignment
    // accessors
    // modifiers
    // operators
    // friends
  private:
    // representation
    // utilities
};
```

In the following sections we will study the parts of this `prototype` class in detail.

[2] C++ also provides a **protected** keyword; we will explain its usage in a later chapter.
[3] It is possible to have more than one **private** or **public** sections in a class definition.

II-2 Initialization and Cleanup

When the representation of a type is hidden some mechanism must be provided for a user to initialize variables of that type. C++ allows the designer of a type to provide a distinguished function to do the initialization. Given this function, allocation and initialization of a variable becomes a single operation (often called *instantiation* or *construction*) instead of two separate operations. Such an initialization function is called a *constructor*. Additionally, in cases where construction of objects of a type is non-trivial, a complementary operation to clean up objects after their last use is provided. In C++, this cleanup function is called a *destructor*. Consider for example a `vector` type:

```
class vector
{
  private:
    int    sz;    // number of elements
    int* data;    // pointer to integers
  public:
    vector(int); // constructor
    ~vector();    // destructor
    // ...
};
```

The `vector` constructor can be defined to allocate space like this:

```
vector::vector(int s)
{
  if (s <= 0) error("bad vector size");
  sz = s;
  data = new int[s]; // allocate an array of "s" integers
}
```

The `vector` destructor frees the storage used:

```
vector::~vector()
{
  delete[] data;
}
```

A2X - C++ Course 1998

C++ does not support garbage collection. This is compensated for, however, by enabling a type to maintain its own storage management without requiring intervention by a user[4].

II-2.1 The scope operator

In the implementations of the constructor and destructor in the previous section we have used the *scope* operator `::` to denote that `vector(int)` is a member function of `vector`. In general, a member function `foo` of a class `X` can be declared in two ways. The first declares *and* implements `foo` within the class `X`:

```
class X
{
  // ...
  public:
    void foo()
    {
      // implementation
    }
};
```

The second way (which we have used for declaring the constructor and destructor) declares `foo` within the class `X` but implements it outside the class `X`:

```
class X
{
  // ...
  public:
    void foo();
};

void X::foo()
{
  // implementation
}
```

[4]This is a common use for the constructor / destructor mechanism, but many uses of this mechanism are unrelated to storage management.

A2X - C++ Course 1998

A2X - C++ Course 1998

II-3 Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many types, but not for all. It can also be necessary to control all copy operations. Consider again the class `vector`:

```
vector v1(100);
vector v2 = v1; // make a new vector v2 initialized to v1
v1 = v2;        // assign v2 to v1
```

In C++ it is possible to define the meaning of the initialization of `v2` and the assignment to `v1`. For example,

```
class vector
{
  private:
    int    sz; // number of elements
    int* data; // pointer to integers
  public:
    // ...
    vector(const vector &);           // initialization
    void operator = (const vector &); // assignment
};
```

specifies that user defined operations are to provided for implementing initialization and assignment of `vector` objects. Assignment may be implemented like this:

```
void vector::operator=(const vector& a)
{
   if (sz != a.sz) error("bad vector sizes");
   for (int i = 0; i < sz; i++)
     data[i] = a.data[i];
}
```

Since the assignment operation relies on the *old value* of the vector being assigned to, the initialization operation must be different. For example:

```
// initialize a vector from another vector
vector::vector(const vector& a)
{
  sz = a.sz;                   // same size
```

```
  data = new int[sz];          // allocate element array
  for (int i = 0; i < sz; i++) // and copy elements
    data[i] = a.data[i];
}
```

In C++ a constructor of the form `X(const X &)` is called a *copy constructor*. In addition to explicit initialization copy, constructors are used to handle arguments passed *by value* and function return values. It is possible to prohibit assignment of an object to another by declaring the assignment **private**:

```
class X
{
  private:
    X(const X &);
    void operator = (const X &);
  public:
    // ...
};
```

II-4 Accessors and Modifiers

The operations on objects either read or modify some property of an object. Read access is a non critical operation in that sense that it leaves the object unchanged. Thus, the object *consistency* remains invariant after an access operation. On the other hand, modifications are much more critical. For example, introducing a bug to a method modifying an object almost always results in a program crash, because data is corrupted. Depending on the program involved this may be connected to significant loss of data and money.
Therefore it is an important issue to be able to distinguish between *accessors*[5] and *modifiers*. The more language support we get for this, the better.

II-4.1 The const keyword

In C++ we can distinguish between constant (not-modifiable) and non-constant (modifiable) variables using the **const** keyword. For example the following sequence will cause a compiler error:

[5] I use the *accessor* as a synonym for *non-modifying reader*.

A2X - C++ Course 1998

A2X - C++ Course 1998


```
const int x = 3; // x is initialized to 3
x = 4;           // compiler error: x is const
...
```

Recall now the `vector` class definition. We may extend it by a `size` method, which returns the size of the allocated space for a `vector`. This method does modify the object, therefore it is also declared **const**:

```
int size() const { return sz; }
```

In this sense a *constant object* is an object which cannot be changed after its initialization, whereas a *constant method* is a method which does not modify the object properties, and therefore can act on constant objects.
A typical example of a modifier method is `set_size`, which reallocates memory for a given vector. Note the missing **const** in the method declaration:

```
int set_size(int);
```

`set_size` may be implemented like this[6]:

```
int vector::set_size(int new_sz)
{
  int old_sz = sz;
  int *old_data = data;
  if (new_sz < 0) error("illegal size");
  sz = new_sz;
  if (sz == 0)
    delete[] data;
  else
    {
      data = new int [sz];
      for (int i = 0; i < min(old_sz, sz); i++)
        data[i] = old_data[i];
      delete[] old_data;
    }
  return old_sz;
}
```

One may ask why it is necessary to add the **const** keyword to a method like `size` which obviously does not change any `vector` property. The answer is one (or all) of the following:

[6]Presuming we have a function `min` which returns the minimum of two integers.

A2X - C++ Course 1998

- by adding **const** you make clear to the potential reader / maintainer of this class that your design intended to use this function as an accessor. It is good programming style to do this.
- If your implementation of a **const** method accidentally modifies the object, this is caught at compile time and you get an error. Therefore, using **const** leads to less bugs.
- by adding **const** you allow the compiler to use this function on constant objects. In C++ you can not apply a non-constant method on a constant object; if you try it, you get an error from the typing system. Thus, if you use **const**, you add safety to your programs.

II-5 Operators

In section II-3 we provided a new meaning for the assignment `=` operator to enable correct control of assignments of `vector` objects. Such an action is called in C++ *operator overloading*. C++ allows a quite large number of operators to be overloaded by the user like `+`, `-`, `*`, `/`, `!`, etc. to mention a few.
For example a *reading* index operator `[]` for `vector` may be defined like this:

```
class vector
{
  private:
    int    sz; // number of elements
    int* data; // pointer to integers

  public:
    // ...
    int operator [] (int i) const;
};
```

and implemented like this:

```
int vector::operator[] (int i) const
{
  if (i < 0 || i >= sz) error("bad vector index");
  return data[i];
}
```

A2X - C++ Course 1998

A typical use of the index operator is in the implementation of a `norm` function, which computes the sum of the squares of the elements of a vector:

```
int norm(vector v)
{
  int sum = 0;
  for (int i = 0; i < v.size(); i++)
    sum = sum + v[i] * v[i];
  return sum;
}
```

`norm` may be used in a context like this:

```
vector v(100);
// ...
int norm_v = norm(v);
```

II-5.1 References to objects

In the previous sections we have implemented a reading index operator for `vector`. This means that the following will produce a compiler error:

```
vector v(100);
int i = v[1]; // ok
v[1] = 123;   // oops: v[1] is not writable
```

In order to be able to do such an assignment, the index operator must return a *reference* to the vector element being accessed. Therefore, we have to provide an additional *writing* index method:

```
int & operator [] (int i);
```

and implement it like this:

```
int & vector::operator[] (int i)
{
  if (i < 0 || i >= sz) error("bad vector index");
  return data[i];
}
```

A2X - C++ Course 1998

Although the implementation within the braces is exactly the same, there are two differences. The syntactical difference is the addition of & to the return type **int**, and the removal of **const**. The semantical difference is that the reading index operator returns a copy of the i-th element, where the writing index operator returns the i-th element *itself*[7]. With these operators the above example compiles now correctly:

```
vector v(100);
int x = v[1]; // ok
v[1] = 123;   // ok: v[1] is writable
```

Of course the reference concept applies also in the case when we want to return more the one multiple results in a function. Consider for example the following min_max procedure. Given two integers a and b, min_max returns the minimum and the maximum in its first two arguments:

```
void min_max(int & min, int & max, int a, int b)
{
  if (a < b) { min = a; max = b; }
  else { min = b; max = a; }
}
```

min_max can be called like this

```
int x, y;
min_max(x, y, 5, 7);
// ...
```

II-5.2 Passing values to functions

References to objects can be used to avoid expensive copying of large objects. Consider again our norm function:

```
int norm(vector v)
{
  int sum = 0;
  for (int i = 0; i < v.size(); i++)
    sum = sum + v[i] * v[i];
  return sum;
}
```

[7] In C++ terminology we say that the writing index operator returns an *l-value*.

A2X - C++ Course 1998

When a user calls `norm` with a `vector x` the compiler creates a copy of `x` and passes this to `norm`. This is necessary to warranty that the value of `x` is not get lost. Passing arguments to functions in this way is called passing *by value*. Now consider what happens if we call `norm` multiple times with vectors of large size: the program will spend too much time in copying.
In conjunction with **const** C++ provides an equally a safe but dramatically more efficient way to do the same:

```
int norm(const vector & v)
{
  // same code as before
}
```

When `norm` is called with a `vector x`, only a reference to `x` is passed, thus copying is avoided[8]. Adding **const** to the function declaration, ensures that the compiler will not allow modification of the object referenced. This way of passing arguments is called passing *by reference* and it should be used whenever large objects must be passed to functions.

II-6 Friends

In section II-5 we have implemented a `norm` function using the `[]` operator of the class `vector`. Recall its (improved) implementation

```
int norm(const vector & v)
{
  int sum = 0;
  for (int i = 0; i < v.size(); i++)
    sum = sum + v[i] * v[i];
  return sum;
}
```

This is not the only possible implementation. We may also implement `norm` as a member-function:

```
int norm() const
{
  int sum = 0;
```

[8] You can think of a reference as a number indicating where the object's memory begins.

A2X - C++ Course 1998

```
  for (int i = 0; i < sz; i++)
    sum = sum + data[i] * data[i];
  return sum;
}
```

The member-function `norm` can now be called from a user program like this:

```
vector v(100);
// ...
int norm_v = v.norm();
```

What are the differences between the two implementations? At a first glance one notices the different calling syntax: the member-function has to be *applied* on the vector `v` using the `.` operator, whereas our first version in section II-5 gets `v` passed as an argument. From a user point of view, this is more near to the mathematical notation one wants to use.
However, there is an impressive difference in terms of efficiency: whereas the functional version has to call the `[]` operator 200 times[9], the method version does not have to call this operator at all; it operates directly on the `data`.
C++ provides a way to implement a version of `norm` satisfying *both* natural notation and speed by declaring it as **friend**. Using this keyword our implementation of `norm` becomes[10]:

```
friend int norm(const vector & v)
{
  int sum = 0;
  for (int i = 0; i < sz; i++)
    sum = sum + v.data[i] * v.data[i];
  return sum;
}
```

In this sense, `friend` functions are non-class functions with access to the private parts of a class.

II-6.1 Type conversions

A natural question arising from the two different implementations of the function `norm` (member, friend) is which one is the best. The answer is, that the one which fits your

[9]We can eliminate the calls by declaring the `[]` operator as **inline** (we will discuss this later). However, we can not eliminate the range-checking which occurs every time we use the `[]` operator.
[10]We can optimize this even further, but this is not the point here.

A2X - C++ Course 1998

purposes is better for you. To explain this, let us leave the `vector` example behind for a while and recall the implementation of the `rational` class and its `+` operator. Our first try implements this operator as a **friend** of `rational`:

```
class rational
  {
     private:
     int num, den;

     public:
     rational()             { num = 0; den = 1; }
     rational(int n)        { num = n; den = 1; }
     rational(int n, int d) { ... }
     ~rational()            { } // yes, it is empty

     friend rational operator + (rational x, rational y);
     ...
  };
```

With this definition the following compiles correctly:

```
rational x = 1; // line 1: x = rational(1)
rational y = 2; // line 2: y = rational(2)
y = 3 + x;      // line 3: implicit: 3 -> rational(3)
```

This is because the compiler produced an automatic *type conversion*, by applying the constructor `rational(int)` to `3`. Replacing the implementation of the **friend** function by a member function

```
rational operator + (rational x);
```

causes a compiler error at line 3. Why? Consider what the compiler does out of line 3:

```
y = 3 + x; // oops: y = 3.operator+(y);
```

`3` is of type **int** and **int** is not a class.
Another (perhaps even more typical) example of using **friend** is the implementation of mathematical functions. Consider the usage of a member function `inv` which return the inverse of a rational number:

A2X - C++ Course 1998

```
rational rational::inv() const
{
  return rational (den, num);
}
```

`inv` can used like this:

```
rational x = -3;
rational y = x.inv(); // y equals now 1/x
...
```

The same code using the **friend** version would look like this:

```
friend rational inv(const rational & x)
{
  return rational(x.den, x.num);
}
```

and might be used like:

```
rational x = -3;
rational y = inv(x);
...
```

The second version provides a *notation* which you will find in most math books. Therefore, in this case and whenever mixed mode arithmetic in numerical work has to be done (for example, matrices, complexes, etc), the **friend** approach combined with the implicit type conversion mechanism, provide a more natural solution.
On the other side, however, the last example shows why one should take care in using **friend**s and implicit type conversion extensively. Notice, for example, that the **friend** function `inv` has to *know* that a `rational` contains a data member `num`, and this breaks data-hiding. Moreover, consider having two classes `rational` and `real`, both providing a constructor for **int**. The following causes a compiler error:

```
real x = inv(-3);     // oops: don't know which inv to call
rational y = inv(-3); // oops: don't know which inv to call
```

In this case the compiler cannot decide which `inv` function to call, since both type conversion alternatives (`real(int)` and `rational(int)`) are equivalently well. On the contrary, the member version poses no problems:

A2X - C++ Course 1998

```
real x = -3;
rational y = -3;
x = x.abs(); // ok: call real::abs()
y = y.abs(); // ok: call rational::abs()
```

Because of the existence of **friend** C++ can not be viewed as a *pure* object-oriented language. Nevertheless, **friend** allows you to make expressions like the one in line 3 (`y = 3 + x;`) legal without modifying the concept of existing types like **int**. This is not possible in pure object-oriented systems. As a rule of thumb one can say, that **friend**s should be used with care. The following section shows an additional method for controlling type conversions.

II-6.2 Explicit type conversions

We have seen that the implicit type conversion can lead to undesirable effects. The C++ keyword **explicit** provides a method for overcoming such problems. Consider for example a `string` class:

```
class string
{
  public:
    string(int n); // construct a string of capacity "n"
    string(const char *s); // construct a string from "s"
    ...
};
```

and the following user program:

```
string s = "hello"; // ok: s = string("hello")
string t = 'a';     // oops: t = int('a')
```

Since the constructor `string(int)` exists and **char** can be converted to **int** without loss of precision, the compiler applied implicit conversion to the character `'a'` This is definitely not what the user meant. To avoid this, the constructor `string(int)` has to be declared **explicit**:

```
class string
{
  public:
    explicit string(int n);
```

A2X - C++ Course 1998

```
    string(const char *s);
    ...
};
```

The behavior of the user program changes to

```
string s = "hello";     // ok: s = string("hello")
string t = 'a';         // error: cannot apply constructor
                        // implicitly
string u = string('b'); // ok: explicit constructor
```

II-7 Statics

In the previous sections we have seen examples of the classes `vector` and `rational` where each object of that class had its own set of **private** data. This data could be accessed by the class's own set of member functions.
There are however cases, where it is desirable to share a *common* piece of data among objects of a given class. An example of such situation could be a default value for initializing the size of a `vector` or the commonly used constant `rational` numbers `1` and `0`. In the first case, this would enable us to provide an empty constructor for `vector`, in the second case it would probably speedup our computations, since these constants do no have to be constructed each time they are used.

II-7.1 Static data members

C++ allows us to achieve the desired behavior in a very elegant manner, by declaring a data member **static** (global) to all objects of a class. Consider the `vector` class:

```
class vector
{
  private:
    static int default_sz;
    int    sz;
    int *data;

  public:
    vector();
    // ...
};
```

A2X - C++ Course 1998

```
int vector::default_sz = 100; // initialization

vector::vector()
{
  sz = vector::default_sz;
  data = new int[sz];
}
```

The case of the constants `1` and `0` for the class `rational` can be implemented in a similar way:

```
class rational
{
  private:
    int num, den;
  public:
    static const rational one, zero;
    // ...
};

const rational rational::one = 1;
const rational rational::zero = 0;
```

In the `rational` implementation we declared the **static** data members `one` and `zero` to be **public**, in order to allow their usage by the user[11]:

```
vector v; // vector of size vector::default_sz;
rational x = rational::one;
```

static data members is an extension of the C **static** mechanism we have seen in I-3.1: they provide the same functionality while also encapsulating *global* parts of a class.

II-7.2 Static member functions

Beside **static** data, C++ allows the definition of **static** functions. Similar to the concept of **static** data, in which variables are shared by all objects of a class, **static** functions apply to all objects of the class[12].

[11] Notice that `one` and `zero` are defined to be **const**, so that they cannot be accidentally changed by a user.

[12] The **static** functions can therefore address only the **static** data of a class.

A2X - C++ Course 1998

For example, assume we want to change the default initialization size for `vector` through a computation. We can do this by providing the following modifier:

```
class vector
{
  private:
    // ...
  public:
    // ...
  static void set_default_size(int);
};

void vector::set_default_size(int s)
{
  if (s < 1) error("invalid default size");
  vector::default_sz = s;
}
```

This can be used like this:

```
vector::set_default_size(150);
vector x; // vector of 150 integers
vector::set_default_size(100);
vector y; // vector of 100 integers
...
```

Note that as for **static** data, it is possible to declare **static** member functions which are **private** or **public**.

II-8 Summary

In this chapter we have shown how to implement *simple* classes following the Object-based Programming Paradigm with C++.
The **class** keyword is used to introduce a new type. **private** and **public** are the used to hide the representation from the user and provide the interface of this class respectively. Initialization and cleanup of objects is done using *constructors* and, where necessary, a *destructor*. Objects can be initialized by other objects on declaration via the *copy constructor* or in a statement by using the *assignment* operator. This operator has to be appropriately *overloaded* as all other operators which are to be used.

A2X - C++ Course 1998

Member functions which do not modify the state of an object are called *accessors*. Is is good programming style to declare accessors to be **const**. Similarly, the **const** keyword is used to denote data which members cannot be changed after initialization. Functions modifying the state of an object are called *modifiers*. Modification can also be done by providing a *reference* to an object; passing objects to functions *by reference* is also a way to avoid copying of large objects.
Wherever possible, the C++ compiler applies the class constructors implicitly. Such *implicit type conversions* are especially useful in mixed mode arithmetic and in conjunction with **friend** functions.
Finally, it is possible to declare data or functions which are common to all objects of a class using the **static** keyword.

II-9 Exercises

1. Complete the implementation of `vector`. Add a method `sort` which sorts the elements of a vector according to the value of a `direction` **static** member. Let `direction` 1 mean *ascending sort* and `direction` -1 mean *descending sort*. Provide a method `set_direction(int)` to set the sort direction.

2. Write a `complex` class for doing complex arithmetic. Provide constructors to **int**, **double**, and pairs of **double**s. Overload the unary - and the binary +, -, * and / operators. Provide accessors and modifiers for the real and imaginary part.

A2X - C++ Course 1998

Support for Object-Based Programming II

In the previous chapter we have introduced the syntax provided by C++ to support Object-Based Programming and we have shown how one can implement a type `vector` for storing vectors of integers. But, why would you want to define a vector of integers anyway? A user typically needs a vector of elements of *some type* unknown to the writer of the vector type. Consequently the vector type ought to be expressed in such a way that it takes the element type as an argument. The topic of this chapter is to show how to do exactly this with C++.

III-1 Parameterized types

A *parameterized type*is introduced in C++ by the keyword **template**. Consider the `vector` definition:

```
// declare a vector of elements of type T
template <class T> class vector
{
  private:
    int sz;
    T *data;
  public:
    vector(int s)
      {
         if (s <= 0) error("bad vector size");
         data = new T[sz = s];
      }
    vector(const vector<T> & v);
    // ...
```

A2X - C++ Course 1998

```
    T & operator[] (int i);
    int size() const { return sz; }
    // ...
};

// initialize a template vector from another vector
template <class T> vector<T>::vector(const vector<T> & a)
{
  sz = a.sz;                      // same size
  data = new T[sz];               // allocate element array
  for (int i = 0; i < sz; i++) // and copy elements
    data[i] = a.data[i];
}
```

Is it that easy? Yes, it is. Whereas a *simple* class specifies *one* type, a **template** specifies a family of types generated by specifying the **template**'s argument(s) (in our case this is T). The only thing one has to change is the notation[1]. The syntactical differences are:

1. Parameterized classes are introduced by the keyword **template** followed by a list of parameters <class T1, class T2, ..., class Tn>.
2. the new type we create is denoted by vector<T1, T2, ..., Tn>.
3. implementations outside the classes must be preceded by template <class T1, class T2, ..., Tn>.

III-1.1 Template instantiation

Having the above definition, vectors of specific types can now be defined and used as follows:

```
vector<int> v1(100);  // v1 is a vector of 100 integers
vector<rational> v2(200); // v2 is a vector of 200 rational numbers
v2[i] = rational(v1[i], v1[i]);
...
```

This is possible because the compiler replaces the parameter T by the actual type (in a process called *template instantiation*) creating a new type. Therefore in our example the compiler creates vector<int> and vector<rational> by replacing T with **int** and rational respectively.

[1]Compare the syntax shown for the vectors of integers in section II-3.

A2X - C++ Course 1998

Code duplication

Unfortunately, since the types created at instantiation are unrelated to each other, the resulting code may contains two versions of the same function. In our example the resulting code contains two versions of the member function `size` (one for each new type), although this function does not depend on the parameter `T`. This effect is known as *code duplication* and can be considered as a problem with parameterized types. However, it is also the reason for the efficiency of parameterized types[2].

III-2 Parameterized functions

Similarly to types, parameterization can be used to produce a family of functions. In the same way as `vector` we can also implement a parameterized `swap` function:

```
template <class T> void swap(T & a, T & b)
{
  T tmp(a); a = b; b = tmp;
}
```

The actual references `a` and `b` could refer to **int**s, **double**s or to any other type. `swap` may be used like this:

```
int x = 12, y = 15;
swap(x, y);
double a = 12, b = 15;
swap(a, b);
```

As soon as the C++ compiler encounters a usage of `swap`, it generates concrete code: `swap(int, int)` and `swap(double, double)`. The reason that these functions do not cause a name conflict is that their names are changed to contain the types of their arguments[3].

III-3 Parameterized functions vs. macros

It is a natural question: Why are templates different from some fancy macro mechanism? For example the previous `swap` function could be easily rewritten like this:

[2] In well-implemented compilers using templates does not impose any run-time overhead.

[3] This technique is called *mangling* and it is also used to provide unique names for operator overloadings.

A2X - C++ Course 1998

```
#define swap2(type,a,b) { type tmp = (a); (a) = (b); (b) = tmp; }
```

This seems to provide a type-safe solution using the preprocessor **define** keyword. Let us now call both version with a wrong input:

```
int x = 2;
swap(1, x);       // line 1
swap2(int, 1, x); // line 2
```

The GNU C++ compiler produces the following error messages:

```
test.cc:1: bad argument 1 for function 'int swap(int &, int &)'
          (type was int)
test.cc:2: non-lvalue in assignment
```

Do you see the difference? In the first case, the compiler checked the type of the formal arguments of `swap` and found that **int** is not the same as **int** `&`. In the second case the compiler expanded the `swap2` macro to

```
{ int tmp = (1); (1) = (x); (x) = tmp; }
```

and recognized much later that an assignment `(1) = (x);` cannot be done, since `1` is a constant. Thus, templates provide substantially better type-checking.

III-3.1 Inlining

Another aspect which must be considered is that of speed. Macros produce code that is as fast as hand-written, since they expand to real code. It may seem that using templates means loss of efficiency but this is not true. In C++ a function can be declared as **inline**: this keyword instructs the compiler to expand (if possible[4]) the code of the function *after* type-checking has been done. Consider again our `swap` example:

```
template <class T>
inline void swap(T & a, T & b)
{
  T tmp(a); a = b; b = tmp;
}
```

This version is as safe as a function and as fast as a macro. Member functions implemented within the **class** declaration are considered to be **inline**. Member functions implemented outside the **class** declaration have to be explicitly preceded by **inline**.

[4]If a function is too large, than the compiler may choose not to inline it, in order to avoid code-bloating.

A2X - C++ Course 1998

III-4 Template specialization

Templates provide a powerful mechanism for building families of functions or types. However, it is often the case, that specific types which are used as parameters to a template, require special handling. For example, an algorithm for computing the determinant of an integer matrix is different from the algorithm used to compute determinants of real matrices, because of rounding problems, the check for zero, etc.
C++ allows to write specialized template classes and/or to specialize specific functions of a template class. The following example illustrates how this is done. We begin with the declaration of the template class:

```
template <class T> class matrix
{
  public:
    matrix() { }
    ~matrix() { }

    T det() const;
};
```

The first specialization possibility is to rewrite the whole generic class for a special type. This is commonly the case when we replace the generic representation by a specialized one, resulting in rewriting most of the parts of the template type.

```
class matrix<int>
{
  public:
    matrix();
    ~matrix();

    int det() const;
};

matrix<int>::matrix() {}
matrix<int>::~matrix() {}

int matrix<int>::det() const
{
  cout << "matrix<int>::det() called" << endl;
  return 0;
}
```

A2X - C++ Course 1998

The usual case however, is when we *overwrite* some generic implementation of a member function by a specific one:

```
double matrix<double>::det() const
{
  cout << "matrix<double>::det() called" << endl;
  return 0.0;
}
```

Finally we (may) also provide a generic implementation for all other types[5].

```
template <class T>
T matrix<T>::det() const
{
  cout << "matrix<T>::det() called" << endl;
  return T();
}
```

Having all this, the user can use matrices in the usual way:

```
matrix<int> mi;
matrix<double> md;
matrix<char> mc;

mi.det();
md.det();
mc.det();
```

and the output of his program is as expected:

```
matrix<int>::det() called
matrix<double>::det() called
matrix<T>::det() called
```

Note that it is good style to create specializations by first declaring the generic class, then declaring and implementing the specialized parts, and finally implementing the generic algorithms. Older C++ compilers may cause problems if you fail to use this order.

[5]If a generic implementation is omitted, then the user receives a compiler error.

III-5 Exercises

1. Rewrite the `complex` class for doing complex arithmetic as a template class with a parameter `T`. Specialize `complex` numbers over the **float**s.

A2X - C++ Course 1998

A2X - C++ Course 1998

IV

Object-Oriented Concepts

In the previous chapters we introduced the C++ support for object-based programming and generic programming with templates. In our examples (`rational`, `vector`, etc) we constructed new types by combining existing types. In this chapter we will give a more formal definition of the concepts presented till now, present the relationships among classes and objects and introduce the concept of *inheritance*.

IV-1 Classes and objects

A *class* is the implementation of an abstract data type (ADT). It defines *data members* (*attributes*) and *member functions* (also called *methods*) which implement the data structure and the set of operations for the ADT. The later is called the *interface* of the ADT.
Instances of a class are called *objects* and vice versa. An *object* of a class is uniquely identified by its *name*. The values of the object's attributes at some particular time is called the *state* of the object. The set of operations that can be applied on an object defines its *behavior*.
Objects interact with each other by calling methods provided by the class interface. Consequently, classes define properties and behavior of sets of objects.

IV-2 Relationships among classes and objects

IV-2.1 Physical

Recall, the implementation of the `rational` class:

```
class rational
  {
```

A2X - C++ Course 1998

```
    private:
    int num, den;

    public:
    // ...
  };
```

This class definition illustrates two kinds of relationships. The first one is derived by viewing the class `rational` from the outside. We can then say, that the class `rational` *has* two integers data members `num` and `den`. On the other hand, we can also say that the integer data members `num` and `den` are *parts of* the class `rational`. If a class contains objects of another class we therefore speak of a *has-a* relationship. The opposite of this is called the *part-of* relationship.

IV-2.2 Conceptual

Recall the definition of the classes we have used to implement a graphics system in section I-5 for drawing points, circles, etc. A possible implementation of `point` might be:

```
class point
{
  public:
    void set_x(int);    // set the x-coordinate of the point
    void set_y(int);    // set the y-coordinate of the point
    int  get_x() const; // get the x-coordinate of the point
    int  get_y() const; // get the y-coordinate of the point
    // ...
  private:
    int x, y;
};
```

A `circle` implementation might be:

```
class circle
{
  public:
    void set_x(int);    // set the x-coord of the circle center
    void set_y(int);    // set the y-coord of the circle center
    int  get_x() const; // get the x-coord of the circle center
```

A2X - C++ Course 1998

```
    int  get_y() const; // get the y-coord of the circle center
    void set_r(int);    // set the radius of the circle
    int  get_r() const; // set the radius of the circle
    // ...
  private:
    int x, y, r;
};
```

If we compare the two class definitions, then we can easily observe that both classes have common data members (`x` and `y`). In the class `point` these describe the coordinates of the point, in the class `circle` these describe the coordinates of the center of the circle. Thus, both classes use a point to denote their origin / position. This origin can be changed by the common functions `set_x` and `set_y`. Finally, the class `circle` adds a new data member `r`.
For a conceptual point of view, since a `circle` can be used in exactly the same way as a `point`, we can say that a `circle` is *a-kind-of* a `point`. An object of the type `circle` *is-a* `point` which is somehow more specialized, since it also has a *radius*.

IV-3 Inheritance

In the previous section we have given an implementation of a class `point` and a class `circle` in order to illustrate the *a-kind-of* and the *is-a* relationship. In this section we will see how to express the relationship with C++.

IV-3.1 Single Inheritance

Recall again the implementation of `point` and `circle`. From a programmer's point of view it poses a severe problem of code duplication: although the implementations of the interface for setting and getting coordinates are the same, we implemented them twice. Fortunately, this can be done more elegantly in C++ using *inheritance*. In our new implementation `point` remains the same. `circle` is changed to inherit from `point`:

```
class circle : public point
{
  public:
    void set_r(int);       // set the radius of the circle
    int  get_r() const;    // set the radius of the circle
    // ...
```

```
  private:
    int  r;
};
```

The syntax for inheriting from a class is rather simple. The inheriting class is introduced as usual. We then add a colon `:` to denote that we are going to inherit from some type. Then we add the keyword **public** and the name of the class to be inherited (in this case `point`). By writing this we denote that we want to inherit the public interface of `point`[1]. Since we inherit from one class, this C++ mechanism is called *single inheritance*.
With the above declaration the following code becomes valid:

```
Circle c;
c.set_x(10);     // inherited from point
c.set_y(10);     // inherited from point
c.set_r(5);      // added by circle
```

This means, that we can use a `circle` like a `point` and this is what we expect since a `circle` is *a-kind-of* `point`. One expects also that any `circle` object can be used wherever a `point` is expected and this is indeed true:

```
void move_x(point & p, int offset)
{
 point.set_x (p.get_x() + offset);
}

circle c;
move_x(c, 10); // ok: a circle is-a point
```

Inheritance and templates

Inheritance can be also used for template types. Consider for example a template type `vector` which implements dynamic vectors over some type `T`:

```
// declare a vector of elements of type T
template <class T> class vector
{
  public:
   int size() const;
```

[1] It is also possible to inherit **private** or **protected**. We will handle this topic later.

A2X - C++ Course 1998

A2X - C++ Course 1998

```
   T & operator[] (int i);
   const T & operator[] (int i) const;
   // ...
};
```

We can use `vector` to implement a class `polynomial` like this:

```
// declare a vector of elements of type T
template <class T> class polynomial : public vector<T>
{
  public:
   int degree() const { return size() - 1; }
   // ...
   // additional operators +, - , *, etc

};
```

This saves re-implementing the handling for the polynomials coefficients: setting and getting size, setting and getting the i-th element, etc. This is now done by `vector`. We only have to implement the additional methods and operators which are available for polynomials but not for vectors.

Single inheritance support

Simply spoken, what the compiler does in our example is

1. adds the data members of a `point` to `circle`.
2. adds the public member functions of a `point` to `circle`.
3. adds the public operators of a `point` to `circle`.

Internally, the class `circle` looks after compiling like this:

```
class circle
{
  public:
    void set_x(int);    // added by the compiler
    void set_y(int);    // added by the compiler
    int  get_x() const; // added by the compiler
    int  get_y() const; // added by the compiler
    void set_r(int);    // set the radius of the circle
```

A2X - C++ Course 1998

```
    int  get_r() const; // set the radius of the circle
    // ...
  private:
    point { int x, y; } // added by the compiler
    int r;
};
```

Inheritance terminology

Inheritance is a mechanism which allows a class A to inherit properties of a class B. We say *A inherits from B*. Objects of class A thus have access to attributes and methods of class B without the need to redefine them. A is said to be *derived* from B. Alternative notations call A a *child class* or a *subclass* of B. B is called the *parent class* or the *superclass* of A.
A common graphical scheme to represent inheritance relationships uses arrows:

```
  circle ----> point
```

The standard for the direction of the arrows is to point to the superclass.

Conversions from base and superclasses

In the previous example of `circle` and `point` we had no *name collisions* in the interfaces of the two classes. But what happens if a derived class provides a member function which exactly the same definition? For example consider the following *hierarchy* which provides a simple model for driving a machine which is able to cook coffee and/or tea:

```
class liquid_machine
{
  public:
  void cook() const
    { cout << "liquid_machine::cook()" << endl; }

};

class coffee_machine : public liquid_machine
{
  public:
  void cook() const
```

A2X - C++ Course 1998

```
    { cout << "coffee_machine::cook()" << endl; }
};

class tea_machine : public liquid_machine
{
  public:
  void cook() const
    { cout << "tea_machine::cook()" << endl; }
};

void cook_liquid(const liquid_machine & l)
{ l.cook(); }
```

Now consider the following sample program:

```
coffee_machine c;
tea_machine t;
c.cook();
t.cook();
cook_liquid(c);
cook_liquid(t);
```

Could you predict that its output is

```
coffee_machine::cook()
tea_machine::cook()
liquid_machine::cook()
liquid_machine::cook()
```

The member function `cook()` of the derived classes (`coffee_machine`, `tea_machine`) has overwritten the one of the superclass (`liquid_machine`). But what happened when `cook_liquid(c)` was called?
The answer is that the compiler did exactly the opposite as it does in the inheritance case: while converting `coffee_machine` to a `liquid_machine` it removed *all* additional data members and functions that were added on inheritance (a mechanism often denoted as *slicing*), revealing the superclass implementation of the `cook()`. In other words after parameter passing is done, there is nothing there anymore which belongs to a `coffee_machine`. If you find this unnatural, ask yourself how a possibly bigger object of a derived class can be stored into a smaller object of the base class. It can't, since the compiler computed and fixed the sizes of types before execution, in a procedure called *static binding*.

A2X - C++ Course 1998

`virtual` functions

It is however possible to enforce *dynamic binding* of member functions (not data members!) by using the **virtual** keyword. For example `liquid_machine`'s implementation changes to:

```
class liquid_machine
{
  public:
  virtual void cook() const
    { cout << "liquid_machine::cook()" << endl; }
};
```

while the implementation of `cook_liquid()` remains the same:

```
void cook_liquid(const liquid_machine & l)
{ l.cook(); }
```

Having this, the output of our sample program becomes

```
coffee_machine::cook()
tea_machine::cook()
coffee_machine::cook()
tea_machine::cook()
```

This is possible, since the compiler now decides *at runtime* which `cook()` function to call. Using a table of functions pointers to the available implementation of `cook()` it is possible to decide using runtime type information which version to call. Such a table is called *virtual table* and it is stored within each object [2].

IV-3.2 Multiple Inheritance

In our previous example `circle` inherited from *one* other superclass, namely `point`. However, there are situations where it is desirable to inherit from *multiple* (more than one) superclasses. In this case we speak of *multiple inheritance*.
For example consider a `ring` over a domain `T`. A ring is a mathematical structure with two operations `+` and `*`. Concerning `+` a ring forms an abelian group, whereas concerning `*` is simply forms a semigroup. This relatively complex relationship can be easily expressed in C++ by the following implementation:

[2] Virtual tables increase the size of an object by a constant amount multiplied by the number of virtual functions.

A2X - C++ Course 1998

```
template <class T> class abelian_group { /* ... */ };
template <class T> class semi_group { /* ... */ };

template <class T>
class ring : public abelian_group<T>,
             public semi_group<T>
{
 // ...
};
```

With this definition which uses multiple inheritance, we can use a `ring` wherever a `abelian_group` or `semi_group` is required.

Graphically, the above example can be represented as

```
            +----> abelian_group<T>
            |
  ring<T> --+----> semi_group<T>
```

Inheritance terminology (continued)

If a class A inherits from more than one class, i.e. B1, B2, ..., Bn, then we speak of *multiple inheritance*.

`virtual` inheritance

Multiple inheritance should be used with care. Consider for example the following situation

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };
```

which is graphically illustrated by

```
      +----> B ----+
      |            |
  D --+----> C ----+----> A
```

A2X - C++ Course 1998

An object of type `D` inherits twice form `A`, i.e. `D` has two copies of the data members and member functions of `A`. If we replace `A`, `B`, `C` and `D` by some real names we see easily that this is not desirable:

```
class base_vector { /* ... */ };
class sort_vector : public base_vector { /* ... */ };
class math_vector : public base_vector { /* ... */ };
class vector : public sort_vector,
               public math_vector { /* ... */ };
```

There is no reason that each object of type `vector` holds twice the representation of the `base_vector`. In C++ we overcome this situation by using **virtual** inheritance. The above example is rewritten as

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : virtual public B, public C { /* ... */ };
```

The addition of **virtual** ensures that the compiler will produce only one copy of `A` in `D`.

IV-4 Exercises

1. Complete the implementation of `polynomial`. Provide operators for addition, subtraction, multiplication and pseudo-division. Implement a pretty-printing output function.

A2X - C++ Course 1998

V

Exception handling

An *exception* is some unpredicted break in the normal program flow. In C and other programming languages there are several methods for handling such irregular situations: these vary from simply issuing warnings, to a fallback error handling and sophisticated mechanisms realized with `setjmp()` and `longjmp()`. In this chapter we describe the C++ solution to exception handling.

V-1 Handling exceptions in C

Exception handling in C may be realized by the following methods:

1. *Exit*
   Immediate abort of the program using `exit()`. This is the most disastrous way to handle an exception.

2. *Issue a warning*
   A function may notice a problem in a program and issue a message, i.e. the programmer provided some conditional checking for catching minor problems. This is a harmless reaction.

3. *Implement fallbacks*
   Another more sophisticated method may be to stop the execution of a function and delegate the problem (via an error code) to the caller function (the so called *fallback error handling*. Together with the error code one may also return data information, which may allow the caller function to restart computation. The caller function may decide to do the same and so forth. The disadvantage of this approach is that the function interfaces increase in order to be able to pass data information back to their caller. Moreover, it is not possible to continue execution to more then just one level back.

4. *Save stack context*
   Moving execution to another execution depth is possible by using the `setjmp()` and `longjmp()` functions, which operate like a `goto` statement for stack contexts. Because of this reason, their use is not recommended. The Linux manual page notes: `setjmp()` / `longjmp()` *make programs hard to understand and maintain. If possible an alternative should be used.*

V-2 The C++ method

The methods described for C are also available in C++. However they can all be handled in a unified (and more powerful) way using the *exception* mechanism[1]. The following sections discuss the usage of exceptions in C++.

V-2.1 Syntax

C++ added 3 new keywords for exception handling:

1. **try** { ... }
   A **try** block contains statements that may cause an exception to be generated (in C++ terms *thrown*).
2. **throw** `expression`
   the **throw** keyword generates (raises) an exception by throwing the expression value as an exception. **throw** should be executed with a **try** block, or within a function which is called from within a **try** block.
3. **catch** `(expression) { ... }`
   The **catch** block does the exception handling for an exception whose type matches that of `(expression)`. It must follow the **try** block.

V-2.2 Example: Safe division of integers

The following example shows how to use exceptions to avoid loss of precision when dividing two integers `a` and `b`. The idea behind the algorithm is that if the remainder `a % b` is equal to zero, then integer division is safe. Otherwise, real division should

[1]Note that exceptions are a rather new feature in C++ and that many compilers do not yet provide sufficient support for handling exceptions. Other compilers need a special option in order to compile exceptions. For example the GNU `g++` compiler requires the option `-fhandle-exceptions` to enable the use of **try**, **throw** and **catch**.

A2X - C++ Course 1998

A2X - C++ Course 1998

used, in which case we deliver the arguments to a function which does real division. This requires implementing the class

```
#include <iostream.h>

// will be used for raising the exception
class loss_of_precision_in_division
{
  public:
    loss_of_precision_in_division(int x, int y)
      { a = x; b = y; }

    int get_a() const { return a; }
    int get_b() const { return b; }

  private:
    int a, b;
};
```

With this we can write a *safe* division function like this

```
int safe_divide(int a, int b)
{
  if (a % b == 0)
    return a / b;
  else
    throw loss_of_precision_in_division(a, b);
}
```

The main program has now the following form:

```
int main()
{
  double d;
  int a, b;

  // read a and b
  cin >> a >> b;

  try
```

A2X - C++ Course 1998

```
  {
    d = safe_divide(a, b);
  }
  catch (loss_of_precision_in_division x)
  {
    cout << "loss of precision in division!" << endl;
    cout << "I will use real division!" << endl;
    d = double(x.get_a()) / x.get_b();
  }

  // no loss of precision
  cout << "result: " << d << endl;
}
```

V-2.3 Multiple throws and catches

A function can throw more than one exceptions, which may have different types. The following example illustrates the use of multiple throws and catches:

```
#include <iostream.h>

class odd { };
class even { };

int main()
{
  int a;

  cin >> a;

  try
  {
    if (a < 0) a = -a;
    if (a % 2 == 1) throw odd();
    else throw even();
  }
  catch (odd i)
  {
    cout << "this number is odd" << endl;
```

```
  }
  catch (even i)
  {
    cout << "this number is even" << endl;
  }
}
```

V-2.4 Handling uncatched exceptions

In the above examples we have seen how to handle exceptions of *known* types. It may be also desirable to handle exceptions of the types which are not handled by a **catch** block. For example, we could consider all unknown exceptions as fatal errors, which terminate the program. For this purpose, C++ provides the default **catch** block:

```
try { ... }
catch (type1 t) { ... }
// ...
catch (typen t) { ... }

// default catch block
catch (...)
{
   cout << "fatal error: program exits" << endl;
   exit(1);
}
```

To be continued ...

V-3 Exercises

1. Extend the example of section V-2.2 to handle division by zero.

A2X - C++ Course 1998

A2X - C++ Course 1998

A

The C++ Language

The aim of this chapter is to serve as a C++ handbook for novice programmers by giving an informal description of the syntax and usage of the C++ language.

A-1 Built-in types

Every C++ implementation comes with a number of predefined types, the so called *built-in* types. Built-in types can be used to implement new types using language mechanisms like the **class** construction, we have described in the previous chapters. In this section we will give an overview of C++ built-in types.

A-1.1 Booleans

The type **bool** represents boolean (logical) values, for which the values `true` and `false` may be used.

A-1.2 Integers

C++ provides two types of integers: **signed** and **unsigned**. Signed integers can store positive and negative numbers, where unsigned integers can store only positive numbers.
The smallest integer type (in terms of bit-size) is a character. The corresponding keyword for a character is **char**[1]. The next larger integer type is **short**, followed by **int** and **long**[2] The size of these types differs between different architectures. The only

[1] Note that a **char** is a *special* integer, since it is also used to store the printable keyboard characters.

[2] It is more correct to say that beside a **char**, C++ provides the large type **int**; this can be (optionally) attributed by **short** or **long** to produce the types `short int` (or briefly **short**), `int` and `long int`. However, I find it more intuitive to categorize the integer types according to the way they are actually

A2X - C++ Course 1998

invariant that can be assumed is that

```
char <= short < int <= long
```

where `type1 < type2` means that one can store an object of `type1` to an object of `type2` without loss of precision.
In 32-bit architectures **unsigned char**s have 8 bit, **unsigned short**s have 16 bit, **unsigned int**s and **unsigned long**s have 32 bit. The **signed** counterparts use 1 bit for storing the sign, thus

```
max_value(signed type) < max_value(unsigned type)
```

(this is also the reason for providing unsigned integers: one can store larger positive numbers in them). Note that the keyword **signed** may be omitted and that **long int** is a synonym for **long**.

A-1.3 Reals

Real number are declared using the keywords **float** or **double** (for *double precision*). Again, we have the invariant

```
float < double
```

A-1.4 Exercises

1. Which built-in types exist in C++?
2. Is **signed int** the same as **int**?
3. On UNIX systems, the maximal values of the built-in types are defined usually in the header files `/usr/include/limits.h`, `/usr/include/values.h`. Find the maximum and minimum values of all C++ built-in types on your machine. How many bits are used to store a **long**?

A-2 Constants

Constants are the simplest language constructs. C++ provides the following types of constants:

---

stored in a machine.

1. *Strings constants*
   are sequences of characters enclosed by quotes ("). For example `"hello!"`, `"Salut!"` and `""` (the empty string) are valid string constants.

2. *Character constants*
   are single characters enclosed in simple quotes ('). Again, `'c'`, `'w'`, are valid character constants. Note that the are also special constants like `'\n'` (newline) `'\t'` (tab), etc. These are used to format output.

3. *Integer constants*
   are sequences of digits (possibly) preceded by a sign (`+` or `-`) and (possibly) followed by the letters `U` (unsigned), `L` (long) or `UL` (unsigned long). This can be given in octal, decimal, or hexadecimal form. For example `017`, `0xf` are the octal and hexadecimal form of the decimal number `15`. Note that octal constants begin with a `0` and hexadecimal constants with `0x`. All other constants are considered to be decimal.

4. *Real constants*
   are sequences of the form `xxx.yyy` where `xxx` and `yyy` are sequences of digits. Again, real constants may be optionally preceded by a sign (`+` or `-`). For example `123.5`, `1.` are valid real constants. Note that there is a convenient form for writing number of the form $xxx.yyy \cdot 10^{zzz}$: one simply writes `xxx.yyyEzzz` (this is called the *scientific notation*). For example `1235E-1` or `12.35e1` are valid reals constants equal to `123.5` (it is allowed to use both `e` and `E`).

A-3 Input and Output

Like C, C++ defines standard input, output and error streams, which are opened when a program is executed. These streams are: `cin`, `cout` and `cerr` and they correspond to C's `stdin`, `stdout` and `stderr`.
`cin`, `cout` and `cerr` are instances of classes for input and output. The classes are declared in `iostream.h`. Reading form streams and writing to streams, can be done using the insertion operator `>>` and the extraction operator `<<`:

```
#include <iostream.h>

void main()
{
  int i;
  cout << "enter a positive integer i: ";
```

A2X - C++ Course 1998

A2X - C++ Course 1998

```
  cin >> i;
  cout << "you have entered " << i << "\n";
}
```

As you see it is possible to output different types of objects using the same output stream `cout`. This is possible for all built-in types. A user-defined type has to overload the insertion / extraction operator in order to use streams like built-in types.

A-4 Overloading operators

In C++ the following operators can be overloaded:

```
+      -      *      /      %      ^      &      |
~      !      ,      =      <      >      <=     >=
++     --     <<     >>     ==     !=     &&     ||
+=     -=     *=     /=     %=     ^=     &=     |=
<<=    >>=    []     ()     ->     ->*    new    delete
```

However, some of these operators may only be overloaded as member functions within a class: the `=`, the `[]`, the `()` and the `->` operators are such operators. Note also, that although it is possible to overload these operators, it is not possible to change their priority in arithmetical evaluation. For example, assume we have overloaded the `^` operator to do powering within some class `X`:

```
X a, b, c;
c = a^3 * b; // oops: evaluates to a^(3 * b)
```

A-5 Free-store management

C++ provides **new** and **delete** for allocating and deallocating objects on the heap. For example:

```
char *cp = new char;    // allocate 1 character
int *ip = new int[10];  // allocate 10 integers
delete cp;              // delete 1 character
delete[] ip;            // delete 10 integers
```

The values of the objects allocated in this way are undefined. This is in contrast to user-defined types where **new** and **delete** also call the type's constructor to do proper initialization:

A2X - C++ Course 1998


```
rational *rp = new rational; // allocate 1 rational and
                             // call the default constructor
vector<int> *vip = new vector<int>(10);
                             // allocate 1 vector of int and
                             // call the constructor vector(int)
```

Calling **delete** calls the destructors and releases the allocated memory. Like `free` in C, **delete** has to be called manually. C++ supports no garbage collection.
Arrays of a type can be also declared in the usual way:

```
rational v[10]; // allocate 10 rationals and call the default
                // constructor on each one
```

Like in C, such arrays are released automatically upon scope exit.

A2X - C++ Course 1998

B

Implementation of a parameterized type

In this chapter we list a sample implementation of a parameterized `vector` type. We will start by giving the specification for `vector` in terms of a **class** definition and continue to the actual implementation and a usage example.

B-1 Specification

B-1.1 Definition

A `vector` is an indexed collection of items of some type `T`. The `size` of a `vector` is the number of its stored items. The `capacity` of a `vector` is the total amount of memory allocated by the `vector` for storing items. Therefore, for a `vector` `v` we always have `v.size() <= v.capacity()`.

B-1.2 Prequisites

The specification is written in the header file `vector.h`. As in C we start by including the necessary system header files. In our case this is only `iostream.h`:

```
// A2X - C++ Course 1998
// Copyright  1998 by Thomas Papanikolaou - All rights reserved.

#ifndef CXX_COURSE_VECTOR_H
#define CXX_COURSE_VECTOR_H

// include the header necessary for input/output
#include <iostream.h>
```

The class definition follows immediately:

A2X - C++ Course 1998

```
// class specification begins here
template <class T> class vector
{
```

B-1.3 Public interface

We now specify the public interface for `vector`. According to the class schema given in II-1.3 we specify constructors, destructor, accessors, modifiers and operator overloadings.

```
  public:

    // constructors and destructor
    vector ( );
    vector ( int i );
    vector ( const vector<T> & x );
    ~vector ( );

    // accessors
    int size ( ) const { return sz; }
    int capacity ( ) const { return alloc; }

    // modifiers
    void set_size ( int i );
    void set_capacity ( int i );

    // overloading of the assignment operator =
    vector<T> & operator = ( const vector<T> & x );

    // overloading of the reading index operator []
    const T & operator [] (int i) const;

    // overloading of the writing index operator []
    T & operator [] (int i);

    // overloading of input and output member functions
    // this illustrate how to avoid overloading of >> and <<
    // as friend functions
```

A2X - C++ Course 1998

```
    // read a vector from the input stream in
    void read ( istream & in );

    // write a vector to the output stream out
    void write ( ostream & out ) const;
```

B-1.4 Private interface

The private interface contains the data structure used for implementing `vector` as well as a routine for handling errors:

```
  private:

    T   *data;            // array to store the vector elements
    int    sz;            // the current size of the vector
    int alloc;            // the allocated size of the vector

    // error handling routine
    static void error ( char *text )
      {
        cerr << "vector<T> error: " << text << endl;
        exit(1);
      }

};
// class specification ends here
```

B-1.5 Further relevant utilities

Finally we specify the I/O operators for `vector`. Note how the member functions `read()` and `write()` are used to provide non-**friend** overloadings for input and output.

```
template <class T>
istream & operator >> ( istream & in  ,       vector<T> & x)
{
  x.read(in);
  return in;
}

template <class T>
```

A2X - C++ Course 1998

```
ostream & operator << ( ostream & out , const vector<T> & x)
{
  x.write(out);
  return out;
}

// include the implementation of vector<T>
#include "vector.cc"

#endif
```

B-2 Implementation

The implementation file `vector.cc` includes the specification file `vector.h` in order to declare the type `vector`.

```
// A2X - C++ Course 1998
// Copyright  1998 by Thomas Papanikolaou - All rights reserved.

// include the header for the template class vector
#include "vector.h"
```

B-2.1 Constructors & destructor

The implementation of the constructors and destructor is straight-forward: at first we allocate memory. If the operating system was not able to deliver the amount of memory we requested, we issue an `error()` call. The destructor simply frees the allocated memory.

Note

We are forced to use a macro for the default initial capacity of a `vector`, since there are some C++ compilers which can not handle **static** data members in parameterized classes. You should change the following macro to a **static** data member if your compiler supports it.

```
#define DEFAULT_INIT_LENGTH 4

// constructors and destructor
```

```
template <class T>
vector<T>::vector ( )
{
  sz = 0;
  alloc = DEFAULT_INIT_LENGTH;
  data = new T[alloc];
  if (data == NULL)
     vector<T>::error("out of memory");
}

template <class T>
vector<T>::vector ( int i )
{
  if (i <= 0)
     vector<T>::error("invalid size");
  sz = 0;
  alloc = i;
  data = new T[alloc];
  if (data == NULL)
     vector<T>::error("out of memory");
}

template <class T>
vector<T>::vector ( const vector<T> & x )
{
  sz = x.sz;
  alloc = x.alloc;
  data = new T[alloc];
  if (data == NULL)
     vector<T>::error("out of memory");
  for (int i = 0; i < sz; i++)
     data[i] = x.data[i];
}

template <class T>
vector<T>::~vector ( )
{
  delete[] data;
}
```

A2X - C++ Course 1998

A2X - C++ Course 1998

B-2.2 Modifiers

The modifier `set_size` has to be implemented carefully, to keep the `vector` invariant `v.size() <= v.capacity()`.

```
// modifiers
template <class T>
void vector<T>::set_size ( int i )
{
  if (i < 0)
    vector<T>::error("illegal size");
  if (i > alloc)
    set_capacity(i);
  sz = i;
}

template <class T>
void vector<T>::set_capacity ( int i )
{
  int old_alloc = alloc;
  T *old_data = data;
  alloc = (i < DEFAULT_INIT_LENGTH) ? DEFAULT_INIT_LENGTH : i;
  data = new T[alloc];
  int min_sz = (old_alloc < alloc) ? old_alloc : alloc;
  if (min_sz < sz) sz = min_sz;
  for (int j = 0; j < sz; j++)
    data[j] = old_data[j];
  delete[] old_data;
}
```

B-2.3 Assignment

Similarly to the constructors, assignment is also straight forward. For efficiency we additionally check that we do not assign a vector to itself. Note also that no re-allocation is necessary if the `capacity` of the destination `vector` is sufficiently large:

```
// overloading of the assignment operator =
template <class T>
```

A2X - C++ Course 1998

```
vector<T> & vector<T>::operator = ( const vector<T> & x )
{
  if (data != x.data) // handle special case x = x
  {
    if (alloc < x.alloc) // here the memory does not suffice
    {
      delete[] data;
      data = new T[x.alloc];
      if (data == NULL)
         vector<T>::error("out of memory");
    }
    sz = x.sz;
    alloc = x.alloc;
    for (int i = 0; i < sz; i++)
      data[i] = x.data[i];
  }
  return *this;
}
```

B-2.4 Index operator

While the read-only `[]` operator is trivial, the writing index operator has to set the size of the `vector` if we try to write above the allocated memory:

```
// overloading of the reading index operator []
template <class T>
const T & vector<T>::operator [] (int i) const
{
  if (i < 0 || i >= alloc)
     vector<T>::error("index out of range");
  return data[i];
}

// overloading of the writing index operator []
template <class T>
T & vector<T>::operator [] (int i)
{
  if (i < 0)
     vector<T>::error("index out of range");
```

A2X - C++ Course 1998

```
  if (i >= alloc)
     set_size(i);
  return data[i];
}
```

B-2.5 I/O

Finally, here is the implementation of the input and output of `vector`:

```
// read a vector from the input stream in
template <class T>
void vector<T>::read ( istream & in )
{
  int n = 0; // number of elements read so far
  char ch;
  bool read_data = false;

  in >> ch;
  if (ch != '[') vector<T>::error("[ expected");

  in >> ch;
  while (ch != ']')
  {
      if (ch != ',')
      {
        in.putback(ch);
        in >> data[n];
        read_data = true;
        n++;
      }
      else
      {
        if (read_data != true)
          n++;
        read_data = false;
      }
      if (n == alloc)
      {
        sz = n;
```

A2X - C++ Course 1998

```
        set_capacity(2 * alloc);
      }
      in >> ch;
  }
  if (n > 0 && read_data == false)
  {
      n++;
      if (n == alloc)
      {
        sz = n;
        set_capacity(2 * alloc);
      }
  }
  sz = n;
}

// write a vector to the output stream out
template <class T>
void vector<T>::write ( ostream & out ) const
{
  out << "[ ";
  for (int i = 0; i < sz; i++)
    out << data[i] << " ";
  out << "]";
}

// undefine the macro
#undef DEFAULT_INIT_LENGTH
```

B-3 Usage

The program `vector_appl.cc` illustrates how the type `vector` can be used:

```
// A2X - C++ Course 1998
// Copyright  1998 by Thomas Papanikolaou - All rights reserved.

// include the header file for vector
// (this includes the implementation)
#include "vector.h"
```

A2X - C++ Course 1998

```
// the main program starts here
int main()
{

  // declare some vectors
  vector<int> a;
  vector<int> b(10);
  vector<int> c = b;

  // output their values using the overloaded << operator
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "b.size() = " << b.size() << endl;
  cout << "b.capacity() = " << b.capacity() << endl;
  cout << "c = " << c << endl;
  cout << "c.capacity() = " << c.capacity() << endl;

  // test the overloaded input operator
  vector<int> x;
  cout << "enter an integer vector x:" << endl;
  cout << "(examples: [], [-1], [1,-2], [,2,3])" << endl;
  cin >> x;
  cout << "you have entered: " << x << endl;
  cout << "(implicit values are initialized randomly)" << endl;
  cout << "x.size() = " << x.size() << endl;
  cout << "x.capacity() = " << x.capacity() << endl;

  // exit nicely
  return 0;

}
// the main program ends here
```

A2X - C++ Course 1998

Bibliography

[1] Stanley B. Lippman: *C++ Primer*, second edition, Addison-Wesley Publishing Co., Reading, Mass., 1991

[2] Bjarne Stroustrup: *The C++ Programming Language*, third edition, Addison-Wesley Publishing Co., ISBN 0-201-88954-4

[3] Bjarne Stroustrup: *What is "Object-Oriented Programming"?*, Bell Labs, 1991

A2X - C++ Course 1998

Readings

- Frank B. Brokken and Karel Kubat, *C++ Annotations*, CCE, State University of Groningen, 1995, available via WWW from `http://www.icce.rug.nl/docs/cplusplus/cplusplus.html`
- Peter Müller, *Object-Oriented Programming Using C++*, Globewide Network Academy (GNA), 1998, available via WWW from `http://uu-gna.mit.edu:8001/uu-gna/text/cc/index.html`
- Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley Publishing Co., ISBN 0-201-54330-3.

A2X - C++ Course 1998

Index

Program code and names of functions are written using the `TypeWriter` font. C++ keywords are written in **Bold**. The page number containing the first appeareance of a definition is underlined.

A

`abelian_group` . . . 53
accessors . . . 24
Application files
- `vector_appl.cc` . . . 73

arguments . . . 24, 29
- passing by reference . . . 29
- passing by value . . . 24, 29

B

base class . . . 16
binding . . . 51, 52
- dynamic . . . 52
- static . . . 51

bool . . . 60

C

C++ Keywords
- **bool** . . . 60
- **catch** . . . 56, 59
- **char** . . . 33, 60
- **class** . . . 13, 19, 20, 36, 41, 60, 65
- **const** . . . 24–26, 28, 29, 35, 37
- **delete** . . . 63, 64
- **double** . . . 37, 40, 61
- **explicit** . . . 33
- **float** . . . 44, 61
- **friend** . . . 30–33, 37, 67
- **inline** . . . 30, 41
- **int** . . . 28, 31–33, 37, 39–41, 60, 61
- **long** . . . 60, 61
- **new** . . . 63
- **private** . . . 13, 20, 24, 34, 36, 48
- **protected** . . . 20, 48
- **public** . . . 13, 20, 35, 36, 48
- **short** . . . 60
- **signed** . . . 60, 61
- **static** . . . 9, 34–37, 68
- **struct** . . . 13
- **template** . . . 38, 39
- **throw** . . . 56
- **try** . . . 56
- **unsigned** . . . 60
- **virtual** . . . 16, 52, 54

catch . . . 56, 59
char . . . 33, 60
child class . . . 50
`circle` . . . 16, 46–50, 52
class . . . 45
- attributes . . . 45
- data members . . . 45
- instance . . . 45
- interface . . . 45
- member functions . . . 45
- methods . . . 45
- object . . . 45

class . . . 13, 19, 20, 36, 41, 60, 65
`coffee_machine` . . . 51
`color` . . . 14
`complex` . . . 37, 44
const . . . 24–26, 28, 29, 35, 37
construction . . . 21
constructor . . . 21
`cook_liquid(c)` . . . 51

A2X - C++ Course 1998

copy constructor 24

D
data hiding 7
datatype 11, 38
  abstract 11
  parameterized 38
  user-defined 11
define 41
delete 63, 64
derived 16, 50
destructor 21
double 37, 40, 61

E
empty 19
exception 55, 56
  handling 55
exit() 55
explicit 33

F
factorial 6
float 44, 61
foo 20
free 64
friend 30–33, 37, 67

H
header file 7
hierarchy 50

I
Implementation files
  vector.cc 68
Inheritance 50
inheritance 45, 47, 48, 53, 54
  multiple 53
  single 48
  virtual 54
inline 30, 41
instance 12
instantiation 21
int 28, 31–33, 37, 39–41, 60, 61
inv 32
iostream.h 65

L
liquid_machine 51, 52
  members
    cook() 51
List of Classes
  abelian_group 53
  circle 16, 46–50, 52
  coffee_machine 51
  color 14
  complex 37, 44
  cook_liquid(c) 51
  empty 19
  foo 20
  liquid_machine 51, 52
  point 14, 46–50, 52
  polynomial 49, 54
  prototype 20
  rational . 12, 13, 31, 32, 34, 35, 39, 45, 46
  real 32
  ring 52, 53
  semi_group 53
  shape 13–16
  stack 11
  string 33
  tea_machine 51
  vector 21–23, 25–27, 29, 31, 34, 36–38, 40, 45, 48, 49, 65–68, 70–73
  X 22, 63
long 60, 61
longjmp() 55, 56

M
mangling 40
method 25
  constant 25
min 25

A2X - C++ Course 1998

modifiers 24
module 7
multiple inheritance 52

N
new 63
`norm` 27–30

O
object 45
- behavior 45
- name 45
- state 45

operator overloading 26

P
paradigm 7, 9
- enable 9
- support 7

parent class 50
`point` 14, 46–50, 52
`polynomial` 49, 54
Preprocessor Keywords
- **define** 41

private 13, 20, 24, 34, 36, 48
procedure 5, 6
programming 5, 7, 10, 13
- modular 7
- object-based 10
- object-oriented 13
- procedural 5
- unstructured 5

protected 20, 48
`prototype` 20
public 13, 20, 35, 36, 48
`push` 11

R
`rational` 12, 13, 31, 32, 34, 35, 39, 45, 46
- friends
  - `inv` 32
- members
  - `den` 46
  - `inv` 31, 32
  - `num` 32, 46
  - `rational(int)` 31, 32
- operators
  - `+` 31
- statics
  - `one` 35
  - `zero` 35

`real` 32
- members
  - `real(int)` 32

recursive 6
reference 27
Relationships 46, 47
- a-kind-of 47
- has-a 46
- is-a 47
- part-of 46

representation 9
`ring` 52, 53

S
scope 22
`semi_group` 53
`setjmp()` 55, 56
`shape` 13–16
- members
  - `draw()` 14, 15
  - `rotate()` 14

short 60
signed 60, 61
slicing 51
Specification files
- `iostream.h` 65
- `vector.h` 65, 68

`sqrt` 6
`stack` 11
static 9, 34–37, 68
`string` 33
struct 13
subclass 16, 50
superclass 16, 50

swap 40, 41
swap2 41

T
tea_machine 51
template 38, 39
throw 56
try 56
type conversion 31

U
unsigned 60
user interface 7

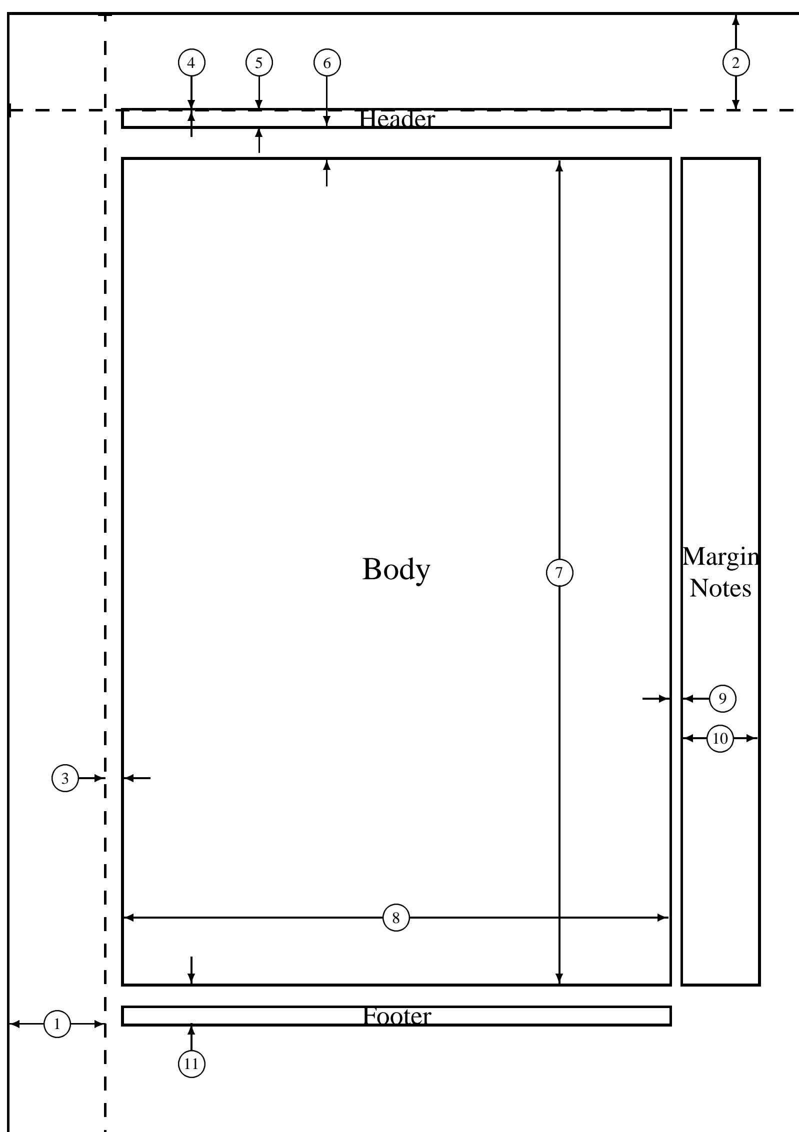
V
vector . 21–23, 25–27, 29, 31, 34, 36–38, 40, 45, 48, 49, 65–68, 70–73
- members
  - data 30
  - norm 29, 30
  - read() 67
  - set_size 25, 70
  - size 25, 40
  - sort 37
  - vector(int) 22
  - write() 67
- operators
  - = ... 26
  - [] 26, 29, 30, 71
- statics
  - direction 37
  - error() 68
  - set_direction(int) 37

vector.cc 68
vector.h 65, 68
vector_appl.cc 73
virtual 16, 52, 54
virtual table 52

X
X 22, 63
- members
  - foo 22
- operators
  - ^ 63

A2X - C++ Course 1998

A2X - C++ Course 1998



1	one inch + \hoffset	2	one inch + \voffset
3	\oddsidemargin = 14pt	4	\topmargin = 0pt
5	\headheight = 12pt	6	\headsep = 25pt
7	\textheight = 621pt	8	\textwidth = 411pt
9	\marginparsep = 10pt	10	\marginparwidth = 57pt
11	\footskip = 30pt		\marginparpush = 7pt (not shown)
	\hoffset = 0pt		\voffset = 0pt
	\paperwidth = 597pt		\paperheight = 845pt