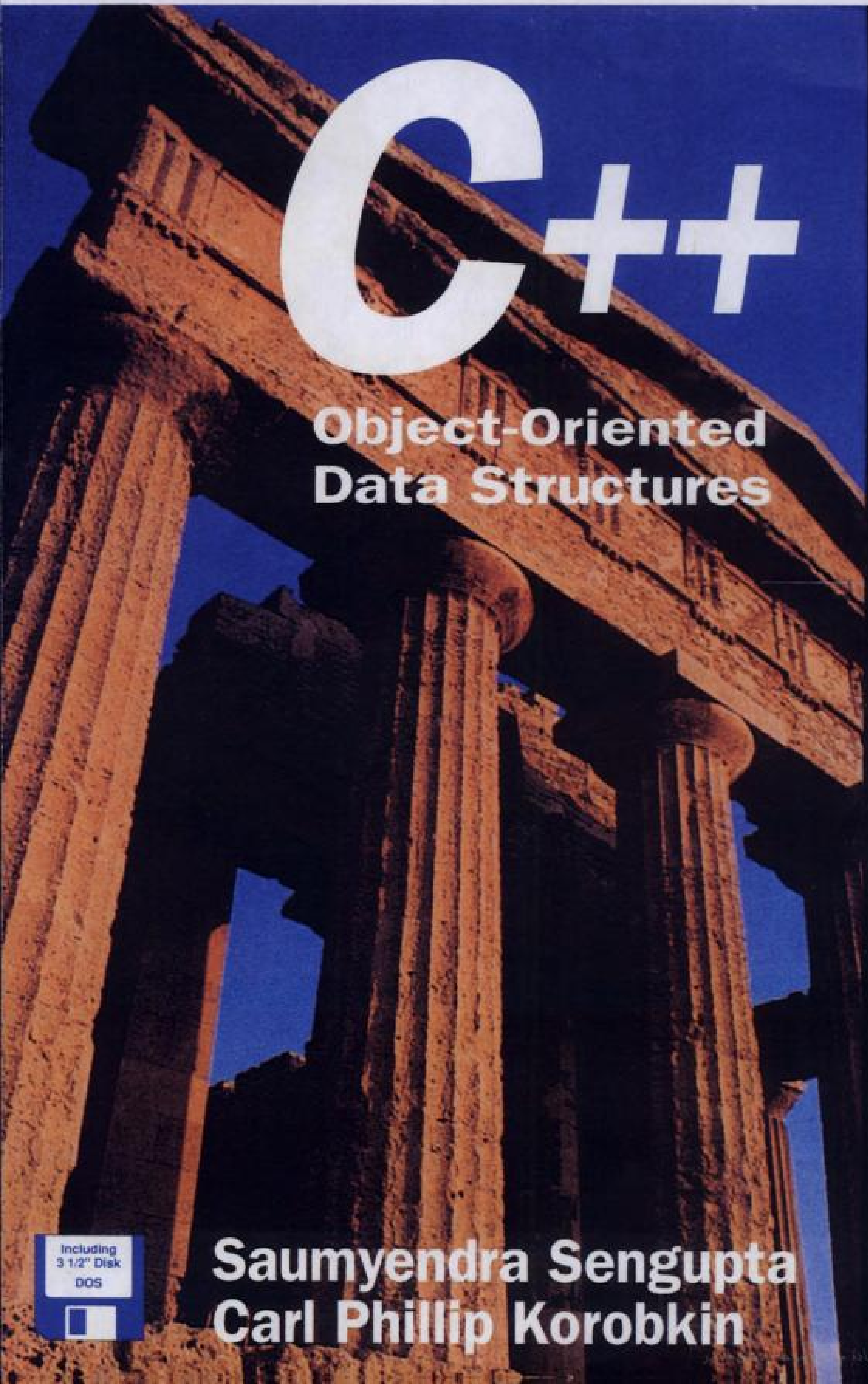
C++

Object-Oriented Data Structures



Including 3 1/2" Disk DOS

Saumyendra Sengupta
Carl Phillip Korobkin

Saumyendra Sengupta
Carl Phillip Korobkin

C++

Object-Oriented Data Structures

With 165 Illustrations

Diskette Included

Springer

Saumyendra Sengupta
Carl Phillip Korobkin
Silicon Graphics Corporation
2011 N. Shoreline Blvd.
Mountain View, CA 94039-7311

Cover photograph of the Temple of the Concordia, Agrigento, Italy.
Courtesy of David Viaggi, U.S.A.

Library of Congress Cataloging-in-Publication Data
Saumyendra, Sengupta.
C++, object-oriented data structures / Saumyendra Senguta, Carl
Phillip Korobkin.
p. cm.
Includes bibliographical references and index.
ISBN 0-387-94194-0
1. C++ (Computer program language) 2. Object-oriented programming
(Computer science) 3. Data structures (Computer science)
I. Korobkin, Carl Phillip. II. Title.
QA76.73.C153S28 1994
005.7′3—dc20 93-40950

Printed on acid-free paper

Production managed by Laura Carlson; manufacturing supervised by Vincent Scelta.
Photocomposed copy prepared from the authors' LaTeX files.
Printed and bound by R.R. Donnelley & Sons, Harrisonburg, VA.
Printed in the United States of America.

9 8 7 6

ISBN 0-387-94194-0 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-94194-0 Springer-Verlag Berlin Heidelberg New York SPIN 10739384

1

Concepts of Function-Oriented and Object-Oriented Data Structures

The processing of information transcends almost all activities of daily life. Whether it is the functioning of the human body or the functioning of a major corporation, the need to represent, store, delete, transform, retrieve, and transfer information is ever present at varying levels of abstraction.

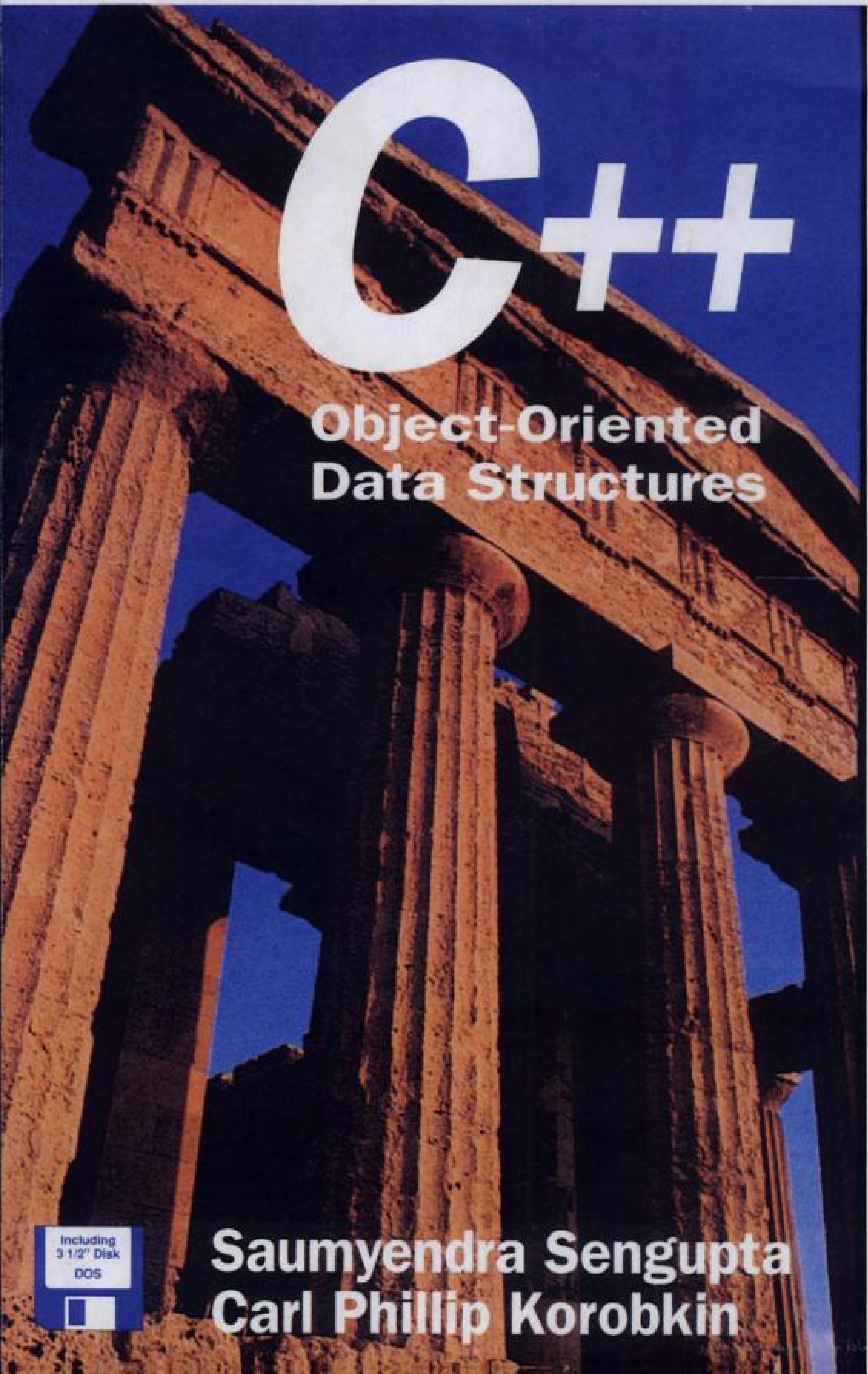
The digital computer aids in the processing of all this information. Fundamental to processing data and solving problems with a computer is the necessity of

- an underlying *data structure* for representing the data, and
- a method or *algorithm* for how to process it.

The discipline of computer science is the study of data, data structures, and algorithms, and the application of the digital computer for such activities. The study of data structures is really the study of algorithms and vice versa. To impose an organization on a set of data is to imply a method for processing the data; to impose a method on processing a set of data is to imply a structure for representing the data. These concepts go hand-in-hand.

1.1 Data Types, Data Objects, and Related Terminologies

The term *data type* refers to the basic kinds of data that variables may contain, as specified by the chosen programming language. The C++ programming language provides for the direct specification of both simple unstructured data types, such as `int`, `float`, and `char`, as well as structured data types, such as an array or `struct`. These are *built-in* data structures available as a primitive call in the language. An array type, formally introduced in Chapter 3, specifies a fixed number of *data items* or *elements* that are of the same type and are stored contiguously in memory. Given an index into the structure, any `Array` element can be accessed in constant time.



C++

Object-Oriented Data Structures

Including
3 1/2" Disk
DOS

Saumyendra Sengupta
Carl Phillip Korobkin

Saumyendra Sengupta
Carl Phillip Korobkin

C++

Object-Oriented Data Structures

With 165 Illustrations

Diskette Included

Springer

Saumyendra Sengupta
Carl Phillip Korobkin
Silicon Graphics Corporation
2011 N. Shoreline Blvd.
Mountain View, CA 94039-7311

Cover photograph of the Temple of the Concordia, Agrigento, Italy.
Courtesy of David Viaggi, U.S.A.

Library of Congress Cataloging-in-Publication Data
Saumyendra, Sengupta.
C++, object-oriented data structures / Saumyendra Senguta, Carl Phillip Korobkin.
p. cm.
Includes bibliographical references and index.
ISBN 0-387-94194-0
1. C++ (Computer program language) 2. Object-oriented programming (Computer science) 3. Data structures (Computer science)
I. Korobkin, Carl Phillip. II. Title.
QA76.73.C153S28 1994
005.7′3—dc20 93-40950

Printed on acid-free paper

Production managed by Laura Carlson; manufacturing supervised by Vincent Scelta.
Photocomposed copy prepared from the authors' LaTeX files.
Printed and bound by R.R. Donnelley & Sons, Harrisonburg, VA.
Printed in the United States of America.

9 8 7 6

ISBN 0-387-94194-0 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-94194-0 Springer-Verlag Berlin Heidelberg New York SPIN 10739384

1

Concepts of Function-Oriented and Object-Oriented Data Structures

The processing of information transcends almost all activities of daily life. Whether it is the functioning of the human body or the functioning of a major corporation, the need to represent, store, delete, transform, retrieve, and transfer information is ever present at varying levels of abstraction.

The digital computer aids in the processing of all this information. Fundamental to processing data and solving problems with a computer is the necessity of

- an underlying *data structure* for representing the data, and
- a method or *algorithm* for how to process it.

The discipline of computer science is the study of data, data structures, and algorithms, and the application of the digital computer for such activities. The study of data structures is really the study of algorithms and vice versa. To impose an organization on a set of data is to imply a method for processing the data; to impose a method on processing a set of data is to imply a structure for representing the data. These concepts go hand-in-hand.

1.1 Data Types, Data Objects, and Related Terminologies

The term *data type* refers to the basic kinds of data that variables may contain, as specified by the chosen programming language. The C++ programming language provides for the direct specification of both simple unstructured data types, such as `int`, `float`, and `char`, as well as structured data types, such as an array or `struct`. These are *built-in* data structures available as a primitive call in the language. An array type, formally introduced in Chapter 3, specifies a fixed number of *data items* or *elements* that are of the same type and are stored contiguously in memory. Given an index into the structure, any `Array` element can be accessed in constant time.

Definition 1.2 *An* ADT *is a collection of data and a set of allowed operations (actions) that are used to define and manipulate the data.*

The collection of data elements of an ADT is referred to as the *data abstraction*, while the set of allowed operations is known as the *program abstraction*. The ADT concept combines the data and program abstraction together. An ADT is specified by:

- a unique type name,
- a set of values of some sort, and
- a set of operations that act on that data type.

The data type may be as simple as built-in types such as `int`, `char`, and `float`, or more complex user-defined types. The ADT concept is fundamental to the data structures introduced and implemented throughout this text. The main features of an ADT are:

- The internal representation of a data type can be changed without changing the operations that are used by external programs or functions to abstract the data. Data abstraction methods remain unchanged.
- An ADT is viewed as a single entity.
- Unessential data in an ADT is concealed from any program that is not entitled to access that data.

As an example of a "built-in" ADT, consider the C++ integer type `int`. One may write programs that use the integer operations `+`, `-`, `*`, `%`, and `/` without having any rights to change the internal representation of `int` in the host computer system. That is, the implementor has an outside concept of `int` without knowing or needing to know the details of its internal implementation.

As an example of a user-defined abstract data type, consider an ADT *closed interval* $[a, b]$. An ADT closed interval is a data structure that contains an ordered pair of real numbers a and b, $a \leq b$, and a set of operations on these pairs of numbers. The number a is called the *left end point* and the number b is called the *right end point*. Typical interval operations include creating or allocating an interval, initializing an interval, deleting an interval, adding two intervals, subtracting two intervals, multiplying two intervals, dividing two intervals, finding the union of two intervals, finding the intersection of two intervals, and printing an interval. The ADT interval has three parts: (1) type name `INTERVAL`, (2) data a, b of type `float`, and (3) operations such as add and subtract. The reader is referred to the work of Moore [Moo66].

Shown below are two representations of ADT interval numbers and arithmetic processing of them. The first representation employs an `Array` construct and the second employs a `struct` construct. A more complete implementation of ADT interval numbers is provided in **Code 1.1**.

```
//  Method I. Define interval number using
//              '[]' for array construct
typedef  float     DATA_TYPE;
typedef  DATA_TYPE INTERVAL[2];
typedef  INTERVAL  *INTERVAL_PTR;

DATA_TYPE lft_end_pt (INTERVAL_PTR A)
{
    return (*A);
}
DATA_TYPE rgt_end_pt (INTERVAL_PTR A)
{
    return (*(++A));
}

//  Method II. Define interval number using
//              'struct' construct
typedef struct INTERVAL
{
    DATA_TYPE  lft_end_pt;
    DATA_TYPE  rgt_end_pt;
} *INTERVAL_PTR;

DATA_TYPE lft_end_pt (INTERVAL_PTR A)
{
    return (A->lft_end_pt);
}
DATA_TYPE rgt_end_pt (INTERVAL_PTR A)
{
    return (A->rgt_end_pt);
}
```

Though the internal representation of the interval number differs between the two implementations, the data abstraction procedures, such as `lft_end_pt()` and `rgt_end_pt()`, are unchanged. The view of the ADT interval numbers to the outside world remains the same.

1.3 Object-Oriented Design and the ADT

An *object-oriented design* (*OOD*) is one based on the ADT abstraction concepts. A straightforward way of understanding the principles of object-oriented design is to differentiate it from the more traditional *function-oriented design* (*FOD*).

Property 1.1 *An object-oriented design treats a software system as a group of interacting objects rather than a set of interacting functions, as in a function-oriented design.*

An *object-oriented program* (*OOP*) is an implementation of an object-oriented design. Alternately stated, object-oriented programming methodologies are employed to implement the ADT concepts. In OOP, data and operations for an ADT are combined into a single entity, called an *object*. Object-oriented programming languages such as C++, Smalltalk, and Object Pascal directly support the object paradigm and simplify the implementation of an OOD. In C++ , the `class` construct is used to implement an ADT and thereby defines a type of an object.

Object-oriented programming is more structured and modular than function-oriented programming, yielding programs that are easily maintained, resilient, and powerful. OOP allows the programmer to more closely mimic real-world situations. Stroustrup [Str89] justifies this.

In this section, we examine and contrast the basic concepts of traditional function-oriented and object-oriented programming. A short preview of an OOP in C++ is included. In succeeding chapters, a wide assortment of data structures are presented as objects and implemented as OOPs. For a detailed discussion of OOD and OOP concepts, the reader is referred to the works of Booch [Boo91] and Meyer [Mey88], and Lafore [Laf91], respectively.

1.3.1 Function-Oriented Data Structures

Function-oriented data structures are built on the concept of a collection of defined data variables and the operations allowed on them without regard to data hiding and protection. Function-oriented programming methods are based on a scheme of

- defining data requirements,
- constructing a data structure, and
- writing a set of functions to process the data.

Traditional programming languages such as C, ALGOL, and Pascal emphasize data representation and the use of fundamental data types. We see such data structure as the array (a unit of adjacent, dissimilar data types implying similar memory space usage) and the record (a unit of adjacent, dissimilar memory spaces) derived from the fundamental data types. With this approach, user-defined data types with data hiding is ignored.

Consider again the ADT interval example to illustrate the concepts of function-oriented data structures. If the data a (`lft_end pt`) and b (`rgt_end_pt`) of the ADT interval $[a, b]$ are implemented using a `struct`

INTERVAL of floats, the traditional programming approach defines the interval data structure and a set of functions in a header file:

```
//  interval.h
//
struct  INTERVAL  {
   float  lft_end_pt,  // Left end point "a"
          rgt_end_pt;  // Right end point "b"
};
void init_interv (INTERVAL *A, float new_lft,
                     float  new_rgt);
INTERVAL  add_interv (INTERVAL A, INTERVAL B);
INTERVAL  sub_interv (INTERVAL A, INTERVAL B);
void      print_interv (INTERVAL A, char * hdr);
```

The corresponding functions are contained in a C++ source file:

```
//  interval.c++
//
void  init_interv (INTERVAL *A, float new_lft,
                   float new_rgt)
{
    A->lft_end_pt = new_lft;
    A->rgt_end_pt = new_rgt;
}

INTERVAL  add_interv (INTERVAL A, INTERVAL B)
{
    //  etc.....;
}
```

This traditional data structure and programming method has a serious deficiency in protecting data from possible misuse. In the above illustration, the data members, lft_end_pt and rgt_end_pt in the data structure INTERVAL are publicly accessible (by default, they both have public access specifiers in C++). These members are not concealed from the functions and programs that may inadvertently corrupt them. This deficiency will be resolved in an object-oriented data structure and programming method.

1.3.2 Object-Oriented Data Structures

Since an ADT is viewed as a type for an object, and object-oriented design (OOD) decomposes most real-world applications as a collection of cooperating objects, it follows that an ADT (and thereby an object-oriented data structure) is an integral feature of OOD. When an OOD is implemented by a program in some OOD supporting language, such as C++ the program is called an object-oriented program (OOP). An OOP employs classes and objects. The basic features of an OOD include the following:

- Abstract Data Typing

  An ADT specifies the data and operations of an object, and is implemented by the C++ `class` construct.

- Data Abstraction

  Data abstraction provides a concept of an object and an interface to an object's data without involving the details of the implementation of the data. As stated by Booch [Boo91], "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

- Encapsulation and Data-hiding

  Encapsulation conceals (hides) the implementation details of the data and methods (see Methods below) of an object from its user program or function. This provides protection of the object's information (data and methods) from unauthorized users. Thus, abstraction complements encapsulation.

- Modularity

  Modularity is a means of decomposing an application into a collection of cooperating objects. As Booch [Boo91] says, "The principles of abstraction, encapsulation, and modularity are synergistic. An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction." By modularity, the logically related abstractions are compacted.

- Classes

  Classes define types or templates of objects. The type concept is derived from ADT. A class is derived from an ADT in addition to protection of data and operations. The classes are templates that describe the data structure and the valid actions that act on the data.

- Identification of Objects and Classes

  An object is an *instance* of a class; it has a unique variable name. It is identified by its behavior and state. A common class is used to define the behavior and structure of objects of the same type.

- Methods

  Valid actions are implemented with member functions of the class that defines an object. Such member functions are called *methods* (or operations or actions). Methods are used to send messages to an object that can only act upon the message.

A traditional function-oriented design presents a complementary approach to object-oriented design. In a function-oriented design, the basic module is a function and not an abstract data entity. In such an environment, the user is typically involved with what a function does *and* how it does it.

In any design, however, be it function-oriented or object-oriented, the implementor or user is usually concerned with the performance of the software system. To this end, a discussion of pertinent space and time tradeoff issues enters into the design considerations. Data structures and their associated algorithms are thus characterized by their efficiencies under varying application conditions. Thus, in an object-oriented design, one may be concerned with not only *what* an object does but *how well* it does it.

In this book, a unified approach to the subjects of traditional function-oriented data structures and object-oriented data structures is presented. Throughout this book, classical data structures – such as lists, stacks, queues, trees, and graphs – are presented. Each data structure is given as a C++ object in the form of a base class. For each base class, various implementations are discussed, with the notion that one implementation may be better suited than another for a given application. These data structures are then implemented as a set of subclasses, from which the application programmer may choose the one which best meets the performance characteristics required by the application.

To illustrate the concepts in abstract terms, consider an application that involves the painting of a house interior and the need to employ a painter. In selecting the painter most appropriate for the job, we may consider various qualifications, such as whether the painter specializes in interiors, exteriors, or both, how fast the painter works, and how much the work would cost. In our search we find three qualified painters. Two of them specialize in interiors—one is faster than the other but charges considerably more for the job. The third painter is a general painter (does interiors as well as exteriors), is priced between the others, but is really slow.

If we think of these painters as objects, we can illustrate their use in terms of a base class and a set of derived subclasses, as follows:

- Abstract Base Class

  The *abstract base class* contains a common set of interfaces to be shared by all the subclasses that *derive from it.* The subclasses that derive from the abstract base class *inherit only interface and not implementation.* The abstract base class defines a common framework for all implementations of the class in terms of a list of methods (actions) as virtual functions.

  For our housepainter example we define the abstract base class `HousePainter`. This base class defines the interface of a set of generic painter operations that are common to all painters of this class.

```
class HousePainter {

        // list of methods
        // (e.g., scrape, sand, mask, paint interior, clean)
};
```

- Derived Classes

  A *derived class* or *subclass* implements the common interfaces specified by the base class. It may also extend the functionality of the base class by defining and implementing additional members required for its specific needs. *The implementation of the derived class characterizes the performance of the object.* Derived classes are alternately referred to as *implementation specific classes.* It is implementor's responsibility to provide a set of uniform interfaces which match the base class such that the application programmer may plug in one implementation specific class or another without changing their code.

  In our example, there are three derived classes: `Fast_Interior_Painter` and `Faster_Interior_Painter` are the interior specialists, while `General_Painter` is the interior-exterior painter. All three know how to paint interiors as part of the interface defined by the abstract base class. The `General_Painter`, however, extends this functionality by providing an exterior painting capability. All three derive from the `HousePainter` base class.

```
class Fast_Interior_Painter : public HousePainter { //..};
class Faster_Interior_Painter : public HousePainter { //..};
class General_Painter : public HousePainter { //..};
```

- Instantiation

  A *derived object* is an instantiation of a derived class and is the top level of object abstraction. It is an object whose methods are given by the base class and whose features and performance are attributed to the plugged-in implementation specific class. As such, it is also referred to as an *implementation specific object.* Application programmers may simply attach their programs to an implementation specific object.

  In our example, we decide that the lower cost interior specialist (`Fast_Interior_Painter`) best suits our requirements. The following instantiation creates a `HousePainter_Object`:

```
Fast_Interior_Painter HousePainter_Object;
```

Figure 1.1 shows the inheritance hierarchy of the abstract base class and its derived subclasses for the housepainter example.

Step 1. Partitioning

Divide the problem into a set of objects. Identify each object by its data and related operations (ADT).

Step 2. Sharing

Determine commonality between objects and hierarchy of classes.

Step 3. Encapsulation Determine levels of protection for data and operations in each object.

Step 4. Inheritance Determine the protection levels for data and operations in the base class(es) that are inherited by subclasses.

Step 5. Polymorphism and Dynamic Binding

Determine which methods in the base class(es) will have the same interface names across subclass(es) but may need to act differently in different subclass(es).

Step 6. Refinement

Repeat the above five steps in an iterative manner until a design is reached for an initial implementation.

1.4 Implementing an OOP in C++

In the software engineering community, the C++ programming language has emerged as a de facto state-of-the-art language for writing OOP. C++ provides the following unique features suitable for object-oriented programming:

- Classes for defining templates
- Single and multiple inheritance; hierarchy of classes and derived classes
- Member functions of a class that are methods
- Protection of data and function members of a class using the `private` and `protected` keywords
- Special functions: Constructor and Destructor
- Inline functions of a class
- Friend functions and classes that support inheritance
- Virtual functions of a class that support polymorphism
- Overloaded functions that support polymorphism

- The internal representation of data structure, storage.
- The internal implementation of the interface.
- The external operations for accessing and manipulating the instances of the class.

A class uses an ADT. The elements belonging to the set of objects described by the class are called instances of the class. Keeping the OOP characteristics (object, class, data-hiding, methods, etc.) in mind, we present the following OOP **Code 1.1** for the interval object $[a, b]$, $a \leq b$, a and b being real numbers. For interval objects, we defined an `Interval` class, overloaded constructor `Interval()`, overloaded operator `+()` function, and one more method. Notice that hiding the data `lft_end_pt` and `rgt_end_pt` is accomplished by declaring these `private` within the `class` construct of an `Interval` object.

Code 1.1

```
//  Program:  interval.c++
//  Purpose:
//    Object-oriented implementation of ADT interval
//    numbers, [a, b], where a \le b.
//
#include <stdio.h>

// Define an "Interval" class
class Interval {
  private:
      //  Declare data as \keyword{private} for hiding
      float  lft_end_pt,  // Left end point "a"
             rgt_end_pt;  // Right end point "a"

  //  Declare methods as \keyword{public}
  public:
    Interval(float new_lft, float new_rgt);  // Constructor
    Interval(float new_lft_rgt);  // Constructor for
                                   // degenerate Interval
                                   // object
    friend Interval operator+(Interval A, Interval B);
    void   print_interval(char *hdr);
};

//  Interval(): First Constructor function.
//    Construct and initialize an "Interval" object
//    using two input values.
//
Interval::Interval (float new_lft, float new_rgt)
{
   lft_end_pt = new_lft;
```

```
Interval B is: [ 6.000000, 9.000000]
Interval A + B is: [ 4.000000, 13.000000]
```

1.5 Example Databases

To aid in the illustration of the topics presented in this book, we have created two example databases. These databases are used by a majority of the sample programs in the chapters that follow. By employing the same sets of data from one example to the next, the various data structuring techniques and implementations presented are more readily contrasted.

The first database, referred to as the PEOPLE database, is a collection of records involving information about a group of individuals and their membership in an organization. Each record of a person contains six attributes: name, age, income, occupation of member, the member identification number, and the years that the member has belonged in the organization. As a matter of implementation, three versions of the PEOPLE database are presented, each differing by the number of attributes of the record that are encoded as keys: thus `people_1d`, `people_2d`, and `people_3d` correspond to records containing one, two, and three keys, respectively.

The second database, referred to as the GEOMETRY database, is a collection of records involving colored points in space. There are two versions, `geometry_2d` and `geometry_3d`. For `geometry_2d`, each record is a point in a two-dimensional space; each contains two keys, coordinate `x` and coordinate `y`, plus the attributes of `color` and `label`. For `geometry_3d`, each record is a point in a three-dimensional space; each contains three keys, coordinate `x` coordinate `y`, and coordinate `z`, plus additional attributes of `color` and `label`. Figure 1.3 illustrates the format of these records. The complete databases are presented in Appendix C.

1.6 Big Oh Notation

To choose an efficient data structure for an application, we employ a tool, called $O(n)$ (Big Oh), to measure the performance of data structures operations. The notation $O(n)$ means "On the Order of n." For example, $O(n^3)$ means on the order of n-cube or proportional to n-cube.

Given two mathematical functions $u(n)$ and $v(n)$, n being an integer, $u(n) = O(v(n))$ if for some positive constants p and q

$$u(n) \leq pv(n) \quad \text{for all values of } n \geq q$$
$$\text{(i.e., for all sufficiently large } n\text{)}$$

This means that $u(n)$ is of order $v(n)$, or $u(n)$ will grow no faster than

Use the `class` construct to implement the object type that combines data and operations into a single entity. Write functions implementing some operations including equality of two matrices.

8. Use the object-oriented design approach for ADT complex numbers. A real number tuple (a, b) of the form $a + bi$, a and b being real numbers, is called a complex number. The number a is the real part, and b is the imaginary part of the complex number. Identify the object by its data and allowed operations. Use the `class` construct to implement the object type that combines data and operations into a single entity. Write and test functions implementing some operations in C++.

9. Use the object-oriented design approach for an ADT file. A file is a logical entity that represents a named portion of information (program source, text and/or binary data). For example, in the PC-DOS operating system on an IBM PC or compatible systems, files with unique names "Example1.txt" and "Example2.txt" are stored on a disk; "COPY Example1.txt Example2.txt" copies one file to the other. Identify the object by its data and allowed operations. Use the `class` construct to implement the object type that combines data and operations into a single entity. Show several examples of user-typed commands that implement some of these operations within a commonly used operating system such as MS-DOS or UNIX.

C++ Operator	Description
DATA_TYPE *	Pointer declaration for a pointer to a variable of type DATA_TYPE.
DATA_TYPE **	Pointer declaration for pointer to a pointer that points to a variable of type DATA_TYPE.
&	Receive address of the variable.
=	The pointer is assigned the address of variable.
*	Content of location pointed to by the pointer.
++	Increase the value of the pointer by adding the size of DATA_TYPE.
--	Decrease the value of the pointer by adding the size of DATA_TYPE.
+= n	Increase the value of the pointer by adding n times the size of DATA_TYPE.
-= n	Decrease the value of the pointer by adding n times the size of DATA_TYPE.

TABLE 2.1. C++ pointer operators.

```
{
   int    a = 5,
         *a_ptr,        // Pointer to variable a
        **a_ptr_ptr;    // Pointer to pointer a_ptr

   printf("\n ** Demo of pointer to a pointer ** \n");
   printf(" a = %d               Address of a = %u \n",
            a, &a );
   a_ptr = &a;          // Assigned address of a
   printf(" *a_ptr = %d    %s = %u %s = %u \n",
            *a_ptr, "     Content of a_ptr", a_ptr,
    "\n                   Address of a_ptr", &a_ptr);
   a_ptr_ptr = &a_ptr; // Assigned address of a_ptr
   printf(" **a_ptr_ptr = %d    %s = %u \n",
     **a_ptr_ptr, "Content of a_ptr_ptr", a_ptr_ptr);
}
```

Code 2.1 for pointer to a pointer produces this output on an IBM PC/386 system:

```
** Demo of Pointer to pointer **
a = 5                 Address of a = 65524
*a_ptr = 5           Content of a_ptr = 65524
                      Address of a_ptr = 65522
**a_ptr_ptr = 5     Content of a_ptr_ptr = 65522
```

Pointer arithmetic	Description
`ptr++`	`ptr = ptr + sizeof(DATA_TYPE);` Use the original value of `ptr` and then `ptr` is incremented after statement execution.
`++ptr`	`ptr = ptr + sizeof(DATA_TYPE);` Original `ptr` is incremented before the execution of statement.
`ptr--`	`ptr = ptr - sizeof(DATA_TYPE);` Use the original value of `ptr` and then `ptr` is decremented after statement execution.
`--ptr`	`ptr = ptr - sizeof(DATA_TYPE);` Original `ptr` is decremented before the execution of statement.
`*ptr++ == *(ptr++)`	Retrieve the content of the location pointed to by `ptr`, and then increment `ptr`.
`*++ptr == *(++ptr)`	Increment `ptr` and then retrieve the content of the new location pointed to by `ptr`.
`(*ptr)++`	Increment content of the location pointed to by `ptr`. For pointer-type content, use pointer arithmetic, else use standard arithmetic.
`++*ptr == ++(*ptr)`	Increment the content of the location pointed to by `ptr` depending upon the type of this content.
`*++ptr == *(++ptr)`	Increment `ptr`, then retrieve the content of the new location pointed to by `ptr`.
`--*ptr == --(*ptr)`	Decrement content of the location pointed to by `ptr` depending upon the type of this content.
`*ptr-- == *(ptr--)`	Retrieve the content of the location pointed to by `ptr`, and then decrement `ptr`.
`*--ptr == *(--ptr)`	Decrement `ptr`, then retrieve the content of the new location pointed to by `ptr`.
`(*ptr)--`	Retrieve content `*ptr` of the location pointed to by `ptr`, then decrement the content of that location; `ptr` is not changed.

TABLE 2.2. Pointer arithmetic and referencing.

Note that the use of `typedef` is not required for the `struct` construct in C++. However, `typedef` can be used to simultaneously define the type of a pointer to this structure. This declaration of `struct` does five things:

- Allocates memory space for the identifier `struct_var_1`.
- Assigns the address of the variable to the pointer `ptr`.
- Recursively defines the structure by including the pointer variable `struct_ptr` for the same structure.
- Allows nesting of structures within a structure (i.e., structure of structures)
- Allows nesting of union within a structure (i.e., structure of union)

For example, in a `people_1d` database management system, the information for a member includes the member's identification, name, occupation, age, yearly gross salary, and the number of years of membership. Though these components are of different types, they can be combined into a single entity using the `struct` construct. For example, the record for the people database `people_1d` is defined as follows:

```
typedef int     Keytype;
typedef int     Nodetype;
typedef int     dummytype;
typedef char    *Name;

struct DataObject {
   Keytype key;         // member id
   Name    name;        // name
   char    *occupation; // occupation
   int     age;         // age
   int     income;      // yearly gross / $1000
   int     years;       // number years member
};
```

2.2.2 Pointers to Structures

When a pointer is used to hold the address of a structure variable, the syntax is:

```
<struct_name>  *<struct_ptr_name>;
```

It is assigned to hold the address of a structure of the same type using the & operator. The syntax is:

```
<struct_ptr_name> = &<struct_var_name>;
```

After this assignment statement, the pointer `struct_ptr_name` can be used to reference any member of `struct_var_name`. Without this assignment, any such reference will cause a *memory fault* during the execution of a program. For example:

```
DataObject record_of_mem1, *mem_ptr;
mem_ptr = &record_of_mem1;
```

2.2.3 Accessing Structures

There are two C++ operators available for accessing the individual members of a structure:

- Operator `.` (period)
- Operator `->` (dash followed by a greater than)

The structure member syntaxes are:

```
<struct_var_name>.<mem_name>
<struct_ptr_name>-><mem_name>
*<struct_ptr_name>.<mem_name>
```

For example:

```
DataObject record_of_mem1, *mem_ptr;

// Initialize values to the fields
record_of_mem1.key = 16;            // ...(1)
record_of_mem1.name = "Bob";
record_of_mem1.occupation = "engineer";
record_of_mem1.age = 29;
record_of_mem1.income = 57;
record_of_mem1.years = 3;

// Assign address of "record_of_mem1" to "mem_ptr"
// Now, "mem_ptr" is a pointer to the structure.
mem_ptr = &record_of_mem1;

// Alternative approach using "->" operator
mem_ptr->key = 16;  // equivalent to (1)

//  OR
(*mem_ptr).key = 16;  // equivalent to (1)
```

2.2.4 Initializing Structures

Initializing a structure requires three key items:

- Only `static` and `extern` storage classes can be used.

```
   protected:
        <type> <data_members>;
        <implementation_operations>;   // methods
   public:
        <type> <data_members>;
        <implementation_operations>;   // methods
};
```

The keyword `typedef` is not required since a class name is a type of name. The keywords `private`, `protected`, and `public` are used to specify the three levels of access protection for hiding data and function members internal to the class:

- `private` means that this member can only be accessed by the member functions and friends of this class. The member functions and friends of this class can always read or write private data members. The private member is not accessible to the outside world (that is, outside of this class).
- `protected` (read/write for data members only) means that this member can only be accessed by the member functions and friends of this class and member functions and friends derived from this class. It is not accessible to the outside world (that is, outside of this class).
- `public` means that this member can be accessed by any function in the outside world. The public implementation-operations (also called functions or methods) are interfaces to the outside world so that any function, in OOP terminology, can send messages to an object of this class through these interface functions.

The data members are always read/write. A member function can be *inline*, which means the member function can be defined within the body of the `class` construct. The keyword `inline` is used for short functions, and for efficient storage like the `register` keyword. A class is an abstract data type. It specifies how objects of its type behave, are created and deleted, and are accessed.

An example of a `Stack`-type `class` is as follows:

```
class  Stack  {
  int  *stk;   // Stack::stk is private by default
private:
  int  size;   // Stack::size is private
protected:
  int  *top;   // Stack::top is protected
public:
  int  top_of_stack;   // Stack::top_of_stack is public
  // Special "public" function "constructor"
  Stack(int sz)  { top = stk = new int[size=sz]; }
  // Special "public" function "destructor"
```

```
#include  "ex2_3.cpp"
#else
#include  "ex2_3.c++"
#endif

void main(void)
{
     //  Declare and initialize two objects "stack_obj1" and
     //  "stack_obj2" of the same type: "Stack" class.
     Stack  stack_obj1(1024);
     Stack  stack_obj2(512);
     //  Define a pointer "stk_ptr" to an object of
     //  "Stack" class
     Stack  *stk_ptr;

     //  Connect "stk_ptr" to the object "stack_obj2"
     stk_ptr = &stack_obj2;

     //  Send a message ("push -20 in stack") to "stack_obj1"
     stack_obj1.push( -21 );

     //  Send a message ("push 11 in stack") to "stack_obj2"
     stk_ptr->push( 11 );
}
```

Note that the objects `stack_obj1` and `stack_obj2` of the same `Stack` class have their own copies of data and function members.

2.4.3 Accessing a Member of a Class

There are two ways we can access a member of a class similar to accessing members of a `struct` or `union` construct. A data or function member of the `class` construct is accessed using the `.` (period) operator in the following manner:

```
<class_object>.<data member>
<class_object>.<function member>
```

When a pointer to an object of a class is used, the `->` operator is used to access data or function members of the class. The syntax is:

```
<class_object_pointer>-><data member>
<class_object_pointer>-><function member>
```

Accessing of a member function `push()` of the objects `stack_obj1` and `stack_obj2` is stated as:

```
stack_obj1.push(-21);

stk_ptr->push(11);
```

Another example of the `class` construct and referencing the members using a pointer is illustrated below in **Code 2.4**.

Code 2.4

```
//  Program:  point.c++
//  Purpose:
//    To demonstrate the uses of class, and
//    pointer to class object, and special function
//    constructor.
//
#include <stdio.h>
#include <math.h>   //  For 'sqrt()' function

typedef  int  COORD;
// Define a "Point" class
class Point {
  private:
    COORD   x,  y;  // x- & y-coordinates
  public:
    Point( COORD x_new, COORD y_new);
    //  To provide "READ-ONLY" access
    inline COORD get_x() { return x; }
    inline COORD get_y() { return y; }
    float find_dist (Point A, Point B);
};

//  Point():
//    Special function: constructor and initializer
//
Point::Point(COORD x_new, COORD y_new)
{
   x = x_new;
   y = y_new;
}

//  find_dist():
//    Compute distance between two point objects
//
float  Point::find_dist(Point A, Point B)
{
   float  dist_sqrd = (B.y - A.y) * (B.y - A.y) +
                      (B.x - A.x) * (B.x - A.x);
   return (sqrt(dist_sqrd));
}

//  main():  Main test driver
//
void  main(void)
{
   //  Declare and initialize two "Point" type
   //  objects.
```

```
    Point  A_pt_obj( 4, 3), B_pt_obj( 0, -1);
    Point *A_ptr = &A_pt_obj,
          *B_ptr = &B_pt_obj;

    printf("\n ** Demo of Class, and Pointer %s \n",
           "to Class Objects **");
    printf(" Point object A = ( %d, %d ) \n",
            A_ptr->get_x(), A_ptr->get_y());
    printf(" Point object B = ( %d, %d ) \n",
            B_ptr->get_x(), B_ptr->get_y() );
    printf(" Distance between A and B is: %f \n",
            A_ptr->find_dist(A_pt_obj, B_pt_obj));
}
```

Code 2.4 for the `Point class` construct produces this output:

```
** Demo of Class, and Pointer to Class Objects **
Point object A = ( 4, 3 )
Point object B = ( 0, -1 )
Distance between A and B is: 5.656854
```

2.4.4 Friend of a Class and Inheritance

In OOP, sometimes functions of one class need to access all private and protected data and function members of another class. To allow inheritance of such an access privilege, the `friend` keyword is only used in the helper class. The syntax is:

```
class  <helper_class_name>  {
  private:
    <data_members>;
    <function_members>;
    friend class  <friend_class_name>;
    friend <type> <friend_function_name>
                ( <arg_type> <arg> );
  public:  // ...
};
```

All member functions of the `friend_class_name` class and the friend function `friend_function_name` can have read/write access of data members and access of function members of the `helper_class_name` class. This friendship is inherited. It is not transitive and inherited, so a friend of the `friend_class_name` cannot inherit the privilege to access private data and function members of the `helper_class_name` class; the friendship is a privilege, and it is not transferable. Only the

```
class <derived_class>:<access_specifier> <base_class>{
   private:
             // "private" members of derived class
   protected:
             // "protected" members of derived class
   public:
             // "public" members of derived class
};
```

If the access_specifier is private in the derived class definition, the protected and public members of the base class are made private members of the derived class. If the access_specifier is public in the derived class definition, the protected and public members of the base class are made public members of the derived class. Private members of the base class are still not accessible to the function members of the derived class unless the friend keyword is used in the base class for a specific function.

When a subclass is derived from two or more previously defined base classes base1_class and base2_class, it is a case of multiple inheritance. The syntax is:

```
class <derived_class> :
        <access_specifier> <base1_class>,
        <access_specifier> <base2_class> {
   private:
            // "private" members of derived class
   protected:
            // "protected" members of derived class
   public:
            // "public" members of derived class
};
```

Multiple inheritance is the ability of a derived class to have multiple parent classes. To illustrate the concepts of derived class and multiple inheritance, consider the following code segment in which we define two base classes, base1 and base2, and a derived class derived:

```
// Define "base1" class
class base1 {
  private:
    int a;
  protected:
    int b;
  public:
    int c;
    int f();
};
// Define "base2" class
```

```
class  base2  {
  private:
    int  d;
  protected:
    int  e;
  public:
    int  g;
    int  f();
};

//  Define "derived" class based on the
//  "base1" and "base2" classes
class  derived : public base1, private base2  {
  private:
    int  x;
  protected:
    int  base1::c;  // redefine access to "base1::c"
  public:
    int  y;
    int  z();
};
```

The members `b`, `c`, and `f()` of the `base1` class are inherited, but `b` and `f()` are `public` and `base1::c` is `protected` in the `derived` class. The members `e`, `g`, and `f()` are `private` in the derived class `derived`. The `private` members `a` and `d` of the `base1` and `base2` classes, respectively, are not inherited to the `derived` class. Notice that ambiguity in referencing the inherited function `f()` occurs since `f()` is found in both of the base classes. To resolve the ambiguity in such a case of multiple inheritance, use the base class name `base1::f()` or `base2::f()` as follows:

```
int  derived::z()
{
  return ( y + base1::f() );
}
```

2.4.6 Nested Class

Like the conventional `struct` construct, the `class` constructs may be nested, one `class` being declared within another. For example, for a singly linked list,

```
class  slist   {      // "slist" is outer class
  private:
   node_obj  *head_ptr;
   class node_obj  {  //  "node_obj" is inner class
     private:
       int         data;
       node_obj   *next;
     public:
       node_obj *set_next (node_obj *new_next);
    };
```

```
  public:
    slist();      // Constructor
    ~slist();     // Destructor
    //  define more methods ...
};
```

2.5 Functions in C++

In C++, there are two categories of functions with regard to the `class` construct,

- the member function, and
- the nonmember function.

The C++ functions that fall into these two categories are listed below in Table 2.3. The C++ keywords `inline`, `friend`, `virtual`, `overload`, and `operator` are used as attributes of the functions listed in Table 2.4. The applicable usage of these attributes for a class member and nonmember function is listed Table 2.4.

Both the class member function and the nonmember function can be defined with the `inline` attribute. Note that the `overload` keyword is not used in C++ 2.0 or above; the overload function can simply be defined without the `overload` keyword.

2.5.1 Special Functions: Constructors

Constructors are special functions in C++. They are member functions of a class, and widely used in OOP. The constructors have the following features:

- The name of a constructor must be the same as the class name.

Class Member Functions	Class Nonmember Functions
Constructors Destructions Implementation operations (methods) Operator function Constant function Static functions	Standard functions Friend functions Operator functions

TABLE 2.3. Class member and nonmember functions.

Attributes	Class Member Function	Class Nonmember Function
inline	Yes	Yes
friend	No	Yes
virtual	Yes	No
overload	Yes	Yes
operator	Yes	Yes

TABLE 2.4. Attributes for class member and nonmember functions.

- They cannot have any return type, not even `void`, but can have arguments.
- They are used to construct an object.
- They can initialize an object.
- They are automatically and implicitly called when an object of a class is declared.

In **Code 2.3**, the statement

```
Stack  stack_obj1(1024);
```

implicitly calls the constructor `Stack()` that dynamically allocates memory storage and initializes the `stack_obj1` object of the `Stack` type.

2.5.2 Special Functions: Destructors

Destructors are special functions in C++. They are member functions of a class, and are widely used in OOP. The destructors have the following features:

- The name of a destructor must be the same as the class name preceded by ~.
- They cannot have any return type and arguments.
- They are used to destroy or deallocate the memory space of an object by using the `delete` operator.
- They are automatically and implicitly called when an object of a class is deleted or the program is exited.
- They can be declared `virtual`.

In the last code segment of Section 2.4, `~Stack()` is the destructor for the `Stack` class, and calls

```
delete  []stk;
```

2.6 Polymorphism, Virtual Functions, and Inheritance

In OOP, polymorphism is an important concept, and an action that is implemented by a method. In OOP, polymorphism means that a method with the same name in a hierarchy of base classes and derived classes acts differently appropriate to the class or subclass. Such a method is called a virtual function. That is, a virtual function is used to implement different behaviors or actions appropriate to the class or subclass in which it is defined. In C++, a virtual function is declared by using the `virtual` keyword, only in the base class:

```
virtual <type> <function>( <argument_type> <argu-
                          ment> );
```

The virtual function must be defined in the base class, but its definition or purpose can be changed in any subsequent derived class. C++ internally uses special pointers to implement virtual functions.

2.6.1 Friend Functions and Inheritance

In OOP, sometimes a nonmember function needs to inherit the access privilege of any data or function members of another class (e.g., `class A`). In order to grant access permission, we declare this nonmember function as a `friend` of this class `A` by using the `friend` keyword. For syntax, see Section 2.4.4.

2.6.2 Overloading and Polymorphism

To achieve polymorphism (a state of assuming different forms of actions) for OOP in C++, we can use overloaded functions and operators.

2.6.3 Overloaded Functions

Functions that have the same name, but that implement many different actions, are called *overloaded functions*. Most of the time the choice of which function to use will be determined by the number and the type of the function's argument list (called *signature*); this is how the ambiguity of having the same function name is resolved. In C++, an overloaded function must be declared with `overload` keyword before it is defined. For example:

```
overload  add;  // Must be first declaration in C++ 1.0.
                // Not needed in C++ 2.0 or later.

int   add( int, int );      // Is called when args are integers
float add( float *, float *); // Is called when args are
                              // pointers to floats
```

2.6.4 Overloaded Operators

An overloaded operator is loaded with different meanings and actions depending on its operands. An overloaded operator must match the C++ built-in operators `op` and their meanings. It is defined as an operator function with the `operator` keyword. The syntax of defining an overloaded function is:

```
<return type> operator<op> ( <type> <arg1>, ...,
                             <type> <argN> )
```

The key points in creating an overloaded operator are:

- For maintainability of program, overloaded operators may carry the same meanings as their C++ built-in counterparts.
- C++ built-in operators cannot be combined to create new overloaded operators.
- The precedence of the new overloaded operators must remain unchanged as their C++ built-in counterparts.
- For binary overloaded operators, one of the operands must be an object of the user-defined class, or the overloaded operators must be friends of the class.

In **Code 2.5**, the operator `+` may be defined for `Complex`-type numbers as follows:

```
Complex operator+ (Complex A, Complex B)
{
    // ...
}
```

The operator `+` is then overloaded with different meanings: one for the addition of `Complex`-type numbers, and the other for the standard C++ built-in meaning. Notice that the overloaded operator functions for `+`, `-`, `*`, and `/` operators are defined as `friends` of the `Complex` class in **Code 2.5**.

2.7 Dangling Pointers and Memory Leaks

A C++ program may dynamically allocate memory space for data objects (ADTs) using the built-in functions `malloc()`, `calloc()`, `realloc()` or the new operator, and deallocate these run-time resources via the C++ `free()` function or `delete` operator.

From the computer system's point of view, memory resources are managed through a *free list* mechanism. When a program dynamically requests memory space, if the requested amount of memory space (in bytes) is available in the free list, the memory manager of the operating system will assign and allocate at least the requested amount of memory space from the *heap* to the C++ program. When the program specifies that previously allocated memory resources are to be freed, the operating system will return these resources to the free list.

While the operating system is keeping track of the overall memory resources, it is the program that is making the requests and thus it is the program that has direct control. As such, it is possible that ill-conceived programs may improperly allocate and deallocate memory, precipitating serious system problems. Two of the most common such problems are identified as *dangling pointers* and *memory leaks.*

2.7.1 Dangling Pointers

To illustrate the concept of a dangling pointer, consider the following code segment:

```
DATA_OBJECT  *ptr_1, *ptr_2;   // Statement 1
ptr_1 = new DATA_OBJECT;       // Statement 2
ptr_1->data = 'A';             // Statement 3
ptr_2 = ptr_1;                 // Statement 4
delete   ptr_2;                // Statement 5
ptr_1->data = 'B';             // Statement 6
```

In this code segment, Statement 1 allocates two ptrs, `ptr_1` and `ptr_2`, to a data type `DATA_OBJECT`. Statement 2 subsequently allocates memory space for the `DATA_OBJECT` and assigns it to `ptr_1`. Statement 3 assigns 'A' as the `data` of this `DATA_OBJECT` while Statement 4 equates `ptr_2` with `ptr_1`. At this point, both pointers hold the base address of the same `DATA_OBJECT`, as shown in Figure 2.2(a). Subsequently, Statement 5 frees the `DATA_OBJECT` memory space, leaving `ptr_1` a dangling pointer, as it references a deallocated resource, as shown by the shaded box in Figure 2.2(b). Further operations on `ptr_1`, as suggested by Statement 6, result in a programming error.

2.7.2 Memory Leaks

A memory leak (also referred to as *garbage*) is the most serious and frequent programming error when using C++ classes. Consider the following code segment as an illustration of the problem:

```
OBJECT  example_obj;                              // Statement 1
example_obj.ptr = example_obj.create("FOO");      // Statement 2
example_obj.ptr = NULL;                           // Statement 3
```

Statement 1 creates an `example_object` as an instance of an `OBJECT` class. Statement 2 creates some internal data structures for the object

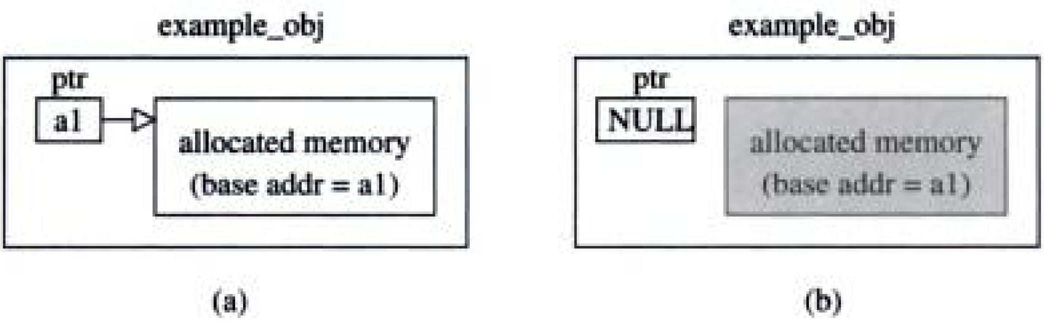


FIGURE 2.3. Memory leak.

2.8 OOP Application: Complex Numbers

While C++ does not provide a built-in type for the complex numbers, a complex number may be viewed as a structured pair (a, b) data object. We define a `Complex` type class and the overloaded operators `+`, `-`, `*`, and `/` as friends of the `Complex` class. These concepts are demonstated in **Code 2.5**.

Code 2.5

```
//  Program:  complex.c++
//  Purpose:  Object-Oriented Implementation of
//             Complex numbers objects "a + bi".
//             Use "Operator Overloading" method.
//
#include <stdio.h>
#include <stdlib.h>
typedef  float      DATA_TYPE;

//  Define a class "Complex" for complex number
//  objects "a + bi"
class  Complex {
  private:
    DATA_TYPE  real;  // Real part "a"
    DATA_TYPE  imag;  // Imaginary part "b"
  public:
    Complex( DATA_TYPE new_re, DATA_TYPE new_im );

    friend Complex  operator+ (Complex A, Complex B);
    friend Complex  operator- (Complex A, Complex B);
    friend Complex  operator* (Complex A, Complex B);
    friend Complex  operator/ (Complex A, Complex B);
    friend Complex  operator- (Complex A);
    friend void     print_complex ( Complex A,
                     char  *Complex_var_name );
};
```

```
//
Complex  operator/ (Complex A, Complex B)
{
   DATA_TYPE   temp1, temp2, temp3;

   // Is the divisor  B = 0 ?
   if (B.real != 0 && B.imag != 0) {
      //  Divisor complex number B is non-zero
      temp1 = A.real * B.real + A.imag * B.imag;
      temp2 = A.imag * B.real - A.real * B.imag;
      temp3 = B.real * B.real + B.imag * B.imag;
      return Complex( temp1/temp3, temp2/temp3 );
    }
    else {
      printf("\n complex_div: ERROR: Division by %s \n",
             "a Complex number ZERO is not allowed ");
      exit(-1);
    }
}

//  print_complex() : Print a complex object
//
void  print_complex (Complex  A,
                     char    *Complex_var_name)
{
   printf(" Complex %s = %10.3f +  %10.3f * i \n",
           Complex_var_name, A.real, A.imag );
}

// main():
//   OBJECT-ORIENTED IMPLEMENTATION of Complex
//   number objects.
//
void  main(void)
{
   //  Declare and initialize "C_obj" and
   //  "D_obj" of the "Complex" class
   Complex  C_obj(10.0, 12.0),
            D_obj(-5.0, 15.0);

   printf("\n ** OOP IMPLEMENTATION %s \n\n",
          "OF COMPLEX NUMBER OBJECTS **");
   print_complex (C_obj, "C    ");
   print_complex (D_obj, "D    ");
   print_complex (C_obj + D_obj, "C + D");
   print_complex (C_obj - D_obj, "C - D");
   print_complex (C_obj * D_obj, "C * D");
   print_complex (C_obj / D_obj, "C / D");
   print_complex (- C_obj, "- C  ");
}
```

FIGURE 3.1. Array base class methods.

As identified by Definition 3.1, an array object is an instance of some derived *implementation specific* class. For the examples in this chapter, we define a *base* class `Array` from which all array classes derive. The `Array` class is defined in `bc_array.h` as follows:

```
// Program:  bc_array.h
// Purpose:  Define base class "Array" for array objects.

extern class Derived_Ary;
// "Derived_Ary" may be a template for a 1- or 2- or
// multi-dimensional array objects.
class  Array {
  //  Declare member functions for matrix
  //  operations as public interfaces
  public:
    virtual void       store(DATA_TYPE data) = 0;
    virtual DATA_TYPE  retrieve(void) = 0;
    friend Derived_Ary operator+ (Derived_Ary A, Derived_Ary B);
    friend Derived_Ary operator- (Derived_Ary A, Derived_Ary B);
    friend Derived_Ary operator* (Derived_Ary A, Derived_Ary B);
    Derived_Ary operator= (Derived_Ary A);
    friend void print_array(Derived_Ary A, char *header);
};
```

Figure 3.1 depicts the base `Array` object and its members. For reasons of convention and efficiency, overloaded operators are defined as `friends` of the `Derived_Ary` class. The `Derived_Ary` class is an implementation specific class derived from the base class `Array`, and may be a template for one-, two-, or multidimensional array objects. `Derived_Ary` is implemented separately in **Code 3.1** for the one-dimensional array object and in **Code 3.2** for the two-dimensional array object. Since `store()` and `retrieve()` use the array object's index or indices, they are implementation specific. However, they could have been declared as `public` members only in the derived class `Derived_Ary`.

3.2 One-Dimensional Arrays

In this section the basic concepts and ADT definitions are first discussed and then the OOP implementation of one-dimensional arrays is presented.

3.2.1 Declaration of Arrays in C++

An array is a sequential form of the same data type objects. The array is identified by a *name* and an *index* and is of fixed size. The syntax for declaring an array is:

```
<data_type>  <array_name>[<size>];
```

For example, the statement `int array[4];`, declares an array named `array` whose data elements are of integer type. The structure `array` has four elements, `array[0]`, `array[1]`, `array[2]`, and `array[3]`. The array index ranges from 0 to 3. The memory layout for `array` is linear and contiguous. That is, memory location `array[0]` is adjacent to `array[1]`. The memory layout of an array is shown in Figure 3.2.

3.2.2 Storing and Retrieving an Element in an Array

Array elements may be randomly accessed. The two ways of accessing array elements are by

- array name and index, and
- a pointer to an array.

The syntax for accessing an array element by name and index is

```
<array_name>[index]
```

For example, the third element in the array `array` is accessed by `array[2]`.

A value, `new_value`, may be stored into and retrieved from `array[2]` with the statements:

```
array[2] = new_value; // store
get_value = array[2]; // retrieve
```

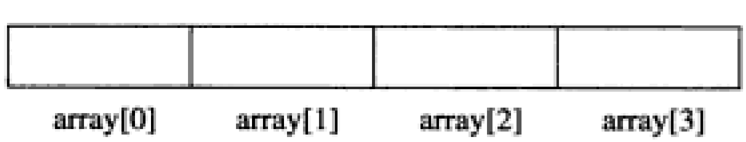


FIGURE 3.2. Memory layout of an array.

```
typedef  int   DATA_TYPE;
void init_ary (DATA_TYPE  array[])
{
    int  index,
    size = sizeof(array)/sizeof(DATA_TYPE);
    for (index = 0; index < size; index++)
        array[index] = 0;
}
```

When a pointer `array_ptr` to an array is used, the code for initializing a one-dimensional array is:

```
void init_ary (DATA_TYPE  array[])
{
    int  index,
    size = sizeof(array)/sizeof(DATA_TYPE);
    register DATA_TYPE *array_ptr = &array[0];
    for (index = size; --index >= 0; *array_ptr++ = 0);
}
```

The *store* (or *put*) and *retrieve* (or *get*) operations, as discussed in Section 3.2.2, are implemented as follows:

```
array[2]  = new_value;  // Store "new_value"
get_value = array[2];   // Get value from array[2]
```

The *retrieve* operation does not destroy the element to be retrieved. This operation may fail in the following cases:

- When the specified array index, say k, is less than 0, or greater than the array size.
- When the specified array index, say k, is within the valid range, but the element `array[k]` is not defined.

Since other operations primarily depend on the store and retrieve operations, these operations do not need to be discussed. The focus is primarily on its OOP implementation shown in **Code 3.1**.

3.2.7 OOP for One-Dimensional Array

For an OOP implementation, the one-dimensional array object is identified by an ADT one-dimensional array, which is derived from the array object's abstract base class `Array` and is implemented by the `Derived_Ary` class. To ensure data-hiding, encapsulation, message-passing for an OOP, the `Derived_Ary` class, defined in `class_Array_def.h` of **Code 3.1**, is:

```
typedef   int  DATA_TYPE;
#include  "bc_array.h"     // For base class "Array"

//  Define a class "Derived_Ary"
```

```
}

// main():
//    Driver to test OBJECT-ORIENTED
//    IMPLEMENTATION of an One-Dimensional array
//
void  main(void)
{
   // Declare and allocate memory space for two
   // objects "A_obj", "B_obj" in "Derived_Ary" class
   Derived_Ary  A_obj(2), B_obj(2);

   // Initialize Derived_Ary objects A_obj and B_obj
   A_obj.element(0) = 2;  A_obj.element(1) = 4;
   B_obj.element(0) = 8;  B_obj.element(1) = 3;
   printf("\n == OOP IMPLEMENTATION %s == ",
          "OF ONE-DIMENSIONAL ARRAY");
   print_array(A_obj, "A");
   print_array(B_obj, "B");
   print_array(A_obj + B_obj, "A + B");
}
```

Here is the output of **Code 3.1**:

```
== OOP IMPLEMENTATION OF ONE-DIMENSIONAL ARRAY ==
*** Following Array: A

    2           4

*** Following Array: B

    8           3

*** Following Array: A + B

   10           7
```

3.3 Two-Dimensional Arrays

Though C++ does not provide any built-in type for multidimensional arrays, such arrays can be built by adding multiple bracket `[]` pairs to the array variable name. For the two-dimensional and three-dimensional arrays, two bracket pairs and three bracket pairs, respectively, are used.

This section discusses basic concepts, the ADT definition and an OOP implementation. Like one-dimensional arrays, multidimensional arrays have

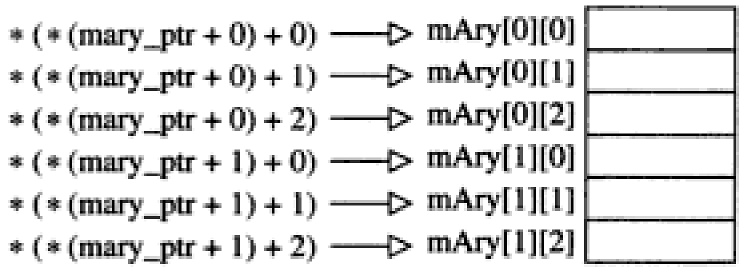


FIGURE 3.6. Pointer to a two-dimensional array.

3.3.3 Initializing a Two-Dimensional Array

Initializing a two-dimensional array can be done at the declaration using the `static` keyword. For example:

```
static int mAry[2][3] = { {-2, 1, -20},    // Row 0
                          { 2, 0,   9} }; // Row 1
```

When a single pointer, e.g., `mary_ptr` is used, initializing the two-dimensional array `mAry` is done by the following statements similar to storing:

```
*mary_ptr = -2;                   // mAry[0][0] = -2
*( *(mary_ptr + 0) + 1) = 1;      // mAry[0][1] =  1
 .
 .
 .

*( *(mary_ptr + 1) + 2) = 9;      // mAry[1][2] =  9
```

Alternatively, an array of pointers to each row may be used for accessing the two-dimensional arrays.

3.3.4 Translating Address of Two-Dimensional Array Elements

This concept is very useful for pointer and object-oriented implementations of arrays. This is the relationship between the base memory address and the memory address of its elements. The formula is:

$$\begin{aligned} Address\ of\ mAry[i][j] &= base\ address + offset \\ &= \&mAry[0][0] \ + \\ &\quad (i \ * \ n\_cols + j) * sizeof(DATA\_TYPE) \end{aligned}$$

where `n_cols` is a number of columns, `DATA_TYPE` is the user-defined data type for elements, and `mAry` is the name of a two-dimensional array.

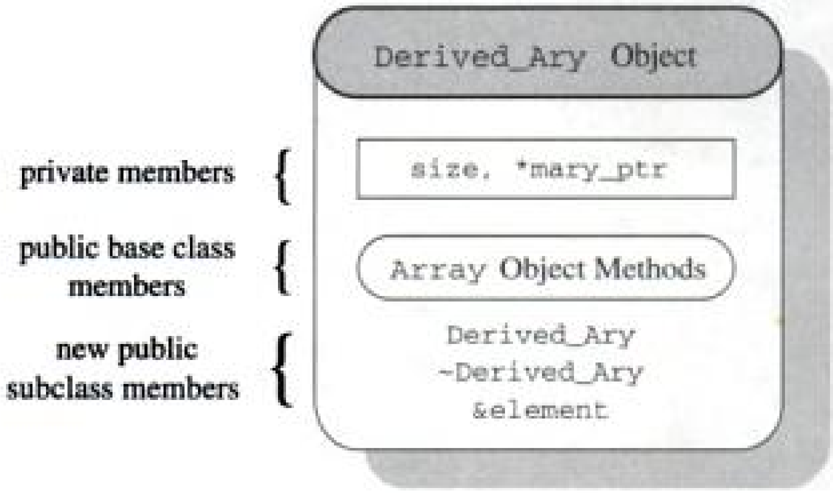


FIGURE 3.7. Derived_Ary object.

```
      DATA_TYPE    &element( int i, int j);
      void        store(DATA_TYPE data);
      DATA_TYPE   retrieve(void);
      friend Derived_Ary operator+ (Derived_Ary A, Derived_Ary B);
      friend Derived_Ary operator- (Derived_Ary A, Derived_Ary B);
      friend Derived_Ary operator* (Derived_Ary A, Derived_Ary B);
      Derived_Ary  operator= (Derived_Ary A);
      friend void  print_array(Derived_Ary A, char *header);
};
```

The Derived_Ary class defines a type for two-dimensional array (matrix) objects, shown in Figure 3.7 encompassing both data and operations. The members DATA_TYPE, n_rows, n_cols, and mary_ptr are private for data-hiding and protection from corruption. The pointer mary_ptr points to the object's data area. The members Derived_Ary(), ~Derived_Ary(), &element(), store(), retrieve() and the friend overloaded operators +, -, *, = for Derived_Ary objects, are public; these are interfaces to the outside world in order to facilitate message passing between Derived_Ary-type object and any object.

Derived_Ary(), with the same name as the type Derived_Ary, is the constructor. It initializes the dimension variables n_rows and n_cols. Using the new operator, it dynamically allocates memory space for n_rows * n_cols number of DATA_TYPE elements for a Derived_Ary type object as follows:

```
mary_ptr = new DATA_TYPE[ n_rows * n_cols ];
```

It is called automatically when Derived_Ary-type object A_obj with two rows and two columns is instantiated:

```
Derived_Ary  A_obj(2, 2);
```

~Derived_Ary(), the destructor, destroys all n_rows * n_cols elements of a Derived_Ary-type object by using the delete operator. The data area being pointed to by mary_ptr, it does:

```
delete [] mary_ptr;
```

The C++ compiler automatically determines the size of the array. This function is implicitly called at the program exit.

The operation, &element(), performs both the store and retrieve operations by using C++ specific feature &. To store a new value, new_value at the position [i, j], use element(i, j). To retrieve a value, use the same function as

```
get_value = element(i, j);  // Retrieve
```

Like their one-dimensional counterparts, the operations store() and retrieve ask user for inputting row and column numbers and subsequently call &element().

The + operator is defined as an overloaded operator for Derived_Ary-type objects. It is a friend of the Derived_Ary class so that it accesses all the private members of the class. It calls Derived_Ary() and element() functions, performs addition of two Derived_Ary-type objects, and returns the result as a Derived_Ary-type object. Its usage is A_obj + B_obj.

The - operator is defined as an overloaded operator for Derived_Ary-type objects. It is a friend of the Derived_Ary class so that it accesses all the private members of the class. It calls Derived_Ary() and element() functions, performs subtraction of two Derived_Ary-type objects, and returns the result as a Derived_Ary-type object. Its usage is A_obj - B_obj.

The * operator is defined as an overloaded operator for Derived_Ary-type objects. It is a friend of the Derived_Ary class so that it accesses all the private members of the class. It calls Derived_Ary() and element() functions, performs multiplication of two Derived_Ary-type objects, and returns the result as a Derived_Ary-type object. Its usage is A_obj * B_obj.

The = operator is defined as an overloaded operator for Derived_Ary-type objects. It is a public member of the Derived_Ary class so that it is accessible outside of this class. It calls Derived_Ary() and element() functions, assigns one Derived_Ary-type object to another, and returns the result as a Derived_Ary-type object. Its usage is B_obj = A_obj.

Print_array() is declared a friend of the Derived_Ary class. It calls element() to retrieve Derived_Ary-type object's data elements, and prints them.

Main() is the test driver for OOP implementation of the two-dimensional array. It instantiates two objects A and B of type Derived_Ary by implicitly calling Derived_Ary(). Since main() is not a member of the

`Derived_Ary` class, it sends messages to the objects `A` and `B` through the public interface routines. The messages include storing values, adding, subtracting, multiplying, assigning, and printing objects.

The complete code and the output are in **Code 3.2**.

Code 3.2

```
// Program: 2dim_ary.c++
//
// Purpose:
//   Object-Oriented Implementation of a two-dimensional
//   array (matrix). Implement "store", "retrieve",
//   matrix "addition", "subtraction", "multiplication"
//   "assignment" as "Overloaded Operators".

#include  <stdio.h>
#include  <iostream.h>
#include  "class_Derived_Ary_def.h"

//   Define a class "Derived_Ary" for two-dimensional
//   array (Matrix) derived from the base class.
class  Derived_Ary : private Array {
  private:
    int         n_rows,     // Number of rows
                n_cols;     // Number of columns
    DATA_TYPE   *mary_ptr;  // Pointer to data area

  //  Declare member functions for matrix
  //  operations as public interfaces
  public:
    Derived_Ary(int rows, int cols);  // Constructor
    ~Derived_Ary(void);               // Destructor
    DATA_TYPE    &element( int i, int j);
    void        store(DATA_TYPE data);
    DATA_TYPE  retrieve(void);
    friend Derived_Ary operator+ (Derived_Ary A, Derived_Ary B);
    friend Derived_Ary operator- (Derived_Ary A, Derived_Ary B);
    friend Derived_Ary operator* (Derived_Ary A, Derived_Ary B);
    Derived_Ary  operator= (Derived_Ary A);
    friend void  print_array(Derived_Ary A, char *header);
};

// Derived_Ary():
//   Initialize the dimension, and allocate
//   memory space for an object of "Derived_Ary" class
//
Derived_Ary::Derived_Ary(int rows, int cols)
{
   // Initialize dimensions of the array object.
   n_rows = rows;
   n_cols = cols;
   // Allocate memory space
```

```
   mary_ptr = new DATA_TYPE[n_rows * n_cols];
}

// ~Derived_Ary():
//   Delete memory space for an object of "Derived_Ary"
//   class.
//
Derived_Ary::~Derived_Ary(void)
{
   delete  [] mary_ptr;
}

// element():
//   Store and retrieve an element A[i][j]
//   of an object in "Derived_Ary" class
//   Usages:
//    To store: A.element(i,j) = new_value;
//    To retrieve: value = A.element(i,j);
//
DATA_TYPE & Derived_Ary::element(int i, int j)
{
   return (mary_ptr [i * n_cols + j]);
}

void Derived_Ary::store(DATA_TYPE data)
{
   int row_index = 0, col_index = 0;
   printf("\nTo store, enter array row index: ");
   cin >> row_index;
   printf("\nTo store, enter array column index: ");
   cin >> col_index;
   element(row_index, col_index) = data;
}

DATA_TYPE Derived_Ary::retrieve(void)
{
   int row_index = 0, col_index = 0;
   printf("\nTo retrieve, enter array row index: ");
   cin >> row_index;
   printf("\nTo retrieve, enter array column index: ");
   cin >> col_index;
   return (element(row_index, col_index));
}

// Derived_Ary Overloaded Operator: "+"
//   Add two matrices A (m x n) & B (m x n)
//   and return a "Derived_Ary" object C (m x n).
//   Usage: C = A + B;
//
Derived_Ary  operator+ (Derived_Ary A, Derived_Ary B)
{
   int  i, // Used for row index, e.g. A[i][j]
        j; // Used for col index, e.g. A[i][j]
   // Declare a temporary matrix for result
```

```
//   Usage: B = A;
//
Derived_Ary  Derived_Ary::operator= (Derived_Ary A)
{
   int  i, // Used for row index, e.g. A[i][j]
        j; // Used for col index, e.g. A[i][j]

   // Allocate a temporary matrix for result
   Derived_Ary tmp( A.n_rows, A.n_cols);

   for (i = 0; i < A.n_rows; i++)
     for (j = 0; j < A.n_cols; j++)
        tmp.element(i, j) = A.element(i, j);
   return (tmp);
}

// print_array():
//   Print a matrix object A (m x n) with a
//   header.
//
void print_array (Derived_Ary A, char *hdr)
{
   int  i, // Used for row index, e.g. A[i][j]
        j; // Used for col index, e.g. A[i][j]
   // Declare a temporary matrix to store result
   Derived_Ary tmp(A.n_rows, A.n_cols);

   printf("\n *** Two-dimensional array (matrix): %s \n\n",
           hdr);
   printf("     ");
   for (j = 0; j < A.n_cols; j++)
      printf("   col %d", j);
   printf("\n\n");   // Skip a line
   for (i = 0; i < A.n_rows; i++) {
     printf("row %d ", i);
     for (j = 0; j < A.n_cols; j++)
       printf(" %5d ", A.element(i, j));
     // Print a new line after a row
     printf("\n");
   }
}

// main():
//   Driver to test OBJECT-ORIENTED
//   IMPLEMENTATION of a two-dimensional object.
//
void  main(void)
{
   // Instantiate two objects "A_obj" & "B_obj"
   // in "Derived_Ary" class
   Derived_Ary A_obj(2, 2), B_obj(2, 2);

   // Initialize Derived_Ary objects A_obj and B_obj
   A_obj.element(0,0) = 2; A_obj.element(0,1) = 4;
```

```
    A_obj.element(1,0) = 5; A_obj.element(1,1) = 6;
    B_obj.element(0,0) = 8; B_obj.element(0,1) = 3;
    B_obj.element(1,0) = 7; B_obj.element(1,1) = 1;
    printf("\n OOP IMPLEMENTATION OF %s",
      "TWO-DIMENSIONAL ARRAY OBJECT");
    print_array(A_obj, "A");
    print_array(B_obj, "B");
    print_array(A_obj + B_obj, "A + B");
    print_array(A_obj - B_obj, "A - B");
    print_array(A_obj * B_obj, "A * B");
    B_obj = A_obj;
    print_array(B_obj, "B = A");
}
```

Here is the output of **Code 3.2**:

```
OOP IMPLEMENTATION OF TWO-DIMENSIONAL ARRAY OBJECT
*** Two-dimensional array (matrix): A

       col 0   col 1

row 0     2       4
row 1     5       6

*** Two-dimensional array (matrix): B

       col 0   col 1

row 0     8       3
row 1     7       1

*** Two-dimensional array (matrix): A + B

       col 0   col 1

row 0    10       7
row 1    12       7

*** Two-dimensional array (matrix): A - B

       col 0   col 1

row 0    -6       1
row 1    -2       5

*** Two-dimensional array (matrix): A * B

       col 0   col 1

row 0    44      10
row 1    82      21
```

'S'	't'	'r'	'i'	'n'	'g'	'\0'
s	s+1	s+2	s+3	s+4	s+5	s+6

FIGURE 3.8. Accessing a pointer-based string.

```
printf("%s", str_ptr);
```

In the case of a pointer-based representation of a string, pointer arithmetic is conveniently used to manipulate and traverse the string. In Figure 3.8, `s` points to the first character `S` of the string, `*s = 'S'`; `s++` (i.e., `s` is incremented by 1) points to the next character `t`, `*(s + 1) = 't'`. This pointer arithmetic process may be continued to access and traverse the string until the end of the string, `\0`, is encountered.

An OOP for a pointer-based string is given as **Code 3.4**. What sets the pointer-based version of the array apart from the array-based version is its use of memory space. An array-based string is allocated a fixed amount of memory space at its declaration; this is potentially wasteful. On the other hand, no fixed memory space is allocated for a pointer-based string; it is of variable length and is dynamically stored in memory.

3.4.2 Array of Array-Based Strings

When a string is represented by an array of characters, an array of such strings is a two-dimensional array of characters, the row dimension being the number of strings, and the column dimension being the maximum string length (i.e., the maximum number of characters among the strings).

In C++, the declaration of an array of strings, for example `menu[][]`, is:

```
char  menu[ARY_SIZE][MAX_STR_LENGTH];
```

where `ARY_SIZE` is the maximum number of menu items, and each menu item is a string of maximum length, `MAX_STR_LENGTH`. The memory spaces of `ARY_SIZE` × `MAX_STR_LENGTH` number of bytes are automatically allocated by this declaration. The characters or substring beyond the `MAX_STR_LENGTH` will be automatically truncated.

Initializing an array of strings is done at the declaration, as shown below.

```
const int ARY_SIZE       = 5;
const int MAX_STR_LENGTH = 26;
static char menu[ARY_SIZE][MAX_STR_LENGTH] = {
     "Add a member record",
     "Remove a member record",
     "Change a member record",
     "Show a member record",
```

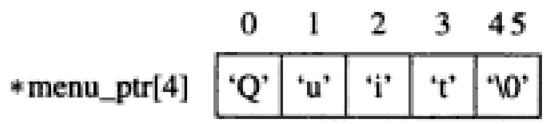


FIGURE 3.10. Memory layout for a pointer-based string.

```
menu_ptr[0] ==   address of "Add a member record"
menu_ptr[1] ==   address of "Remove a member record"
menu_ptr[2] ==   address of "Change a member record"
menu_ptr[3] ==   address of "Show a member record"
menu_ptr[4] ==   address of "Quit"
```

The following code shows how to print a menu string, an array of pointers to each menu item:

```
//
// print_str_ptr():
//   Print a variable length string, strings
//   are stored in an array.
//
void print_str_ptr (char *menu_ptr[], int index)
{
    printf(" %s \n", menu_ptr[index]);
}
```

3.4.4 String Object and Its OOP Implementation

A *string object* is specified by an ADT string. An ADT string is a data structure that contains a sequence of characters as data, and the associated operations to process the data. Some operations for an ADT string are shown in `bc_strng.h`. The OOP implementation of string objects whose data are of variable length is shown in **Code 3.4**. For string objects, an abstract base class `String` shown in Figure 3.11 is defined in `bc_strng.h`:

```
//  Program:  bc_strng.h

extern class Derived_String;
//  Define base class "String" for "String" objects
class  String {
  public:
    virtual int     get_length (void) = 0;
    virtual Derived_String parse_str_obj(Derived_String Delim)=0;

    // Declare Overloaded Operators for "String"
    virtual void   operator=(Derived_String A) = 0;
    friend  Derived_String operator+(Derived_String A,
```

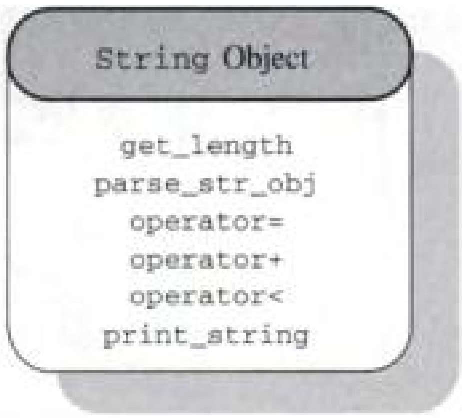


FIGURE 3.11. String base class methods.

```
                                        Derived_String B);
    friend  int  operator<( Derived_String A, Derived_String B);
    friend  void print_string(char *hdr);
};
```

`Derived_String` may be a template for array-based (i.e., fixed-length) or pointer-based (i.e., variable-length) string objects.

When the string is of a fixed length `STR_SIZE`, that is, data of the string is maintained as an array `str_data_ptr[STR_SIZE]` of characters, the array-based string object is defined by the `Derived_String` class, derived from the base class `String`, as follows:

```
//  Define class "Derived_String" derived from "String"
class  Derived_String : private String {
  private:
    int    STR_SIZE;               // Array str_data_ptr[STR_SIZE]
    char   *str_data_ptr;          // of fixed length STR_SIZE
  public:
    Derived_String(char *str, int size);  //  Constructor
    ~Derived_String();            //  Destructor
    int     get_length (void);
    Derived_String  parse_str_obj(Derived_String Delim);

    // Declare Overloaded Operators for "String"
    void    operator=( Derived_String A);
    friend  Derived_String operator+
                          (Derived_String A, Derived_String B);
    friend  int operator<( Derived_String A, Derived_String B);
    void    print_string(char *hdr);
};
```

The complete OOP implementation for an array-based string is left as Exercise 3-3.

An OOP implementation of string objects whose data are of variable length is shown in **Code 3.4**. This string object is depicted in Figure 3.12.

```
//  "String" Overloaded Operator: "+"
//    Join two strings together by
//    adding str2 after str1 object.
//    Usage:  Derived_String  str1, str2, str3;
//              str3 = str1 + str2;
//
Derived_String
operator+ (Derived_String str1, Derived_String str2)
{
  //  Declare the resulting "Derived_String" object
  Derived_String   sum_obj(" ");  // initialized blank

  char *str1_data_ptr = str1.str_data_ptr;
  char *str2_data_ptr = str2.str_data_ptr;
  char *sum_data_ptr  = sum_obj.str_data_ptr;

  //  The data of the first Derived_String object
  //  is copied into data of "sum_obj" string.
  while (*str1_data_ptr != '\0')
        *sum_data_ptr++ = *str1_data_ptr++;

  //  add a space between str1 & str2 objects
  *sum_data_ptr++ = ' ';
  //  now the second Derived_String object is copied.
  while (*str2_data_ptr != '\0')
        *sum_data_ptr++ = *str2_data_ptr++;

  //  put '\0' to indicate end of string.
  *sum_data_ptr = '\0';
  return (sum_obj);
}

//  "String" Overloaded Operator: "<"
//
//   Returns
//    -1 if numerical value of str1 object is  < str2
//     0 if   "    "     "  "     "       "     =   "
//     1 if   "    "     "  "     "       "     >   "
//
//  "str_data_ptr" members of the two "String" objects
//  "str1", "str2" are compared.
//
int
operator< (Derived_String str1_obj, Derived_String str2_obj)
{
  // ASCII str1 & str2 are coverted into floating
  // point values.
  float str1_value = atof(str1_obj.str_data_ptr);
  float str2_value = atof(str2_obj.str_data_ptr);
  if (str1_value < str2_value)
        return (-1);
  else if (str1_value > str2_value)
        return (1);
  else          //  str1_value = str2_value
```

```
        return (0);
}

//  print_string():
//     Print the data of a "String" object
//
void  Derived_String::print_string(char *hdr)
{
  printf("\n %s: \n    %s \n", hdr, str_data_ptr);
}

//
//  main():
//     Object-Oriented Implementation of ADT Derived_String
void   main(void)
{
   //  Declare and initialize "Derived_String" class'
   //  objects "str_obj1", "str_obj2"
   Derived_String   str_obj1("Object-Oriented");
   Derived_String   str_obj2("Strings");

   printf("\n ** OOP IMPLEMENTATION %s %s %s \n",
           "OF STRING OBJECTS","\n    USING",
           "POINTERS ***");
   str_obj1.print_string("String object 1 is");
   str_obj2.print_string("String object 2 is");
   //  Test overloaded "String" operators: +, <, =
   (str_obj1 + str_obj2).print_string(
                "Concatenated string is");
   if ((str_obj1 < str_obj2) == 1)
     printf("\n String1 is greater than String2 \n");
   str_obj1 = str_obj2;
   str_obj1.print_string(
     "After assigning String2 to String1");
}
```

Code 3.4 for an OOP implementation of pointer-based string objects yields this output:

```
** OOP IMPLEMENTATION OF STRING OBJECTS
   USING POINTERS **

String object 1 is:
   Object-Oriented

String object 2 is:
   Strings

Concatenated string is:
   Object-Oriented Strings
```

```
//  "People_Database" class
class  People: private  People_Database {
  private:
    int     total_members,
            curr_last;
  protected:
    int     db_size;
  public:
    People(int size);   //  Constructor
    int     search_member (Keytype srch_key);
    void    read_str_input (char **str);
    void    add_member(void);
    void    remove_member(void);
    void    change_member_record(void);
    void    wait_for_any_key(void);
    void    show_member_record(void);
    void    exit_people_db(void);
};

//  Define "People_Menu" class that uses all the
//  components and functions of the base class
//  "People".
class  People_Menu : private People {
  private:
    char *title,      *menu_item1, *menu_item2,
         *menu_item3, *menu_item4, *menu_item5;
  public:
    People_Menu(int db_sz);    // Constructor
    ~People_Menu();   // Destructor
    void   display_menu_title(char *title);
    void   show_menu(void);
    void   do_menu(void);
};

//  People():
//    Construct and initialize people
//    database object
//
People::People(int size) : People_Database(size)
{
    People_Database::build_initial_db();
    curr_last     = 9;
    total_members = 10;
}

//  search_member():
//    Search for people from people_db
//    people number as key. If it is not found,
//    return an invalid array index, -1.
//
int  People::search_member(Keytype srch_key)
{
   int  i;
```

```
    // Increase curr_last by 1
    people_db[++curr_last].key = member_id;
    printf(" Enter Member's Name:        ");
    read_str_input (&people_db[curr_last].name);
    printf(" Enter Member's Occupation: ");
    read_str_input (&people_db[curr_last].occupation);
    printf(" Enter Member's Age <integer>: ");
    scanf("%d", &people_db[curr_last].age);
    printf(" Enter Member's Income <integer x $1000>: ");
    scanf("%d", &people_db[curr_last].income);
    printf(" Enter Member's Years <integer>: ");
    scanf("%d", &people_db[curr_last].years);

    total_members = curr_last + 1;
}

//  wait_for_any_key():  Pause screen
//
void  People::wait_for_any_key(void)
{
   printf("\n\n Press y to return to main menu ... ");
#ifdef  MSDOS
   while (getch() != 'y')
#else
   while (getc(stdin) != 'y')
#endif
      ;
}

//  remove_member():
//     Remove a member from people_db database object
//
void  People::remove_member(void)
{
   printf(" Remove menu is not implemented.");
   wait_for_any_key();
}

//  change_member_record():
//     Change people record in people_db database.
//
void  People::change_member_record(void)
{
   printf(" Change menu is not implemented.");
   wait_for_any_key();
}

//  show_member_record():
//      Search a member by member's id (key),
//      and if successful, show that member's
//      record from an object in "People_Database".
//
void  People::show_member_record(void)
{
```

```
  menu_item2 = "\n Remove a member          ..  r";
  menu_item3 = "\n Change a member record   ..  c";
  menu_item4 = "\n Show a member record     ..  s";
  menu_item5 = "\n Exit from database       ..  e";
}

//  ~People_Menu():
//     Destructor
//
People_Menu::~People_Menu()
{
   delete title;        delete menu_item1;
   delete menu_item2;   delete menu_item3;
   delete menu_item4;   delete menu_item5;
}

//  display_menu_title():  Print header of a menu
void  People_Menu::display_menu_title (char *title)
{
   printf("\n\n    =========================== \n");
   printf("     *  PEOPLE %s MENU \n", title);
   printf("    =========================== \n");
}

//  show_menu():
//     Show menu items as the user interface
//     for People Database
//
void  People_Menu::show_menu(void)
{
   printf ("%s %s %s %s %s %s",
     title,           // Main menu title
     menu_item1,      // Menu: ADD people
     menu_item2,      // Menu: REMOVE people
     menu_item3,      // Menu: CHANGE
     menu_item4,      // Menu: SHOW
     menu_item5);     // Menu: EXIT
}

//  do_menu():
//     Get the correct choice number from 1 to 5.
//     Transfer control to the corresponding function.
void  People_Menu::do_menu(void)
{
   char  selection;

   printf("\n\n Select menu (a, r, c, s, e): ");
#ifdef  MSDOS
   switch (getch()) {
#else
   fflush(stdout);
   cin >> selection;
   switch (selection) {
#endif
```

```
    case 'a':
    case 'A':
         //  Add a people record in people_db
         display_menu_title("ADD");
         People::add_member();
         break;
    case 'r':
    case 'R':
         //  Remove people record in people_db
         display_menu_title("REMOVE");
         People::remove_member();
         break;
    case 'c':
    case 'C':
         //  Change people record in people_db
         display_menu_title("CHANGE");
         People::change_member_record();
         break;
    case 's':
    case 'S':
         //  Find people record in people_db & show.
         display_menu_title("SHOW");
         People::show_member_record();
         break;
    case 'e':
    case 'E':
         //  Exit from database people_db
         People::exit_people_db();
         break;
    default:
         printf("\n Unknown choice !!! \n");
         People::wait_for_any_key();
         break;
   }
}

//   main():
//     Test driver for OBJECT-ORIENTED people_1d
//     DATABASE
//
void  main(void)
{
   //  Define an object "pmenu_obj" of "People_Menu",
   //  which will initialize "People_Database" and
   //  "People" type objects with size 50.
   People_Menu   pmenu_obj(50);
   int            REPEATED = 1;

   setbuf(stdin, NULL);
   while (REPEATED) {    // infinite loop until "e"
     pmenu_obj.show_menu();
     pmenu_obj.do_menu();
   }
```

```
}
```

Code 3.5 for an OOP implementation of `people_1d` database produces this output:

```
   PEOPLE ORGANIZATION
People DataBase MAIN MENU

Add a member             ..  a
Remove a member          ..  r
Change a member record   ..  c
Show a member record     ..  s
Exit from database       ..  e

Select menu (a, r, c, s, e): a

===========================
*  PEOPLE ADD MENU
===========================

Enter Member's Id <integer>: 23
Please wait_for_any_key. Checking whether
same member id already exists ...
Enter Member's Name:    Joe Barish
Enter Member's Occupation: Marketing Engineer
Enter Member's Age <integer>: 34
Enter Member's Income <integer x $1000>: 56
Enter Member's Years <integer>: 5


   PEOPLE ORGANIZATION
People DataBase MAIN MENU

Add a member             ..  a
Remove a member          ..  r
Change a member record   ..  c
Show a member record     ..  s
Exit from database       ..  e

Select menu (a, r, c, s, e): s

===========================
*  PEOPLE SHOW MENU
===========================

Enter Member's Id: 23
Member's Number:    23
Member's Name:      Joe Barish
Member's Occupation:Marketing Engineer
Member's Age:       34
Yearly Income:      $56000
Year's of Membership:  5
```

```
Press y to return to main menu ... y


   PEOPLE ORGANIZATION
People DataBase MAIN MENU

Add a member                .. a
Remove a member             .. r
Change a member record      .. c
Show a member record        .. s
Exit from database          .. e

Select menu (a, r, c, s, e): r

===========================
*   PEOPLE REMOVE MENU
===========================
Remove menu is not implemented.

Press y to return to main menu ... y


   PEOPLE ORGANIZATION
People DataBase MAIN MENU

Add a member                .. a
Remove a member             .. r
Change a member record      .. c
Show a member record        .. s
Exit from database          .. e

Select menu (a, r, c, s, e): c

===========================
*   PEOPLE CHANGE MENU
===========================
Change menu is not implemented.

Press y to return to main menu ... y


   PEOPLE ORGANIZATION
People DataBase MAIN MENU

Add a member                .. a
Remove a member             .. r
Change a member record      .. c
Show a member record        .. s
Exit from database          .. e

Select menu (a, r, c, s, e): s

===========================
*   PEOPLE SHOW MENU
```

```
===========================

Enter Member's Id: 63
Member's Number:     63
Member's Name:       Peter
Member's Occupation:sales
Member's Age:        51
Yearly Income:       $95000
Year's of Membership:  1

Press y to return to main menu ... y


   PEOPLE ORGANIZATION
People DataBase MAIN MENU

Add a member               ..  a
Remove a member            ..  r
Change a member record     ..  c
Show a member record       ..  s
Exit from database         ..  e

Select menu (a, r, c, s, e): e
Exiting from People DataBase ...
```

3.6 Exercises

1. For the OOP implementation of an ADT one-dimensional array in **Code 3.1**, write the overloaded operator functions

   (a) `-()` to subtract one `Array` type object from another.
   (b) `=()` to assign one `Array` type object to another.
   (c) `*()` for the vector product of two `Array` type objects.

2. For the OOP implementation of a variable-length string (i.e., pointer version) in **Code 3.4**, write the overloaded operator -() for subtracting (removing) one `String` type object from another on the condition that there is a complete match of the specified string in the target string.

3. Write an OOP for a fixed-length string (i.e., array version) that includes a test driver `main()` and the overloaded operators

   (a) `+()` to append one string object to another.
   (b) `-()` to subtract one string object from another.
   (c) `*()` for appending one string object to another.
   (d) `>()` for appending one string object to another.

(e) `=()` for assigning one string object to another.

4. Write an OOP to read ten variable-length strings from a file and print those strings with lengths of greater than 66 characters into another file.

5. Write an OOP that reads a string from the user keyboard, counts the number of words in the string, and prints the string and the count.

6. The graphical form of the well-known *Pascal's Triangle* is shown below. Each row begins and ends with 1.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

   Write an OOP that implements and displays the graphical form of Pascal's triangle.

7. (Sengupta and Edwards [SE91]) Write an OOP for a string parser. Given a file of input strings, the parser should search each string for braces, parentheses, brackets, and quotes, finding the matching member of the pairs (a) left and right braces, (b) left and right parentheses, (c) left and right brackets, and (d) double quotes. The number of occurrences of the matching member may be more than one, and must be reported. The parser should indicate the location of invalid syntax. The following C++ syntax should be followed:

   - For every open (left) brace, parenthesis, bracket, or quote, there should be a matching close (right). The number of opens should equal the number of closes.
   - Before the innermost open brace, parenthesis, or bracket close, others cannot be closed. For example, `{{[}]}`, `(]`, and `{)` are invalid.
   - Braces cannot be nested inside parentheses or brackets. For example, `([]{()})` is invalid.
   - After an open quote, any character can follow any other character until the close quote. For example, `"xyz{]{]]]]]"` is valid.
   - Quotes cannot be nested. For example, `"xyz("abc")"` is considered to be two strings, `"xyz("` and `")"`.

8. Write an OOP that reads an array-based string, reverses the string, and prints the original and reversed strings.

4

Recursion

The concept of *recursion* is central to the disciplines of computer science and mathematics. Proper application of recursive programming techniques can give way to elegant, compact, and readily maintainable programs. A characteristic situation where recursive programming techniques are useful involves applications for which solutions can be devised as a process of dividing the given problem into a series of smaller and similar subproblems, each requiring the same procedural steps and stopping criterion. Recursive techniques are not always applicable or may be too costly to consider — in such situations, *iterative* (nonrecursive) solutions are best.

This chapter discusses the basic concepts of recursion, how it works, the concepts of the C++ *run-time stack*, as well as the space and time trade-offs associated with recursive implementations. As evidence of its general applicability, recursive programming techniques are to be found throughout the remaining chapters of this book.

4.1 Concept of Recursion

Definition 4.1 *A function is said to be recursive if it is defined in terms of itself. Similarly, a program is said to be recursive if it calls itself.*

There are two main types of recursion to be considered. With *direct recursion*, procedures may call themselves before they are completed. That is, a direct recursive function will call itself from within its own body. The function, `factorial_n_recur()`, given in **Code 4.1** below, is an example of a direct recursive function. With *indirect recursion*, procedures may call other procedures that in turn invoke the procedure that called them. For example, a function `f()` calls a function `g()`, which calls a function `h()`, which in turn calls the original calling function `f()`.

Recursive functions are powerful. They are compact in their representation, as compared to their nonrecursive (iterative) counterparts. However, this does not necessarily come without a price. A major potential drawback of recursive implementations is that many levels of recursion (repeated calls for the same types of tasks) can consume excessive memory space and computing time.

4.2 Divide-and-Conquer and Recursion

Intimately related to recursion is the *divide-and-conquer* approach to problem solving. A divide-and-conquer strategy is used in devising a recursive function for a given problem.

The principal methodology of divide-and-conquer is to repeatedly split or *divide* a given problem into a number of smaller or simpler subproblems until these subproblems are *conquered*; the solution of the individual subproblems merges to a solution of the whole problem. Each division of the problem is a recursive call, and recursion continues until a *terminating condition* is reached — that is, when a solvable subproblem is encountered. The existence of a terminating condition, also referred to as the *anchor* of a recursive function, is an absolute requirement, as the recursive process cannot continue forever. A given problem is conquered when all recursive calls successfully terminate.

In a typical divide-and-conquer scheme, the given problem is divided approximately in half at each stage. A recursive program implementing such a divide-and-conquer strategy would contain two recursive calls, each handling about half of the given input data.

4.3 Recursive and Nonrecursive Functions in C++

We now discuss how to derive a recursive function for a given application, say $n!$ (factorial of n), n being a nonnegative integer.

$$n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1.$$

The first step in the derivation is to identify the terminating condition. Since by definition,

$$0! = 1, 1! = 1$$

these are the termination conditions.

Next, consider a simple case and redefine it in terms of its previous case, using inductive steps to deduce generality. For example:

$$\begin{aligned}
n &= 2, & 2! &= 2 * 1 = 2 * 1! \\
n &= 3, & 3! &= 3 * 2 * 1 = 3 * (2 * 1) = 3 * 2! \\
n &= m, & m! &= m * (m-1) * ... * 2 * 1 \\
n &= m+1, & (m+1)! &= (m+1) * m * (m-1) * ... * 2 * 1 \\
& & &= (m+1) * (m * (m-1) * ... * 2 * 1) \\
& & &= (m+1) * m!
\end{aligned}$$

$$\begin{aligned} power(A,B) &= A^B \\ &= A^1 * A^{B-1} \\ &= A * power(A,(B-1)),\ if B > 1 \end{aligned}$$

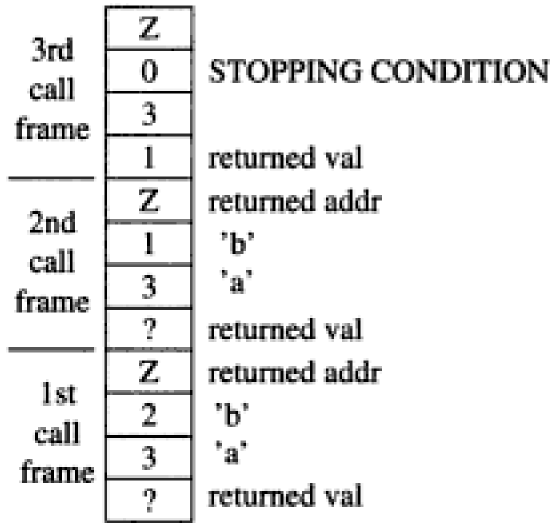
In **Code 4.2**, the direct recursive function, `power_recur()`, and its nonrecursive counterpart, `power_recur()`, are given. Shown below is the sequence of recursive calls for `power_recur(3, 4)`. Note that the termination for this sequence of direct recursive calls is reached for the values of $A = 3$ and $B = 0$.

```
power_recur(3, 4)                              1st call
 = 3 * power_recur(3, 4 - 1)                   2nd call
 = 3 * (3 * power_recur(3, 2))                 3rd call
 = 3 * (3 * (3 * power_recur(3,1)))            4th call
 = 3 * (3 * (3 * (3 * power_recur(3,0))))      5th call
 = 3 * (3 * (3 * (3 * 1)))           /\
 = 3 * (3 * (3 * 3))           terminating condition
 = 3 * (3 * 9)
 = 3 * 27
 = 81
```

Code 4.2

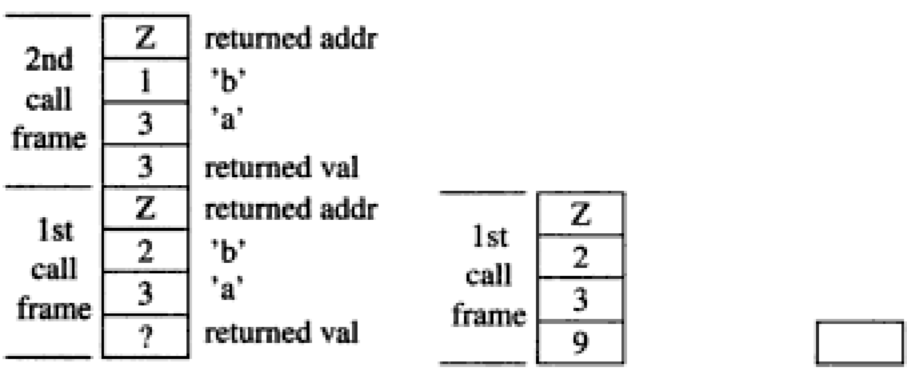
```
//
//  Program:  power.c++
//  Purpose:
//    Raise integer 'a' to the power 'b', i.e., a^b.
//       (i)  Recursive implementation
//       (ii) Nonrecursive (iterative ) implementation.
//
#include  <stdio.h>
#include  <stdlib.h>

//  Recursive version
int  power_recur (int a, int b)
{
     if ((a == 0) && (b == 0)) {
        printf ("\n power_recur: Undefined 0^0 \n");
        exit (0);
     }
     if (a == 0)
        return (0);
     if ((a != 0) && (b == 0))
        return (1);
     if (b > 0)
        return (a * power_recur (a, b - 1));
}
```

The third recursive call, `power_recur(3,0)`, encounters the stopping condition for any further recursive calls:



Call power_recur(3,0)

Backtracking now begins, popping the run-time stack for the process until the stack is empty:



After 1st return of power_recur(3,0)

After 2nd return of power_recur(3, 1)

Empty stack after 3rd return of power_recur(3, 2)

When the C++ stack for the recursive function `power_recur(3,2)` is empty the value returned to `main()` is 9.

Note that the C++ stack of this example could grow larger in size if a larger value of `b` was considered. This would require the consumption of more stack space and more time for pushes and pops.

4.5 OOP Application: The Towers of Hanoi

The game problem *Towers of Hanoi* is a famous application of recursion. As the underlying story goes, three priests in the Temple of Brahma were given three needles and asked to move 64 golden disks from one needle to another. They were to do this in such a way that only one disk would be moved at a time and no disk would ever rest on a smaller one. The world would be destroyed by the time they were to complete their tasks. The problem is restated below.

The Towers of Hanoi Problem

Given three poles (needles), move a tower of N disks from one pole, the *start* pole, to another pole, the *finish* pole, with the added conditions that

- only the start pole contains all of the disks (see Figure 4.2),
- only one disk may be moved at a time, and
- no disk ever rests on a smaller one.

In Figure 4.2, the disks on pole 1 will be moved to pole 2, using pole 3 as the temporary working storage, while adhering to the previously stated conditions. The moves are to be recursively performed until the start pole is empty and the finish pole has all the disks in the prescribed order.

Initially, pole 1 has five disks and poles 2 and 3 are empty. Note that the smallest disk and the largest disk are at the top and the bottom of pole 1, respectively. It is easy to move the smallest disk from one pole to another. The complex part is that, for a pole of height N, the disk at the bottom of the pole is not accessible until all the previous $(N-1)$ smaller disks are moved somewhere and placed subject to the third condition.

The divide-and-conquer approach will be used to solve this problem. We proceed by breaking up the whole problem into a set of three smaller subproblems, identified as the *one-disk*, *two-disk*, and *three-disk* problems. In doing so, similar steps used in solving these subproblems are identified.

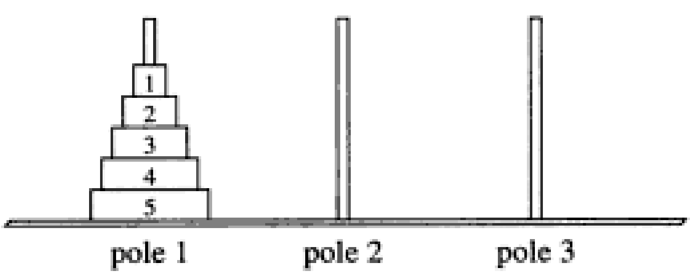


FIGURE 4.2. The Towers of Hanoi with three poles.

Step 1. Move disk 1 (i.e., the smallest disk at the top) from the start pole (pole 1) to the finish pole (pole 2).

Step 2. Move the larger disk (disk 2) from the start pole to the temporary pole (pole 3).

Step 3. Move disk 1 from the finish pole to the temporary pole (pole 3).

Step 4. Pole 1 has only the largest disk (disk 3), and both of the smaller disks are moved to the temporary pole, pole 3. Move the largest disk (disk 3) from pole 1 to the finish pole (pole 2).

Step 5. Move the smallest disk (disk 1) from the temporary pole (pole 3) to the start pole (pole 1).

Step 6. Move the smaller disk (disk 2) from the temporary pole (pole 3) to the finish pole (pole 2), and place it on top of the largest disk (disk 3) on the finish pole.

Step 7. Move the smallest disk (disk 1) from the start pole (pole 1) to the finish pole (pole 2).

Step 8. The start pole (pole 1) is empty, and the finish pole (pole 2) has all the disks in the desired order. The problem is solved.

Alternatively, the three-disk version of the Towers of Hanoi problem can be viewed in terms of the two-disk version. To move the whole pile to pole 2, move the $(N-1)$ stack for $N = 3$ to pole 3, move the largest disk to pole 2, and then move the stack on top of it. In this way, the problem is stated in terms of divide-and-conquer and reduces the complexity of the size-N problem to a problem of size $N-1$.

An analysis of the solutions for one-disk, two-disk, and three-disk versions shows that all the disks, except for the largest disk on the start pole, must be moved to the temporary pole before the largest disk is moved from the start pole to the finish pole.

An OOP implementation of the Towers of Hanoi problem with three poles and N disks is presented as **Code 4.3**. For the current example, we consider the five-disk problem; that is, we have set $N = 5$. To illustrate the actual workings of this code, Figure 4.3 provides a graphical trace of some of the intermediate disk moves involved, starting with the first disk moved.

The object `hanoi_obj` is identified of type `Towers_of_Hanoi`. The data elements of `hanoi_obj` are maximum disks, three poles, pole number, an array of disks associated with each pole, and the height of each pole with disks (rings). The operations for this object are implemented with the methods

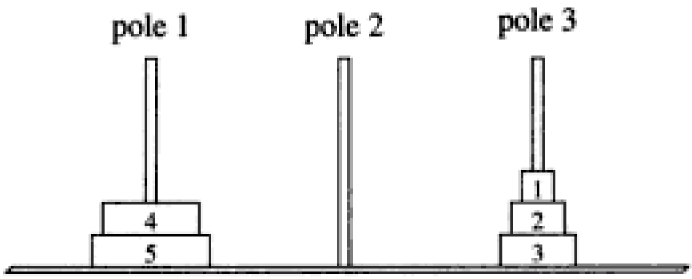
- `Towers_of_Hanoi()`, `~Towers_of_Hanoi()`,

- `build_and_init_poles()`,
- `move_disk()`,
- `solve1_hanoi()`, `solve_hanoi()`,
- `draw_pole()`, and
- `get_max_disks()`.

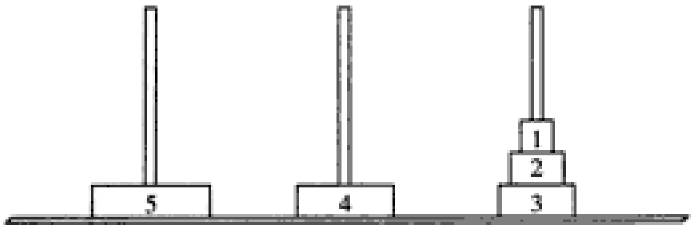
In the header file `def_class_Towers_of_Hanoi.h` we define a class `Towers_of_Hanoi` as follows:

```
class  Towers_of_Hanoi {
  private:
    typedef  int  POLE_NO;
    typedef  int  DISK_NO;
    typedef  int  POLE_HEIGHT;
    int  MAX_DISKS;
    int  MAX_HEIGHT;  // max = MAX_DISKS
    int  MAX_DISK_NO;

    typedef struct POLE_WITH_DISKS {
       // DISK_NO  DISK_ARRAY[MAX_HEIGHT];
       DISK_NO  *DISK_ARRAY;
       DISK_NO  ring_height;
       POLE_NO  pole_no;
    } *POLE_PTR;
    POLE_PTR  pole_1_ptr, pole_2_ptr, pole_3_ptr;
  public:
    POLE_WITH_DISKS pole_1,  // start pole
                    pole_2,  // finish pole
                    pole_3;  // temporary pole
    Towers_of_Hanoi(int max_disks);
    ~Towers_of_Hanoi();
    void  build_and_init_poles(void);
    void  move_disk(POLE_PTR from, POLE_PTR to);
    void  solve1_hanoi(DISK_NO no_of_disks,
            POLE_PTR start_pole,
            POLE_PTR finish_pole,
            POLE_PTR temp_pole);
    void  solve_hanoi(int no_of_disks);
    void  draw_pole (POLE_WITH_DISKS  pole);
    int   get_max_disks() { return MAX_DISKS; }
};
```

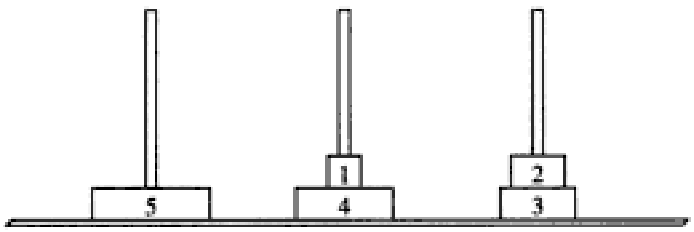
The space `DISK_ARRAY[]` is dynamically allocated when the input `MAX_DISKS` is given, allowing this implementation to support any specified number of disks on three poles. However, the current implementation of `draw_pole()` only allows for the display of the five disks specific to the five-disk version of the Towers of Hanoi problem. Modification of this implementation to allow for the display of any specifiable number of disks is left as an exercise. The methods of **Code 4.3** are now briefly discussed.



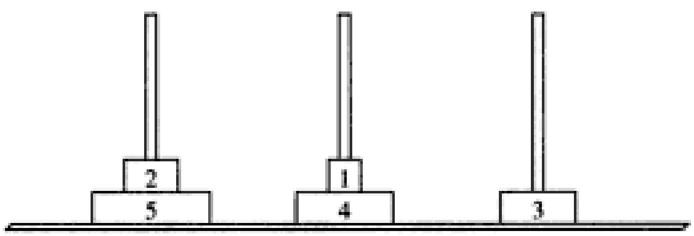
(g) Disk 1 is moved from pole 1 to pole 3.



(h) Disk 4 is moved from pole 1 to pole 2.



(i) Disk 1 is moved from pole 3 to pole 2.



(j) Disk 2 is moved from pole 3 to pole 1.

FIGURE 4.3. *Continued.*

`Towers_of_Hanoi()` is a publicly accessible member function of the `Towers_of_Hanoi` class. As it accepts the input `max_disks`, it sets `MAX_DISKS`. Using the new operator, it dynamically allocates three arrays `DISK_ARRAY[]` of type `DISK_NO` for the three poles `pole_1`, `pole_2`, and `pole_3`.

`~Towers_of_Hanoi()` is also a publicly accessible member function of the `Towers_of_Hanoi` class. This member function is a destructor for an object of type `Towers_of_Hanoi`. It dynamically deallocates memory space for the three disk arrays `DISK_ARRAY[]`.

`Build_and_init_poles()` is a publicly accessible member function of the `Towers_of_Hanoi` class. It builds disk numbers for `pole_1` and initializes disk numbers for `pole_2` and `pole_3`. It also initializes the fields `ring_height` and `pole_no` of each pole structure. The three point-

```
// Constructor for "Towers_of_Hanoi" class
Towers_of_Hanoi::Towers_of_Hanoi(int max_disks)
{
   MAX_DISKS    = max_disks;
   MAX_DISK_NO = MAX_DISKS - 1;
   MAX_HEIGHT  = MAX_DISKS;

   pole_1.DISK_ARRAY = new DISK_NO [MAX_DISKS];
   pole_2.DISK_ARRAY = new DISK_NO [MAX_DISKS];
   pole_3.DISK_ARRAY = new DISK_NO [MAX_DISKS];
}

// Destructor for "Towers_of_Hanoi" class
Towers_of_Hanoi::~Towers_of_Hanoi()
{
   delete pole_1.DISK_ARRAY;
   delete pole_2.DISK_ARRAY;
   delete pole_3.DISK_ARRAY;
}

// build_and_init_poles():
//   Build and initialize three poles with disks.
//
void Towers_of_Hanoi::build_and_init_poles(void)
{
  for (int i = 0; i < MAX_DISKS; i++ ) {
   // pole_1.DISK_ARRAY = ({ 0, 1, 2, 3, 4, 5}, 5};
   // Bottom (i.e., largest) disk = DISK_ARRAY[MAX_DISKS - 1]
   // Top (i.e., smallest) disk = DISK_ARRAY[0]
   pole_1.DISK_ARRAY[i] = MAX_DISKS - i;
   pole_2.DISK_ARRAY[i] = 0;
   pole_3.DISK_ARRAY[i] = 0;
  }
  pole_1.ring_height = MAX_DISKS - 1;
  pole_1.pole_no = 1;
  pole_2.ring_height = 0;
  pole_2.pole_no = 2;
  pole_3.ring_height = 0;
  pole_3.pole_no = 3;

  // Set up the pointers to poles' data structures
  pole_1_ptr = &pole_1;
  pole_2_ptr = &pole_2;
  pole_3_ptr = &pole_3;
}

//
// move_disk():
//   Move disks from one pole "from_pole" to another
//   pole "to_pole". This is the heart of
//   "Towers of Hanoi" problem.
//
void  Towers_of_Hanoi::move_disk (POLE_PTR from_pole,
                                  POLE_PTR to_pole)
```

```
//  draw_pole():
//     Graphically display picture of a pole with disks.
//
void Towers_of_Hanoi::draw_pole (POLE_WITH_DISKS  pole)
{
   static char *pole_picture[] =
    {"             ",
     "             ",
     "       --       ",
     "     01     ",
     "      ----     ",
     "     02     ",
     "     ------     ",
     "    03    ",
     "    --------    ",
     "   04   ",
     "   ----------   ",
     "  05  ",
     "  ------------  " };

   printf("\n");

   for (int i = MAX_DISKS - 1; i > 0; i--) {
      DISK_NO disk_no = pole.DISK_ARRAY[i];

      switch(disk_no) {
        case 0:
              printf("%s\n%s\n",
                pole_picture[0], pole_picture[1]);
              break;
        case 1:
              printf("%s\n%s\n%s\n%s\n",
                pole_picture[0], pole_picture[1],
                pole_picture[2], pole_picture[3]);
              break;
        case 2:
              printf("%s\n%s\n",
                pole_picture[4], pole_picture[5]);
              break;
        case 3:
              printf("%s\n%s\n",
                pole_picture[6], pole_picture[7]);
              break;
        case 4:
              printf("%s\n%s\n",
                pole_picture[8], pole_picture[9]);
              break;
        case 5:
              printf("%s\n%s\n",
                pole_picture[10], pole_picture[11]);
              break;
      }
   }
```

```
    printf(" =============== %s %d\n\n",
           " \n     Pole", pole.pole_no);
}

//  main(): Test driver for OOP implementation of a
//          "Towers of Hanoi" with 3 poles.
//
void main(void)
{
    // Declare a "Towers_of_Hanoi" type object
    // with 5 disks; MAX_DISKS = 6
    Towers_of_Hanoi  hanoi_obj(6);
    Towers_of_Hanoi::DISK_NO
      max_disk_no = hanoi_obj.get_max_disks() - 1;

    printf("\n **  OOP IMPLEMENTATION OF %s  **\n %s \n",
      "TOWERS OF HANOI", "             USING RECURSION");
    hanoi_obj.build_and_init_poles();
    printf("\n\n **  States of Poles before moves **\n");
    hanoi_obj.draw_pole(hanoi_obj.pole_1);
    hanoi_obj.draw_pole(hanoi_obj.pole_2);
    hanoi_obj.solve_hanoi (max_disk_no);
    printf("\n\n **  States of Poles after moves **\n");
    hanoi_obj.draw_pole(hanoi_obj.pole_1);
    hanoi_obj.draw_pole(hanoi_obj.pole_2);
}
```

Code 4.3, an OOP of the Towers of Hanoi with three poles and five disks, yields this output:

```
 **  OOP IMPLEMENTATION OF TOWERS OF HANOI  **
              USING RECURSION


 **  States of Poles before moves **

        !!
        !!
        --
       !01!
       ----
      ! 02 !
      ------
     !  03  !
     --------
    !   04   !
    ----------
   !    05    !
 ===============
     Pole 1
```

```
      !!
      !!
      !!
      !!
      !!
      !!
      !!
      !!
      !!
      !!
===============
    Pole 2


Move disk 1 from pole 1 to pole 3
Move disk 2 from pole 1 to pole 2
Move disk 1 from pole 3 to pole 2
Move disk 3 from pole 1 to pole 3
Move disk 1 from pole 2 to pole 1
Move disk 2 from pole 2 to pole 3
Move disk 1 from pole 1 to pole 3
Move disk 4 from pole 1 to pole 2
Move disk 1 from pole 3 to pole 2
Move disk 2 from pole 3 to pole 1
Move disk 1 from pole 2 to pole 1
Move disk 3 from pole 3 to pole 2
Move disk 1 from pole 1 to pole 3
Move disk 2 from pole 1 to pole 2
Move disk 1 from pole 3 to pole 2
Move disk 5 from pole 1 to pole 3
Move disk 1 from pole 2 to pole 1
Move disk 2 from pole 2 to pole 3
Move disk 1 from pole 1 to pole 3
Move disk 3 from pole 2 to pole 1
Move disk 1 from pole 3 to pole 2
Move disk 2 from pole 3 to pole 1
Move disk 1 from pole 2 to pole 1
Move disk 4 from pole 2 to pole 3
Move disk 1 from pole 1 to pole 3
Move disk 2 from pole 1 to pole 2
Move disk 1 from pole 3 to pole 2
Move disk 3 from pole 1 to pole 3
Move disk 1 from pole 2 to pole 1
Move disk 2 from pole 2 to pole 3
Move disk 1 from pole 1 to pole 3
Move disk 1 from pole 3 to pole 2
Move disk 2 from pole 3 to pole 1
Move disk 1 from pole 2 to pole 1
Move disk 3 from pole 3 to pole 2
Move disk 1 from pole 1 to pole 3
Move disk 2 from pole 1 to pole 2
Move disk 1 from pole 3 to pole 2
Move disk 4 from pole 3 to pole 1
```

request to display the chessboard.

Any number of queens is supported. For example, a 12-Queens problem is implemented by declaring `Queen Twelve_queen_obj(12)` in `main()`.

Code 4.4

```
//
//  Program:  queens.c++
//  Purpose:  OBJECT-ORIENTED IMPLEMENTATION OF
//             THE FAMOUS EIGHT-QUEENS PROBLEM
//
#include <stdio.h>
const int TRUE  = 1;
const int FALSE = 0;
//  Define a "Queens" class
#include "def_class_Queens.h"

// Queens():
//   Initialize the dimension and allocate
//   memory space for the square chessboard.
Queens::Queens(int bd_size)
{
   BOARD_SIZE = bd_size;
   chessboard = new int [BOARD_SIZE * BOARD_SIZE];
   init_chessboard();
}

// ~Queens():
//   Deallocate memory space for the chessboard.
Queens::~Queens(void)
{
   delete  [] chessboard;
}

//  init_chessboard():
//     Initialize chessboard[BOARD_SIZE][BOARD_SIZE]
//
void Queens::init_chessboard(void)
{
   for (int i = 0; i < BOARD_SIZE; i++)
       for (int j = 0; j < BOARD_SIZE; j++)
          //  chessboard[i][j] = FALSE;
          chessboard[i * BOARD_SIZE + j] = FALSE;
}

int Queens::is_safe_to_move(void)
{
   // Check for attack (i.e., presence of any queen,
   // identified by TRUE location value) at each
   // queen/TRUE location.
   //
   for (int i = 0; i < BOARD_SIZE; i++)
```

```
    for (int j = 0; j < BOARD_SIZE; j++)
       // if (chessboard[i][j] == TRUE)
       if (chessboard[i * BOARD_SIZE + j] == TRUE) {
         // Check for attack on vertical axis
         for (int k = 0; k < BOARD_SIZE; k++)
            if ((chessboard[k * BOARD_SIZE + j] == TRUE)
                 && k != i)
               return (FALSE);

         // Check for attack on horizontal axis
         for (int l = 0; l < BOARD_SIZE; l++)
            if ((chessboard[i * BOARD_SIZE + l] == TRUE)
                 && l != j)
               return (FALSE);

        //Check for attack on south east diagonal axis
        for (k = i + 1, l = j + 1;
             k < BOARD_SIZE && l < BOARD_SIZE; k++, l++)
           if (chessboard[k * BOARD_SIZE + l] == TRUE)
               return (FALSE);

         //Check for attack on north west diagonal axis
         for (k = i - 1, l = j - 1;
             k >= 0 && l >= 0; k--, l--)
            if (chessboard[k * BOARD_SIZE + l] == TRUE)
               return (FALSE);

         //Check for attack on south west diagonal axis
         for (k = i + 1, l = j - 1;
             k < BOARD_SIZE && l >= 0; k++, l--)
            if (chessboard[k * BOARD_SIZE + l] == TRUE)
               return (FALSE);

         //Check for attack on north east diagonal axis
         for (k = i - 1, l = j + 1;
             k >= 0 && l < BOARD_SIZE; k--, l++)
            if (chessboard[k * BOARD_SIZE + l] == TRUE)
               return (FALSE);
       } // -- end of "if (chessboard[i ..."
  return (TRUE);
}

//  add_nonattacking_queen():
//    Add nonattacking queens in a chessboard
//    of size BOARD_SIZE x BOARD_SIZE.
//    (Implemented using Recursion.)
//
int Queens::add_nonattacking_queen (int position)
{
   int  outcome = FALSE;
   for (int k = 0; k < BOARD_SIZE && outcome == FALSE;
             k++) {
      if (position >= BOARD_SIZE)
         return (TRUE);
```

14. Modify and test `add_nonattacking_queen()` in **Code 4.4** so that the first queen can be placed in any row and any column position of the chessboard.

15. Write an OOP for parsing pointer-based strings. To do this, you first define a `String` class. The parsing function must be recursive and a `public` member function of the `String` class. The program reads strings from a file until the end of file is reached. The parser scans and reports the total number of occurrences of each string for any of these matching pairs in the string:

    (a) a pair of left "(" and right ")" parentheses

    (b) a pair of left "{" and right "}" braces

    (c) a pair of left "[" and right "]" brackets

    (d) a pair of left "/*"and right "*/"

16. *Maze Problem Project*: Write and test an OOP for an $n \times n$ *maze problem* using recursion. A maze problem demonstrates the concept of *backtracking and tracing*. An $n \times n$ maze is a confusing intricate network of passages and is created when an $n \times n$ matrix is divided into a set of cells (squares), some of them being blocked. The problem is that a rat is set free inside the maze and is searching for a path to exit the maze. While moving through the unblocked (unused or blank) cells in any direction inside the maze, the rat may encounter a blocked cell (B) that is not a through street. It backtracks to find another way and will not come back to the same blocked path. The intricacy lies in finding a path from start (S) to finish (F) in the maze. Create the maze object for $n = 5$ and print the path, if any, from the start cell (denoted by $(0,0)$) to the finish cell (denoted by $(4,4)$).

	0	1	2	3	4
0	S	B		B	
1		B			
2				B	
3	B	B		B	
4			B		F

17. Redo Exercise 4-16 for a 7×7 maze object. Construct a maze to test your program.

5

Lists

An *ordered* or *linear list* is a collection of data elements organized and accessed in an *explicit* sequential fashion. It is a fundamental structure upon which a wide variety of more complex data structures are built.

A list is similar to an array in that both structures store collections of data elements in a sequential manner. In an array, however, the sequential organization of the elements is *implicit* to the position which elements occupy in the array. Thus, an array representation is equivalent to an ordered list if the order of the elements in the array, stored in consecutive memory locations (consecutive indexes), happens to correspond to the desired sequential order of the elements of the list. When an ordered list is directly represented in an array memory, there exists a one-to-one mapping between the order of the list elements and contiguous array locations — for a given element in the list, the *next* element of the list (if it exists) resides in the next incrementally (or decrementally) indexed array location.

A *linked* ordered list, or *linked list*, is a linear list for which the data elements are not necessarily organized with such an implicit sequential memory mapping. In a linked list, a data element in sequence is accessed by an index or pointer to it. As the data elements of a linked list are visited in a contiguous manner, the memory locations corresponding to the elements need not be contiguous. This is in contrast to the array, where sequential access implies indexing contiguous memory locations.

Since its size is fixed at declaration, an array representation of a list is of limited use in applications where the number of data elements is initially not known, and the number grows and shrinks in size during the execution of the application. Additionally, an array is not flexible enough to allow for elements to be rearranged efficiently. These are two major areas where linked lists hold a distinct advantage.

Linked lists are useful in applications where there is a direct sequential ordering of the data elements, access of elements is not at random, dynamic memory allocation and deallocation of data elements are frequently needed, data elements need not be stored in contiguous memory locations, and sequential access to data elements in a forward and/or backward direction is desired.

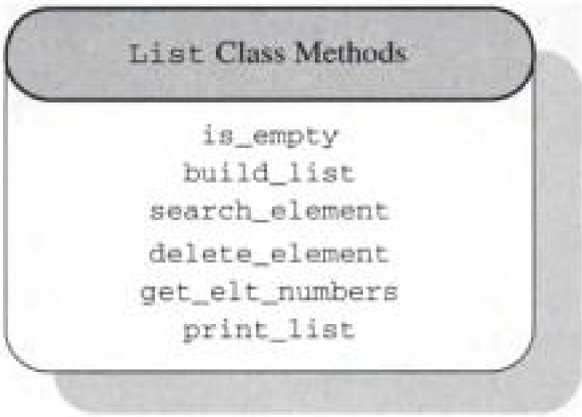


FIGURE 5.1. Public methods for the list class.

A list object is an instance of a derived *implementation specific* class. The *base* class, from which the list object is derived, is implementation unspecific; it provides a set of uniform methods (actions or interfaces) for a list object, which are implemented in the derived class. For the current discussion, a base class `List` is defined in the file `bc_list.h`:

```
//  Program: bc_list.h (Abstract Base class for List object)

class List {
  public:
    virtual BOOLEAN is_empty() = 0;
    virtual void    build_list (DATA_TYPE *A) = 0;
    virtual void    search_element (DATA_TYPE srch_key) = 0;
    virtual void    delete_element (DATA_TYPE target_key) = 0;
    virtual int     get_elt_numbers(void) = 0;
    virtual void    print_list(char *hdr) = 0;
};
```

Figure 5.1 shows the `List` object and its members. The names and arguments for the methods in the base class `List` and in the derived class must exactly match each other.

An implementation specific class is created from the abstract base class with a statement of the form:

```
class <implementation_specific_class>:public List{
     //  ...
};
```

A list object `list_obj` is created by instancing a selected implementation specific class:

```
<implementation_specific_class>  list_obj;
```

5.2 Implementation Specific Linked List Classes

Several OOP implementations for a list object are presented as implementation specific classes.

The lists presented in this chapter fall into one of four structural categories:

- *Singly linked list* — each node (element) contains a single link that connects the node to its successor node. The list is efficient at *forward* traversals.
- *Doubly linked list* — each node contains two links, one to its predecessor node and the other to its successor node. The list is efficient at *forward* as well as *backward* traversal.
- *Singly linked circular list* — a singly linked list for which the last element (tail) is linked to the first element (head) such that the list may be traversed in a circular (ring) fashion.
- *Doubly linked circular list* — a doubly linked list for which the last element is linked to the first and vice versa. As such, the list may be traversed in a circular (ring) fashion in both a *forward* and *backward* direction.

For each of these four types of list structures, one may choose either an array-based implementation or a pointer-based implementation. Fundamentally, these implementations differ in the way memory is allocated for the data elements, how the elements are linked together, and how they are accessed. More specifically, the implementations presented here are categorized as either

- those that employ fixed array memory allocation, or
- those that employ dynamic memory allocation and pointers.

5.3 Array-Based Linked Lists

Linked lists that are implemented using fixed-size array memory typically exist in two varieties, distinguished by the manner in which the elements are stored in the array:

- *Sequential element storage* — predecessor and/or successor links are *implicit* to the array structure.
- *Nonsequential element storage* — predecessor and/or successor links are *explicit* to the array structure, specified in terms of array indexes.

	LIST
list[0]	elem_1
list[1]	elem_2
	⋮
list[i−1]	elem_i−1
list[i]	elem_i
list[i+1]	elem_i+1
	⋮
list[LAST]	last_element
	⋮
list[LIST_SIZE−1]	

FIGURE 5.2. Array-based list with data elements stored sequentially.

In the sequential storage scheme, the elements of an ordered list are arranged in the exact consecutive order as the locations of the array. A key feature of this sequential storage scheme is that *the predecessor and successor indexes of each data element need not be stored as a part of the data record for each element.* These indexes are *implicit* since the list elements are stored in successive elements of the array. Traversing such a list amounts to sequentially indexing the array structure. However, if elements are to be inserted into or deleted from the list, the existing elements must be shifted around. This process is not efficient.

Consider a list array, `list[]`, of size `LIST_SIZE`. The maximum number of elements that may be stored in this list corresponds to `LIST_SIZE`. The successor and predecessor of each element, if either or both exist, is easily referenced in terms of the index of the array. Figure 5.2 illustrates this list using sequential element storage. The following items characterize the predecessor-successor relationship of the data elements:

- `list[0]` has no predecessor; its successor is `list[1]`.
- `list[LIST_SIZE-1]` has no successor; its predecessor is `list[LIST_SIZE-2]`.
- For any `i`, `1 <= i <= LIST_SIZE-1`, `list[i]` has predecessor `list[i-1]`, and successor `list[i+1]`.

A nonsequential storage scheme, on the other hand, relies on explicit storage of predecessor and successor indexes. Insert and delete operations are more efficient in this case, as elements need not be moved. The link indexes are updated instead. This scheme requires additional memory for each element.

5.3.2 Adding an Element After a Given Element

For the array-based list, a new element `new_elt` is added after a given element, `after_elt` using the following steps:

Algorithm: Add an Element After Another

Step 1. Search for `after_elt` in `list[]`. If search is not successful, return; otherwise, assume `i` is the array index for `elt_after`, i.e., `list[i] = elt_after`, and proceed to Step 2.

Step 2. Check whether the list is full. If it is full, return; otherwise proceed to Step 3.

Step 3. Make space for `new_elt`; move `list[LAST]`, ..., `list[i]` downward to respectively `list[LAST+1]`, ..., `list[i + 1]`.

Step 4. Insert `new_elt`: `list[i + 1] = new_elt`.

Step 5. Increment `LAST` counter: `LAST` = `LAST` + 1.

In **Code 5.1**, the `public` member function `add_after()` of the class `Ary_List` implements this algorithm. It calls `search_element1()`, a `private` member function of the `Ary_List` class. Note that a similar function for inserting before a given element can be written; this is left as an exercise.

5.3.3 Deleting an Element from an Array List

For the array-based list, a specified element `target_elt` is deleted using the following steps:

Algorithm: Delete an Element

Step 1. Search for `target_elt` in `list[]`. If search is not successful, return; otherwise, assume `i` is the array index for `target_elt`, i.e., `list[i] = target_elt`, and proceed to Step 2.

Step 2. Overwrite `list[i]` by its successor `list[i + 1]` in the following manner: move up the list elements `list[i + 1]`, ..., `list[LAST]` to respectively `list[i]`, ..., `list[LAST - 1]`.

Step 3. Decrement `LAST` counter: `LAST` = `LAST - 1`.

The implementation of this deletion process is left as an exercise.

```
      printf("\n");
    }
    else
      printf("\n List is empty \n");
}

//  main():
/     Test driver for Object-Oriented
/     implementation of an ADT array list
//
void main(void)
{
   // Declare an object of the "Ary_List" class
   // and set the size of array to 7
   Ary_List  alist_obj(7);

   printf("\n** OOP IMPLEMENTATION %s \n",
          "OF AN ARRAY LIST **");
   alist_obj.build_list ("BALORI");
   alist_obj.print_list ("before inserting \'L\'");
   alist_obj.add_after ('L', 'L');
   alist_obj.print_list ("after inserting \'L\' ");
}
```

Code 5.1 for an OOP implementation of an array-based list object produces this output:

```
** OOP IMPLEMENTATION OF AN ARRAY LIST **
 List before inserting 'L' is:   B  A  L  O  R  I
 List after inserting 'L'  is:   B  A  L  L  O  R  I
```

The array-based list implemented in **Code 5.1** has a couple of general deficiencies. First, the list implements a sequential element storage scheme, and as such, its insertion and deletion procedures require a physical movement of the data elements. This process is inefficient.

Second, as with any array implementation, the size of the array is fixed. In **Code 5.1** the object `alist_obj` is declared with the size of its data area, which is initialized to 7. This limits the list in two ways:

- The growth of the list object cannot exceed the initially declared and allocated size of the array.
- Memory space is wasted if the list does not fully occupy the allocated space.

For the singly linked lists depicted in Figure 5.4 we find that each data element contains two fields, the data contained in the node, labeled `data`, and the pointer to the successor element of the current element, labeled `next`. The elements of the doubly linked lists contain the same fields plus an additional pointer to the predecessor element of the current element, labeled `prev`.

Notice that for all the linked lists depicted in Figure 5.4, the element with a `data` field 'I' and a `next` field with a value of `NULL` is called the last element. Thus, the end of a singly or doubly linked list is determined by the `NULL` value of the `next` field of an element in the list. It follows that an empty list contains `NULL` values for `head_ptr` and `tail_ptr`.

For circularly linked lists, the concept of a tail (or last) data element does not apply. For these types of linked lists it is appropriate to speak of a head pointer only. These structures will be considered in a later section.

5.4.1 Non-OOP Implementation of the Singly Linked List

In the following, a simple non-OOP implementation of a singly linked list object is given. This simple approach will provide motivation for using an OOP approach.

For the singly linked list in Figure 5.4(a), a data element is defined by the following structure:

```
typedef int    DATA_TYPE;
typedef struct SLIST_ELEMENT {
   DATA_TYPE      data;
   SLIST_ELEMENT  *next;
} *LIST_PTR;
```

For simplicity, there are three data elements `elem_1` (first element), `elem_2`, and `elem_3` (last element) of type `SLIST_ELEMENT`, `head_ptr` of type `LIST_PTR` pointing to the first element `elem_1`. As shown below in **Code 5.2**, the singly linked list is formed by making `elem_2` the successor of `elem_1`, and `elem_3` the successor of `elem_2`. This sort of chaining is done by manually using the single link variable `next`. The printing of the first element is done using `head_ptr`, and the next two successive elements are printed using the links `head_ptr->next` and `head_ptr->next->next`.

Code 5.2

```
//  Program:  simple_slist.c++
//  Purpose:  Implementation of a singly linked list
//            by simply linking its each element
//            (character-type data) using pointers.
```

```
//
#include  <stdio.h>
typedef int DATA_TYPE;  // Data type for list's element
typedef struct SLIST_ELEMENT {
     DATA_TYPE       data;
     SLIST_ELEMENT   *next;
} *LIST_PTR;

void main(void)
{
     // Declare and initialize elements
     static SLIST_ELEMENT  elem_1 = { 1, NULL},
                            elem_2 = {-3, NULL},
                            elem_3 = {10, NULL};
     LIST_PTR  head_ptr = &elem_1;

     // Form the singly linked list by making
     // 'elem_2' the successor of 'elem_1', &
     // 'elem_3' the successor of 'elem_2'. This is
     // done by using the single link variable, 'next'.

     elem_1.next = &elem_2;
     elem_2.next = &elem_3;

     // Print the singly linked list
     printf("\n ** A Simple Singly Linked List **\n");
     printf("\n The list is: ");
     while (head_ptr != NULL) {
        printf(" %d -> ", head_ptr->data);
        head_ptr = head_ptr->next;
     }
     printf("NULL \n");
}
```

Code 5.2 for a simple singly linked list produces this output:

```
 ** A Simple Singly Linked List **

 The list is:  1 ->  -3 ->  10 -> NULL
```

The deficiencies of the above simple approach in **Code 5.2** are:

- It does not conceal `data`, `next`, and `head_ptr`.
- It is not designed to handle a large number of the list's data elements.
- It does not have a capability of dynamically increasing and decreasing the number of data elements in the list.

To conceal the data elements and compact the data elements and operations into a single entity, an OOP approach is required. Dynamic list sizing can be accomplished with the dynamic memory allocation and deallocation techniques in C++. In particular, one may employ the `new` operator or `malloc()` call for allocating memory space, and the `delete` operator or `free()` call for deallocating memory space. For example,

```
LIST_PTR elem_1_ptr = new SLIST_ELEMENT;
elem_1_ptr->data = 1;
elem_1_ptr->next = NULL;
delete elem_1_ptr;
```

As pointed out in Chapter 2, the use of the dynamic memory allocation may lead to a problem with memory leaks. This possibility is well noted and resolved in the OOP approach shown below in **Code 5.3**.

5.4.2 OOP Implementation of the Singly Linked List

As discussed above, the member fields of the `SLIST_ELEMENT`-type data structure are not hidden from users or client programs. In the OOP implementation, the data-hiding is performed by using the C++ `class` construct. From the ADT definition of list, the `Singly_Linked_List`-type object is identified. This list object contains list elements as data and ADT list operations as methods. Each list element contains a data field, and a link field. Then, in C++ the singly linked list object is defined by the following `Singly_Linked_List` class in the header file `slist.h`:

```
class  Singly_Linked_List : public List {
  private:
    typedef struct SLIST_ELEMENT {
       DATA_TYPE      data;
       SLIST_ELEMENT  *next;
    } *LIST_PTR;
    LIST_PTR  head_ptr; // Ptr to first element in list
    void      init_slist() { head_ptr = NULL;};
  protected:
    LIST_PTR  search_element1(LIST_PTR, DATA_TYPE);
    LIST_PTR  search_previous(LIST_PTR, DATA_TYPE);
    LIST_PTR  get_head(void) { return head_ptr; }
    BOOLEAN   is_sublist_empty(LIST_PTR lst_ptr);
  //  Declare the interface routines as "public"
  public:
    Singly_Linked_List() { init_slist(); }
    ~Singly_Linked_List();

    BOOLEAN  is_empty() {return (head_ptr == NULL);}
    void     build_list (DATA_TYPE *A);
    void     search_element (DATA_TYPE srch_key);
    void     add_after (DATA_TYPE elt_after,
```

```
Singly_Linked_List slist_obj;
```

In the next sections, we discuss some of the algorithms that may be employed to implement the interface routines (methods) of this class.

5.4.3 Building a Singly Linked List

An algorithm for building a singly linked list object from a given string is outlined with the steps below:

Algorithm: Build a Singly Linked List

Step 1. Declare the data type `SLIST_ELEMENT`, and the `head_ptr`, and `new_ptr` of type `LIST_PTR`.

Step 2. Allocate memory for a new element of type `SLIST_ELEMENT` using the `new` operator; the address of the new element is `new_ptr`.

Step 3. Iteratively create the first (head) element and the successive elements of the singly linked list object in steps 3.1 and 3.4.

Step 3.1.
```
new_ptr->data = string_data;
new_ptr->next = NULL;
```

Step 3.2. If the singly linked list object does not have any elements (i.e., `head_ptr` is NULL), do:
```
head_ptr = new_ptr; tmp_ptr = new_ptr;
```

Step 3.3. Using the `new` operator, allocate memory spaces for the next elements pointed to by `new_ptr`. Connect it with its previous element by the following statements:
```
tmp_ptr->next = new_ptr;
tmp_ptr = tmp_ptr->next;
```

Step 4. Repeat until there is no more input for the element.

These steps are implemented by the nonrecursive function `build_list()` in **Code 5.3**. The last element in the singly linked list object has a NULL value for `next` pointer because it does not have any successor.

5.4.4 Inserting an Element in a Singly Linked List

The algorithm employed for adding or inserting an element in a singly linked list varies depending on where the incoming element is to be inserted. The location for insertion can be:

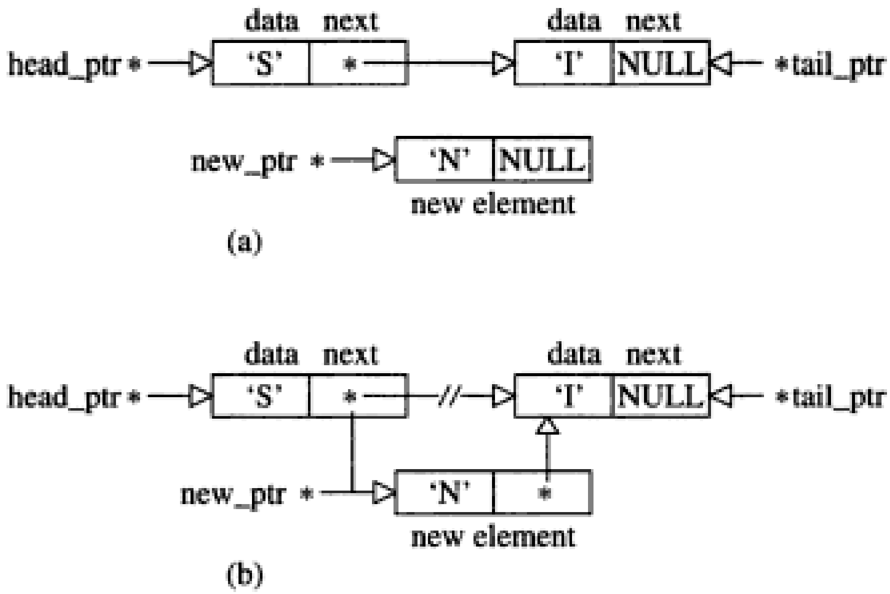


FIGURE 5.6. (a) Singly linked list before adding 'N' before 'I'. (b) Adjusting links during addition of 'N' before 'I'.

Step 6. Add the element between elements 'S' and 'I' by adjusting the `next` link field for the element 'S':

Step 6.1. `new_ptr->next  = element_ptr;`

Step 6.2. `previous_ptr->next = new_ptr;`

Step 6.3. `return;`

Of note is the fact that these steps do not require any physical movement of the elements. Instead, links are adjusted, as in steps 4 and 6. As pointed out earlier, this is a major advantage of the pointer-based list implementations.

The above steps for inserting an element before another specified element may be easily modified for inserting an element after a specified element. In the latter case, steps 4 and 5, the process of searching for the previous element of a given element, would not be necessary.

The actual implementations of the "add-before" and "add-after" algorithms are left as exercises for the reader.

5.4.5 Deleting an Element in a Singly Linked List

The process of deleting a specified element from a singly linked list is straightforward and is illustrated in Figure 5.7. For this example, the element with the data 'I' is to be deleted. Figure 5.7(a) shows the initial state of the list. In Figure 5.7(b) the link to the element containing 'I' is broken and the element is deleted using the `delete` operator. In **Code**

Main(), a client function of the Singly_Linked_List class, instantiates this class's object slist_obj. It calls slist_obj.build_list() to create the slist_obj object from the given string "SNIGDHA". It calls slist_obj.print_list(), slist_obj.delete_element() and slist_obj.get_elt_numbers(). In this manner, main() sends requests for such actions as creating, printing, deleting a specified element, and counting total number of elements, to the object slist_obj. The program's memory is freed by delete str and ~Singly_Linked_List().

Code 5.3

```
//
//  Program:  slist.c++
//  Purpose:  Object-oriented implementation of a singly linked
//            list with a simple data of 'char' type.
//
#include <stdio.h>
typedef   int  BOOLEAN;
typedef   char DATA_TYPE;
#include  "bc_list.h"  // For base class
//  Define a singly linked list type class
//  "Singly_Linked_List" with base class "List"
#include   "slist.h"
//
//  ~Singly_Linked_List():
//    Delete entire singly linked list object and free
//    up its memory spaces (Iterative implementation).
//
Singly_Linked_List::~Singly_Linked_List(void)
{
    LIST_PTR  next_ptr, // pointer to next element
              tmp_ptr;

    tmp_ptr = head_ptr;
    while  (!is_sublist_empty(tmp_ptr)) {
       //  Save pointer to next element
       next_ptr = tmp_ptr->next;
       delete tmp_ptr;    //  Dispose of its space
       tmp_ptr = next_ptr;
     }
     // Assure 'head_ptr' is not a "dangling pointer"
     head_ptr = NULL;
}

BOOLEAN Singly_Linked_List::is_sublist_empty(
                            LIST_PTR lst_ptr)
{
     return (lst_ptr == NULL);
}

//
```

```
//  build_list():
//    Build a singly linked list object from a
//    given string of variable length.
//    (Iterative implementation)
//
void Singly_Linked_List::build_list(DATA_TYPE *str)
{
    LIST_PTR tmp_ptr, new_ptr;

    while (*str != '\0') {
        new_ptr       =  new SLIST_ELEMENT;
        new_ptr->data =  *str++ ;
        new_ptr->next =  NULL;
        if (head_ptr == NULL) {
           head_ptr = new_ptr;
           tmp_ptr = new_ptr;
        }
        else {
           tmp_ptr->next =  new_ptr;
           tmp_ptr = tmp_ptr->next;
        }
    }
}

//  search_element1():
//    Search for an element identified by its key
//    'search_key' (search data) in a list object,
//    pointed to by 'lst_ptr'. Return the pointer
//    if found; otherwise return NULL.
//    (Recursive method)
//
Singly_Linked_List::LIST_PTR
Singly_Linked_List::search_element1(
            LIST_PTR lst_ptr, DATA_TYPE search_key)
{
    if (!is_sublist_empty(lst_ptr)) {
       if (search_key == lst_ptr->data)
          return (lst_ptr);
       search_element1 (lst_ptr->next, search_key);
    }
    else {
      printf("\n search_element: %s \n",
             "Element is not found in list");
      return (NULL);
    }
}

void
Singly_Linked_List::search_element(DATA_TYPE search_key)
{
    if (search_element1 (head_ptr, search_key) != NULL)
      printf("\n Element is found %s \n",
             "in singly linked list");
}
```

```
//  search_previous():
//    Search for an element in a singly linked
//    list object given its data as key 'search_key',
//    and return a pointer to the previous element
//    (predecessor) if found. Return NULL if the
//    given data element is not found or the first
//    element in the list. (Recursive method)
//
Singly_Linked_List::LIST_PTR
Singly_Linked_List::search_previous (LIST_PTR lst_ptr,
                                     DATA_TYPE search_key)
{
    if (lst_ptr != NULL) {
      if (search_key == lst_ptr->next->data)
        return (lst_ptr);
      search_previous (lst_ptr->next, search_key);
    }
    else {
      printf("\n search_previous: Previous %s \n",
             "element is not found in list");
      return (NULL);
    }
}

//  delete_element():
//    Delete an element in a singly linked list
//    object given its data as the search key.
//
void
Singly_Linked_List::delete_element(DATA_TYPE search_key)
{
     LIST_PTR  element_ptr, previous_ptr;

     // Check to see if the given element is the
     // first one in the list object.
     if ((head_ptr != NULL) &&
         (head_ptr->data == search_key)) {
        element_ptr = head_ptr->next;
        delete head_ptr;
        head_ptr = element_ptr;
     }
     if ((element_ptr = search_element1 (head_ptr,
                          search_key)) != NULL) {
        previous_ptr = search_previous (head_ptr,
                                  search_key);
        previous_ptr->next = element_ptr->next;
        delete element_ptr;
     }
}

//  get_elt_numbers():
//    Get elements in a singly linked list object.
//    (Non-recursive, that is, iterative procedure)
```

```
head_ptr = tmp_ptr; head_ptr->prev = NULL;
return;
```

Step 4. Check to see if the element to be deleted (e.g., with value 'I'), is the last one in the list object. If it is, delete the last element, and make its predecessor the last (tail) element of the list. Do:

```
tail_ptr = search_ptr->prev;
tail_ptr->next = NULL; delete search_ptr;
return;
```

Step 5. Now the element to be deleted (pointed to by `search_ptr`) is between the head and the tail elements of the doubly linked list. Drop the link to the target element, and then adjust the links of its next (pointed to by `search_ptr->next`) and previous (pointed to by `search_ptr->prev`) elements to retain the doubly linked list structure.

```
search_ptr->prev->next = search_ptr->next;
search_ptr->next->prev = search_ptr->prev;
```

Step 6. Free the memory space of the element to be deleted (e.g., element with value 'I'): `delete search_ptr; return;`

In Figure 5.11(b), the links to the element with key value 'I' are broken, and then the element is deleted using the `delete` operator. In **Code 5.4**, `delete_elt_obj()` implements this algorithm.

5.4.10 METHODS OF THE `Doubly_Linked_List` CLASS

The following functions are all member functions (some of them being `public` interface routines) of `Doubly_Linked_List` class. These are interface routines to the outside world including `main()`. These routines allow nonmember functions to pass messages (e.g., requests for actions) to `dlist_obj` object of `Doubly_Linked_List` class. All the member fields are accessible to the member functions of the same `Doubly_Linked_List` class. Therefore, it is not necessary to pass these member fields, `head_ptr` and `tail_ptr`, as arguments of the member functions. Since the class `Doubly_Linked_List` is a `friend` of the `DLIST_ELEMENT` class, its member functions do not require the passing of the data members `data`, `next`, and `prev` of `DLIST_ELEMENT` class as arguments. arguments

`Doubly_Linked_List()`, the constructor and `public` member function of `Doubly_Linked_List` class, is implicitly called when this class's object `dlist_obj` is instantiated. It calls `init_dlist()`.

`~Doubly_Linked_List()`, a `public` member function of the class `Doubly_Linked_List`, is implicitly called in `main()` before the program exits. It is iteratively implemented. It destroys the entire doubly linked list object. First, it calls `chk_empty_dlist()` to check whether the list object pointed to by `tmp_ptr` (= `head_ptr`) is empty. For an empty doubly linked list object, it sets `head_ptr` to `NULL` and returns. For a nonempty list object, the pointer to the successor of the current element pointed to by `head_ptr` is saved in `tmp_ptr`. It traverses the doubly linked list object using the forward link pointer `next`, and deallocates the memory space by using the `delete` operator.

`Init_dlist()` initializes an object of `Doubly_Linked_List` class. It sets `head_ptr` to `NULL`. It is made `private` to disallow possible misuse that may cause memory leak.

`Chk_empty_dlist()`, a `private` member of `Doubly_Linked_List` class, compares `lst_ptr` with `NULL`. It returns 1 if list object is empty, and 0 otherwise.

`Build_list()`, a `public` member of `Doubly_Linked_List` class, is an iterative function. An address of a string is passed to this function. Like its counterpart in `Singly_Linked_List` class, it iteratively builds a doubly linked list object `dlist_obj` with character data from a string. The only additional work is properly setting up `prev` pointer field of each `DLIST_ELEMENT`-type object.

`Search_element()` is a `public` member of `Doubly_Linked_List` class. It calls `search_elt_obj()` and if successful, it displays a message "Element is found in doubly linked list object."

`Search_elt_obj()`, a `private` member of `Doubly_Linked_List` class, is a recursive function. It has two arguments, the list pointer `lst_ptr` of type `LIST_PTR`, and `search_key` of type `DATA_TYPE`. It performs like `search_element1()` in **Code 5.3**. It directly calls itself until a match of the target key or the end of list is reached.

`Delete_element()` is a `public` member of `Doubly_Linked_List` class. It calls `search_elt_obj()` to find the given element object specified by `search_key`. If the given element object is not found, it displays a message and returns. Otherwise, the element object is found in the doubly linked list object, and is pointed to by `search_ptr`. It considers three cases for the element object to be deleted at the (1) head of the list, (2) tail of the list, (3) intermediate position. In either of these positions, it efficiently uses the back link `prev` to break the link to the given element object and maintains the list object. Finally, it frees up the memory space held by the given element using the `delete` operator.

`Get_elt_numbers()` and `Print_list()` are `public` member functions of the `Doubly_Linked_List` class. Both perform exactly the same operations as their counterparts in the `Singly_Linked_List` class of **Code 5.3**.

`Print_dlist_obj_backward()`, a `public` member function of the `Doubly_Linked_List` class, is an iterative function. It displays this class's object by linearly traversing the list from the last element object

```
//  search_elt_obj():
//    Search for an element with its key value in
//    a list object. Return the pointer if found;
//    otherwise NULL. (Recursive procedure)
//
LIST_PTR Doubly_Linked_List::search_elt_obj(
              LIST_PTR lst_ptr, DATA_TYPE search_key)
{
     if (!chk_empty_dlist(lst_ptr)) {
          if (search_key == lst_ptr->data)
             return (lst_ptr);
          search_elt_obj (lst_ptr->next, search_key);
     }
     else {
       printf("\n search_elt_obj: %s \n",
              "Element is not found");
       return (NULL);
     }
}

void Doubly_Linked_List::search_element(DATA_TYPE search_key)
{
    if (search_elt_obj(head_ptr, search_key) != NULL)
       printf("\n Element is found %s \n",
              "in doubly linked list object.");
}

//  delete_element():
//    Delete a specified DLIST_ELEMENT object
//
void Doubly_Linked_List::delete_element(DATA_TYPE element_key)
{
    LIST_PTR  lst_ptr = head_ptr,
                 search_ptr, tmp_ptr;

    //  Search for the object to be deleted
    search_ptr = search_elt_obj(lst_ptr, element_key);
    if (search_ptr == NULL) { // object is not found
       printf("\n delete_element: Object to be %s \n",
              "deleted is not found");
       return;
    }
    //  DLIST_ELEMENT class object is found.
    //  Is the object to be deleted the head of the list?
    if (search_ptr == head_ptr) {
       tmp_ptr = head_ptr->next;
       if (tail_ptr == head_ptr)
          tail_ptr = tmp_ptr;
       delete head_ptr;             //  Free up memory
       head_ptr = tmp_ptr;
       head_ptr->prev = NULL;
       return;
     }
```

a for loop. If the list object is empty, it returns zero. It returns the total number of CLIST_ELEMENT-type objects in the singly linked circular list object.

Print_list(), a public member function of Circ_Linked_List class, is similar to print_list() of the Singly_Linked_List class in **Code 5.3** with the exception of the "wraparound" check.

Main(), a client function, instantiates clist_obj which is an object of the Circ_Linked_List class:

```
Circ_Linked_List clist_obj;
```

For a string "PRATIVA" pointed to by str, main() sends messages to the object clist_obj through this object's methods build_list(), delete_element(), add_after(), and print_list() for their associated actions. At the program exit, memory is cleaned up by delete str and the implicit call of ~Circ_Linked_List().

Code 5.5

```
//  Program:  cirlist.c++
//  Purpose:
/     Object-oriented implementation of a
/     singly linked circular list; assumed
/     'char' type data for each element.
//
#include <stdio.h>
typedef  int    BOOLEAN;
typedef  char   DATA_TYPE;
#include "bc_list.h"  // For base class "List"
#include "cirlist.h"  // For derived class
                       // "Circ_Linked_List"
// ~Circ_Linked_List():
//    Destroy the entire singly linked circular
//    list object. This destructor function is auto-
//    matically called before exiting program.
//
Circ_Linked_List::~Circ_Linked_List()
{
  LIST_PTR  tmp_ptr, next_ptr;

  if (!is_empty()) {
    // To stop 'for' loop, do "wrap around" check
    for (tmp_ptr = head_ptr;
         tmp_ptr->next != head_ptr; // "wrap around" check
         tmp_ptr = tmp_ptr->next) {
      next_ptr = tmp_ptr;
      delete  next_ptr;   // free up memory space
    }
    head_ptr = NULL;
  }
}
```

```
  if ((head_ptr != NULL) &&
      (head_ptr->data == search_key)) {
         elem_ptr = head_ptr->next;
         delete head_ptr;
         head_ptr = elem_ptr;
   }
   if ((elem_ptr = search_element1 (head_ptr,
                         search_key)) != NULL) {
     prev_ptr = search_previous (search_key);
     prev_ptr->next = elem_ptr->next;
     delete elem_ptr;
  }
}

// add_after():
//  Add a new element (given its 'given_data')
//  after a given element ('search_key') in a
//  singly linked circular list object.
//
void  Circ_Linked_List::add_after(
      DATA_TYPE search_key, DATA_TYPE given_data)
{
  LIST_PTR  new_ptr, elem_ptr;

  if ((elem_ptr = search_element1 (head_ptr,
                                   search_key))
         != NULL) {
    new_ptr = new  CLIST_ELEMENT;
    new_ptr->data  = given_data;
    new_ptr->next  = elem_ptr->next;
    elem_ptr->next = new_ptr;
  }
}

// get_elt_numbers():
//  Get the number of elements in a
//  singly linked circular list object.
//  (Iterative or nonrecursive method)
//
int Circ_Linked_List::get_elt_numbers(void)
{
  int element_numbers = 0;

  if (head_ptr == NULL)
    return (0);
  for (LIST_PTR  tmp_ptr = head_ptr;
       tmp_ptr->next != head_ptr;
       tmp_ptr = tmp_ptr->next)
    ++element_numbers;
  return (++element_numbers);
}

// print_list():
//  Print the elements of a singly linked
```

```
The singly linked circular list object is:
    P -> R -> A -> T -> I -> V -> A -> head
Number of elements in this list object is: 7

The singly linked circular list after deleting V is:
    P -> R -> A -> T -> I -> A -> head
Number of elements in this list object is: 6

The singly linked circular list after adding V after I is:
    P -> R -> A -> T -> I -> V -> A -> head
Number of elements in this list object is: 7
```

5.5.3 Doubly Linked Circular List and Its OOP Implementation

A doubly linked circular list is shown in Figure 5.14. Each element contains two pointer fields, `next` and `prev`, and a data field `data`. Any one element, say 'S', can be considered as the first element pointed to by `head_ptr` and by the pointer field `next` of the element 'I'. The element 'I' is pointed to by the pointer field `prev` of the element 'S'. The "circular connection" between the elements are defined in this manner. An OOP implementation of a doubly linked circular list is similar to that of a doubly linked list (see **Code 5.4**) with additions of logic for *circular connection*. It is left as an exercise.

5.6 Performance Analyses of List Operations

In this chapter, several implementations of the list object are discussed using: (1) array list, (2) singly linked list, (3) doubly linked list, (4) singly linked circular list, and (5) doubly linked circular list. Uses of these data structures depend on the applications and the operations required to perform in these applications. Thus, for the selection of an appropriate and efficient form of list data structure for a list, it is important to know at least

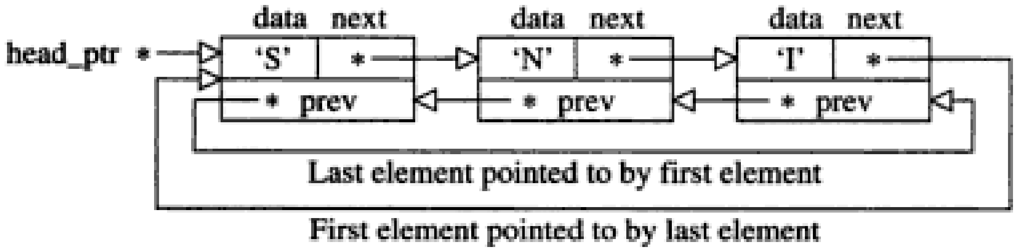


FIGURE 5.14. A doubly linked circular list.

```
 == Enter Polynomial Term ==
 Enter Coefficient and Power of X for each Term.
 Polynomial Terms will be automatically ordered.
 New coefficient will replace old for like powers of X.
 Zero coefficient will erase old for like powers of X.
<ENTER> with no Value at Coefficient Prompt when Finished
============================================================
Coefficient:  5
Power of X :  0

 OOP for Single Variable Polynomials
-------------------------------------

 Current Polynomial Object 2 is:

10.00x + 5.00

 == Enter Polynomial Term ==
 Enter Coefficient and Power of X for each Term.
 Polynomial Terms will be automatically ordered.
 New coefficient will replace old for like powers of X.
 Zero coefficient will erase old for like powers of X.
 <ENTER> with no Value at Coefficient Prompt when Finished
============================================================
Coefficient:  ENTER

 OOP for Single Variable Polynomials
-------------------------------------

Polynomial Object 1:

20.00x^2 + 10.00x

Polynomial Object 2:

10.00x + 5.00

Sum of Polynomial Object 1 and Polynomial Object 2:
20.00x^2 + 20.00x + 5.00

Press any key to continue!
```

5.8 OOP Application: Memory Management

As discussed in Chapter 2, two common programming errors associated with data structure implementations making use of dynamic memory allocation are *dangling pointers* and *memory leaks*.

allocated, it returns an overflow condition and the requesting process must go to sleep at this condition, and wake it up when available. Otherwise, `get_mem_block()` allocates a block using the *best fit* allocation policy, and returns its character-type pointer location. It expects two inputs, `block_size` and `block_type`. The algorithm for `get_mem_block()` is:

```
BEGIN
  ptr = head of MEMORY Structure;
  if (size > the maximum allowable block)
     return (address 0);
  while (not at end of the free list)
  BEGIN
    if (ptr->size just fits the block_size)
    BEGIN
       if (block is free or can be shared)
          return (location of block);
       else ptr = next element in list;
    END
    else ptr = next element in list;
  END
  If (reached this point)
     set overflow flag & return(address 0);
END
```

`Allocate_mem_block()` is a `public` member function of the class `Free_List`. It expects a pointer `pid` of type `PCB_LIST_PTR`, and an integer argument `block_type`. Its algorithm is:

```
allocate_mem_block(pid, block_type)

BEGIN
  Call get_mem_block() to allocate a block.
  If (return address 0) exit this procedure now.
  Allocate block for the process.
  Appropriately set characteristics for the allocated block.
  Increment process size according to allocated block.
  Update the MEMORY Structure and the PROCESS Structure.
  Return (the location of the allocate block).
END
```

`Release_mem_block()` is a `public` member function of the class `Free_List`. It expects a pointer `pid` of type `PCB_LIST_PTR`. Its algorithm is:

```
release_mem_block(pid)

BEGIN
  ptr = head of MEMORY Structure;
  while (ptr->size not equal to process memory limits)
          ptr = next element;

  Decrement reference counter of the block by 1;
```

This book provides a broad coverage of fundamental and advanced concepts of data structures and algorithms. Its aim is to provide readers with a modern synthesis of concepts with examples of applications that find practical use. Throughout, C++ is used to illustrate the construction and use of abstract data types and to demonstrate object-oriented implementations. As a result, it will make a superb textbook for students taking courses in data structures and software engineering as well as for software professionals. Readers are assumed to have basic working familiarity with C and C++, but it is otherwise self-contained.

The book is based on courses presented to undergraduate students both in engineering and computer science departments, and contains numerous classroom-tested examples and exercises. All the C++ codes presented are complete object-oriented programs and have been tested to run under both Borland International and AT&T C++ compilers. They are available on a disk at the back of the book. Amongst the many data types covered and implemented as objects are:

- arrays and strings
- lists
- stacks and queues
- trees and tries
- graphs and digraphs
- multidimensional search trees and search tries

ISBN 0-387-94194-0
EAN
9 780387 941943 >
ISBN 0-387-94194-0
ISBN 3-540-94194-0