

Chapter 16

Modules and Object Classes

Overview

For clarity and easy debugging, a small program should be written as a collection of functions. However, use of subprograms alone is inadequate for a large, complex system. Encapsulated modules were developed as tools for managing team efforts and complex tasks. An encapsulated module is bigger than a function definition, yet smaller than a program. These modules facilitate the creation of code libraries, separate compilation, the grouping of logically related elements for the implementation of abstract data types, the sharing of public data, and the protection of private information.

Separately compiled C files, Ada packages, and object classes in C++ are three strategies for achieving modularity. Before module-related facilities were included within programming languages, files and separate compilation were used to achieve the goals of modularity. In a modularized C program, an object or function in one file can be shared by a program in another file by using an “extern” specification, and kept private through the use of the keyword “static”. The operating system’s linking loader completed the connections among the program’s parts. An automated method, the makefile, was invented to make linking UNIX applications easier and more foolproof.

Packages in Ada provide the framework to group the variables and functions of an abstract data type. A package consists of a header which declares shared data and a body which defines private symbols and the ADT functions. Unlike C, Ada’s goals of modularity are achieved from within the language.

In C++, the class serves a similar role as the package in Ada. Classes contain both functions and data. C++ also allows classes to be constructed in a hierarchical fashion so that they are related to each other and can inherit data and functions. Within a

Exhibit 16.1. Terms for modules.

C		a separately compiled file
Ada		package
Modula		module
CLU		cluster
C++, Simula, Smalltalk		class

class, the definition of a function is called a method. The C++ class is a template that must be instantiated by the use of the class name in a declaration. The result is an object whose fields will be initialized if the class contains an initialization function called a constructor.

Some relationships among classes are difficult to achieve using only public and private parts. There are times when we would like parts shared with some modules but not all. C++ provides a solution for this problem: a class or a function may be declared to be a friend of another class. Access is then shared by the parent class and the friend class.

16.1 The Purpose of Modules

It is universally accepted by computer scientists that programs should be designed and implemented in a modular fashion. For small programs, this merely means that a program is written as a collection of function calls. Each function definition must be short, its purpose clearly defined, and its interface clearly declared and documented. Global variables are avoided in order to minimize unwanted interaction between functions and side effects.

When writing large systems programs, though, this basic modular methodology is helpful but inadequate. The complexity of a large system is so great that it must be organized as something other than a list of thousands of function definitions. Another level of program structure is needed so that things that belong together can be grouped and isolated from all possible outside interference. The semantic bases of many modern languages have been extended to support a unit that is larger than a single function but smaller than a program. Such units are called by varied names [Exhibit 16.1] but have similar purposes and properties, which we examine here. Sample modules are given in Ada and in C++.

Encapsulated modules were developed in response to several needs that have been mentioned earlier. These are:

- To facilitate separate compilation and creation of code libraries.

- To enable the logical grouping of type declarations and relevant representation-dependent functions.
- To support implementation-independent abstract data types.
- To implement private information (static local data).
- To implement controlled, nonhierarchical sharing of data.

Grouping Logically Related Elements. A module provides a framework for grouping together a set of related functions and is thus an ideal way to implement an ADT. Within the module, the details of the implementation and functions are defined that access specific fields in the representation. The module must include all representation-dependent functions that are needed to use the ADT domains. These can be written, compiled, and tested in relative isolation. The ADT functions often call each other and might use information that is private within the module. Their interdependence is documented by their inclusion in the same module. The `array_stack`, defined in Exhibit 15.9, is a good example; in Section 16.3 we give an implementation of this ADT in Ada.

Sharing Public Facilities. To be useful, a module must have some functions or data objects that are public and can be accessed by other parts of the program. These symbols are “made known” to other modules, on demand. If a program, P , wishes to use a module, M , then P must contain a command to *include* M . Then within P , the public symbols in M can be referenced.

Protecting Private Information. An encapsulated module contains declarations for types, functions, and data objects. These declarations are separated into two groups, of *public* and *private* symbols. Public symbols may be accessed by other modules and form the interface between a module and the rest of the world. In an ADT implementation, the ADT functions would be declared to be public symbols, so that other parts of a program could call them. In addition, the module might contain private functions which are called by the ADT functions but cannot be called from outside the module.

The `array_stack` ADT is particularly simple and does not require any locally defined types. To implement more complex ADTs, however, we must be able to make private type declarations. Having the possibility of internally defined types also enables the representation of the ADT to be changed easily, simply by changing the private type declaration. The effects of that change are limited to the scope of the module definition. Once the module’s functions are redefined to work on the new representation, all code that depends on those functions would automatically work. For example, suppose a module were used to define a stack, and the representation chosen initially was an array. Later, it became clear that a linked list representation would fit the data characteristics better. To make this change, the data type of the stack storage and definitions of the stack functions (all within the module) would have to be changed. However, when the outer program called `push` or `pop`, the call would function as expected and store or return a value. The caller would never know that the representation of the ADT had been changed.

A module must provide for private data. Many large systems have modules that operate as coroutines, acting asynchronously to carry out a set of related concurrent tasks. For example, an application that uses dynamic storage allocation will often have a separate memory management module which gets storage from the operating system in efficient quantities and manages lists of new and freed cells. I/O buffering and caching are done by coroutines. To support coroutines we need private data objects with a static lifetime that can be shared among all the functions in the module. These must be created and initialized when the module is first entered.

Often, a group of private data declarations can be used in a module in place of a single object of a record type. In Pascal, for example, a stack is usually implemented as a record consisting of an array of data slots and an integer top-of-stack subscript. We declare a record data type with these parts so that a stack can be passed around as one coherent argument. However, when we implement a stack module, the stack-data-object is internal to the module, and within the module it is global to all the module's functions. It is not passed anywhere as a parameter. In such cases, there is no real advantage to gathering the two parts of the stack into one record.

The data parts of the ADT would be declared in the private part of the module. Private symbols can be used only within the module; we say they are *hidden* within it. In an ordinary function, the local variables are similarly hidden. However, local variables cannot do all that is needed. By concealing the data-parts of the ADT, we force the programmer to access them *only* through the ADT functions. The nature, type, extent, and value of all private data is concealed from the program outside the module. This organization forces the programmer to use an implementation-independent coding style, making the entire package easier to debug and modify.

Implementation Strategies. We will examine three basic strategies for achieving modularity that are in common use:

1. C's separately compiled files.
2. Ada packages.
3. Object classes in C++.

16.2 Modularity Through Files and Linking

Before module-related facilities were included within programming languages, files and separate compilation were used to achieve the goals of modularity. This was not an altogether satisfactory solution, since it relied on the operating system environment, specifically the linker, to complete the connections among the parts of the program. The structure of the application as a whole was not expressed anywhere within the program code! Instead, the user determined this structure by telling the system linker which modules to include.

Giving complex commands to a linker is error prone and cumbersome, so an automated method, the *makefile*, was invented for linking UNIX applications. Currently, several systems support some kind of "make" facility. The example we give and explain here is written in ANSI C and UNIX.

Exhibit 16.2. Header file: “modules.h”.

```
/* Set this constant to the number of data items in your array. */
#define LEN 4

/* Set these definitions to the desired data type and format specifier. */
/* NUMBER must be defined to be a standard numeric type. */
#define D "%ld"
typedef long int NUMBER;
extern NUMBER reduce(NUMBER (*)(), NUMBER)
```

A modularized program in C consists of four or more files. We list their purposes here, then examine an example of each written in C. The files are as follows:

- A makefile, which represents the entire program. It lists the components and describes how each depends on the others. The contents of this file are not C code. Rather, they are operating system commands, interpreted by the system’s “make” facility. When you “make” a program, any necessary compilation and linking is done, according to the instructions given in the makefile. The result is a machine-code module that is ready to load and run.
- A header file, containing declarations and definitions of symbols that must be shared by two or more code modules in order to attain consistency.
- A main module, containing the main program, where execution begins. Any declarations in the main program are local to it and are not shared by other modules.
- One or more modules containing functions that are called by main. These share the header file with main, but are compiled separately and may contain private declarations of any sort. These modules often are used for coroutines that handle buffering and storage management.

Library Packages

We present a short interactive C application consisting of four files to show how the parts of an application are written and combined. The header file [Exhibit 16.2] must be included in the other files when they are compiled; note the `#include` statements at the top of Exhibits 16.3 and 16.4. A header file provides a way to coordinate the assumptions made in the code modules. In this example, the code modules both need to know the length and base type of the arrays that will be processed. Using the particular constant definitions in Exhibit 16.2, we would work with arrays of four long integers and print the answers out in a “%ld” (long decimal) format field. However, we could change these three declarations to handle any other length and base type. For example, to work with arrays of ten floating-point numbers, we would edit the header file to say:

Exhibit 16.3. Main module: “sumup.c”.

```

#include <stdio.h>
#include "modules.h"
/* This constant is the number of array operations that are defined. */
#define OPS 6

NUMBER fi_plus(a, b)    NUMBER a, b; {return a + b;}
NUMBER fi_times (a, b)  NUMBER a, b; {return a * b;}
NUMBER fl_or(a, b)     NUMBER a, b; {return a || b;}
NUMBER fl_and(a, b)   NUMBER a, b; {return a && b;}
NUMBER fb_or(a, b)    NUMBER a, b; {return a | b;}
NUMBER fb_and(a, b)   NUMBER a, b; {return a & b;}

static NUMBER (*op_ar[OPS])() = {fi_plus, fl_or, fb_or, fi_times, fl_and, fb_and};
static char *label[OPS] = {"plus ", "logical or ", "bitwise or ",
                          "times ", "logical and", "bitwise and"};

main()
{ /* Header file must define NUMBER to be a standard numeric type. */
  NUMBER ar[LEN];           /* Declare an array of numbers. */
  NUMBER *ar_last = &ar[LEN-1]; /* Mark the last slot of the array. */
  NUMBER *p;                /* A scanning pointer for the array. */
  int k;                    /* An index for the function array. */

  puts("This program demonstrates some whole-array operations.\n\n");
  do {
    printf ("Please enter %d integers separated by spaces.\n", LEN);
    p=ar;                    /* Start input at head of array. */

    while( p<=ar_last && scanf(D, p)==1 ) ++p;
    if (p <= ar_last)
      printf("Premature EOF or conversion error; job terminated.\n"),
      exit(1);
    while (getchar()!='\n'); /* Flush rest of input line. */
    putchar ('\n');        /* Prepare for output. */

    /* Apply each operation in the op array to the number array. */
    for(k=0; k<OPS; k++)
      printf ("%s "D"\n", label[k], reduce(op_ar[k], ar));

    printf("\nDo you want to enter more data? (y/n)\n");
  }
  while (getchar() == 'y');
}

```

Exhibit 16.4. Subroutine module: “reduce.c”.

```

#include "modules.h"                /* a */
NUMBER reduce(NUMBER (*op)(), NUMBER ar) /* b */
{
    int k;
    NUMBER sum;                    /* c */
    sum = ar[0];
    for(k=1; k<LEN; k++) sum = (*op)(sum, ar[k]); /* d */
    return sum;
}

```

```

#define LEN 10
#define D "%f"
typedef float NUMBER;

```

Note the `extern` declaration for the function `reduce`. When this line is included in the main module, it tells the compiler two things:

- The `reduce` function is defined in another module.
- It takes two arguments, a pointer to a function and a number, and returns a number.

This information permits the compiler to compile correct and meaningful calls to a function it has never seen.

/subsubsection Keeping Private Information

In a modularized C program, a programmer can create both shared functions and private functions by judicious use of header files and “`static`” declarations. The modifier “`static`” is the opposite of “`extern`”. The keyword `extern` is used to denote an object or function that is to be shared by other modules, while `static` denotes a private item. A global symbol (function or data object) that is not declared to be either one is `extern`.

The main module, shown in Exhibit 16.3 is intended only as a demonstration of how the parts of a modularized program work together.¹ It sets up an array of integers, then sends the array to `reduce` to be processed, in turn, by each of six dyadic integer functions. Finally, it prints all the answers and queries the user about more data.

Sharing Information

The subroutine module shown in Exhibit 16.4 contains the function `reduce`.² This performs a “running operation”, using whatever dyadic function is passed to it as an argument. For example,

¹This main program is for demonstration purposes—it does not do anything that is particularly useful.

²Compare the syntax for functional parameters in C to the Pascal syntax shown in Exhibit 9.29.

Exhibit 16.5. The makefile for the sumup program.

```

sumup:  main.o reduce.o
        cc -o sumup main.o reduce.o

main.o:  main.c modules.h
        cc -o main.o main.c

reduce.o:  reduce.c modules.h
        cc -o reduce.o reduce.c

```

if the argument operation is “+”, `reduce` computes the running sum: $a[0] + a[1] + a[2] + \dots + a[n]$. If the argument function is “*”, the answer is the “running product”.

Note that this module includes the header file (line a). Thus the local variable, `sum`, (line c), will be type long or short or integer or float, whichever is selected in the header file. The important thing is that it will be the same type as the array elements created in `main`, and an appropriate type for the function argument. Each time the header file is modified, both the subroutine module and the main module, which depend on it, must be recompiled.

The type declaration “`NUMBER (*op)()`”, in the header of `reduce`, declares “`op`” to be a pointer to a function that returns a `NUMBER`. We can refer to this function through the pointer by writing its name with a dereference operator: “`(*op)`”. Thus line d calls the function which was passed to `reduce` as an argument. The arguments to *that* function are the current partial sum and the current array element.

Defining the Application

The `#include` commands in the two code modules express the relationship among the three files, but not in a coherent way. By reading all the files, a user could deduce the relationship, but that relationship is not presented in a single place or in a way that is convenient to process. Although the problem is small for a program with only two code modules and a header, it can become considerable with a large many-module application. The UNIX “`makefile`” makes the relationships and dependencies explicit. A makefile is a file of executable system commands; the example shown here contains UNIX commands to compile and link a program.

The makefile is the root of a tree of files that comprise the application, and it is the basis of an automated version-control system which serves two purposes:

- The most recent version of every file involved will be used to construct the executable module. No object module will be sent to the linker if a corresponding source or header file has been modified after the last compilation for that module.
- If any step in the compiling and linking process is unnecessary, it will be skipped. No compiling or linking operation will ever be redone unless the files on which it depends have been modified.

A makefile explicitly defines how the final application depends on separately compiled object modules, and how each of these depends on source code and header files. In our example [Exhibit 16.5], the first line tells the linker that the finished program will be called “`sumup`”, and that it can be produced by linking two user-created object files, `main.o` and `reduce.o`. (The object files that belong to the C library do not need to be listed.) The third and fifth lines define the source files on which the two user-defined object modules depend.

The second line of the makefile contains a call to link the object files that comprise this application, and to produce an executable file named “`sumup`”. The fourth and sixth lines contain calls on the compiler to produce the necessary object files.³ These lines are invoked only when needed. The make facility checks the date-last-modified on source files, object files, and executable files. If a linked module already exists, and none of its component object files have been modified or need modification, the linker can return immediately without wasting the effort to redo a job that is already done. An object file needs to be recompiled if any of the source files on which it depends have been changed since the creation time of the object file. These dependencies are defined by the last two lines in the makefile, where we see that the object file `main.o` depends on `main.c` and `modules.h`, and, similarly, `reduce.o` depends on a code module and a header module. If one of these three files were modified, the relevant object module or modules would be recompiled.

To use a makefile, that is, to compile and link the program, the programmer simply says “`make`”, or “`make sumup`”. (The longer form is needed if the user’s file directory contains makefiles for more than one application.) Any source module that has been edited since the last call on `make` will be recompiled, then the object modules will be linked, and the result stored as an executable file named `sumup`. To run the compiled program, the programmer will say “`sumup`”.

Makefiles, separate compilation, and declarations for external and static symbols thus combine to provide a language/system that achieves the goals of grouping, sharing, and protection.

16.3 Packages in Ada

Grouping Related Elements

Packages in Ada provide a framework under which to group together the variables and functions of an ADT, uninterrupted by unrelated variables and code. A package has much better lexical coherence than a comparable set of definitions in Pascal. An Ada package has a header, which declares all the externally visible symbols, and a body, which can include type definitions, static local variables, definitions for the publicly accessible functions, and private local functions. We will show how an Ada package definition can be used to implement the ADT `array_stack` [Exhibit 16.6].⁴ This package contains definitions for a stack representation and the representation-dependent stack functions, `PUSH`, `POP` and `TOP`.

³These two lines are not strictly necessary because they only state explicitly what the system would do by default if no compile commands were given.

⁴In Chapter 17 we show how a package can be embedded within a generic framework which allows for varying the base type of this ADT.

Exhibit 16.6. Defining a stack package in Ada.

```

-- The package header declares the externally visible symbols of the ADT.
package STACK is
    function PUSH(k: integer) return boolean;
    function POP return integer;
    function TOP return integer;
end STACK;

-- The package body defines private symbols and the ADT functions.
-- The private variables cannot be referenced outside this package.
package body STACK is
    STKLEN: constant:= 10;
    stk: array(1..STKLEN)of integer;           -- static local storage.
    tos: integer range -1..STKLEN;           -- Negative 1 is an error code.

    -- These functions can be called from outside the package because they
    -- have been declared as external symbols in the package header.
    function PUSH(k: integer) is
    begin
        if tos<STKLEN then
            tos:=tos+1;  stk(tos):= k;
            return true;
        else return false;
        end if
    end PUSH;

    function TOP return integer is
    begin
        if tos >0 then return stk(tos);
        else return SYSTEM.MIN_INT;  -- Use value from the package SYSTEM.
        end if;
    end TOP;

    function POP return integer is  -- Remove & return top stack item.
    ans: integer;
    begin
        ans:= TOP;  tos:=tos-1;
        return ans;
    end POP;

-- This code is run at load time, and initializes the package data.
begin
    tos:=0;
end STACK;

```

Sharing Public Facilities

Any symbol declared in the header of an Ada package can be shared by other modules. In our example, this includes the functions `PUSH`, `POP`, and `TOP`. To use an Ada package, the programmer writes a `with` command within the main program [Exhibit 16.7]. Having done this, one can refer to a part of that module by its complete name: the module name followed by the function or data name (see the left side of the exhibit). Using complete names avoids a possible problem caused by using the same identifier for parts of different modules.

Because packages are used extensively in Ada programs, this syntax for naming the parts of a package becomes annoying. Ada provides an alternate syntax that can usually be used and is more convenient. Using the `use` command causes all the public names in a package to be included with the local symbols in the calling program, eliminating the need to use the module name in every reference (see the right side of exhibit).⁵

By selectively including modules where they are needed, we can construct a nonhierarchical sharing structure. Each part of a program can be given access to exactly those other parts it needs. The possibility of many kinds of unintentional interactions between functions and data can be minimized, and the resulting software becomes easier to debug and maintain.

Private Information

Any symbol declared in the package body but not listed in the header is private and can only be accessed from within the package. In our Ada example, the stack-array and the stack-pointer are declared as pieces of private data, and the length of the stack is a private constant [Exhibit 16.6, lines 10 to 12]. The stack-pointer (`tos`) and the array (`stk`) are not grouped into a record-object, as they would be in Pascal, because there is no advantage to that grouping. The two variables are global within the package and invisible outside it, so they will never be passed anywhere as arguments.

Private variables declared in a package are static; that is, the value computed during one activation of a stack function remains until the next stack function is called. Package variables are not deallocated when control leaves the package, and they are not reinitialized when control returns. This permits the programmer to declare protected, local objects that can be used globally. This system is a tremendous improvement over the way similar problems must be handled in Pascal. In a Pascal program, data structures that are to be used throughout the program must be declared globally. They are usually initialized by the main program at the beginning of program execution. Pascal provides no way to protect the data structures from the rest of the program, or to ensure that they are accessed only through the proper functions.

Many variables and most data structures must be initialized when they are allocated. If a data structure is local and private within a package, the package itself must initialize it. The code on the last three lines of this Ada package serves this purpose [Exhibit 16.6]. At load time, before

⁵Much like the “`with`” command in Pascal.

Exhibit 16.7. Using an Ada package.**Basic Syntax:**

```
with STACK;

procedure MyJob is
    ...
    STACK.push(MyValue);
```

Sugared Syntax:

```
with STACK;
use STACK;
procedure MyJob is
    ...
    push(MyValue);
```

beginning to execute the main program, storage is allocated for all private variables in all included packages. The “main” initializing code in each of these packages is then executed.

Comparison to C

The goals of grouping, sharing, and privacy are achieved much the same way in an Ada package as they are in a C module. The primary differences between the two systems is that Ada packages have been brought entirely within the Ada language, whereas the C system relies on using a command file that is outside the C language.

16.4 Object Classes.

In object-oriented languages, the class serves much the same roles as the package does in Ada: it organizes things that belong together and allows for private information. In addition, the classes can be constructed so that there are hierarchies of classes, all related to each other.⁶ The best-known object-oriented languages are Simula, (the first object-oriented language), Smalltalk, and C++. In all of these, a class may contain data fields and functions, which we will call *members of the class*. Like Ada packages, classes may have private and public members.

Terminology varies somewhat among these languages. The term *method* was invented for Smalltalk,⁷ and it means one definition, in one class, for a function. This term is useful, and we will use it whenever we need to distinguish between an entire function and a single definition for that function. A function name represents a conceptual process. A class function might be (and usually is) given different (but related) method definitions in various parts of a class hierarchy.⁸

Terminology used for function definitions and calls is also variable. Simula has procedures, C++ has functions, and Smalltalk has messages. In Simula and C++, we speak of calling a procedure or function, while in Smalltalk we “send a message to an object”. In this discussion, we will use the terms “function” and “function call”.

⁶See Chapter 18.

⁷This term is not generally used in describing Simula and C++.

⁸Virtual functions are covered in Chapter 18.

Exhibit 16.8. Defining a class in C++.

```

class char_stack {                // Here are the private parts of the class.
    int size;
    char *tos, *end;
    char* s;
public:                            // All remaining symbols are public.
    char_stack(int sz)            // This is the class's constructor.
    {   s=new char[size=sz];      // Allocate an array of sz chars.
        tos=s;                    // Initialize top to point at first char.
        end=tos+(size-1);        // Mark last slot in stack.
    }

    ~char_stack() { delete s; }    // This object destructor is used when
                                    // control leaves the object's scope.

    int push(char c)              // Return 0 if stack is full.
    {   return ( tos<=end ? (*tos++ =c) : 0 ); }

    char pop()                    // Return null char if stack is empty.
    {   return ( tos>s ? *--tos : '\0' ); }

    char top();

};                                // End of class char_stack.

char char_stack::top()            // Return top but don't pop.
{   return ( tos>s ? *tos : '\0' ); }

```

16.4.1 Classes in C++

For the rest of this discussion, we will focus attention primarily on the semantics and syntax of C++, since it promises to become the most widely used object-oriented language.

A C++ class has the same elements as an Ada package, and it is declared with similar syntax.⁹ A sample class, named `char_stack`, is defined in Exhibit 16.8. It has four private data-members and four public function-members.

⁹In C++, a “`struct`” data type is also a kind of class. It may contain function declarations, and any `struct` declaration actually defines a class. However, it is a class with no private parts. The difference between a `struct` variable and a class variable is the accessing restrictions on the private data and functions. We will limit further discussion to classes declared as `class`.

Exhibit 16.9. Instantiating a C++ class.

```
main ()
{   char c;
    char_stack stk1(100);           // We can instantiate a class more than once,
    char_stack stk2(10);           // and get multiple copies of the objects.
    ...
    stk3 = new char_stack(20);
    stk1.push('#');                 // We must specify which "push" to use.
    stk2.push('%');
    c = stk2.pop();
}
```

Instances and Naming

A C++ class is a template, like a type declaration in Pascal, and must be instantiated to create objects. To instantiate a class, you use the class name in a declaration, as you would use a Pascal type name. The result of instantiation is an *object*, which is represented by a record-type storage-object with one field for each of the class variables. Those fields will be initialized if the class contains an initialization function.

When we instantiate a class, we bind the resulting object to a name. The object's name is used to refer to both data fields and the function members. For example, in Exhibit 16.9, the object `stk1` is an instance of the C++ class `char_stack`, and it has the function-members `stk1.push` and `stk1.pop` and the data members `stk1.size`, `stk1.top`, `stk1.s`, and `stk1.end`.

Within the context of the class, the members may be referenced by using a simple name. A public member can be referenced outside the class using a dotted notation:

$$\langle \text{class_name} \rangle . \langle \text{function_name} \rangle (\langle \text{argument_list} \rangle)$$

For example, the last three lines in Exhibit 16.9 call functions defined in the class `char_stack`.

The Implicit Argument. When we call a function that is a class member, we must specify which instance of the function we are calling. To do this, we write an object name followed by the member function name. (In Smalltalk, these are separated by spaces, while Simula and C++ use dotted notation, as in Exhibit 16.9.) The data members of an object do not need to be passed as arguments to the function members: each function may operate on the data fields of its own instance. Thus the record of data-members is an implicit argument to all of its function members.

As a result of having an implicit argument, each class function needs one fewer explicit argument. However, the name of the object that includes both the function instance and the implicit argument must be specified as part of the function name. To a programmer accustomed to traditional languages, such as Pascal, this looks like writing the name of the first argument on the left side of

a function name instead of on the right side with the rest of the argument list. For example, in Pascal, the name `stk1_stack` would be written as part of the argument list to `push` or `pop`, but in an object-oriented language it is written as part of the name of the function [Exhibit 16.9].

Having an implicit argument leads to one difficulty. Unlike an explicit argument, the syntax provides no way to give a local name to this record. Yet some functions, particularly recursive ones, must refer explicitly to the implicit argument. This problem is solved by a keyword that can be used to refer to any implicit argument. In *Smalltalk*, the keyword is `self`. In C++, the keyword “`this`” refers to a pointer to the implied argument.

Implementation

Classes contain both functions and data. The functions are declared within the class definition and may be either public or private. Theoretically, a “copy” of each class function is made each time the class is instantiated. The newly instantiated functions are part of the new class-object and are the only functions that have access to its private data fields. In practice, though, there is no need to duplicate the code of the class functions for every class instance. Thus an instance of a class can be implemented by a record variable with one field for each class data-member. This differs from an ordinary record-type variable because the access to the private fields is restricted to member functions.¹⁰ Initialization is done by a member function called a constructor, which must have the same name as the class.

Constructors and Initialization. Each class may contain a set of constructor functions.¹¹ A class variable may be created in a declaration or by using the `new` operator, which creates a storage object dynamically. In both cases, a constructor function is called to initialize the new object. Like *Ada* (see Exhibit 14.28), C++ permits the programmer to call a constructor function in an expression to create a pure value of the class out of its components.¹² This is often done inside a function to create a return value of the new type, and it is also used in declarations to create appropriate initializers.

Constructors are ordinary functions, with or without parameters. The only restriction is that all the constructors in a class must have different types of argument lists.¹³ If a constructor has parameters, an argument list must be supplied in a declaration, a call on `new`, or in an explicit call on the constructor. Exhibit 16.9 creates three instances of the class `char_stack` with different arguments and binds them to the names `stk1`, `stk2`, and `stk3`. The class `char_stack` must have a definition for its constructor which takes an integer argument. This function is freely definable and can perform any appropriate initialization.

¹⁰A *Simula* class has an associated “main procedure”, like that in an *Ada* package, which is called when a class object is constructed to initialize it.

¹¹In current terminology, the name of the constructor function may be “overloaded”.

¹²Review Chapter 14, Section 14.3.1.

¹³The power of C++ constructors goes well beyond initialization, and will be further developed in Chapter 17.

A C++ class may also have a destructor function, which does the opposite of the constructor function. If a destructor is defined for the class, it is invoked automatically at block exit to dispose of any class instances that were created at block entry or during execution of the block. This permits automatic recovery of storage locations that were created using `new`, a major improvement over C and Pascal.

Function Declarations and Definitions. A function is a class member if it is *declared* within the class. Remember that a function declaration is simply a header line—stating the return type, the function name, and the argument types. Compare the C++ code in Exhibit 16.8 to the Ada code in Exhibit 16.6. Ada requires the programmer to write each function declaration twice: once to declare whether it is public or private, and once with the definition. C++ syntax is simpler. It permits the function definition either to follow the declaration immediately, or to be given elsewhere. Practical considerations determine whether the definition should be with or separated from the declaration. Definitions written *with* the declaration are expanded as an in-line code; that is, the translator replaces each function call, in line, by a new copy of the compiled code for the function. All the time and space overhead of stack frames, jump-to-subroutine, and subroutine return is avoided.

If a member function is declared but not defined in a class, its definition must eventually be given; often, this is done just after the end of the class. Defining a method outside the class does not affect the status of the method as a class member. The method still has member's access rights. The difference is practical, not semantic; functions defined this way are compiled separately using the ordinary function call and return mechanism.

Giving a definition outside its class has one affect on syntax. Because some functions have multiple defining methods, we must specify the full name of the method when we define it, so that the compiler can tell with which class the method belongs. To denote the full name of a method we use the *scope resolution operator*, “`::`”. It can be used in dyadic or monadic form:

```

<class_name>::<function_name>( <argument_list>)
      ::<function_name>( <argument_list>)

```

Thus the full name of the method for `pop` in the class `char_stack` is `char_stack::pop`, and the constructor function for the class is `char_stack::char_stack`.

Denoting a Single Method The meaning of the scope resolution operator, “`char_stack::push`”, is easily confused with the dotted notation used in a function call, “`stk1.push`”, but they do not mean the same thing. The first notation denotes a particular method which is a member of a particular class. Every reference to a method outside its defining class must be written using “`::`”. Methods in programmer-defined classes are denoted using the dyadic “`::`”, and globally defined methods are denoted by using “`::`” in its monadic form.

In contrast, the dotted notation is used refer to a function, not one of its methods. It is important to be able to denote a single function method, as opposed to the entire function, which

Exhibit 16.10. A represented domain in C++.

```

class imag
{
    im: float;
    operator float() { return im; }
public:
    imag(){ im=0.0; }
    imag(float f){ im=f; }
    friend imag operator+(imag a, imag b){imag( float(a)+float(b) ); }
    friend float operator*(imag a, imag b){ -1*(float(a) * float(b)); }
} // Operators + and * are friends of this class.

```

is a collection of methods. A function call must be dispatched,¹⁴ while a method may simply be called directly. The dotted reference “`stk1.push`” means “Start with the object `stk1`. Run the function dispatcher to select the most appropriate method for the function “`push`”, then call that method with `stk1` as its implied argument.” If no method for `push` is defined in the class of its implied argument, some other method, higher in the class hierarchy, will be dispatched.

The most common reason for using a method name instead of a function name is that one method for a function is being defined in terms of another. The philosophy of object-oriented programming dictates that we should be able to use a single function name for both old and new methods, so long as they implement a common external function. For example, a customized printing function might be defined for a user-defined type by making several calls on one of the predefined print functions. Within the definition of the new print method, we must be able to denote the old method by its method name, in order to avoid a hopeless circular definition.

16.4.2 Represented Domains

Chapter 15, Section 15.4.3, discusses the problems inherent in using one domain to represent another. Briefly, the language must allow some connection between the domains in order to permit the programmer to define functions intrinsic to the new domain. However, once these are defined, all further compatibility between the domains is undesirable. In Pascal, this unwanted compatibility is unavoidable. In Ada, one-way protection is provided, as described in Exhibits 15.22 and 15.23. In C++ we have more control over the situation.

A language with classes provides a different and much more satisfactory approach to this problem. Where in Ada, we would define a new represented domain, in C++ we would define a class with one private data member from the old domain. The class `imag` in Exhibit 16.10 is an analog of the Ada type `imag` defined in Exhibit 15.26. The information provided by the programmer and

¹⁴The distinction cannot be fully understood until the material in Chapter 17 is mastered. However, we give a brief explanation here.

even the syntax are similar in the two languages. The important difference is the semantics.

No casts or compatibilities are predefined for C++ classes, so the programmer is not stuck with unwanted ones. In return, the two casts that are needed to define arithmetic for imaginary numbers must be defined explicitly. The first is the constructor `imag(float)`, which changes a float value to type `imag`.¹⁵ Because these two functions are private, the relationship between types `float` and `imag` is completely hidden from the outside world, and the semantics of type `imag` are fully protected. In contrast, the semantics of the Ada type in Exhibit 15.26 make the cast `imag(float)` public, and make the cast `float()` automatic. (That is, an `imag` is acceptable in any context that requires a float.)

The other cast needed in our package is one that changes an `imag` number to its underlying `float` representation so that we may operate on it. This is the *operator function* named “`float`”, defined in Exhibit 16.10.¹⁶ In C and C++ syntax, a cast is (syntactically) a prefix operator, not a function, and it is called by placing the name of the cast to the left of the expression which is its operand. Since the name of a cast is a type name enclosed in parentheses, this looks like a function call with the parentheses around the wrong thing. The designer of C++ gave the option of calling any operator, including a cast, using either operator syntax or standard function call syntax. Thus we can call `operator float()` either of these two ways:

```
float(17); float (x+2)
(float) 17; (float)(x+2)
```

16.4.3 Friends of Classes

The class mechanism lets the programmer impose restrictions on type compatibility, object visibility, and access to parts of objects. These restrictions are immensely powerful and can be a huge help in achieving semantically valid programs. However, when one uses a restricting mechanism, it is often difficult to achieve exactly the right degree of restrictions.

This is true of classes, also. There can be relationships among classes that are important but hard to capture using only public and private parts. This gives us tree-structured sharing among classes. However, sometimes graph-structured sharing is needed. We could say that, sometimes, we want “semiprivate” parts: parts that can be shared with some other program modules but not with all.

We use classes to make programs more modular and more reliable. The important semantic mechanism for achieving these goals is accessing restrictions. The data type of a class is *hidden* within the class, and other functions are forced to work through the accessing functions defined for the class. By using this methodology, we can easily change the representation of a class without affecting the correctness of other parts of the program. However, each function call has a cost in time and space efficiency. Although the cost of a single function call is small, these costs add up. The cost of doing everything through function calls can be great. When the access operation

¹⁵Conversion functions are treated more fully in Chapter 18, Section 18.1.

¹⁶The word “operator” in lines 3, 7, and 8 is a keyword.

itself is trivial, like the `pop` function in Exhibit 16.8, the cost of calling the function can exceed the cost of executing it. Thus applying modular methodology throughout a program could introduce unacceptable inefficiency.

Two solutions are provided in C++ for this problem: friends and in-line code. A method may be declared to produce *in-line code*, like a macro, rather than a separately compiled code module. If the method is actually defined between the begin-class and end-class brackets, it will be expanded in-line. If it is declared inside the class but defined elsewhere, it will be compiled as a separate code module. The semantics and syntax are otherwise the same, as is the ability to access the private members of the class. With short functions, such as `push` and `pop`, in-line code is preferred, because it is more time- and space-efficient than separate compilation. For long functions, in-line code would still be faster, but it could make the compiled program much longer if there are two or more calls on the function. Thus in-line compilation is a bad idea for moderate-length and long functions, and separate compilation is more commonly used.

The *friend* mechanism can also be used to avoid the inefficiency involved in constant calls to trivial functions. A class will share its private members with friend functions, but not with other functions. Friendship is given, not taken. A class declares who its friends are; a function cannot declare which classes are its friends. The friendship relationship can be declared function-by-function, or an entire class can be declared to be a friend, which means that all its functions are friends.

For example, assume that we have defined two classes, “`forest`” and “`tree`”. A `forest` is to be a collection of trees, in this case an array. In general though, whether it is a set, an array, or a list, it would be hidden within the class `forest` [Exhibit 16.11]. Both constructors for class `tree` are called by function `forest::grow_tree`; this would be permitted even if `forest` were not a friend, because these constructors are public in class `tree`. However, `grow_tree` also sets the value field of the new `tree` node it creates. To do this, it uses the member name `value`, which is private to class `tree`. This is permitted because class `forest` is a friend of class `tree`.

Comparisons

The class mechanism in C++ differs from the Ada package mechanism in many ways. Among the most important are the use of classes to form type hierarchies and govern function inheritance and automatic type conversion, as explained in Chapter 17. These facilities grow directly from the fact that the C++ class is an extension of the facility for defining record types.

The Ada package has one property, though, that is lacking in the class. A package can contain objects, functions, and more than one type declaration. Thus if we were writing a package, both types `tree` and `forest` would be written in the same package. Having done this, the right of forests to access tree parts would be established. A back door mechanism, such as the *friend* mechanism, is not needed to establish a relationship of “trust” between the two types.

Exhibit 16.11. Friend classes.

```
class tree
{
    tree * l_son;
    tree * r_son;
    char value;
public:
    tree(){ l_son= r_son= NULL; value= '\0'; }
    tree(tree *a, tree *b){ l_son =a; r_son =b; value= '\0'; }
    friend class forest;
}

class forest
{
    int n;
    tree woods[100];
public:
    forest(){ n=0; }
    forest(tree * t)
    {
        if( n==100 ) cout <<"Forest is full.";
        else woods[n++] = t;
    }
    void grow_tree(int n, char c)
    {
        tree * t = new tree;
        t->value = c;
        forest[n] = tree(t, forest[n]);
    }
}
}
```

Exercises

1. Why should programs be modularized?
2. What are the dangers in using global variables?
3. What is the purpose of encapsulating a module?
4. How can a module share data and yet protect private information?
5. How do private type declarations make an ADT representation invisible to the programmer?
6. How does C use files and separate compilation to achieve the goals of modularity?

7. What is the role of the makefile facility? A header file? The main module?
8. What is the role of `extern` and `static` in C declarations?
9. How did `Ada` and C++ extend the goals of grouping, sharing, and protection to their languages?
10. How does `Ada` distinguish between public and private symbols?
11. What is a method in `Smalltalk`?
12. A C++ class is a template. How is it instantiated? Initialized?
13. Explain two ways in which constructor function might be treated differently than other functions in a C++ class.
14. What is an implicit argument? How is it used in C++?
15. What is the role of a destructor function?
16. What are the differences in the declaration of a function in C++ and `Ada` syntax?
17. What is the role of the scope resolution operator?
18. How does C++ give the programmer more control than `Ada` over semantics and compatibility when using one domain to represent another?
19. What is the role of a friend class?
20. What is in-line code? When is its usage preferred? Not preferred?
21. Contrast classes in C++ and packages in `Ada`.