Chapter 2

Concise Introduction to C++

C++ seems to be most suitable for many practical applications. Indeed, C++ is sufficiently high-level to allow transparent object-oriented programming and efficient use of human resources, yet is also sufficiently low-level to have types, variables, pointers, arrays, and loops to allow efficient use of computer resources.

In this chapter, we give a concise description of C++ and illustrate its power as an object-oriented programming language. In particular, we show how to construct and use abstract mathematical objects such as vectors and matrices. We also explain the notion of a template class, which can be filled with a concrete type later on in compilation time. We also discuss inheritance and illustrate its potential.

2.1 Objects

As we have seen above, C is a language based on functions. Every command is also a function that returns a value that can be further used or abandoned, according to the wish of the programmer. Furthermore, programmers can write their own functions, which may also return variables of the type specified just before the function name. When the function is called, a temporary, unnamed variable is created to store the returned value until it has been used.

C++, on the other hand, is an object-oriented programming language. In this kind of language, the major concern is not the functions that can be executed but rather the objects upon which they operate. Although C++ supports all the operations and features available in C, its point of view is different. C++ allows users to create not only their own functions but also their own objects and types, which can then be used in much the same way as the "int", "float", or "double" types that are built into C. The new types defined by C++ programmers can be used not only in the specific application for which they have been implemented but also in many other potential applications. Thus, C++ may be viewed as a dynamic extension of C, which constantly develops and grows with the definition of more and more useful objects.

The major advantage of object-oriented programming is the clear separation of the abstract (mathematical) concepts from their detailed implementation. Let us consider a team of two C++ programmers: Programmer A, who implements the objects, and Programmer B, who uses them in the actual application. Just as a C programmer is not interested in

the actual implementation of integer or real numbers in the hardware of the computer, Programmer B is not interested in the detailed implementation of the objects prepared by Programmer A. All that Programmer B needs are the objects themselves and the functions that operate upon them. Once these objects are available, Programmer B is free to use them in his particular application, regardless of their internal structure.

In fact, we could also consider a larger team of C++ programmers, in which Programmer A implements the objects required by all other programmers in their particular applications. Of course, Programmer A should also use feedback from these programmers to develop new functions that may be useful to potential users.

It is important that these programmers (or users) have a convenient interface to use the objects. This interface should not be changed, because any change in it would require changing every application that uses it. The actual implementation of objects, however, can be modified by Programmer A if necessary, so long as the modification doesn't affect the interface. The users should have no access to the internal implementation of objects, or they could change it inadvertently and spoil the entire framework.

Thus, just as C programmers cannot change the properties of integer or real types, users of the objects prepared by Programmer A cannot change their properties or functions. The data structures used in the objects should be accessible to external users only through public interface functions implemented and maintained by Programmer A. In fact, even Programmer A cannot modify the implementation unless he/she makes sure that the public interface functions are called and used in the same way.

The minds of the users of the objects prepared by Programmer A are thus free to develop the algorithms required in their own applications. They can use these objects through the interface functions, with no concern about any detail of implementation. They can treat the objects as perfect abstract concepts, which is particularly useful in the development of new methods and approaches. This is particularly important in numerical modeling, where complex mathematical objects are often involved.

2.2 Classes

As discussed above, C++ is particularly suitable for implementing abstract mathematical structures, which can then be used by other programmers as new objects or types. Assume, for example, that one wants to implement a point in the two-dimensional Cartesian plane. The well-implemented point should then be used by external users as if it were a standard type such as "int" or "double", leaving the users completely unaware of how it is stored or manipulated. This would free the minds of the users to concentrate on their particular application without being distracted by the details of the implementation of the points.

In particular, users should be able to write commands like

```
point P;
point Q=P;
```

to define a point 'P' and use it to define and initialize another point 'Q'. As we'll see below, this objective is achieved by defining the "point" class with its interface functions.

A new object in C++ is defined in a class as follows:

```
class point{
  public:
```

```
    double x;
    double y;  //  not object oriented
};
```

This is called a class block. The symbol "//" indicates the start of a comment line; the words that follow it are intended to describe or explain the code and are ignored by the C++ compiler.

The above block defines a new class called "point", which contains two data fields of type "double". The class can now be used to define variables of type "point". Once a concrete "point" variable 'P' is defined, "P.x" refers to its first field (representing the 'x'-coordinate) and "P.y" refers to its second field (representing the 'y'-coordinate). The reserved word "public:" in the above code indicates that these fields are accessible by every user of the class.

This implementation, however, is not in the spirit of object-oriented programming. Because the data fields 'x' and 'y' are accessible, the "point" object cannot be viewed as a complete "black box" implementation of the point. Because users are not familiar with the internal structure, they could change it inadvertently and spoil the object. Furthermore, if Programmer A, who wrote the "point" class, wanted to change the implementation at some stage, then he/she would have to tell all the users, who would have to change their own codes accordingly. This problem is avoided in the following object-oriented implementation:

```
class point{
    double x;
    double y;  //  an object-oriented implementation
  public:
    double X() const{
      return x;
    }  //  read x

    double Y() const{
      return y;
    }  //  read y

    void zero(){
      x=y=0.;
    }  //  set to zero
};
```

In this version, the data fields 'x' and 'y' are no longer accessible to external users. Because they appear before the reserved word "public", they are considered by default "private": users who define a "point" object 'P' cannot access its coordinates simply by "P.x" or "P.y" as before. The data in the "point" object are thus safe from inadvertent change by careless users. Still, users can read (but not change) the data fields in 'P' only through the public interface functions "X()" and "Y()". For instance, "P.X()" returns the 'x'-coordinate of 'P', and "P.Y()" returns the 'y'-coordinate of 'P'. We refer to 'P' as the current object or variable with which the function "X()" or "Y()" is called. The calls "P.X()" and "P.Y()" are no more expensive than the corresponding calls "P.x" and "P.y" in the previous

implementation, because the functions contain only one code line each and create no new objects.

The rule is that fields and functions in the class block that have not been declared "public" are by default private. This is why the above 'x' and 'y' fields are private. One may also declare some more private fields and functions at the end of the class block by writing the reserved word "private:" before their declarations.

The functions "X()" and "Y()" read the coordinates of the current "point" object without changing them. This is indicated by the reserved word "const" that precedes the symbol '{' that opens the function block. This reserved word guarantees that the current "point" object can never be changed by the function, and every attempt to change it will lead to a compilation error.

The public function "zero" in the above code lacks the word "const" before its block, because it changes the current "point" object and sets it to zero. Thus, users of class "point" may change the data in a "point" object only through nonconstant functions like "zero".

Interface functions like "X()", "Y()", and "zero" can be modified at any time by Programmer A, who is responsible for the implementation, provided that they still take the same number of arguments and return the same type as before, so they can be used by other users in the same way as before. This way, the users are completely unaware of the actual implementation or any change in it. All they need to know is how to use the interface functions. In fact, they can think of a "point" variable like 'P' as a point in the two-dimensional Cartesian plane. The interface functions associated with it can also be thought of as operations on actual points.

As we've seen above, interface functions are placed inside the class block, right after the definitions of data fields. This style is suitable for short functions that contain no more than a few code lines. These functions are then recompiled every time the function is called. A more efficient style, which is suitable for longer functions as well, declares the function inside the class block, leaving its actual definition until later. The definition is placed outside the class block, and the function name in it is preceded by a prefix containing the class name followed by the symbol "::", to indicate that this is indeed an interface function in this class. This way, it is treated as if it had been placed inside the class block. For example, the "point" class could have been written equivalently as follows:

```
class point{
    double x;
    double y;  //  an object-oriented implementation
  public:
    double X() const;
    double Y() const;  //  declarations only
    void zero();
};

double point::X() const{
  return x;
}  //  definition of X()

double point::Y() const{
  return y;
}  //  definition of Y()
```

```
void point::zero(){
  x=y=0.;
}  //  definition of "zero()"
```

Here, the interface functions are only declared in the class block, while their actual definitions are left outside it. Each definition is compiled only once, which creates a finite state machine (automaton). This machine is then invoked every time the function is called, with the concrete arguments that are passed to the function as input and the returned value as output.

The prefix "point::" that precedes the function names in the above code may actually be considered as an operator that “transfers” the definition back into the class block. This format, however, is unnecessary in the present "point" example, which uses very short definitions. The original style, in which the complete definitions appear inside the class block, is preferable in this case.

2.3 Constructors

When the user defines a point object by writing

```
point P;
```

the computer executes this command by allocating memory for the data fields of the new "point" variable ’P’. These data fields, the "double" variables "P.x" and "P.y", are not yet initialized by any meaningful value; in fact, they initially contain random, meaningless values. Only interface functions of the "point" class have access to these variables and can assign meaningful values to them.

C++ has a default constructor that allocates memory for the data fields and initializes them with random values. However, this is not always good enough. Programmer A might have a different idea about what should happen when the user writes code lines like "point P", and he/she might also want users to be able to write more sophisticated code lines like "point P = 0", in which ’P’ is constructed and initialized to zero. Indeed, if the "point" object is to behave like standard types, then such commands should be available to users.

It is, thus, good practice to write an explicit constructor in the class. The constructor is a public interface function that tells the computer how to construct a new object and allocate memory for its data fields. The name of the constructor function must be the same as the name of the class itself. For example, if one writes in the block of the "point" class

```
point(){
}  //  default constructor for the "point" class
```

then this constructor is invoked every time the compiler encounters a command of the form "point P;". The body of the constructor function contains no code lines at all. However, it still does something: as soon as it is called, the constructor implicitly invokes the default constructor built into C++, which allocates storage for the data fields "P.x" and "P.y" and fills them with random numbers. The above constructor is also called the default constructor for the "point" class, because it is invoked by commands like "point P;", with no initialization at all.

It is possible to write more than one constructor in the class block. In particular, one may also define a more informative constructor, which not only allocates memory for the

data fields in the constructed object but also initializes them with meaningful values. For example, if one writes in the block of the "point" class

```
point(double xx,double yy){
  x=xx;
  y=yy;
}
```

then this constructor is invoked every time the compiler encounters a command of the form

```
point P(3.,5.);
```

to create a new "point" object 'P' with the value 3 in its 'x'-field and the value 5 in its 'y'-field.

Here is how this constructor works. When it is called, it first implicitly invokes the default constructor built into the C++ compiler, which allocates memory to the 'x'- and 'y'-fields and fills them with random numbers. Then, the 'x'- and 'y'-fields are reassigned their correct values from the corresponding arguments "xx" and "yy", respectively.

This process is somewhat inefficient. Why not initialize the 'x'- and 'y'-fields immediately with their correct values? This can indeed be done by using an initialization list as follows:

```
point(double xx,double yy):x(xx),y(yy){
}  //  constructor with initialization list
```

The initialization list that follows the character ':' is a list of data fields in the constructed object, separated by commas. Each data field is followed by its initial value in parentheses. When this constructor is called, the fields are allocated memory and initialized properly in the order in which they appear (are declared) in the class block, and not necessarily in the order in the initialization list. The construction and initialization are then complete, so there is nothing left to do in the function block, and it remains empty.

Better yet, one may rewrite the above constructor as follows:

```
point(double xx=0.,double yy=0.):x(xx),y(yy){
}  //  arguments with default values
```

This way, the arguments "xx" and "yy" take the default value 0, which is used whenever they are not specified explicitly. This constructor also serves as a default constructor in the "point" class, so there is no need to write any other constructor. When the compiler encounters commands like

```
point P(3.,5.);
point Q(3.);  //  or point Q=3.;
point W;
```

it completes the missing arguments with zeroes and constructs three point objects: 'P' with 'x'-field initialized to 3 and 'y'-field initialized to 5, 'Q' with 'x'-field initialized to 3 and 'y'-field initialized to 0 (because the second argument "yy" is not specified and therefore takes the default value 0), and 'W' with both fields initialized to 0 (because both arguments "xx" and "yy" are unspecified).

2.4 Explicit Conversion

The above constructor also provides an explicit conversion operator from type "double" to type "point". As in C, where "(double)n" produces a temporary "double" variable with the same value as the integer variable 'n', "(point)a" produces a temporary "point" object whose first field contains the same value as the "double" variable 'a' and whose second field is zero. This is done by invoking the above constructor, with 'a' serving as the first argument and no second argument. Thanks to the default values in the above code, the second argument implicitly takes the zero value, which is then assigned to the second field in the temporary variable returned by the "(point)" function. This is called explicit conversion.

The term "conversion" is somewhat confusing. Actually, 'a' is never converted or changed in any way. It is only used as an argument for the "(point)" function. In fact, one can write equivalently "point(a)" and obtain the same result: a temporary "point" object with first field equal to 'a' and second field zero. This object can be used only in the very code line in which it was constructed and disappears soon after. Although the term "conversion" is inaccurate, it is convenient and commonly used to refer to this function.

2.5 Implicit Conversion

The above constructor also provides implicit conversion from type "double" to type "point". In code where a variable of type "point" is expected and a variable of type "double" appears instead, the above constructor is invoked implicitly to convert the "double" variable into the required "point" variable. This feature may be particularly useful in functions that take "point" arguments. When such a function is called with a "double" argument, it is converted implicitly into the required "point" argument. This feature is analogous to type conversion in C. On one hand, it may make codes more transparent and straightforward; on the other hand, it may also be too expensive, as we'll see below.

Implicit conversion requires an extra call to the constructor. The memory allocation in this construction requires extra computer time to complete. Although this overhead may be negligible for small objects such as points, it may be significant for larger objects, particularly if the implicit conversion is repeated many times in long loops. One should thus consider carefully whether or not to use implicit conversion.

2.6 The Default Copy Constructor

The above constructor can be used to construct new "point" objects and initialize them with prescribed values. However, users of the "point" class should also be able to use existing objects to initialize new ones. For example, they would surely like to write code such as

```
point P(3.,5.);
point Q(P); //  or point Q=P;
```

where 'P' is first constructed as before, and then 'Q' is constructed and initialized to have the same value as 'P'. This is done by the copy constructor.

The copy constructor constructs (allocates memory for) a new object and initializes it with the value of the object passed to it as an argument. The construction and initialization

are done in the same order in which the fields appear in the class block. In the above example, memory is allocated for the new fields "Q.x" and "Q.y", which are initialized with the values "P.x" and "P.y", respectively.

If no copy constructor is defined explicitly in the class block, then the construction is executed by the default copy constructor, available in the C++ compiler. This constructor just copies the data from the fields of the object passed to it as an argument to the corresponding fields in the constructed object. In the "point" class, this is exactly what we want, so there is actually no need to write an explicit copy constructor. Still, it is good practice to write an explicit copy constructor, because the default copy constructor may do the wrong thing. We'll return to this subject in Section 2.10.

The copy constructor is invoked every time an argument is passed to a function by value. In this case, a local copy of the concrete argument is constructed. Consider, for example, the following ordinary (noninterface) function, written outside the class block:

```
const point negative(const point p){
  return point(-p.X(),-p.Y());
}
```

This function returns the negative (minus) of a point in the Cartesian plane. However, its current implementation is rather expensive, because a constructor is used three times in each call to it. First, the concrete argument passed to the function is copied to a local variable 'p' by the copy constructor. Then, the constructor with two "double" arguments is used to create the negative of 'p'. Finally, the copy constructor is used once again to construct the object returned by the function. (Some compilers support a compilation option that avoids this third construction.) In Section 2.9, we'll show how the argument can be passed to the function by address rather than value, avoiding the first call to the copy constructor.

The "point" object returned by the "negative" function has no name and is stored only temporarily. It disappears soon after it is used in the code line in which the function is called. For this reason, it is a good idea to declare it as a constant, as is indeed done in the above code by putting the reserved word "const" before the definition of the function. This way, the returned variable cannot be changed by further calls to other functions. Usually, temporary returned objects have no need to change, because they disappear anyway at the end of the current code line. Declaring them as constants guarantees that they cannot be changed inadvertently.

When an object that is returned from a function is not declared constant, it can further change in the same code line in which it is created. For example, it can be used as a current object in interface functions. However, it cannot be passed by address to serve as an argument of any other function. The C++ compiler would refuse to create a local pointer that points to a temporary object, out of fear that it would change further in the function. The compiler would suspect that this wasn't the real intention of the programmer and would issue a compilation error.

For example, the temporary object "point(1.)", although nonconstant, cannot be passed by address to any function with a pointer-to-(nonconstant)-point argument. However, it can be used as a current object in interface functions such as "zero()":

```
Q = point(1.).zero();  //  or Q=((point)1.).zero();
```

The "negative()" function can also be called with a "double" argument, e.g., "negative(1.)" or "negative(a)", where 'a' is a "double" variable. In this call, the "double" argument is first

converted implicitly to a "point" object, which is then used as a concrete argument in the "negative" function.

2.7 Destructor

At the end of the block of a function, the local variables are destroyed, and the memory allocated for them is freed for future use. This is done by the destructor invoked implicitly by the computer. If no destructor is defined explicitly in the class block, then the default destructor available in the C++ compiler is used. This destructor goes over the data fields in the object and destroys them one by one. This is done in reverse order: "point" objects, for example, are destroyed by freeing their 'y'-field and then their 'x'-field.

The default destructor, however, does not always do a proper job. It is thus good practice to write an explicit destructor in the class block:

```
~point(){
}  //  destructor
```

Here, the actual destruction is done by the default destructor, which is invoked implicitly at the end of this destructor. This is why the body of this destructor is empty.

The default destructor, however, cannot properly destroy more complicated objects with data fields that are themselves pointers. Indeed, when the default destructor encounters such a field, it only destroys the address in it, not its content. The object in this address, although inaccessible because its address is no longer available, still occupies valuable memory. This is why an explicit destructor is required to delete this field properly using the reserved "delete" command. This command not only deletes the address in the field but also implicitly invokes the appropriate destructor to destroy the object in it and free the memory it occupies.

2.8 Member and Friend Functions

Interface functions may be of two possible kinds: member functions or friend functions. In what follows, we'll describe the features of these kinds of functions.

Constructors, destructors, and assignment operators must be member functions. The above "X()", "Y()", and "zero" functions are also member functions: they are defined inside the class block and act upon the current object with which they are called. For example, the call "P.X()" applies the function "X()" to the "point" object 'P' and returns its 'x'-field, "P.x".

Since member functions are defined inside the class block, their definitions can use (or call) only other interface functions declared in the class block; they cannot use ordinary (noninterface) functions defined outside the class block unless they are declared friends of the class.

Member functions are called with a current object, e.g., 'P' in "P.X()". When a member function such as "X()" is executed, it is assumed that the fields 'x' and 'y' mentioned in its definition refer to the corresponding fields in the current object; 'x' is interpreted as "P.x", and 'y' is interpreted as "P.y".

Friend functions, on the other hand, have no current object and can only take arguments as in ordinary functions.

The most important property of member functions is that they have access to all the fields of the current object and objects passed to them as concrete arguments, including private fields. In what follows, we'll explain how this access is granted.

When the user calls "P.X()" for some "point" variable 'P', the address of 'P' is passed to the function "X()", which stores it in a local variable named "this" of type constant-pointer-to-constant-point. (The word "this" is reserved in C++ for this purpose.) The type of "this" guarantees that neither "this" nor its content may change. Indeed, "X()" is a constant function that never changes its current object, as is indicated by the reserved word "const" before the function block.

Now, the member function "X()" can access the private members of 'P', "P.x", and "P.y", through the address of 'P', contained in "this". In the definition of "X()", 'x' and 'y' are just short for "this->x" (or "(*this).x") and "this->y" (or "(*this).y"), respectively. In fact, the definition of "X()" can be rewritten with the command:

```
return this->x;  //  same as (*this).x
```

In nonconstant functions like "zero", "this" is of a slightly different type: it is constant-pointer-to-point, rather than constant-pointer-to-constant-point. This allows changes to the current object through it. Indeed, "zero" is a nonconstant function, which lacks the reserved word "const" before its block. When "P.zero()" is called by the user, "this->x" and "this->y" are set to zero, which actually means that "P.x" and "P.y" are set to zero, as required.

The "this" variable is also useful for returning a value. For example, if we want the function "zero" to return a pointer to the current "point" object with which it is called, then we should rewrite it as follows:

```
point* zero(){
  x=y=0.;
  return this;
}  //  returns pointer-to-current-point
```

This way, a temporary, unnamed variable of type pointer-to-point is created at the end of the block of the function and initialized to the value in "this". This unnamed variable can be further used in the same code line in which the "zero" function is called and can serve as an argument for another function.

Because the pointer returned from the above "zero" function exists only temporarily, it is not good practice to use it to change its content. Usually, contents should change only through permanent, well-defined pointers, not temporary, unnamed pointers returned from functions as output. A better style is, therefore, the following, in which the returned pointer is a pointer-to-constant-point, so it cannot be used to change its content further. This is indicated by the reserved word "const" before the type of the function:

```
const point* zero(){
  x=y=0.;
  return this;
}  //  returns pointer-to-constant-point
```

The pointer returned by the "zero" function can be used, e.g., in the "printf" function, as follows:

```
int main(){
  point P;
  printf("P.x=%f\n",P.zero()->X());
  return 0;
}  //  print P.x after P has been set to zero
```

Here, the function "printf" prints the 'x'-field of the "point" object 'P' whose address is returned by the "zero()" function.

Later on, we'll show how the "zero" function can also be rewritten as a "friend" function. The reserved word "friend" should then precede the function name in the declaration in the class block. No current object or "this" pointer is available; objects must be passed explicitly as arguments, as in ordinary functions. Next, we'll see that arguments should be passed not by name (or value) but rather by reference.

2.9 References

In C++, one can define a reference to a variable. A reference is actually another name for the same variable. For example,

```
point p;
point& q = p;
```

defines a variable 'q' of type reference-to-point, initialized to refer to the "point" object 'p'. (Because 'q' is a reference, it must be initialized.) Every change to 'q' affects 'p' as well, and vice versa.

In the previous chapter, we saw that if a function is supposed to change a variable, then this variable must be passed to it by address, that is, by passing a pointer argument that points to it. A more transparent method is to pass a reference to this variable, which allows the function to refer to it and change it. For example, the above "zero" function can also be implemented as a friend function, which sets its argument to zero. This is done as follows. First, it should be declared as a friend in the block of the "point" class:

```
friend const point* zero(point&);
```

The actual definition can be made outside the block of the "point" class:

```
const point* zero(point&p){
  p.x=p.y=0.;
  return &p;
}  //  set "point" argument to zero
```

With this implementation, the function "zero" can be declared as a friend of other classes as well, so it can also use their private fields, if necessary. In the present example, this is not needed, so one can actually declare and define the function at the same time inside the block of the "point" class as follows:

```
friend const point* zero(point&p){
  p.x=p.y=0.;
  return &p;
}  //  declare as friend and define
```

This way, the "zero" function can be called from "main()" as follows:

```
printf("P.x=%f\n",zero(P)->X());
```

In the above definition, the point argument is passed to the "zero" function by reference. Therefore, when the function is called, no local "point" variable is created; instead, a local variable of type reference-to-point that refers to the same object is created. Every change to this local reference in the function affects the concrete argument as well, as required.

Passing an argument by reference is far more convenient than by address. Indeed, the programmer of the function is exempted from bothering with the address of the argument. Furthermore, the user of the function is exempted from passing the address of the variable, and can pass it in the same way as passing by value.

In both passing by reference and passing by address the computer does exactly the same thing: it creates a local copy of the address of the concrete argument, through which it can be changed. Still, when passing-by-reference is used, this is done implicitly, with no need to deal with any explicit address or pointer.

The style of the "zero" function may further improve by returning a reference to its argument rather than a pointer to it. For this purpose, the definition should read

```
friend const point& zero(point&p){
  p.x=p.y=0.;
  return p;
}
```

The type of function is now "const point&" rather than "const point*", indicating that a reference is returned rather than a mere address. The function can then be used as follows:

```
printf("P.x=%f\n",zero(P).X());
```

Here, "zero(P)" returns a reference to 'P', which is further used in conjunction with the function "X()" to print the 'x'-field in 'P'.

Although it is possible to implement the "zero" function as a friend of the "point" class, it is not considered very elegant. Friend functions are usually used to read data from private fields of a class, and their advantages and disadvantages in doing this are discussed in Section 2.14 below. Friend functions are also useful for accessing private data in more than one class. However, when the private data in the class are not only read but also changed, it is more natural to use member functions. The above "zero" function can be written equivalently as a member function in the block of the "point" class as follows:

```
const point& zero(){
  x=y=0.;
  return *this;
}
```

This way, the "zero" member function is of type constant-reference-to-point, as is indeed indicated by the reserved words "const point&" before its name. The function returns a reference to the current object contained in the address "this". The returned reference can be further used in the same code line as follows:

```
printf("P.x=%f\n",P.zero().X());
```

Here, the reference to 'P' returned by "P.zero()" serves as the current object in a further call to the "X()" function.

Passing arguments by reference is not only convenient but also efficient. Indeed, when the argument is passed by reference rather than by value, the need to invoke the copy constructor to construct a local copy is avoided. For example, if the "negative" function in Section 2.6 had been rewritten as

```
const point negative(const point& p){
  return point(-p.X(),-p.Y());
}  //  passing argument by reference
```

then no local "point" object would be constructed, only a local reference to the concrete argument. Creating this reference requires only copying the address of the concrete argument rather than copying the entire concrete argument physically. The total number of constructions in the call to the "negative" function would then decrease from three to two.

The "negative" function still requires two calls to constructors of "point" objects: one to construct the local negative and the other to return a copy of it. This number cannot be reduced further. Look what happens if one attempts to avoid the second construction by writing

```
const point& negative(const point& p){
  return point(-p.X(),-p.Y());
}  //  wrong!!! returns reference to nothing
```

This version returns by reference rather than by value. (Indeed, the type of function is "const point&" rather than "const point".) It returns a reference to the local variable that contains the negative of the "point" argument. However, the negative of the "point" argument is a temporary local variable, which no longer exists at the end of the function, so it actually returns a reference to nothing. One should therefore drop this version and stick to the previous one.

2.10 Copy Constructor

As mentioned in Section 2.6, it is good practice to define an explicit copy constructor. For example, a suitable copy constructor can be written in the block of the "point" class as follows:

```
point(const point& p):x(p.x),y(p.y){
}  //  copy constructor
```

Here, the copied "point" object is passed to the constructor by reference, and its fields are used to initialize the corresponding fields in the constructed "point" object.

Actually, it is not necessary to write this constructor, because the default copy constructor available in the C++ compiler does exactly the same thing. Still, it is good practice to write your own copy constructor, because in many cases the default one does the wrong thing, as we'll see below.

The above copy constructor is invoked whenever the compiler encounters an explicit copying such as

```
point Q = P;  //  same as  point Q(P);
```

or an implicit copying such as passing an argument to a function or returning an object from it by value.

2.11 Assignment Operators

Users of the "point" class may want to assign values in a natural way as follows:

```
point P,W,Q(1.,2.);
P=W=Q;
```

Here, the point objects 'P', 'W', and 'Q' are created in the first code line by the constructor in Section 2.3, which uses "double" arguments with default value 0. This way, the fields in 'Q' are initialized with the specified values 1 and 2, whereas the fields in 'P' and 'W' take the default value 0. In the second code line, the default assignment operator built into the C++ compiler is invoked to assign the value of fields in 'Q' to the corresponding fields in 'W' and 'P'. This is done from right to left as follows. First, the values of fields in 'Q' are assigned to the corresponding fields in 'W' one by one in the order in which they are declared in the class block. In other words, first "W.x" is assigned with "Q.x", and then "W.y" is assigned with "Q.y". This assignment operation also returns a reference to 'W'. This reference is used further to assign the updated 'W' object to 'P', so eventually all three point objects have the same value, as required.

Although the default assignment operator does the right thing here, it is good practice to define your own assignment operator. According to the rules of C++, it must be a member function. Here is how it is defined in the class block:

```
const point& operator=(const point& p){
```

This is the heading of a function, in which the type of argument is declared in parentheses and the type of returned object is declared before the function name, "operator=". Note that the argument is passed and the output is returned by reference rather than by value to avoid unnecessary calls to the copy constructor. Furthermore, the argument and the returned object are also declared as constants, so they cannot change inadvertently. Indeed, both the argument and the function name are preceded by the words "const point&", which stand for reference-to-constant-point.

Look what happens if the argument is declared nonconstant by dropping the word "const" from the parentheses. The compiler refuses to take any constant concrete argument, out of fear that it will change through its nonconstant local reference. Furthermore, the compiler refuses to take even a nonconstant concrete argument that is returned from some other function as a temporary object, out of fear that it might change during the execution of the assignment operator. Because it makes no sense to change a temporary object that will disappear soon anyway, the compiler assumes that the call is mistaken and issues a compilation error. Declaring the argument as constant as in the above code line prevents all these problems. The function can now be called with either a constant or a nonconstant argument, as required.

The body of the function is now ready to start. The following "if" question is used to make sure that the compiler hasn't encountered a trivial assignment like "P = P". Once it is made clear that the assignment is nontrivial, it can proceed:

```
    if(this != &p){
      x = p.x;
      y = p.y;
    }
```

Finally, a reference to the current object is also returned for further use:

```
    return *this;
  }  //  point-to-point assignment
```

We refer to this operator as a point-to-point assignment operator.

It is also possible to assign values of type "double" to "point" objects. For example, one can write

```
P=W=1.;
```

When this command is compiled, the constructor is first invoked to convert implicitly the "double" number "1." into a temporary unnamed point object with 'x'- and 'y'-fields containing the values 1 and 0, respectively. This object is then assigned to 'W' and 'P' as before. The "zero()" function of Section 2.2 is, thus, no longer necessary, because one can set 'P' to zero simply by writing "P = 0.".

As discussed in Section 2.5 above, implicit conversion may be rather expensive, as it requires the construction of an extra "point" object. This issue is of special importance when assignment is used many times in long loops. In order to avoid this extra construction, one may write an assignment operator that takes a "double" argument:

```
const point& operator=(double xx){
  x = xx;
  y = 0.;
  return *this;
}  //  double-to-point assignment
```

We refer to this operator as a double-to-point assignment operator. It must also be a member function that is at least declared (or even defined) inside the class block.

When the compiler encounters a command of the form "P = 1.", it first looks for a double-to-point assignment operator. If such an operator exists, then it can be used here, avoiding implicit conversion. More specifically, the double-to-point assignment operator assigns 1 to "P.x" and 0 to "P.y" as required, avoiding any construction of a new object. A reference to the current object is also returned to allow commands of the form "W = P = 1.".

The assignment operator allows compact elegant code lines like "P = Q" and "P = 1.". Still, it can also be called as a regular function:

```
P.operator=(W.operator=(0.));  //  same as P=W=0.;
```

This is exactly the same as "P = W = 0." used above. The original form is, of course, preferable.

Below we'll see many more useful operators that can be written by the programmer of the class. These operators use the same symbols as standard operators in C, e.g., the '=' symbol in the above assignment operators. However, the operators written by the programmer are not necessarily related to the corresponding operators in C. The symbols used here only preserve the standard priority order.

2.12 Operators

The programmer of the "point" class may also define other operators for convenient manipulation of objects. The symbols used to denote arithmetic and logical operators may be given new meaning in the context of the present class. The new interpretation of an operator is made clear in its definition.

Although the new operator may have a completely different meaning, it must still have the same structure as in C; that is, it must take the same number of arguments as in C. The type of these arguments and returned object, however, as well as what the operator actually does, is up to the programmer of the class. For example, the programmer may give the symbol "&&" the meaning of a vector product as follows:

```
double operator&&(const point&p, const point&q){
  return p.X() * q.Y() - p.Y() * q.X();
}  //  vector product
```

This way, although the "&&" operator in C has nothing to do with the vector product, it is suitable to serve as a vector-product operator in the context of "point" objects because it takes two arguments, as required. One should keep in mind, though, that the "&&" operator in C is a logical operator, with priority weaker than that of arithmetic operators. Therefore, whenever the "&&" symbol is used in the context of "point" objects, it must be put in parentheses if it should be activated first.

Note that the above operator is implemented as an ordinary (nonmember, nonfriend) function, because it needs no access to any private member of the "point" class. In fact, it accesses the data fields in 'p' and 'q' through the public member functions "X()" and "Y()". The arguments 'p' and 'q' are passed to it by reference to avoid unnecessary copying. These arguments are also declared as constant, so the function can take either constant or nonconstant concrete arguments. The user can now call this function simply by writing "P&&Q", where 'P' and 'Q' are some "point" variables.

2.13 Inverse Conversion

Another optional operator is inverse conversion. This operator is special, because its name is not a symbol but rather a reserved word that represents the type to which the object is converted.

In the context of the "point" class, this operator converts a "point" object to a "double" object. Exactly how this is done is up to the programmer of the class. However, here the programmer has no freedom to choose the status of the function or its name: it must be a member function with the same name as the type to which the object is converted, that is, "double". Here is how this operator can be defined in the block of the "point" class:

```
operator double() const{
  return x;
}  //  inverse conversion
```

With this operator, users can write "(double)P" or "double(P)" to read the 'x'-coordinate of a "point" object 'P'. Of course, 'P' never changes in this operation, as is indeed indicated by the reserved word "const" before the '{' character that opens the function block.

The term “conversion” is inexact and is used only to visualize the process. Actually, nothing is converted; the only thing that happens is that the first coordinate is read, with absolutely no change to the current object.

Inverse conversion can also be invoked implicitly. If the "point" object ’P’ is passed as a concrete argument to a function that takes a "double" argument, then the compiler invokes the above operator implicitly to convert ’P’ into the required "double" argument with the value "P.x".

Implicit calls to the inverse-conversion operator are also risky. The programmer is not always aware of them or able to decide whether or not they should be used. It seems to be better practice not to define inverse conversion and to let the compiler issue an error whenever an argument of the wrong type is passed to a function. This way, the programmer becomes aware of the problem and can decide whether to convert the argument explicitly. In the present classes, we indeed define no inverse conversion.

2.14 Unary Operators

The "negative" function in Section 2.6 can actually be implemented as an operator that takes one argument only (unary operator). For this purpose, one only needs to change the function name from "negative" to "operator-". With this new name, the function can be called more elegantly by simply writing "-Q" for some "point" object ’Q’:

```
point W,Q=1.;
W=-Q;  //  same as W=operator-(Q);
```

In this code, the "point" object ’W’ is assigned the value $(-1,0)$.

The "operator-" may also be more efficient than the original "negative" function. For example, the code

```
point W=negative(1.);
```

uses implicit conversion to convert the "double" argument 1 into the "point" argument $(1,0)$ before applying the "negative" function to it. Once the "negative" function has been renamed "operator-", the above code is rewritten as

```
point W=-1.;
```

which interprets -1 as a "double" number and uses it to initialize ’W’ with no conversion whatsoever.

One may define other optional operators on "point" objects. In the definition, the reserved word "operator" in the function name is followed by the symbol that should be used to call the operator.

For example, we show below how the "+=" operator can be defined as a member function of the "point" class. The definition allows users to write the elegant "P+=Q" to add the "point" argument ’Q’ to the current "point" object ’P’:

```
const point& operator+=(const point& p){
  x += p.x;
  y += p.y;
  return *this;
}  //  adding a point to the current point
```

Here, the "point" argument is passed to the "+=" operator by reference to avoid unnecessary copying. Then, the values of its 'x'- and 'y'-coordinates are added to the corresponding fields in the current object stored in "this". The updated current object is also returned by reference as output. This output can be further used in the same code line as follows:

```
P=W+=Q;
```

This code line is executed right to left: first, 'W' is incremented by 'Q', and the resulting value of 'W' is then assigned to 'P'. This assignment can take place thanks to the fact that the point-to-point assignment operator in Section 2.11 takes a reference-to-*constant*-point argument, so there is no fear that it will change the temporary object returned from "W+=Q", which is passed to it as a concrete argument.

The above code is actually equivalent to the following (less elegant) code:

```
P.operator=(W.operator+=(Q));  //  same as P=W+=Q;
```

Like the assignment operator in Section 2.11, the "+=" operator also accepts a "double" argument through implicit conversion. For example, the call

```
W+=1.;
```

first implicitly converts the "double" argument 1 to the temporary "point" object (1,0), which in turn is used to increment 'W'. As discussed above, implicit conversion can be used here only thanks to the fact that "operator+=" takes a reference-to-*constant*-point argument rather than a mere reference-to-point, so it has no problem accepting as argument the temporary "point" object returned from the implicit conversion.

The above implicit conversion is used only if there is no explicit version of the "+=" operator that takes the "double" argument. If such a version also exists, then the compiler will invoke it, because it matches the type of arguments in the call "W+=1.". This version of "operator+=" can be defined in the class block as follows:

```
const point& operator+=(double xx){
  x += xx;
  return *this;
}  //  add real number to the current point
```

This version increases the efficiency by avoiding implicit conversion. This property is particularly important in complex applications, where it may be used many times in long loops.

The above "+=" operators are implemented as member functions, which is the natural way to do it. However, they can also be implemented as friend functions as follows:

```
friend const point&
operator+=(point&P,const point& p){
  P.x += p.x;
  P.y += p.y;
  return P;
}
```

In this style, the function takes two arguments. The first one is nonconstant, because it represents the object that is incremented by the second, constant, argument. Similarly, a "friend" version can also be written for the "+=" operator with a "double" argument. The calls to the "+=" operators are done in the same way as before.

The "friend" implementation, although correct, is somewhat unnatural in the context of object-oriented programming. Indeed, it has the format of a C function that changes its argument. In object-oriented programming, however, we think in terms of objects that have functions to express their features, rather than functions that act upon objects. This concept is better expressed in the original implementation of the "+=" operators as member functions.

The "friend" version has another drawback. Although it correctly increments well-defined "point" variables, it refuses to increment temporary, unnamed "point" objects that have been returned from some other function. Indeed, since the incremented argument in the "friend" version is of type reference-to-(nonconstant)-point, it rejects any temporary concrete argument because it sees no sense in changing an object that is going to vanish soon and assumes that this must be a human error. Of course, the compiler might not know that this was intentional. In the original, "member", version of the "+=" operator, on the other hand, even temporary objects can be incremented, because they serve as a nonconstant current object. This greater flexibility of the "member" implementation is also helpful in the implementation of the '+' operator below.

2.15 Binary Operators

In this section, we define binary operators that take two arguments to produce the returned object. In this respect, these operators have the same structure as ordinary C functions. The difference is, however, that they use objects rather than just integer or real numbers as in C. Furthermore, the operators can be called conveniently and produce code that imitates the original mathematical formula.

The '+' operator that adds two "point" objects can be written as an ordinary (non-member, nonfriend) function that requires no access to private data fields of point objects and hence needs no declaration in the class block:

```
const point
operator+(const point& p, const point& q){
  return point(p.X()+q.X(),p.Y()+q.Y());
}  //  add two points
```

Unlike the "+=" operator, this operator doesn't change its arguments, which are both passed as reference-to-constant-point, but merely uses them to produce and return their sum. Note that the returned variable cannot be declared as a reference, or it would refer to a local "point" object that vanishes at the end of the function. It must be a new "point" object constructed automatically in the "return" command by the copy constructor to store a copy of that local variable. This is indicated by the words "const point" (rather than "const point&") that precede the function name.

The above '+' operator can be called most naturally as

```
P=W+Q;  //  the same as P=operator+(W,Q);
```

As mentioned at the end of Section 2.13, we assume that no inverse conversion is available, because the "operator double()" that converts "point" to "double" is dropped. Therefore, since both arguments in "operator+" are of type reference-to-constant-point, "operator+" can be called not only with two "point" arguments (like "W+Q") but also with one "point" argument and one "double" argument (like "W+1." or "1.+W"). Indeed, thanks to the implicit double-to-point conversion in Section 2.5, the "double" number "1." is converted to the point $(1,0)$ before being added to 'W'. Furthermore, thanks to the lack of inverse conversion, there is no ambiguity, because it is impossible to convert 'W' to "double" and add it to "1." as "double" numbers.

If one is not interested in implicit conversion because of its extra cost and risks and wants the compiler to announce an error whenever it encounters an attempt to add a "double" number to a "point" object, then one can drop conversion altogether by not specifying default values for the "double" arguments in the constructor in Section 2.3. This way, the constructor expects two "double" arguments rather than one and will not convert a single "double" number to a "point" object.

In the above implementation, the '+' operator is defined as an ordinary function outside the block of the "point" class. This way, however, it is unavailable in the class block, unless it is declared there explicitly as a "friend":

```
friend const point operator+(const point&,const point&);
```

With this declaration, the '+' operator can also be called from inside the class block. Furthermore, it has access to the data fields of its "point" arguments. In fact, it can be defined inside the class block as follows:

```
friend const point operator+(
        const point& p, const point& q){
  return point(p.x+q.x,p.y+q.y);
}  //  defined as "friend" in the class block
```

The '+' operator can also be implemented as a member function inside the class block as follows:

```
const point operator+(const point& p) const{
  return point(x+p.x,y+p.y);
}  //  defined as "member" in the class block
```

With this implementation, the '+' operator is still called in the same way (e.g., "W+Q"). The first argument ('W') serves as the current object in the above code, whereas the second argument ('Q') is the concrete argument passed by reference to the above function.

This, however, is a rather nonsymmetric implementation. Indeed, implicit conversion can take place only for the second argument, which is a reference-to-constant-point, but not for the first argument, the current object. Therefore, mixed calls such as "W+1." are allowed, but not "1.+W". This nonsymmetry makes no apparent sense.

The original implementation of "operator+" as an ordinary function outside the class block is more in the spirit of object-oriented programming. Indeed, it avoids direct access to the private data field 'x' or 'y' in "point" objects and uses only the public member functions "X()" and "Y()" to read them. This way, the '+' operator is independent of the internal implementation of "point" objects.

One could also write two more versions of "operator+" to add a "double" number and a "point" object explicitly. These versions increase efficiency by avoiding the implicit conversion used above. They are also implemented as ordinary functions outside the class block:

```
const point operator+(const point& p, double xx){
  return point(p.X()+xx,p.Y());
}  //  point plus real number

const point operator+(double xx, const point& p){
  return point(p.X()+xx,p.Y());
}  //  real number plus point
```

These versions are invoked by the compiler whenever a call such as "W+1." or "1.+W" is encountered, avoiding implicit conversion.

The original implementation of the '+' operator as an ordinary function can also be written in a more elegant way, using the "operator+=" member function defined in Section 2.14:

```
const point
operator+(const point& p, const point& q){
  return point(p) += q;
}  //  point plus point
```

Thanks to the fact that the "+=" operator is defined in Section 2.14 as a member (rather than a mere friend) of the "point" class, it accepts even temporary "point" objects as concrete arguments. In particular, even the first argument (the current object that is incremented in the "+=" operator) may be a temporary object. This property is used in the above code, where the temporary object "point(p)" returned from the copy constructor is incremented by the "+=" operator before being returned as the output of the entire '+' operator. This elegant style of programming will be used in what follows.

2.16 Example: Complex Numbers

In Fortran, the "complex" type is built in and available along with the required arithmetic operations. The programmer can define a complex variable 'c' by writing simply "complex c" and apply arithmetic operations and some other elementary functions to it.

The C and C++ compilers, on the other hand, don't support the "complex" type. If one wants to define and use complex numbers, then one must first define the "complex" class that implements this type. Naturally, the block of the class should contain two "double" fields to store the real and imaginary parts of the complex number and some member operators to implement elementary arithmetic operations. Once the implementation is complete, one can define a "complex" object 'c' simply by writing "complex c" and then apply arithmetic operations to it as if it were built into the programming language.

Some standard C libraries do support complex numbers. Still, the "complex" object implemented here is a good example of object-oriented programming in C++.

This example illustrates clearly how the object-oriented approach works: it provides new objects that can then be used as if they were built into the programming language.

These new objects can then be viewed as an integral part of the programming language and can add new dimensions and possibilities to it. The programming language develops dynamically by adding more and more objects at higher and higher levels of programming.

Here is the detailed implementation of the "complex" class:

```
#include<stdio.h>
class complex{
    double real;   //  the real part
    double image;  //  the imaginary part
  public:
    complex(double r=0.,double i=0.):real(r), image(i){
    } //  constructor

    complex(const complex&c):real(c.real),image(c.image){
    }  //  copy constructor

    ~complex(){
    }  //  destructor
```

In the above constructors, all the work is done in the initialization lists, where memory is allocated for the data fields "real" and "image" with the right values. The bodies of these functions remain, therefore, empty. The destructor above also needs to do nothing, because the default destructor called implicitly at the end of it destroys the data fields automatically.

Because the data fields "real" and "image" are declared before the reserved word "public:", they are private members of the class. Thus, only members and friends of the class can access and change data fields of an object in the class. Still, we'd like other users to be able to read these fields from ordinary functions that are neither members nor friends of the class. For this purpose, we define the following two public member functions that can only read (but not change) the data fields:

```
    double re() const{
      return real;
    }  //  read real part

    double im() const{
      return image;
    } //  read imaginary part
```

Next, we define the assignment operator. This operator will enable users to assign the value of a "complex" object 'd' to a "complex" object 'c' simply by writing "c = d":

```
    const complex&operator=(const complex&c){
      real = c.real;
      image = c.image;
      return *this;
    }  //  assignment operator
```

Next, we define some member arithmetic operators that change the current "complex" object. For example, the "+=" operator allows users to write "c += d" to add 'd' to 'c':

```
const complex&operator+=(const complex&c){
  real += c.real;
  image += c.image;
  return *this;
}  //  add complex to the current complex

const complex&operator-=(const complex&c){
  real -= c.real;
  image -= c.image;
  return *this;
}  //  subtract complex from the current complex

const complex&operator*=(const complex&c){
  double keepreal = real;
  real = real*c.real-image*c.image;
  image = keepreal*c.image+image*c.real;
  return *this;
}  //  multiply the current complex by a complex

const complex&operator/=(double d){
  real /= d;
  image /= d;
  return *this;
}  //  divide the current complex by a real number
```

In the latter function, the current complex number is divided by a real number. This operator will be used later in the more general version of the "/=" operator that divides the current complex number by another complex number.

In the end, we'll have two "/=" operators: one that takes a real argument and one that takes a complex argument. When the C++ compiler encounters a command of the form "c /= d", it invokes the first version if 'd' is real and the second if 'd' is complex.

The division of a complex number by a complex number will be implemented later in another version of "operator/=". As a member function, this function will not be able to recognize any ordinary function that is defined outside the class block and is neither a member nor a friend of the class. This is why the two following functions are defined as friends of the "complex" class: they have to be called from the "operator/=" member function defined afterward.

The first of these two functions returns the complex conjugate of a complex number. The name of this function, "operator!", has nothing to do with the logical "not" operator used in C. In fact, this name is chosen here only because it represents a unary operator, which can later be called as "!c" to return the complex conjugate of 'c'.

Note that the returned object must be complex rather than reference-to-complex ("complex&"), or it would refer to a local variable that disappears at the end of the function. The word "complex" before the function name in the above code indicates that the local variable is copied to a temporary unnamed variable to store the returned value. Since this variable is not declared "constant", it can be further changed in the "/=" operator below:

```
    friend complex operator!(const complex&c){
      return complex(c.re(),-c.im());
    }  //  conjugate of a complex
```

The second function defined below, "abs2()", returns the square of the absolute value of a complex number:

```
    friend double abs2(const complex&c){
      return c.re() * c.re() + c.im() * c.im();
    }  //  square of the absolute value of a complex
```

Because they are declared as friends, these two functions can now be called from the "operator/=" member function that divides the current "complex" object by another one:

```
    const complex&operator/=(const complex&c){
      return *this *= (!c) /= abs2(c);
    }  //  divide the current complex by a complex
};
```

In this "/=" operator, the current complex number is divided by the complex argument 'c' by multiplying it by the complex conjugate of 'c' (returned from "operator!") divided by the square of the absolute value of 'c' (returned from the "abs2()" function). Because its argument is real, this division is carried out by the early version of the "/=" operator with a real argument. Thanks to the fact that this operator is a member function, it can change its nonconstant unnamed current object "!c" and divide it by "abs2(c)". The result is used to multiply the current complex number, which is equivalent to dividing it by 'c', as required. This completes the block of the "complex" class.

The following functions are ordinary noninterface (nonmember, nonfriend) functions that implement basic operations on complex numbers. Note that there are two different '-' operators: a binary one for subtraction and a unary one for returning the negative of a complex number. When the C++ compiler encounters the '-' symbol in the program, it invokes the version that suits the number of arguments: if there are two arguments, then the binary subtraction operator is invoked, whereas if there is only one argument, then the unary negative operator is invoked:

```
const complex
operator-(const complex&c){
  return complex(-c.re(),-c.im());
}  //  negative of a complex number

const complex
operator-(const complex&c,const complex&d){
  return complex(c.re()-d.re(),c.im()-d.im());
}  //  subtraction of two complex numbers
```

Here are more binary operators:

```
const complex
operator+(const complex&c,const complex&d){
```

```
  return complex(c.re()+d.re(),c.im()+d.im());
}  //  addition of two complex numbers

const complex
operator*(const complex&c,const complex&d){
  return complex(c) *= d;
}  //  multiplication of two complex numbers

const complex
operator/(const complex&c,const complex&d){
  return complex(c) /= d;
}  //  division of two complex numbers
```

In the above functions, the returned object cannot be of type reference-to-complex ("const complex&"), or it would refer to a local variable that no longer exists. It must be of type "const complex", which means that the local variable is copied by the copy constructor to a temporary "complex" object that is constructed at the end of the function and also exists after it terminates.

Finally, we also define the unary "operator+" to return the complex conjugate of a complex number. This way, the user can write "$+t$" for any numerical variable t, may it be either real or complex. Indeed, if t is real, then $+t = t$, as required; if, on the other hand, t is complex, then $+t = \bar{t}$.

```
complex operator+(const complex&c){
  return complex(c.re(),-c.im());
}  //  conjugate complex
```

This concludes the arithmetic operations with complex numbers. Finally, we define a function that prints a complex number:

```
void print(const complex&c){
  printf("(%f,%f)\n",c.re(),c.im());
}  //  printing a complex number
```

Here is how complex numbers are actually used in a program:

```
int main(){
  complex c=1.,d(3.,4.);
  print(c-d);
  print(c/d);
  return 0;
}
```

2.17 Templates

Above, we have implemented the "point" object and the "zero()" function that sets its value to zero. Now, suppose that we need to implement not only points in the two-dimensional Cartesian plane but also points in the three-dimensional Cartesian space. One possible implementation is as follows:

```
class point3{
    double x;
    double y;
    double z;
  public:
    void zero(){ x=y=z=0.; }
};
```

This implementation, however, is neither elegant nor efficient in terms of human resources, because functions that have already been written and debugged in the "point" class (such as arithmetic operators) will now be written and debugged again in the "point3" class. A much better approach is to use templates.

A template can be viewed as an object with a parameter that has not yet been specified. This parameter must be specified in compilation time, so that functions that use the object can compile. When the compiler encounters a call to a function that uses this object, it interprets it with the specified parameter and compiles it like regular objects. The compiled function is actually a finite state machine that can be further used in every future call to the function with the same parameter.

The "point" class can actually be defined as a template class with an integer parameter 'N' to determine the dimension of the space under consideration. This way, the class is defined only once for 'N'-dimensional vectors; "point" and "point3" are obtained automatically as special cases by specifying 'N' to be 2 or 3.

The template class is called "point<N>" instead of "point", where 'N' stands for the dimension. The reserved words "template<int N>" that precede the class block indicate that this is indeed a template class that depends on a yet unspecified parameter 'N':

```
#include<stdio.h>
template<int N> class point{
    double coordinate[N];
  public:
    point(const point&);
```

The copy constructor that is declared here will be defined explicitly later.

Because the "coordinate" field appears before the reserved word "public:", it is by default a private member of the "point" class, which can be accessed only by members and friends of the class. Still, we want users to be able to read (although not change) the 'i'th coordinate in a "point" object from ordinary functions written outside the class block. For this purpose, we define the following public member operator:

```
    double operator[](int i) const{
      return coordinate[i];
    }  //  read ith coordinate
};
```

This completes the block of the "point" template class.

The above "operator[]" allows users to read the 'i'th coordinate in a "point" object 'P' simply as "P[i]". This is a read-only implementation; that is, the 'i'th coordinate can be read but not changed. This property is obtained by returning a copy of the 'i'th coordinate

rather than a reference to it, as is indicated by the word "double" (rather than "double&") that precedes the function name.

Another possible strategy to implement the read-only operator is to define the returned variable as a constant reference to double:

```
const double& operator[](int i) const{
  return coordinate[i];
}  //  read-only ith coordinate
```

This approach is preferable if the coordinates are themselves big objects that are expensive to copy; here, however, since they are only "double" objects, it makes no difference whether they are returned by value or by constant reference.

The "operator[]" function can also be implemented as a “read-write” operator as follows:

```
double& operator[](int i){
  return coordinate[i];
}  //  read/write ith coordinate (risky)
```

This version returns a nonconstant reference to the ’i’th coordinate, which can be further changed even from ordinary functions that are neither members nor friends of the "point" class. With this version, a call of the form "P[i]" can be made only if the "point" object ’P’ is nonconstant; otherwise, its ’i’th coordinate is constant and cannot be referred to as nonconstant, for fear that it will be changed through it. For this reason, the reserved word "const" before the ’{’ character that opens the function block is missing: this indicates that the current object can be changed by the function and, therefore, cannot be constant.

The latter version, however, is somewhat risky, because the coordinates can be changed inadvertently, which will spoil the original "point" object. This version must, therefore, be used with caution and only when necessary.

It is also possible to define "operator()" rather than "operator[]" by using parentheses. The ’i’th coordinate in the "point" object ’P’ is then read as "P(i)" rather than "P[i]". Here, there is no real difference between the two styles. In other cases, however, "operator()" may be more useful because it may take any number of arguments, whereas "operator[]" must take exactly one argument.

The only thing left to do in the "point" template class is to define explicitly the copy constructor declared above. This task is discussed and completed below.

The "point" template class must have a copy constructor, because the default copy constructor provided by the compiler does the wrong thing. Indeed, this copy constructor just copies every data field from the copied object to the constructed object. Since the field in the "point<>" template class is an array, it contains the address of a "double" variable. As a result, the default copy constructor just copies this address to the corresponding field in the constructed object, and no new array is created, and both objects have the same array in their data field. The result is that the constructed object is not really a new object but merely a reference to the old one, which is definitely not what is required. The copy constructor in the code below, on the other hand, really creates a new object with the same data as in the old one, as required.

The words "template<int N>" before the definition indicate that this is a template function. The prefix "point<N>::" before the function name indicates that this is a definition of a member function:

```
template<int N> point<N>::point(const point&P){
   for(int i = 0; i < N; i++)
     coordinate[i] = P.coordinate[i];
}  //  copy constructor
```

The default constructor available in the C++ compiler is used here to construct new "point" objects. In fact, it allocates memory for an array of length 'N' with components of type "double". This is why 'N' must be known in compilation time. The components in this array are not yet assigned values and are initialized with meaningless, random values.

This constructor is also invoked automatically at the start of the above copy constructor, since no initialization list is available in it. The correct values are then copied from the coordinates in the copied object to the corresponding coordinates in the constructed object.

Here is how the "point<N>" template class is actually used in a program:

```
int main(){
  point<2> P2;
  point<3> P3;
  printf("P2=(%f,%f)\n",P2[0],P2[1]);
  return 0;
}
```

When a concrete "point<N>" object is created, the parameter 'N' must be specified numerically and used in every function that uses the object as a current object or a concrete argument. For example, "P2" and "P3" in the above code are constructed as points in the two-dimensional and three-dimensional spaces, respectively, and the functions that are applied to them also use 'N' $= 2$ and 'N' $= 3$, respectively.

The above template class uses only a single parameter, the integer 'N'. More advanced template classes may use several parameters of different types. In particular, a parameter may specify not only the value but also the class that is used within the template class. For example, the above "point" template class can use a coordinate of type 'T', where 'T' is a parameter that should be specified in compilation time as integer, double, complex, or any other type or class. This gives the user greater freedom in choosing the type of coordinate in the "point" object.

When the template class is defined, the type 'T' is not yet specified. It is only specified at the call to the constructor of the class. The template class "point<T,N>" that implements 'N'-dimensional points with coordinates of type 'T' is written similarly to the "point<N>" class above, except that "<int N>" is replaced by "<class T, int N>"; "<N>" is replaced by "<T,N>"; and "double" is replaced by 'T':

```
template<class T, int N> class point{
    T coordinate[N];
};

int main(){
  point<double,2> P2;
  return 0;
}
```

In the next section we provide a more complete version of this template class with many useful functions, from which both two-dimensional and three-dimensional points can be obtained as special cases.

2.18 Example: The Vector Object

In this section, we present the "vector<T,N>" template class that implements an N-dimensional vector space, in which each vector has N components of type 'T'. The arithmetic operators implemented in this class provide a useful and convenient framework to handle vectors. The two-dimensional and three-dimensional point objects are also obtained as a special case.

Most operators and functions, except very short ones, are only declared inside the class block below, and their actual definition is placed outside it later on. The prefix "vector<T,N>::" that precede a function name indicates that a member function is defined. The words "template<class T, int N>" that precede the definition of a function indicate that this is a template function that uses the as yet unspecified type 'T' and integer 'N':

```
#include<stdio.h>
template<class T, int N> class vector{
    T component[N];
  public:
    vector(const T&);
    vector(const vector&);
    const vector& operator=(const vector&);
    const vector& operator=(const T&);
```

So far, we have only declared the constructor, copy constructor, and assignment operators with scalar and vector arguments. The actual definitions will be provided later. Next, we define the destructor. Actually, the destructor contains no command. The actual destruction is done by the default destructor invoked at the end of the function:

```
    ~vector(){
    }  //  destructor
```

Because the components in the vector are private class members, we need public functions to access them. These functions can then be called even from nonmember and nonfriend functions:

```
    const T& operator[](int i) const{
      return component[i];
    }  //read ith component

    void set(int i,const T& a){
      component[i] = a;
    }  //  change ith component
```

Here we declare more member arithmetic operators, to be defined later:

```
    const vector& operator+=(const vector&);
    const vector& operator-=(const vector&);
    const vector& operator*=(const T&);
    const vector& operator/=(const T&);
};
```

This concludes the block of the "vector" class. Now, we define the member functions that were only declared in the class block: the constructor, copy constructor, and assignment operators with vector and scalar arguments:

```
template<class T, int N>
vector<T,N>::vector(const T& a = 0){
   for(int i = 0; i < N; i++)
     component[i] = a;
}  //  constructor
template<class T, int N>
vector<T,N>::vector(const vector<T,N>& v){
  for(int i = 0; i < N; i++)
    component[i] = v.component[i];
}  //  copy constructor

template<class T, int N>
const vector<T,N>& vector<T,N>::operator=(
          const vector<T,N>& v){
  if(this != &v)
    for(int i = 0; i < N; i++)
      component[i] = v.component[i];
  return *this;
}  //  assignment operator

template<class T, int N>
const vector<T,N>& vector<T,N>::operator=(const T& a){
  for(int i = 0; i < N; i++)
    component[i] = a;
  return *this;
}  //  assignment operator with a scalar argument
```

Next, we define some useful arithmetic operators:

```
template<class T, int N>
const vector<T,N>&
vector<T,N>::operator+=(const vector<T,N>&v){
    for(int i = 0; i < N; i++)
      component[i] += v[i];
    return *this;
}  //  adding a vector to the current vector
```

```
template<class T, int N>
const vector<T,N>
operator+(const vector<T,N>&u, const vector<T,N>&v){
  return vector<T,N>(u) += v;
}  //  vector plus vector
```

The following are unary operators that act upon a vector as in common mathematical formulas: $+v$ is the same as v, and $-v$ is the negative of v.

```
template<class T, int N>
const vector<T,N>&
operator+(const vector<T,N>&u){
  return u;
}  //  positive of a vector

template<class T, int N>
const vector<T,N>
operator-(const vector<T,N>&u){
  return vector<T,N>(u) *= -1;
}  //  negative of a vector
```

The following functions return the inner product of two vectors and the sum of squares of a vector:

```
template<class T, int N>
const T
operator*(const vector<T,N>&u, const vector<T,N>&v){
    T sum = 0;
    for(int i = 0; i < N; i++)
      sum += u[i] * +v[i];
    return sum;
}  //  vector times vector (inner product)

template<class T, int N>
T squaredNorm(const vector<T,N>&u){
    return u*u;
}  //  sum of squares
```

Finally, here is a function that prints a vector to the screen:

```
template<class T, int N>
void print(const vector<T,N>&v){
  printf("(");
  for(int i = 0;i < N; i++){
    printf("v[%d]=",i);
    print(v[i]);
  }
  printf(")\n");
}  //  printing a vector
```

The template class "vector<T,N>" and its functions are now complete. The two-dimensional and three-dimensional point classes can be obtained from it as special cases:

```
typedef vector<double,2> point;
typedef vector<double,3> point3;
```

The "typedef" command gives a short and convenient name to a type with a long and complicated name. This way, "point" is short for a two-dimensional vector, and "point3" is short for a three-dimensional vector. In what follows, we show how the "point" class can be further used to derive an alternative implementation for the "complex" class.

2.19 Inheritance

In Section 2.16, we defined the "complex" class that implements complex numbers. The data hidden in a "complex" object are two "double" fields to store its real and imaginary parts. In this section we introduce a slightly different implementation, using the geometric interpretation of complex numbers as points in the two-dimensional Cartesian plane, with the x-coordinate representing the real part and the y-coordinate representing the imaginary part. For example, one may write

```
class complex{
      point p;
    public:
      complex(const point&P):p(P){}

      complex(const complex&c):p(c.p){}

      const complex&operator=(const complex&c){
        p=c.p;
        return *this;
      }

      const complex&operator+=(const complex&c){
        p+=c.p;
        return *this;
      }
};
```

and so on. This implementation uses the “has a” approach: the "complex" object has a field "point" to contain the data. In this approach, one must explicitly write the functions required in the "complex" class.

The above approach is not quite natural. Mathematically, the complex number doesn’t “have” any point, but rather “is” a point in the two-dimensional Cartesian plane. This leads us to the “is a” approach available in C++. In this approach, the "complex" object is actually a "point" object with some extra features or functions. This approach is more in the spirit of object-oriented programming in this case, because it allows one to implement complex numbers precisely as they are in mathematical terms: points in the Cartesian plane with some extra algebraic features.

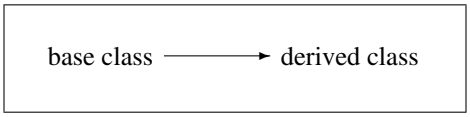

Figure 2.1. *Schematic representation of inheritance.*

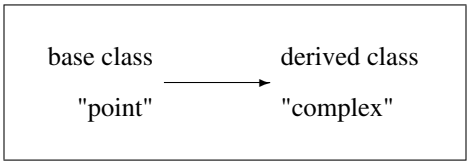

Figure 2.2. *Schematic representation of inheritance from the base class "point" to the derived class "complex".*

The “is a” concept in C++ is implemented by inheritance or derivation. The new class is derived from the “base” class and inherits its properties (see Figure 2.1). In the present example, the "complex" class is derived from the "point" class and inherits its properties as a point in the two-dimensional Cartesian plane (see Figure 2.2). On top of these properties, more arithmetic operations that are special to complex numbers can be defined in the derived "complex" class. These algebraic operations (multiplication and division of two complex numbers) complete the Cartesian plane from a mere vector space into a complete mathematical field.

The definition of the derived class (derivation) is similar to the standard class definition, except that the name of the derived class is followed by the character ’:’ followed by the reserved word "public" and the name of the base class from which it is derived. All these words precede the ’{’ character that opens the block of the derived class:

```
class complex : public point{
```

The word "public" before the name of the base class is optional. However, without it the derivation is private in the sense that the nonpublic functions in the base class are unavailable from objects or functions of the derived class. In other words, the derived class has the same access rights as ordinary classes. This way, the users are completely "unaware" of the derivation in the sense that they are unable to use objects from the derived class in functions written in terms of the base class. This is definitely not what we want here: in fact, we definitely want users of the "complex" class to be able to add and subtract complex numbers as if they were mere "point" objects. Therefore, we use here public derivation by writing the word "public" before the name of the base class as in the above code line.

Next, we implement the constructors in the body of the derived "complex" class:

```
public:
  complex(const point&p){
    set(0,p[0]);
    set(1,p[1]);
  }  //  constructor with "point" argument
```

```
    complex(double re=0., double im=0.){
      set(0,re);
      set(1,im);
    }  //  constructor with "double" arguments
```

When a "complex" object is constructed, the underlying "point" object is first constructed by its own default constructor, which sets the values of its fields to 0. These values are then reset to their correct values obtained from the argument. This resetting must use the public "set" function in the "vector" template class in Section 2.18. It is impossible to change the components of a vector object directly because they are declared as private in the "vector" class and are thus inaccessible not only by users but also by derivers.

Next, we define a friend function that calls the constructor to return the complex conjugate:

```
    friend complex operator+(const complex&c){
      return complex(c[0], -c[1]);
    }  //  complex conjugate
```

Next, we declare member arithmetic operations that do not exist in the base "point" class or should be rewritten. The actual definition will be given later:

```
    const complex&operator+=(double);
    const complex&operator-=(double);
    const complex&operator*=(const complex&);
    const complex&operator/=(const complex&);
};
```

This completes the block of the derived "complex" class.

The derived class has no access to the private fields of the base class. However, it has access to “half private” fields: fields that are declared as "protected" in the block of the base class by simply writing the reserved word "protected:" before their names. These fields are accessible by derivers only, not by other users. In fact, if the "component" field is declared "protected" in the base "vector" class, then it can be accessed from the derived "complex" class and set in its constructors directly. With the present implementation of the "vector" class in Section 2.18, however, the "component" field is private, so the constructors of "complex" objects must access it indirectly through the public "set" function as above.

In summary, the members of a class are of three possible kinds: (a) public members that can be used by everyone; (b) private members that are accessible only to members and friends; and (c) protected members that are accessible to members and friends of derived classes (or when called in conjunction with current objects that have been derived by public derivation), but not to ordinary functions, even if they take arguments from the class or any other class derived from it (see Figure 2.3).

When the derived object is constructed, the data fields that belong to the underlying base class are constructed first by the default constructor of the base class. Thus, the constructors in the derived class cannot use an initialization list for these fields, as they are already initialized to their default values. If these data fields are declared protected in the base class, then they can be reset in the body of the constructors of the derived class. If, on the other hand, they are private members of the base class, then they can be reset only

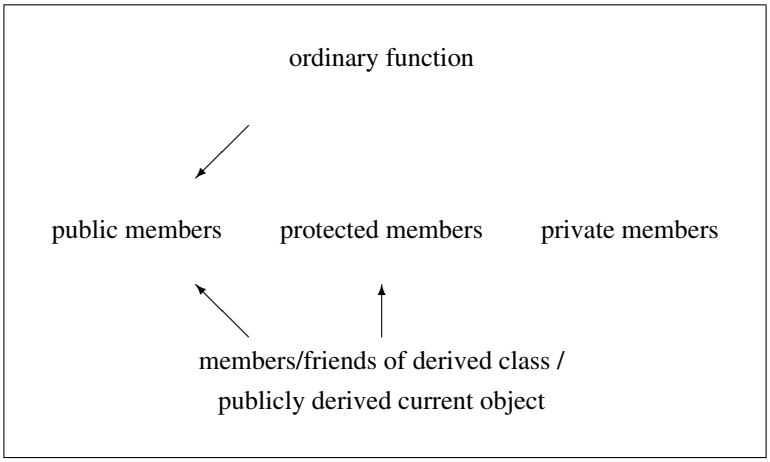

Figure 2.3. *The three kinds of members of a class (public, protected, and private) and their access pattern.*

indirectly through public member functions like "set" in the above example. In this example, the "set" function is applied to the "complex" object that is currently constructed. Since no such function is available in the derived "complex" class, the "complex" object is interpreted as a "point" object, and the "set" function of Section 2.18 is invoked to reset the data fields that represent the real and imaginary parts of the complex number to their correct values.

Similarly, when a "complex" object is passed as a concrete argument to a function, the compiler first looks for this function among the functions that take a "complex" argument. Only if no such function exists does it interpret the passed "complex" object as a "point" object and look for a function that takes a "point" argument.

When a derived object is destroyed, the data members inherited from the base class are destroyed last. This is done by the default destructor of the base class, which is invoked automatically at the end of the destruction and destroys the members inherited from the base class in reverse order to the order in which they are declared in it. For this reason, the destructor in the "complex" class needs to do nothing: the destructor in the base "point" class does the actual destruction implicitly. To invoke it, the default destructor available automatically in the "complex" class is sufficient, and no explicit destructor needs to be written.

The above discussion indicates that the process of inheritance suffers from slight overhead in terms of both time and storage due to the base object hidden behind every derived object. Still, it may be well worth it for the sake of elegant and transparent programming and for using objects that are already implemented properly to derive various kinds of more advanced objects. Furthermore, inheritance gives us the opportunity to follow the true concept of mathematical objects and the relation between them. It is particularly useful in high-level programming, where the programmer can concentrate on the special properties of the new object derived from more technical elementary objects.

The members of the base class also function as members of the derived class and can thus be used in the definition of its own members. Furthermore, since the derivation is public, the public members of the base class remain public in the derived class and can thus be used by its users.

Members of the base class can also be rewritten in the derived class. In this case, the version in the derived class overrides the version in the base class. For example, the "*=" and "/=" operators in the "complex" class override the corresponding operators in the base "point" class and are therefore used for "complex" objects. One can still call the version in the base class by adding a prefix of the form "base::" (where "base" stands for the name of the base class) before the function name to indicate that the old version is called. For example, "point::operator*=" invokes the "*=" operator in the base "point" class.

The unary '+' operator defined above also overrides the corresponding operator in the "point" class and returns the complex conjugate of a complex number. The '+' symbol is chosen for this purpose because it represents the only operator that leaves real numbers unchanged, as indeed does the complex-conjugate operator when applied to complex numbers with zero imaginary part. This operator is used in the division of complex numbers:

```
const complex&
complex::operator*=(const complex&c){
  double keep0 = (*this)[0];
  set(0,(*this)[0]*c[0]-(*this)[1]*c[1]);
  set(1,keep0*c[1]+(*this)[1]*c[0]);
  return *this;
}  //  multiplying by complex

const complex&
complex::operator/=(const complex&c){
  return *this *= (complex)((point)(+c)/=squaredNorm(c));
}  //  dividing by complex
```

The above "/=" operator works as follows. Dividing by the complex number c is the same as multiplying by $\bar{c}/|c|^2$. First, $|c|^2$ is calculated by the "squaredNorm" function. Since the derived "complex" class has no such function, the argument 'c' is interpreted as a "point" object, and the "squaredNorm" function of the base "vector" class in Section 2.18 is invoked. The unary '+' operator, on the other hand, is available in both base and derived classes, so the version in the derived "complex" class overrides the version in the base "vector" class. As a result, "+c" is interpreted as the required complex conjugate $\bar{c}$. Now, the compiler needs to invoke a "/=" operator to divide the "complex" object $\bar{c}$ by the "double" number $|c|^2$. However, if one attempts to do this naively, then, since no division by "double" is available in the "complex" class, the compiler will implicitly convert the number $|c|^2$ from "double" to "complex" and invoke the "/=" operator recursively. Of course, this will lead to infinitely many recursive calls, with no result.

The cure to the above problem is to convert $\bar{c}$ explicitly from "complex" to "point" by adding the prefix "(point)", which converts a derived object to the underlying base object, while preserving its value. The "point" object $\bar{c}$ is then divided by the scalar $|c|^2$ unambiguously by the "/=" operator of the "vector" class in Section 2.18, as required.

The resulting "point" object $\bar{c}/|c|^2$ is then converted back to a "complex" object by the prefix "(complex)", which invokes the constructor that takes a "point" argument and constructs a "complex" object with the same value. Since $\bar{c}/|c|^2$ is now interpreted as a complex number, it can multiply the current "complex" object stored in "this", which completes the required division by the complex number c.

The above implementation involves two extra conversions, which may cause considerable overhead in applications that divide many times. These conversions are avoided in the following alternative implementation:

```
const complex&
complex::operator/=(const complex&c){
 return *this *= (+c).point::operator/=(squaredNorm(c));
}  //  dividing by complex
```

In this code, the prefix "point::" before the inner call to the "/=" operator indicates that the "/=" operator of the base "point" class is to be used. This implementation produces the correct result with no conversion at all.

There is no need to rewrite the "+=" and "−=" operators with a "complex" argument, because the corresponding operators in the base "point" class work just fine. However, there is a need to write explicitly the "+=" and "−=" operators with a "double" argument. Otherwise, the compiler would implicitly convert the "double" argument a to the "point" object (a,a), as in the constructor that takes a double argument in Section 2.18. Of course, this object has the wrong value; the correct value should be $(a,0)$. Therefore, these operators must be implemented explicitly as follows:

```
const complex& complex::operator+=(double a){
  set(0,(*this)[0] + a);
  return *this;
}  //  adding a real number

const complex& complex::operator-=(double a){
  set(0,(*this)[0] - a);
  return *this;
}  //  subtracting a real number
```

The same applies to the (nonmember) binary '+' and '−' operators. These should be rewritten for mixed "complex" and "double" arguments as follows:

```
const complex
operator+(double a, const complex&c){
  return complex(c) += a;
}  //  double plus complex

const complex
operator+(const complex&c, double a){
  return complex(c) += a;
}  //  complex plus double

const complex
operator-(double a, const complex&c){
  return complex(a) - c;
}  //  double minus complex
```

```
const complex
operator-(const complex&c, double a){
  return complex(c) -= a;
}  //  complex minus double
```

The above member operators "*=" and "/=" are now used as in Section 2.16 to implement binary multiplication and division operators:

```
const complex
operator*(const complex&c, const complex&d){
  return complex(c) *= d;
}  //  complex times complex

const complex
operator/(const complex&c, const complex&d){
  return complex(c) /= d;
}  //  complex divided by complex
```

The present implementation clearly shows the risks taken in implicit conversion. For example, when the compiler encounters an expression like "3. * c", where 'c' is a complex number, it doesn't know whether to convert implicitly the real number '3.' into a complex number and invoke a complex-times-complex multiplication, or treat 'c' as a "point" object and invoke the scalar-times-point operator of the base "point" class. Although the mathematical result is the same, the compiler doesn't know that in advance and issues a compilation error due to ambiguity.

In order to avoid this problem, operators with mixed "complex" and "double" arguments must be defined explicitly as follows:

```
const complex
operator*(double a, const complex&c){
  return (point)c *= a;
}  //  double times complex

const complex
operator*(const complex&c, double a){
  return (point)c *= a;
}  //  complex times double

const complex
operator/(const complex&c, double a){
  return (point)c /= a;
}  //  complex divided by double
```

In some cases, it is also recommended to implement explicitly the binary addition and subtraction operators to avoid ambiguity:

```
const complex
operator+(const complex&c, const complex&d){
  return complex(c) += d;
}  //  complex plus complex
```

```
const complex
operator-(const complex&c, const complex&d){
  return complex(c) -= d;
}  //  complex minus complex
```

Here is how complex numbers are actually used in a program:

```
int main(){
  complex c=1.,i(0.,1.);
  complex d=c*3+4.*i;
  print(c+1);
  print(c/d);
  return 0;
}
```

It seems that the original implementation of the "complex" object in Section 2.16 is simpler and cheaper than the latter one, because it avoids constructing a base "point" object and converting implicitly to and from it whenever a base-class function is called. Still, inheritance may be invaluable in many applications, as we'll see below. The present implementation of the "complex" class can be viewed as a good exercise in using inheritance.

2.20 Example: The Matrix Object

Here we use inheritance to implement the "matrix" object and useful arithmetic operations with it. First, we describe briefly the object and its main functions.

Suppose that we want to implement an $N \times M$ matrix, that is, a matrix with M columns of N elements each:

$$A \equiv \left(A_{i,j}\right)_{0 \le i < N,\ 0 \le j < M} = (c_0 \mid c_1 \mid \cdots \mid c_{M-1}).$$

Here the matrix A is represented as a set of M columns $c_0, c_1, \ldots, c_{M-1}$, each of which contains N elements.

For an N-dimensional vector u, the product u times A is the M-dimensional vector

$$uA \equiv ((u,c_0),(u,c_1),\ldots,(u,c_{M-1})),$$

where (u,c_j) is the inner product of u and c_j.

For an M-dimensional vector $v = (v_0, v_1, \ldots, v_{M-1})$, the product A times v is the N-dimensional vector

$$Av \equiv \sum_{j=0}^{M-1} v_j c_j.$$

Let B be a $K \times N$ matrix. The product B times A is the $K \times M$ matrix

$$BA \equiv (Bc_0 \mid Bc_1 \mid \cdots \mid Bc_{M-1}).$$

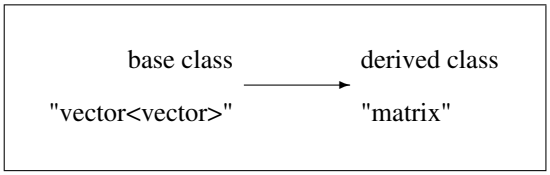

Figure 2.4. *Schematic representation of inheritance from the base class "vector<vector>" to the derived class "matrix".*

In order to implement the "matrix" object, we use the vector as in Section 2.18, with components that are themselves vectors, representing the columns in the matrix. In other words, the matrix is implemented as a vector of vectors. For this purpose, the "matrix" class is derived from the "vector<vector>" class (see Figure 2.4). The three different kinds of products (vector-matrix, matrix-vector, and matrix-matrix) are then implemented exactly as in the above mathematical definitions.

The implementation of the template class uses three parameters, to be specified later in compilation time: 'T' to specify the type of elements, 'N' to specify the number of rows, and 'M' to specify the number of columns. This way, the definition is general, with unspecified type and dimensions:

```
template<class T, int N, int M>
class matrix : public vector<vector<T,N>,M>{
  public:
      matrix(){}  //  default constructor
```

With this default constructor, the user can define a "matrix" object with no arguments whatsoever. The compiler then invokes implicitly the default constructor of the base "vector" class, to construct the underlying "vector" object.

Furthermore, the following constructor converts implicitly a base "vector" object into the derived "matrix" object:

```
    matrix(const vector<vector<T,N>,M>&){
    }  //  implicit convertor
```

With this convertor, functions of type "matrix" (namely, functions that return a "matrix" object) may return a base "vector" object in their "return" command. This object will then be converted implicitly into the required "matrix" object upon the termination of the function.

Next, we turn to a more meaningful constructor that actually does something. More specifically, although the underlying "vector" object that is created by the default constructor of the base "vector" class contains initially trivial zero elements only, it is further changed throughout the block of the function to take more meaningful values:

```
      matrix(const vector<T,N>&u, const vector<T,N>&v){
        set(0,u);
        set(1,v);
      }  //  constructor with 2 columns
```

In some compilers, however, including advanced versions of the GNU compiler, members of templated base classes such as "vector" cannot be accessed from derived classes by their names only. With these compilers, the above calls to the function "set" inherited from the base "vector" class must be replaced by "vector::set" or "this->set":

```
matrix(const vector<T,N>&u, const vector<T,N>&v){
  this->set(0,u);
  this->set(1,v);
}  //  constructor with 2 columns
```

Fortunately, here we use an earlier version of GNU such as Version 3.2, so this extra programming is unnecessary.

Here is also a useful function that reads only a prescribed matrix element:

```
const T& operator()(int i,int j) const{
  return (*this)[j][i];
}  //  read the (i,j)th matrix element
```

Finally, we also declare some member operators that must be defined later on to let the compiler know that they are applied to the underlying "vector" object in the same way as in the base "vector" class:

```
        const matrix& operator+=(const matrix&);
        const matrix& operator-=(const matrix&);
        const matrix& operator*=(const T&);
        const matrix& operator/=(const T&);
};
```

This concludes the block of the "matrix" class. The copy constructor, assignment operator, and destructor don't have to be defined here, because the corresponding functions in the base "vector<vector>" class work just fine. The operators that multiply and divide by a scalar, on the other hand, must be rewritten, because the corresponding operators in the base "vector" class return base "vector<vector>" objects rather than the required "matrix" objects. The actual definition of these operators is left as an exercise.

We can now define new types "matrix2" and "matrix3" of square matrices of orders 2 and 3, respectively, as special cases of the above template class:

```
typedef matrix<double,2,2> matrix2;
typedef matrix<double,3,3> matrix3;
```

The addition and subtraction of two "matrix" objects are done by the corresponding operators inherited from the base "vector" class, so they don't have to be rewritten here. The actual implementation of the above products of vector times matrix, matrix times vector, and matrix times matrix is left as an exercise, with detailed solution in Section A.5 of the Appendix. Assuming that these functions are available, one can define and manipulate matrices as follows:

```
int main(){
  matrix2 m(point(2,1),point(1,1));
```

```
    print(m+m);
    print(m-m);
    print(m*m);
    return 0;
}
```

Although multiplication operators exist in both the base and derived classes, derived-class functions override base-class functions, so they are the ones that are invoked, as required.

2.21 Determinant and Inverse of a Square Matrix

Two difficult and important tasks are the calculations of the determinant $\det(A)$ and the inverse A^{-1} of a square matrix A (of order N). The best way to do this is by using the LU decomposition of A described below.

Let $e^{(j)}$ be the jth standard unit vector, namely, the vector whose jth component is equal to 1, and all other components are equal to 0. The LU decomposition of a nonsingular square matrix A is given by

$$A = LUP,$$

where L, U, and P are square matrices (of order N) with the following properties.

1. L is lower triangular with main-diagonal elements that are equal to 1:

   $$L_{i,j} = 0, \quad 0 \le i < j < N,$$

   and

   $$L_{i,i} = 1, \quad 0 \le i < N.$$

2. U is upper triangular with nonzero main-diagonal elements (pivots):

   $$U_{i,j} = 0, \quad 0 \le j < i < N,$$

   and

   $$U_{i,i} \neq 0, \quad 0 \le i < N.$$

3. P is a permutation matrix, namely, a matrix whose columns are distinct standard unit vectors.

The above LU decomposition is obtained from Gaussian elimination, with suitable permutations of columns to avoid zero pivots. Here, we present a simplified version of this algorithm, in which it is assumed that the pivots are not too small in magnitude, so no permutation is needed, and P is just the identity matrix I.

Algorithm 2.1.

1. *Initialize* $L = \left(L_{i,j}\right)_{0 \le i,j < N}$ *to be the identity matrix* I.
2. *Initialize* $U = \left(U_{i,j}\right)_{0 \le i,j < N}$ *to be the same matrix as* A.
3. *For* $i = 1, 2, 3, \ldots, N-1$, *do the following:*

- *For* $j = 0, 1, 2, \ldots, i-1$, *do the following:*
  (a) *Define*
  $$factor = U_{i,j} / U_{j,j}.$$
  (b) *For* $k = j, j+1, \ldots, N-1$, *set*
  $$U_{i,k} \leftarrow U_{i,k} - factor \cdot U_{j,k}.$$
  (c) *Set*
  $$L_{i,j} \leftarrow factor.$$

The determinant of A can be calculated most efficiently as follows:

$$\det(A) = \det(L)\det(U)\det(P) = \pm\det(U) = \pm U_{0,0}U_{1,1}U_{2,2}\cdots U_{N-1,N-1}.$$

In other words, the determinant of A is just the product of pivots $U_{i,i}$ obtained during Gaussian elimination.

The LU decomposition is also useful in calculating the inverse of A, A^{-1}. Indeed, the jth column in A^{-1} is just the solution x of the vector equation

$$Ax = e^{(j)}.$$

This equation can be solved by substituting the above LU decomposition for A:

$$LUPx = e^{(j)}.$$

This equation is solved in three steps. First, the equation

$$Lz = e^{(j)}$$

is solved for the unknown vector z (forward elimination in L). Then, the equation

$$Uy = z$$

is solved for the unknown vector y (back substitution in U). Finally, since P is orthogonal, its inverse is the same as its transpose: $P^{-1} = P^t$. Therefore, the required vector x is obtained as

$$x = P^t y.$$

This way, neither L nor U needs to be inverted explicitly, which saves a lot of computation. Furthermore, the triangular factors L and U calculated during Gaussian elimination can be stored for further use. In fact, if A is no longer required, then they can occupy the same array occupied previously by A, to save valuable computer memory.

2.22 Exponent of a Square Matrix

The exponent of a square matrix A of order N is defined by the converging infinite series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{A^n}{n!},$$

where I is the identity matrix of order N.

This function can be approximated by either the Taylor or the Padé approximation in Chapter 1, Section 22 above, by just replacing the scalar x used there by the matrix A. For this purpose, the present "matrix" class is most helpful.

As in Chapter 1, Section 22, one must first find a sufficiently large integer m such that the l_2-norm of $A/2^m$ is sufficiently small (say, smaller than $1/2$). Since the l_2-norm is not available, we estimate it in terms of the l_1- and l_∞-norms:

$$\|A\|_2 \le \sqrt{\|A\|_1 \|A\|_\infty},$$

where the l_1- and l_∞-norms are given by

$$\|A\|_1 = \max_{0 \le j < N} \sum_{i=0}^{N-1} |A_{i,j}|,$$

$$\|A\|_\infty = \max_{0 \le i < N} \sum_{j=0}^{N-1} |A_{i,j}|.$$

Thus, by finding an integer m so large that

$$2\sqrt{\|A\|_1 \|A\|_\infty} < 2^m,$$

we also guarantee that the l_2-norm of $A/2^m$ is smaller than $1/2$, as required.

The algorithm to approximate $\exp(A)$ proceeds as in Chapter 1, Section 22, with either Taylor or Padé polynomials. The scalar x used in Chapter 1, Section 22, is replaced by the square matrix A. The codes in Chapter 1, Section 22, are easily adapted to apply also to square matrices, provided that the required arithmetic operations are well defined.

The power of templates is apparent here. Indeed, if the functions in Chapter 1, Section 22 were written as template functions, then they could be used immediately to calculate the exponent of a matrix by simply specifying their template to be of the "matrix" class. This approach is left as an exercise below.

2.23 Diagonalization

Here we present and implement an algorithm that diagonalizes a given square matrix A of order n. In other words, the algorithm produces two upper triangular matrices W and Z (whose main-diagonal elements are equal to 1) and a diagonal matrix D (whose only nonzero elements are of the form $D_{i,i}$, $1 \le i \le n$) such that

$$W^t A Z = D.$$

The algorithm is based on a generalized Gram–Schmidt process that starts from the columns of the identity matrix of order n and gradually produces from them two sets of vectors (the columns of W and Z in their final form) that are biconjugate to each other with respect to the bilinear form induced by A [15]. To make sure that there is no division by zero, we assume here for simplicity that the $i \times i$ upper-left block submatrices of A ($1 \le i < n$) are nonsingular.

Let $A^{(i)}$ denote the ith column in A, so $(A^t)^{(i)}$ is the ith row in A. The algorithm is now defined as follows [15]:

Algorithm 2.2.

- *Initialize both W and Z to be the identity matrix of order n.*
- *For $i = 1,2,\ldots,n$, do the following:*
  1. *For $j = 1,2,\ldots,i-1$, update $W^{(i)}$ (the ith column in W) by*

$$W^{(i)} \leftarrow W^{(i)} - \frac{\left((A^t)^{(i)}, Z^{(j)}\right)}{D_{j,j}} W^{(j)}$$

     [where the numerator in the above coefficient is the so-called real inner product (that declines to take the complex conjugate) of the ith row in A with the jth column in Z].

  2. *Similarly, for $j = 1,2,\ldots,i-1$, update $Z^{(i)}$ (the ith column in Z) by*

$$Z^{(i)} \leftarrow Z^{(i)} - \frac{\left(W^{(j)}, A^{(i)}\right)}{D_{j,j}} Z^{(j)}$$

     (where the numerator in the above coefficient is the real inner product of the jth column in W with the i column in A).

  3. *Finally, define*

$$D_{i,i} \equiv \left(W^{(i)}, AZ^{(i)}\right)$$

     (real inner product, with no complex conjugate).

Fortunately, this algorithm uses mainly column operations, so the implementation of the "matrix" object in Section 2.20 as a collection of column vectors is particularly suitable. To benefit from this structure fully, however, we must first add an extra "operator()" member function to the block of the base "vector" class in Section 2.18:

```
T& operator()(int i){
  return component[i];
}  //  read/write ith component
```

This operator can now be called from another "operator()" member function that should be added to the block of the derived "matrix" class in Section 2.20:

```
vector<T,N>& operator()(int i){
  return vector<vector<T,N>,M>::operator()(i);
}  //  read/write ith column
```

With this new operator, one may not only read the 'i'th column of a "matrix" object 'm' by writing "m[i]", but also change it by writing "m(i)", which returns a nonconstant reference to it.

Moreover, the above new operators are now used to add yet another version of "operator()" to the block of the derived "matrix" class in Section 2.20. This version returns a nonconstant reference to a particular matrix element:

```
T& operator()(int i,int j, char*){
  return (*this)(j)(i);
}  //  read/write A(i,j)
```

With this new version, one can not only read the (i,j)th element of a well-defined matrix 'm' by writing "m(i,j)", but also change it by writing, e.g., "m(i,j,"write")". This way, one can add to the class block in Section 2.20 another constructor that takes a "double" argument to construct a 2×2 matrix:

```
matrix(double d){
  (*this)(0,0,"write") = d;
  (*this)(1,1,"write") = d;
}  //  constructor of a 2 by 2 matrix
```

Furthermore, the above operator can also be used to produce the transpose of a given matrix:

```
template<class T, int N, int M>
const matrix<T, M, N>
transpose(const matrix<T, N, M>&A){
  matrix<T, M, N> At;
  for(int i=0; i<N; i++)
    for(int j=0; j<M; j++)
      At(j,i,"write") = A(i,j);
  return At;
}  //  transpose of a matrix
```

We are now ready to implement the diagonalization algorithm by simply following its original mathematical formulation:

```
template<class T, int N>
void diagonalize(matrix<T,N,N>&W,
    const matrix<T,N,N>&A,
    matrix<T,N,N>&Z, vector<T,N>&D){
  for(int i=0; i<N; i++){
    W(i,i,"write") = 1.;
    Z(i,i,"write") = 1.;
  }
  matrix<T,N,N> At = transpose(A);
  for(int i=0; i<N; i++){
    vector<T,N> r = At[i];
    for(int j=0; j<i; j++)
      W(i) -= r * Z[j] / D[j] * W[j];
    vector<T,N> c = A[i];
    for(int j=0; j<i; j++)
```

```
        Z(i) -= W[j] * c / D[j] * Z[j];
      D(i) = W[i] * (A * Z[i]);
    }
  }  //  diagonalize a matrix
```

In particular, the diagonal matrix D, whose main diagonal is stored in the vector 'D' produced in the above code, can now be used to compute the determinant of the original matrix A as the product of the $D_{i,i}$'s:

```
template<class T, int N>
const T det(const vector<T,N>&D){
  T result = 1.;
  for(int i=0; i<N; i++)
    result *= D[i];
  return result;
}  //  determinant using diagonalization
```

2.24 Exercises

1. Implement complex numbers as a template class "complex<T>", where 'T' is the type of the real and imaginary parts. Define the required arithmetic operations and test them on objects of type "complex<float>" and "complex<double>".

2. Implement complex numbers in polar coordinates: a "complex" object contains two fields, 'r' and "theta", to store the parameters $r \geq 0$ and $0 \leq \theta < 2\pi$ used in the polar representation $r\exp(i\theta)$. Define and test the required arithmetic operations.

3. Do users of the "complex" class have to be informed about the modification made above? Why?

4. Complete the missing operators in Section 2.18, such as subtraction of vectors and multiplication and division by a scalar. The solutions are given in Section A.4 of the Appendix.

5. Implement the operators that add and subtract two matrices. The solutions are given in Section A.5 of the Appendix.

6. Implement the vector-matrix, matrix-vector, matrix-matrix, and scalar-matrix products that are missing in the code in Section 2.20. The solutions are given in Section A.5 of the Appendix.

7. Write functions that return the transpose, inverse, and determinant of 2×2 matrices ("matrix2" objects in Section 2.20). The solution can be found in Section A.5 of the Appendix.

8. Rewrite the "expTaylor" and "expPade" functions in Chapter 1, Section 22, as template functions that take an argument of type 'T', and apply them to an argument A of type "matrix2" to compute $\exp(A)$. Make sure that all the required arithmetic operations between matrices are available in your code.

9. Apply the above functions to objects of type "matrix<complex,4,4>", and verify that, for a complex parameter λ,

$$\exp\left(\begin{pmatrix} \lambda & & & \\ 1 & \lambda & & \\ & 1 & \lambda & \\ & & 1 & \lambda \end{pmatrix}\right) = \exp(\lambda) \begin{pmatrix} 1 & & & \\ 1/1! & 1 & & \\ 1/2! & 1/1! & 1 & \\ 1/3! & 1/2! & 1/1! & 1 \end{pmatrix}$$

   (the blank spaces in the above matrices indicate zero elements).

10. Compare your code to the Fortran code (461 lines) that can be found on the Web page of the book at http://www.siam.org/books/cs09. Are the numerical results the same? Which code is easier to read and use?