Introduction to Stacks

2

THIS CHAPTER introduces the study of stacks, one of the simplest but most important of all data structures. The application of stacks to the reversal of data is illustrated with a program that calls on the standard-library stack implementation. A contiguous implementation of a stack data structure is then developed and used to implement a reverse Polish calculator and a bracket-checking program. The chapter closes with a discussion of the general principles of abstract data types and data structures.

2.1 STACK SPECIFICATIONS

2.1.1 Lists and Arrays

Soon after the introduction of loops and arrays, every elementary programming class attempts some programming exercise like the following:

> *Read an integer n, which will be at most 25, then read a list of n numbers, and print the list in reverse order.*

This simple exercise will probably cause difficulty for some students. Most will realize that they need to use an array, but some will attempt to set up the array to have n entries and will be confused by the error message resulting from attempting to use a variable rather than a constant to declare the size of the array. Other students will say, "I could solve the problem if I knew that there were 25 numbers, but I don't see how to handle fewer." Or "Tell me before I write the program how large n is, and then I can do it."

lists and arrays

The difficulties of these students come not from stupidity, but from thinking logically. A beginning course sometimes does not draw enough distinction between two quite different concepts. First is the concept of a ***list*** of n numbers, a list whose size is variable; that is, a list for which numbers can be inserted or deleted, so that, if $n = 3$, then the list contains only 3 numbers, and if $n = 19$, then it contains 19 numbers. Second is the programming feature called an ***array*** or a vector, which contains a constant number of positions, that is, whose size is fixed when the program is compiled. A list is a ***dynamic*** data structure because its size can change, while an array is a ***static*** data structure because it has a fixed size.

implementation

The concepts of a list and an array are, of course, related in that a list of variable size can be implemented in a computer as occupying part of an array of fixed size, with some of the entries in the array remaining unused. We shall later find, however, that there are several different ways to implement lists, and therefore we should not confuse implementation decisions with more fundamental decisions on choosing and specifying data structures.

2.1.2 Stacks

A ***stack*** is a version of a list that is particularly useful in applications involving reversing, such as the problem of Section 2.1.1. In a stack data structure, all insertions and deletions of entries are made at one end, called the ***top*** of the stack. A helpful analogy (see Figure 2.1) is to think of a stack of trays or of plates sitting on the counter in a busy cafeteria. Throughout the lunch hour, customers take trays off the top of the stack, and employees place returned trays back on top of the stack. The tray most recently put on the stack is the first one taken off. The bottom tray is the first one put on, and the last one to be used.

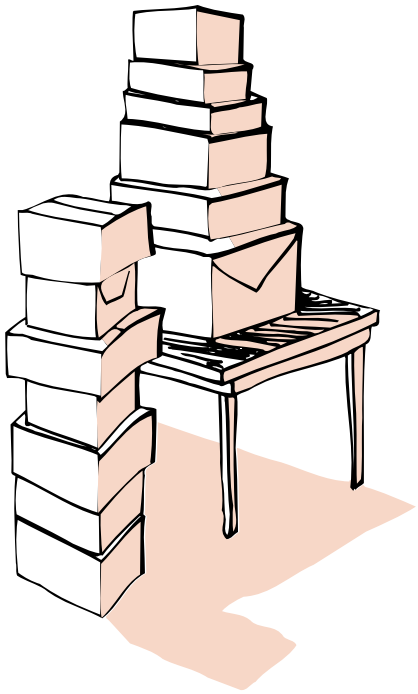


Figure 2.1. Stacks

Sometimes this picture is described with plates or trays on a spring-loaded device so that the top of the stack stays near the same height. This imagery is poor and should be avoided. If we were to implement a computer stack in this way, it would mean moving every item in the stack whenever one item was inserted or deleted. It is far better to think of the stack as resting on a firm counter or floor, so that only the top item is moved when it is added or deleted. The spring-loaded imagery, however, has contributed a pair of colorful words that are firmly embedded in computer jargon and that we shall use to name the fundamental operations on a stack. When we add an item to a stack, we say that we ***push*** it onto the stack, and when we remove an item, we say that we ***pop*** it from the stack. See Figure 2.2. Note that the last item pushed onto a stack is always the first that will be popped from the stack. This property is called ***last in, first out***, or *LIFO* for short.

push and pop

2.1.3 First Example: Reversing a List

As a simple example of the use of stacks, let us write a program to solve the problem of Section 2.1.1. Our program must read an integer n, followed by n floating-point numbers. It then writes them out in reverse order. We can accomplish this task by pushing each number onto a stack as it is read. When the input is finished, we pop numbers off the stack, and they will come off in the reverse order.

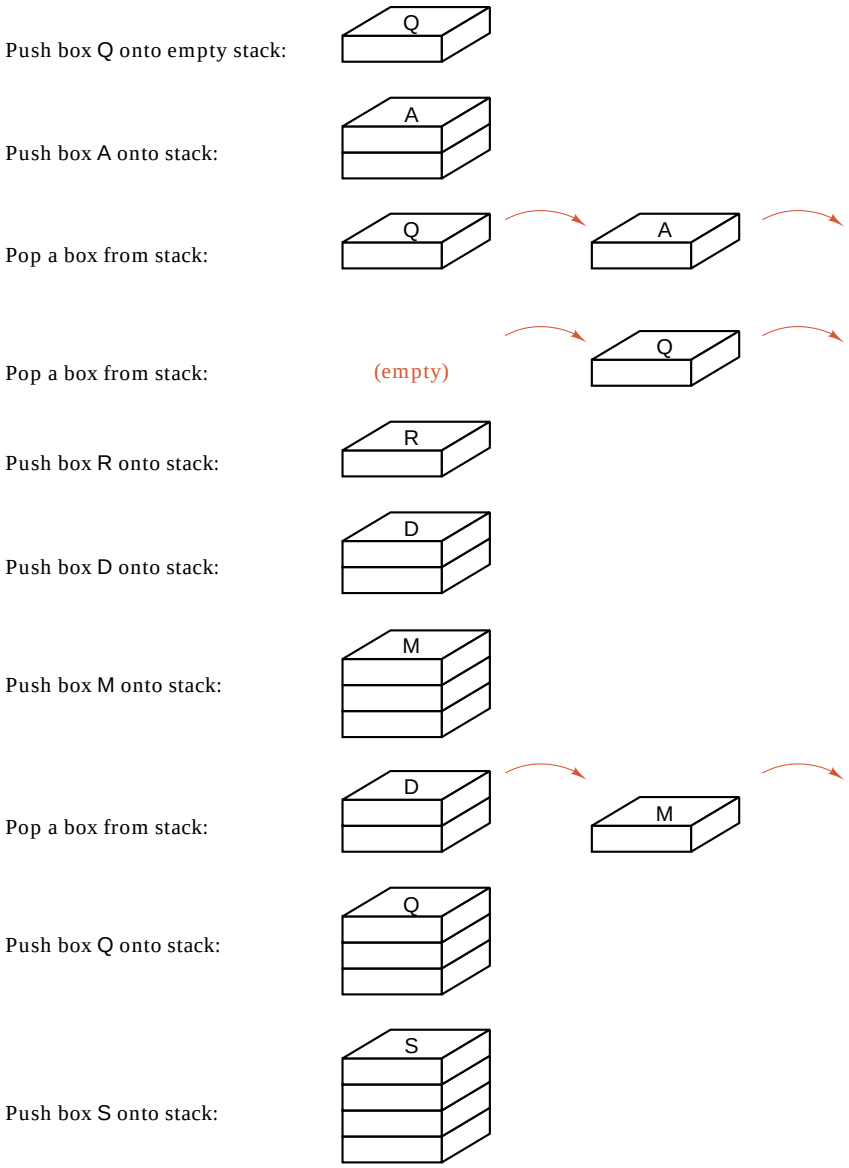


Figure 2.2. Pushing and popping a stack

standard template library

In our program we shall rely on the ***standard template library*** of C++ (usually called the ***STL***) to provide a **class** that implements stacks.[1] The STL is part of the standard library of C++. This standard library contains all kinds of useful information, functions, and classes. The STL is the part of the standard library that

[1] If the STL stack implementation is not available, the stack class that we implement in the next section can be used in its place.

provides convenient implementations for many common data structures, including almost all the data structures we shall study in this book.

We can include the STL stack implementation into our programs with the directive #include <stack> (or, on some older, pre-ANSI compilers, the directive #include <stack.h>). Once the library is included, we can define initially empty stack objects, and apply methods called push, pop, top, and empty. We will discuss these methods and the STL itself in more detail later, but its application in the following program is quite straightforward.

```
#include <stack>

int main()
/* Pre:   The user supplies an integer n and n decimal numbers.
   Post:  The numbers are printed in reverse order.
   Uses:  The STL class stack and its methods */
{
   int n;
   double item;
   stack<double> numbers;    // declares and initializes a stack of numbers
   cout  << " Type in an integer n followed by n decimal numbers." << endl
         << " The numbers will be printed in reverse order." << endl;
   cin    >> n;
   for (int i = 0;  i < n;  i++) {
      cin >> item;
      numbers.push(item);
   }
   cout << endl << endl;
   while (!numbers.empty()) {
      cout << numbers.top() << " ";
      numbers.pop();
   }
   cout << endl;
}
```

initialization

In this number-reversing program, we have used not only the methods push(), top(), and pop() of the stack called numbers, but we have made crucial use of the implicit initialization of numbers as an empty stack. That is, when the stack called numbers is created, it is automatically initialized to be empty. Just as with the standard-library classes, whenever we construct a class we shall be careful to ensure that it is automatically initialized, in contrast to variables and arrays, whose initialization must be given explicitly.

capitalization

We remark that, like the atomic classes **int**, **float**, and so on, the C++ library **class** stack has an identifier that begins with a lowercase letter. As we decided in Section 1.3, however, the classes that we shall create will have identifiers with an initial capital letter.

Contents
Index
Help

One important feature of the STL stack implementation is that the user can specify the type of entries to be held in a particular stack. For example, in the reversing program, we create a stack of elements of type **double** with the definition stack<**double**> numbers, whereas, if we had required a stack of integers, we would have declared stack<**int**> numbers. The standard library uses a C++ construction known as a *template* to achieve this flexibility. Once we are familiar with more basic implementations of data structures, we shall practice the construction and use of our own templates, starting in Chapter 6.

template

2.1.4 Information Hiding

We have been able to write our program for reversing a line of input without any consideration of how a stack is actually implemented. In this way, we have an example of ***information hiding***: The methods for handling stacks are implemented in the C++ standard library, and we can use them without needing to know the details of how stacks are kept in memory or of how the stack operations are actually done.

As a matter of fact, we have already been practicing information hiding in the programs we have previously written, without thinking about it. Whenever we have written a program using an array or a structure, we have been content to use the operations on these structures without considering how the C++ compiler actually represents them in terms of bits or bytes in the computer memory or the machine-language steps it follows to look up an index or select a member.

built-in structures

One important difference between practicing information hiding with regard to arrays and practicing information hiding with regard to stacks is that C++ provides just one built-in implementation of arrays, but the STL has several implementations of stacks. Although the code in a client program that uses stacks should not depend on a particular choice of stack implementation, the performance of the final program may very much depend on the choice of implementation. In order to make an informed decision about which stack implementation should be used in a given application, we need to appreciate the different features and behaviors of the different implementations. In the coming chapters, we shall see that for stacks (as for almost all the data types we shall study) there are several different ways to represent the data in the computer memory, and there are several different ways to do the operations. In some applications, one implementation is better, while in other applications another implementation proves superior.

alternative implementations

Even in a single large program, we may first decide to represent stacks one way and then, as we gain experience with the program, we may decide that another way is better. If the instructions for manipulating a stack have been written out every time a stack is used, then every occurrence of these instructions will need to be changed. If we have practiced information hiding by using separate functions for manipulating stacks, then only the declarations will need to be changed.

change of implementation

Another advantage of information hiding shows up in programs that use stacks where the very appearance of the words *push* and *pop* immediately alert a person reading the program to what is being done, whereas the instructions themselves

clarity of program

might be more obscure. We shall find that separating the use of data structures from their implementation will help us improve the top-down design of both our data structures and our programs.

top-down design

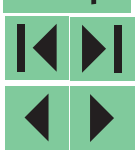
2.1.5 The Standard Template Library

The standard C++ library is available in implementations of ANSI C++. This library provides all kinds of system-dependent information, such as the maximum exponent that can be stored in a floating-point type, input and output facilities, and other functions whose optimal implementation depends on the system. In addition, the standard library provides an extensive set of data structures and their methods for use in writing programs. In fact, the standard library contains implementations of almost all the data structures that we consider in this text, including stacks, queues, deques, lists, strings, and sets, among others.

library of data structures

To be able to use these library implementations appropriately and efficiently, it is essential that we learn the principles and the alternative implementations of the data structures represented in the standard library. We shall therefore give only a very brief introduction to the standard library, and then we return to our main goal, the study of the data structures themselves. In one sense, however, most of this book can be regarded as an introduction to the STL of C++, since our goal is to learn the basic principles and methods of data structures, knowledge that is essential to the discerning use of the STL.

Contents
Index
Help



As we have already noted, the STL stack implementation is a class template, and therefore a programmer can choose exactly what sort of items will be placed in a stack, by specifying its template parameters between < > symbols. In fact, a programmer can also utilize a second template parameter to control what sort of stack implementation will be used. This second parameter has a default value, so that a programmer who is unsure of which implementation to use will get a stack constructed from a default implementation; in fact, it will come from a deque—a data structure that we will introduce in Chapter 3. A programmer can choose instead to use a vector-based or a list-based implementation of a stack. In order to choose among these implementations wisely, a programmer needs to understand their relative advantages, and this understanding can only come from the sort of general study of data structures that we undertake in this book.

template parameter

alternative implementations

Regardless of the chosen implementation, however, the STL does guarantee that stack methods will be performed efficiently, operating in constant time, independent of the size of the stack. In Chapter 7, we shall begin a systematic study of the time used by various algorithms, and we shall continue this study in later chapters. As it happens, the constant-time operation of standard stack methods is guaranteed only in an averaged sense known as *amortized* performance. We shall study the amortized analysis of programs in Section 10.5.

algorithm performance

The STL provides implementations of many other standard data structures, and, as we progress through this book, we shall note those implementations that correspond to topics under discussion. In general, these library implementations are highly efficient, convenient, and designed with enough default options to allow programmers to use them easily.

Exercises 2.1

IRM

E1. Draw a sequence of stack frames like Figure 2.2 showing the progress of each of the following segments of code, each beginning with an empty stack s. Assume the declarations

```
#include <stack>
stack<char> s;
char x, y, z;
```

(a)
```
s.push('a');
s.push('b');
s.push('c');
s.pop();
s.pop();
s.pop();
```

(b)
```
s.push('a');
s.push('b');
s.push('c');
x = s.top();
s.pop();
y = s.top();
s.pop();
s.push(x);
s.push(y);
s.pop();
```

(c)
```
s.push('a');
s.push('b');
s.push('c');
while (!s.empty())
   s.pop();
```

(d)
```
s.push('a');
s.push('b');
while (!s.empty()) {
   x = s.top();
   s.pop();
}
s.push('c');
s.pop();
s.push('a');
s.pop();
s.push('b');
s.pop();
```

E2. Write a program that makes use of a stack to read in a single line of text and write out the characters in the line in reverse order.

E3. Write a program that reads a sequence of integers of increasing size and prints the integers in decreasing order of size. Input terminates as soon as an integer that does not exceed its predecessor is read. The integers are then printed in decreasing order.

E4. A stack may be regarded as a railway switching network like the one in Figure 2.3. Cars numbered 1, 2, ..., n are on the line at the left, and it is desired to rearrange (permute) the cars as they leave on the right-hand track. A car that is on the spur (stack) can be left there or sent on its way down the right track, but it can never be sent back to the incoming track. For example, if $n = 3$, and we have the cars 1, 2, 3 on the left track, then 3 first goes to the spur. We could then send 2 to the spur, then on its way to the right, then send 3 on the way, then 1, obtaining the new order 1, 3, 2.

stack permutations

(a) For $n = 3$, find all possible permutations that can be obtained.

(b) For $n = 4$, find all possible permutations that can be obtained.

(c) [Challenging] For general n, find how many permutations can be obtained by using this stack.

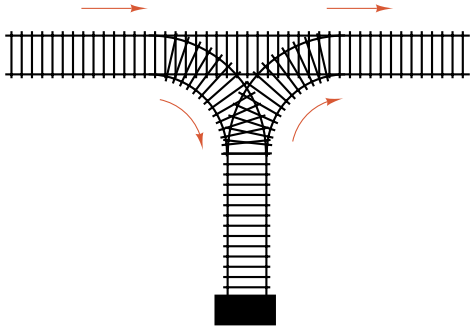


Figure 2.3. Switching network for stack permutations

2.2 IMPLEMENTATION OF STACKS

contiguous implementation

We now turn to the problem of the construction of a stack implementation in C++. We will produce a ***contiguous*** Stack implementation, meaning that the entries are stored next to each other in an array. In Chapter 4, we shall study a *linked* implementation using pointers in dynamic memory.

classes

In these and all the other implementations we construct, we shall be careful always to use classes to implement the data structures. Thus, we shall now develop a **class** Stack whose data members represent the entries of a stack. Before we implement any class, we should decide on specifications for its methods.

Contents
Index
Help

2.2.1 Specification of Methods for Stacks

stack methods

The methods of our **class** Stack must certainly include the fundamental operations called empty(), top(), push(), and pop(). Only one other operation will be essential: This is an *initialization* operation to set up an empty stack. Without such an initialization operation, client code would have to deal with stacks made up of random and probably illegal data, whatever happened beforehand to be in the storage area occupied by the stack.

1. Constructors

The C++ language allows us to define special initialization methods for any class. These methods are called ***constructors*** for the class. Each constructor is a function with the same name as the corresponding class. A constructor has no return type. Constructors are applied automatically whenever we declare an object of the class. For example, the standard library implementation of a stack includes a constructor that initializes each newly created stack as empty: In our earlier program for reversing a line of input, such an initialization was crucial. Naturally, we shall create a similar Stack constructor for the **class** that we develop. Thus, whenever one of our clients declares a Stack object, that object is automatically initialized as empty. The specification of our Stack constructor follows.

initialization

Stack::Stack();
precondition: None.
postcondition: The Stack exists and is initialized to be empty.

2. Entry Types, Generics

entry type

The declarations for the fundamental methods of a stack depend on the type of entries that we intend to store in the stack. To keep as much generality as we can, let us use Stack_entry for the type of entries in our Stack. For one application, Stack_entry might be **int**, for another it might be **char**. A client can select an appropriate entry type with a definition such as

```
typedef char Stack_entry;
```

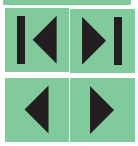
By keeping the type Stack_entry general, we can use the same stack implementation for many different applications.

generics

The ability to use the same underlying data structure and operations for different entry types is called ***generics***. Our use of a **typedef** statement to choose the type of entry in our Stack is a simple way to achieve generic data structures in C++. For complex applications, ones that need stacks with different entry types in a single program, the more sophisticated **template** treatment, which is used in the standard library **class** stack, is more appropriate. After we have gained some experience with simple data structures, we shall also choose to work with templates, beginning with the programs in Chapter 6.

templates

Contents
Index
Help



3. Error Processing

In deciding on the parameters and return types of the fundamental Stack methods, we must recognize that a method might be applied illegally by a client. For example, a client might try to pop an empty stack. Our methods will signal any such problems with diagnostic ***error codes***. In this book, we shall use a single enumerated type called Error_code to report errors from all of our programs and functions.

error codes

The enumerated type Error_code will be part of our utility package, described in Appendix C. In implementing the Stack methods, we shall make use of three values of an Error_code, namely:

success, overflow, underflow

If a method is able to complete its work normally, it will return success as its Error_code; otherwise, it will return a code to indicate what went wrong. Thus, a client that tries to pop from an empty Stack will get back an Error_code of underflow. However, any other application of the pop method is legitimate, and it will result in an Error_code of success.

stack error codes

error handling

This provides us with a first example of *error handling*, an important safeguard that we should build into our data structures whenever possible. There are several different ways that we could decide to handle error conditions that are detected in a method of a data structure. We could decide to handle the error directly, by printing

out an error message or by halting the execution of the program. Alternatively, since methods are always called from a client program, we can decide to return an error code back to the client and let it decide how to handle the error. We take the view that the client is in the best position to judge what to do when errors are detected; we therefore adopt the second course of action. In some cases, the client code might react to an error code by ceasing operation immediately, but in other cases it might be important to ignore the error condition.

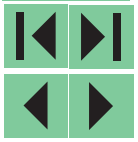
Programming Precept

After a client uses a class method,
it should decide whether to check the resulting error status.
Classes should be designed to allow clients to decide
how to respond to errors.

exception handling

We remark that C++ does provide a more sophisticated technique known as ***exception handling***: When an error is detected an exception can be thrown. This exception can then be caught by client code. In this way, exception handling conforms to our philosophy that the client should decide how to respond to errors detected in a data structure. The standard library implementations of stacks and other classes use exception handling to deal with error conditions. However, we shall opt instead for the simplicity of returning error codes in all our implementations in this text.

Contents
Index
Help



4. Specification for Methods

Stack *methods*

Our specifications for the fundamental methods of a Stack come next.

```
Error_code Stack :: pop();
```

precondition: None.

postcondition: If the Stack is not empty, the top of the Stack is removed. If the Stack is empty, an Error_code of underflow is returned and the Stack is left unchanged.


```
Error_code Stack :: push(const Stack_entry &item);
```

precondition: None.

postcondition: If the Stack is not full, item is added to the top of the Stack. If the Stack is full, an Error_code of overflow is returned and the Stack is left unchanged.

The parameter item that is passed to push is an input parameter, and this is indicated by its declaration as a **const** reference. In contrast, the parameter for the next method, top, is an output parameter, which we implement with call by reference.

Error_code Stack :: top(Stack_entry &item) **const**;

precondition: None.

postcondition: The top of a nonempty Stack is copied to item. A code of fail is returned if the Stack is empty.

The modifier **const** that we have appended to the declaration of this method indicates that the corresponding Stack object is not altered by, or during, the method. Just as it is important to specify input parameters as constant, as information for the reader and the compiler, it is important for us to indicate constant methods with this modifier. The last Stack method, empty, should also be declared as a constant method.

bool Stack :: empty( ) **const**;

precondition: None.

postcondition: A result of **true** is returned if the Stack is empty, otherwise **false** is returned.

Contents
Index
Help

2.2.2 The Class Specification

stack type

For a contiguous Stack implementation, we shall set up an array that will hold the entries in the stack and a counter that will indicate how many entries there are. We collect these data members together with the methods in the following definition for a **class** Stack containing items of type Stack_entry. This definition constitutes the file `stack.h`.

```
const int maxstack = 10;          //  small value for testing

class Stack {
public:
   Stack( );
   bool empty( ) const;
   Error_code pop( );
   Error_code top(Stack_entry &item) const;
   Error_code push(const Stack_entry &item);
private:
   int count;
   Stack_entry entry[maxstack];
};
```

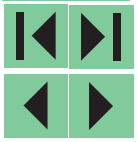
As we explained in Section 1.2.4, we shall place this class definition in a header file with extension `.h`, in this case the file `stack.h`. The corresponding code file, with the method implementations that we shall next develop, will be called `stack.c`. The code file can then be compiled separately and linked to client code as needed.

2.2.3 Pushing, Popping, and Other Methods

The stack methods are implemented as follows. We must be careful of the extreme cases: We might attempt to pop an entry from an empty stack or to push an entry onto a full stack. These conditions must be recognized and reported with the return of an error code.

```
Error_code Stack::push(const Stack_entry &item)
/* Pre:  None.
   Post: If the Stack is not full, item is added to the top of the Stack. If the Stack
         is full, an Error_code of overflow is returned and the Stack is left un-
         changed. */
{
  Error_code outcome = success;
  if (count >= maxstack)
    outcome = overflow;
  else
    entry[count++] = item;
  return outcome;
}
```

Contents
Index
Help



```
Error_code Stack::pop()
/* Pre:  None.
   Post: If the Stack is not empty, the top of the Stack is removed. If the Stack is
         empty, an Error_code of underflow is returned. */
{
  Error_code outcome = success;
  if (count == 0)
    outcome = underflow;
  else --count;
  return outcome;
}
```

We note that the data member count represents the number of items in a Stack. Therefore, the top of a Stack occupies entry[count − 1], as shown in Figure 2.4.

```
Error_code Stack::top(Stack_entry &item) const
/* Pre:  None.
   Post: If the Stack is not empty, the top of the Stack is returned in item. If the
         Stack is empty an Error_code of underflow is returned. */
{
  Error_code outcome = success;
  if (count == 0)
    outcome = underflow;
  else
    item = entry[count − 1];
  return outcome;
}
```

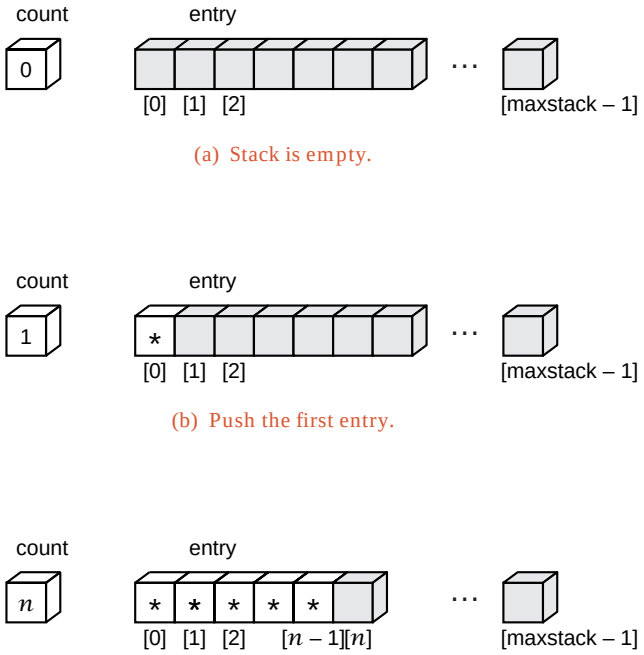


Figure 2.4. Representation of data in a contiguous stack

```
bool Stack :: empty( ) const
/* Pre:   None.
   Post:  If the Stack is empty, true is returned. Otherwise false is returned. */
{
   bool outcome = true;
   if (count > 0) outcome = false;
   return outcome;
}
```

constructor

The other method of our Stack is the ***constructor***. The purpose of the constructor is to initialize any new Stack object as empty.

```
Stack :: Stack( )
/* Pre:   None.
   Post:  The stack is initialized to be empty. */
{
   count = 0;
}
```

2.2.4 Encapsulation

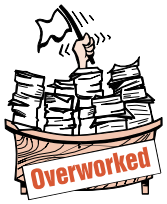
data integrity

Notice that our stack implementation forces client code to make use of information hiding. Our declaration of **private** visibility for the data makes it is impossible for a client to access the data stored in a Stack except by using the official methods push(), pop(), and top(). One important result of this data privacy is that a Stack can never contain illegal or corrupted data. Every Stack object will be initialized to represent a legitimate empty stack and can only be modified by the official Stack methods. So long as our methods are correctly implemented, we have a guarantee that correctly initialized objects must continue to stay free of any data corruption.

encapsulation

We summarize this protection that we have given our Stack objects by saying that they are ***encapsulated***. In general, data is said to be encapsulated if it can only be accessed by a controlled set of functions.

The small extra effort that we make to encapsulate the data members of a C++ class pays big dividends. The first advantage of using an encapsulated class shows up when we specify and program the methods: For an encapsulated class, we need never worry about illegal data values. Without encapsulation, the operations on a data structure almost always depend on a precondition that the data members have been correctly initialized and have not been corrupted. We can and should use encapsulation to avoid such preconditions. For our encapsulated **class** Stack, all of the methods have precondition specifications of *None*. This means that a client does not need to check for any special situations, such as an uninitialized stack, before applying a public Stack method. Since we think of data structures as services that will be written once and used in many different applications, it is particularly appropriate that the clients should be spared extra work where possible.



Contents
Index
Help

Programming Precept

The public methods for a data structure should be implemented without preconditions. The data members should be kept private.

We shall omit the precondition section from public method specifications in all our encapsulated C++ classes.

The private member functions of a data structure cannot be used by clients, so there is no longer a strong case for writing these functions without preconditions. We shall emphasize the distinction between public and private member functions of a data structure, by reserving the term *method* for the former category.

Exercises 2.2

IRM

E1. Assume the following definition file for a contiguous implementation of an extended stack data structure.

```
class Extended_stack {
public:
   Extended_stack( );
   Error_code pop( );
   Error_code push(const Stack_entry &item);
   Error_code top(Stack_entry &item) const;
   bool empty( ) const;
   void clear( );              // Reset the stack to be empty.
   bool full( ) const ;        // If the stack is full, return true; else return false.
   int size( ) const;          // Return the number of entries in the stack.
private:
   int count;
   Stack_entry entry[maxstack];
};
```

Write code for the following methods. [Use the private data members in your code.]

(a) clear **(b)** full **(c)** size

E2. Start with the stack methods, and write a function copy_stack with the following specifications:

> Error_code copy_stack(Stack &dest, Stack &source);
>
> *precondition*: None.
>
> *postcondition*: Stack dest has become an exact copy of Stack source; source is unchanged. If an error is detected, an appropriate code is returned; otherwise, a code of success is returned.

Write three versions of your function:

(a) Simply use an assignment statement: dest = source;

(b) Use the Stack methods and a temporary Stack to retrieve entries from the Stack source and add each entry to the Stack dest and restore the Stack source.

(c) Write the function as a friend[2] to the **class** Stack. Use the private data members of the Stack and write a loop that copies entries from source to dest.

[2] Friend functions have access to all members of a C++ class, even private ones.

Which of these is easiest to write? Which will run most quickly if the stack is nearly full? Which will run most quickly if the stack is nearly empty? Which would be the best method if the implementation might be changed? In which could we pass the parameter source as a constant reference?

E3. Write code for the following functions. [Your code must use Stack methods, but you should not make any assumptions about how stacks or their methods are actually implemented. For some functions, you may wish to declare and use a second, temporary Stack object.]

(a) Function **bool** full(Stack &s) leaves the Stack s unchanged and returns a value of **true** or **false** according to whether the Stack s is or is not full.

(b) Function Error_code pop_top(Stack &s, Stack_entry &t) removes the top entry from the Stack s and returns its value as the output parameter t.

(c) Function **void** clear(Stack &s) deletes all entries and returns s as an empty Stack.

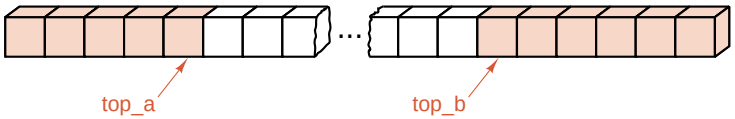
(d) Function **int** size(Stack &s) leaves the Stack s unchanged and returns a count of the number of entries in the Stack.

(e) Function **void** delete_all(Stack &s, Stack_entry x) deletes all occurrences (if any) of x from s and leaves the remaining entries in s in the same relative order.

Contents
Index
Help

two coexisting stacks

E4. Sometimes a program requires two stacks containing the same type of entries. If the two stacks are stored in separate arrays, then one stack might overflow while there was considerable unused space in the other. A neat way to avoid this problem is to put all the space in one array and let one stack grow from one end of the array and the other stack start at the other end and grow in the opposite direction, toward the first stack. In this way, if one stack turns out to be large and the other small, then they will still both fit, and there will be no overflow until all the space is actually used. Define a new class Double_stack that includes (as private data members) the array and the two indices top_a and top_b, and write code for the methods Double_stack( ), push_a( ), push_b( ), pop_a( ), and pop_b( ) to handle the two stacks within one Double_stack.



Programming Projects 2.2

P1. Assemble the appropriate declarations from the text into the files `stack.h` and `stack.c` and verify that `stack.c` compiles correctly, so that the **class** Stack can be used by future client programs.

P2. Write a program that uses a Stack to read an integer and print all its prime divisors in descending order. For example, with the integer 2100 the output should be

prime divisors

7 5 5 3 2 2.

[*Hint*: The smallest divisor greater than 1 of any integer is guaranteed to be a prime.]

2.3 APPLICATION: A DESK CALCULATOR

reverse Polish calculations

This section outlines a program to imitate the behavior of a simple calculator that does addition, subtraction, multiplication, division, and perhaps some other operations. There are many kinds of calculators available, and we could model our program after any of them. To provide a further illustration of the use of stacks, however, let us choose to model what is often called a ***reverse Polish*** calculator. In such a calculator, the operands (numbers, usually) are entered *before* an operation is specified. The operands are pushed onto a stack. When an operation is performed, it pops its operands from the stack and pushes its result back onto the stack.

examples

We shall write ? to denote an instruction to read an operand and push it onto the stack; + , −, * , and / represent arithmetic operations; and = is an instruction to print the top of the stack (but not pop it off). Further, we write a, b, c, and d to denote numerical values such as 3.14 or −7. The instructions ? a ? b + = mean read and store the numbers a and b, calculate and store their sum, and then print the sum. The instructions ? a ? b + ? c ? d + * = request four numerical operands, and the result printed is the value of (a + b) * (c + d). Similarly, the instructions ? a ? b ? c − = * ? d + = mean push the numbers a, b, c onto the stack, replace the pair b, c by b − c and print its value, calculate a * (b − c), push d onto the stack, and finally calculate and print (a * (b − c)) + d. The advantage of a reverse Polish calculator is that any expression, no matter how complicated, can be specified without the use of parentheses.

no parentheses needed

If you have access to a UNIX system, you can experiment with a reverse Polish calculator with the command dc.

Polish notation is useful for compilers as well as for calculators, and its study forms the major topic of Chapter 13. For the present, however, a few minutes' practice with a reverse Polish calculator will make you quite comfortable with its use.

It is clear that we should use a stack in an implementation of a reverse Polish calculator. After this decision, the task of the calculator program becomes simple. The main program declares a stack of entries of type **double**, accepts new commands, and performs them as long as desired.

In the program, we shall apply our generic Stack implementation. We begin with a **typedef** statement to set the type of Stack_entry. We then include the Stack definition file `stack.h`.

```
typedef double Stack_entry;
#include "stack.h"

int main()
/* Post: The program has executed simple arithmetic commands entered by the
         user.
   Uses: The class Stack and the functions introduction, instructions, do_command,
         and get_command. */
```

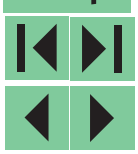
```
{
  Stack stored_numbers;
  introduction();
  instructions();
  while (do_command(get_command(), stored_numbers));
}
```

The auxiliary function get_command obtains a command from the user, checking that it is valid and converting it to lowercase by using the string function tolower() that is declared in the standard header file cctype. (The file cctype, or its older incarnation ctype.h, can be automatically included via our standard utility package; see Appendix C.)

user commands

In order to implement get_command, let us make the decision to represent the commands that a user can type by the characters ?, =, +, −, *, /, where ? requests input of a numerical value from the user, = prints the result of an operation, and the remaining symbols denote addition, subtraction, multiplication, and division, respectively.

Contents
Index
Help



```
char get_command()
{
  char command;
  bool waiting = true;
  cout ≪ "Select command and press < Enter > :";
  while (waiting) {
    cin ≫ command;
    command = tolower(command);
    if (command == '?' || command == '=' || command == '+' ||
        command == '−' || command == '*' || command == '/' ||
        command == 'q') waiting = false;
    else {
      cout ≪ "Please enter a valid command:" ≪ endl
           ≪ "[?]push to stack  [=]print top" ≪ endl
           ≪ "[+] [−] [*] [/]  are arithmetic operations" ≪ endl
           ≪ "[Q]uit." ≪ endl;
    }
  }
  return command;
}
```

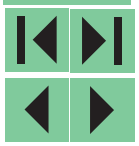
The work of selecting and performing the commands, finally, is the task of the function do_command. We present here an abbreviated form of the function do_command, in which we have coded only a few of the possible commands in its main switch statement.

Do a user command


```
bool do_command(char command, Stack &numbers)
/* Pre:  The first parameter specifies a valid calculator command.
   Post: The command specified by the first parameter has been applied to the
         Stack of numbers given by the second parameter.  A result of true is re-
         turned unless command == 'q'.
   Uses: The class Stack. */
{
  double p, q;
  switch (command) {
  case '?':
read  cout << "Enter a real number: " << flush;
    cin >> p;
    if (numbers.push(p) == overflow)
      cout << "Warning: Stack full, lost number" << endl;
    break;
  case '=':
print if (numbers.top(p) == underflow)
      cout << "Stack empty" << endl;
    else
      cout << p << endl;
    break;
  case '+':
add if (numbers.top(p) == underflow)
      cout << "Stack empty" << endl;
    else {
      numbers.pop();
      if (numbers.top(q) == underflow) {
        cout << "Stack has just one entry" << endl;
        numbers.push(p);
      }
      else {
        numbers.pop();
        if (numbers.push(q + p) == overflow)
          cout << "Warning: Stack full, lost result" << endl;
      }
    }
    break;
  // Add options for further user commands.

  case 'q':
quit  cout << "Calculation finished.\n";
    return false;
  }
  return true;
}
```

Contents
Index
Help



In calling this function, we must pass the Stack parameter by reference, because its value might need to be modified. For example, if the command parameter is +, then we normally pop two values off the Stack numbers and push their sum back onto it: This should certainly change the Stack.

The function do_command allows for an additional user command, q, that quits the program.

Exercises 2.3

IRM

E1. If we use the standard library **class** stack in our calculator, the method top() returns the top entry off the stack as its result. Then the function do_command can then be shortened considerably by writing such statements as

```
case '-': numbers.push(numbers.pop() - numbers.pop());
```

(a) Assuming that this statement works correctly, explain why it would still be bad programming style.

(b) It is possible that two different C++ compilers, both adhering strictly to standard C++, would translate this statement in ways that would give different answers when the program runs. Explain how this could happen.

E2. Discuss the steps that would be needed to make the calculator process complex numbers.

Programming Projects 2.3

P1. Assemble the functions developed in this section and make the necessary changes in the code so as to produce a working calculator program.

P2. Write a function that will interchange the top two numbers on the stack, and include this capability as a new command.

P3. Write a function that will add all the numbers on the stack together, and include this capability as a new command.

P4. Write a function that will compute the average of all numbers on the stack, and include this capability as a new command.

2.4 APPLICATION: BRACKET MATCHING

CASE Study

Programs written in C++ contain several different types of brackets. For example, brackets are used to enclose expressions, function arguments, array indices, and blocks of code. As we know, the brackets used within a program must pair off.

For example, the following string

```
{a = (1 + v(b[3 + c[4]]));
```

cannot possibly have matched brackets, because it has five opening brackets and only four closing brackets: Like the first drafts of many C++ programs, it is missing a final brace. The string

```
{ a = (b[0) + 1]; }
```

has equal numbers of opening and closing brackets, but we can see that it has unmatched brackets, because its first closing bracket) does not correspond to the most recent opening bracket [. On the other hand, the bracket sequence

```
{()[()]}
```

is matched, although it is not a legitimate part of any C++ program.

specifications

In this section we shall implement a program to check that brackets are correctly matched in an input text file. For simplicity, we will limit our attention to the brackets {, }, (,), [, and]. Moreover, we shall just read a single line of characters, and ignore all input other than bracket characters. In checking the bracketing of an actual C++ program, we would need to apply special rules for brackets within comments and strings, and we would have to recognize that the symbols <, > can also denote brackets (for example, in the declaration stack<**double**> numbers; that we used in the program of Section 2.1.3).

Contents
Index
Help

algorithm

If we formalize the rules for pairing brackets, we quickly obtain the following algorithm: Read the program file character by character. Each opening bracket (, [, or { that is encountered is considered as unmatched and is stored until a matching bracket can be found. Any closing bracket),], or } must correspond, in bracket style, to the last unmatched opening bracket, which should now be retrieved and removed from storage. Finally, at the end of the program, we must check that no unmatched opening brackets are left over.

data structure: stack

We see that a program to test the matching of brackets needs to process an input file character by character, and, as it works its way through the input, it needs some way to remember any currently unmatched brackets. These brackets must be retrieved in the exact reverse of their input order, and therefore a Stack provides an attractive option for their storage.

Once we have made this decision, our program need only loop over the input characters, until either a bracketing error is detected or the input file ends. Whenever a bracket is found, an appropriate Stack operation is applied. We thus obtain the following program.


```
int main()
/* Post: The program has notified the user of any bracket mismatch in the standard
         input file.
   Uses: The class Stack. */
```

```
{
   Stack openings;
   char symbol;
   bool is_matched = true;
   while (is_matched && (symbol = cin.get()) != '\n') {
      if (symbol == '{' || symbol == '(' || symbol == '[')
         openings.push(symbol);
      if (symbol == '}' || symbol == ')' || symbol == ']') {
         if (openings.empty()) {
            cout << "Unmatched closing bracket " << symbol
                 << " detected." << endl;
            is_matched = false;
         }
         else {
            char match;
            openings.top(match);
            openings.pop();
            is_matched = (symbol == '}' && match == '{')
                      || (symbol == ')' && match == '(')
                      || (symbol == ']' && match == '[');
            if (!is_matched)
               cout << "Bad match " << match << symbol << endl;
         }
      }
   }
   if (!openings.empty())
      cout << "Unmatched opening bracket(s) detected." << endl;
}
```

Contents
Index
Help

Programming Projects 2.4

P1. Modify the bracket checking program so that it reads the whole of an input file.

P2. Modify the bracket checking program so that input characters are echoed to output, and individual unmatched closing brackets are identified in the output file.

P3. Incorporate C++ comments and character strings into the bracket checking program, so that any bracket within a comment or character string is ignored.

2.5 ABSTRACT DATA TYPES AND THEIR IMPLEMENTATIONS

2.5.1 Introduction

In any of our applications of stacks, we could have used an array and counter in place of the stack. This would entail replacing each stack operation by a group

of array and counter manipulations. For example, the bracket checking program might use statements such as:

```
if (counter < max) {
   openings[counter] = symbol;
   counter++;
}
```

In some ways, this may seem like an easy approach, since the code is straightforward, simpler in many ways than setting up a class and declaring all its methods.

A major drawback to this approach, however, is that the writer (and any reader) of the program must spend considerable effort verifying the details of array index manipulations every time the stack is used, rather than being able to concentrate on the ways in which the stack is actually being used. This unnecessary effort is a direct result of the programmer's failure to recognize the general concept of a stack and to distinguish between this general concept and the particular implementation needed for a given application.

Another application might include the following instructions instead of a simple stack operation:

```
if ((xxt ==  mxx) || (xxt > mxx))
    try_again();
else {
    xx[xxt] = wi;
    xxt++;
}
```

In isolation, it may not even be clear that this section of code has essentially the same function as the earlier one. Both segments are intended to push an item onto the top of a stack.

Researchers working in different subjects frequently have ideas that are fundamentally similar but are developed for different purposes and expressed in different language. Often years will pass before anyone realizes the similarity of the work, but when the observation is made, insight from one subject can help with the other. In computer science, even so, the same basic idea often appears in quite different disguises that obscure the similarity. But if we can discover and emphasize the similarities, then we may be able to generalize the ideas and obtain easier ways to meet the requirements of many applications.

analogies

The way in which an underlying structure is implemented can have substantial effects on program development and on the capabilities and usefulness of the result. Sometimes these effects can be subtle. The underlying mathematical concept of a real number, for example, is usually (but not always) implemented by computer as a floating-point number with a certain degree of precision, and the inherent limitations in this implementation often produce difficulties with round-off error. Drawing a clear separation between the logical structure of our data and its implementation in computer memory will help us in designing programs. Our first step is to recognize the logical connections among the data and embody these con-

implementation

nections in a logical data structure. Later we can consider our data structures and decide what is the best way to implement them for efficiency of programming and execution. By separating these decisions they both become easier, and we avoid pitfalls that attend premature commitment.

To help us clarify this distinction and achieve greater generality, let us now consider data structures from as general a perspective as we can.

2.5.2 General Definitions

1. Mathematical Concepts

Mathematics is the quintessence of generalization and therefore provides the language we need for our definitions. We start with the definition of a type:

Definition

A ***type*** is a set, and the elements of the set are called the ***values*** of the type.

We may therefore speak of the type ***integer***, meaning the set of all integers, the type ***real***, meaning the set of all real numbers, or the type ***character***, meaning the set of symbols that we wish to manipulate with our algorithms.


49

Notice that we can already draw a distinction between an abstract type and its implementation: The C++ type **int**, for example, is not the set of all integers; it consists only of the set of those integers directly represented in a particular computer, the largest of which depends on the word size of the computer. Similarly, the C++ types **float** and **double** generally mean certain sets of floating-point numbers (separate mantissa and exponent) that are only small subsets of the set of all real numbers.

Contents
Index
Help

2. Atomic and Structured Types

Types such as **int**, **float**, and **char** are called ***atomic*** types because we think of their values as single entities only, not something we wish to subdivide. Computer languages like C++, however, provide tools such as arrays, classes, and pointers with which we can build new types, called ***structured*** types. A single value of a structured type (that is, a single element of its set) is a structured object such as a contiguous stack. A value of a structured type has two ingredients: It is made up of ***component*** elements, and there is a ***structure***, a set of rules for putting the components together.

building types

For our general point of view we shall use mathematical tools to provide the rules for building up structured types. Among these tools are sets, sequences, and functions. For the study of lists of various kinds the one that we need is the ***finite sequence***, and for its definition we use mathematical induction.[3] A definition by induction (like a proof by induction) has two parts: First is an initial case, and second is the definition of the general case in terms of preceding cases.

Definition

A ***sequence of length*** 0 is empty. A ***sequence of length*** $n \geq 1$ of elements from a set T is an ordered pair (S_{n-1}, t) where S_{n-1} is a sequence of length $n-1$ of elements from T, and t is an element of T.

[3] See Appendix A for samples of proof by induction.

From this definition we can build up longer and longer sequences, starting with the empty sequence and adding on new elements from T, one at a time.

From now on we shall draw a careful distinction between the word ***sequential***, meaning that the elements form a sequence, and the word ***contiguous***, which we take to mean that the elements have adjacent addresses in memory. Hence we shall be able to speak of a *sequential* list in a *contiguous* implementation.

sequential versus contiguous

3. Abstract Data Types

definition of list

The definition of a finite sequence immediately makes it possible for us to attempt a definition of a list: a ***list*** of items of a type T is simply a finite sequence of elements of the set T.

Next we would like to define a stack, but if you consider the definitions, you will realize that there will be nothing regarding the sequence of items to distinguish these structures from a list. The primary characteristic of a stack is the set of *operations* or *methods* by which changes or accesses can be made. Including a statement of these operations with the structural rules defining a finite sequence, we obtain

Definition

A ***stack*** of elements of type T is a finite sequence of elements of T, together with the following operations:

1. *Create* the stack, leaving it empty.
2. Test whether the stack is *Empty*.
3. *Push* a new entry onto the top of the stack, provided the stack is not full.
4. *Pop* the entry off the top of the stack, provided the stack is not empty.
5. Retrieve the *Top* entry from the stack, provided the stack is not empty.

Note that this definition makes no mention of the way in which the abstract data type *stack* is to be implemented. In the coming chapters we will study several different implementations of stacks, and this new definition fits any of these implementations equally well. This definition produces what is called an ***abstract data type***, often abbreviated as ***ADT***. The important principle is that the definition of any abstract data type involves two parts: First is a description of the way in which the components are related to each other, and second is a statement of the operations that can be performed on elements of the abstract data type.

abstract data type

2.5.3 Refinement of Data Specification

Now that we have obtained such a general definition of an abstract data type, it is time to begin specifying more detail, since the objective of all this work is to find general principles that will help with designing programs, and we need more detail to accomplish this objective.

There is, in fact, a close analogy between the process of top-down refinement of algorithms and the process of top-down specification of data structures that we have now begun. In algorithm design we begin with a general but precise statement of the problem and slowly specify more detail until we have developed a complete

top-down specification

program. In data specification we begin with the selection of the mathematical concepts and abstract data types required for our problem and slowly specify more detail until finally we can implement our data structures as classes.

stages of refinement

The number of stages required in this specification process depends on the application. The design of a large software system will require many more decisions than will the design of a single small program, and these decisions should be taken in several stages of refinement. Although different problems will require different numbers of stages of refinement, and the boundaries between these stages sometimes blur, we can pick out four levels of the refinement process.

conceptual

1. On the *abstract* level we decide how the data are related to each other and what operations are needed, but we decide nothing concerning how the data will actually be stored or how the operations will actually be done.

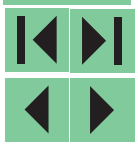
algorithmic

2. On the *data structures* level we specify enough detail so that we can analyze the behavior of the methods and make appropriate choices as dictated by our problem. This is the level, for example, at which we might choose a contiguous structure where data is stored in an array.

programming

3. On the *implementation* level we decide the details of how the data structures will be represented in computer memory.
4. On the *application* level we settle all details required for our particular application, such as names for variables or special requirements for the operations imposed by the application.

Contents
Index
Help



The first two levels are often called ***conceptual*** because at these levels we are more concerned with problem solving than with programming. The middle two levels can be called ***algorithmic*** because they concern precise methods for representing data and operating with it. The last two levels are specifically concerned with ***programming***.

Our task in implementing a data structure in C++ is to begin with conceptual information, often the definition of an ADT, and refine it to obtain an implementation as a C++ class. The methods of the C++ class correspond naturally to the operations of the ADT, while the data members of the C++ class correspond to the physical data structure that we choose to represent our ADT. In this way, the process of moving from an abstract ADT, to a data structure, and then on to an implementation leads directly to a C++ class definition.

Let us conclude this section by restating its most important principles as programming precepts:

Programming Precept

Let your data structure your program.
Refine your algorithms and data structures at the same time.

Programming Precept

Once your data are fully structured,
your algorithms should almost write themselves.

Exercises 2.5

IRM

E1. Give a formal definition of the term *extended stack* as used in Exercise E1 of Section 2.2.

E2. In mathematics the ***Cartesian product*** of sets $T_1, T_2, \ldots, T_n$ is defined as the set of all n-tuples $(t_1, t_2, \ldots, t_n)$, where t_i is a member of T_i for all $i, 1 \le i \le n$. Use the Cartesian product to give a precise definition of a class.

POINTERS AND PITFALLS

1. Use data structures to clarify the logic of your programs.
2. Practice information hiding and encapsulation in implementing data structures: Use functions to access your data structures, and keep these in classes separate from your application program.
3. Postpone decisions on the details of implementing your data structures as long as you can.
4. Stacks are among the simplest kind of data structures; use stacks when possible.
5. In any problem that requires a reversal of data, consider using a stack to store the data.
6. Avoid tricky ways of storing your data; tricks usually will not generalize to new situations.
7. Be sure to initialize your data structures.
8. In designing algorithms, always be careful about the extreme cases and handle them gracefully. Trace through your algorithm to determine what happens in extreme cases, particularly when a data structure is empty or full.
9. Before choosing implementations, be sure that all the data structures and their associated operations are fully specified on the abstract level.

Contents
Index
Help

REVIEW QUESTIONS

2.1 IRM

1. What is the standard library?

2. What are the methods of a stack?

3. What are the advantages of writing the operations on a data structure as methods?

2.2

4. What are the differences between information hiding and encapsulation?

5. Describe three different approaches to error handling that could be adopted by a C++ class.

6. Give two different ways of implementing a generic data structure in C++.

2.3 **7.** What is the reason for using the reverse Polish convention for calculators?

2.5 **8.** What two parts must be in the definition of any abstract data type?

9. In an abstract data type, how much is specified about implementation?

10. Name (in order from abstract to concrete) four levels of refinement of data specification.

REFERENCES FOR FURTHER STUDY

stacks

For many topics concerning data structures, such as stacks, the best source for additional information, historical notes, and mathematical analysis is the following series of books, which can be regarded almost like an encyclopædia for the aspects of computing science that they discuss:

encyclopædic reference: KNUTH

DONALD E. KNUTH, *The Art of Computer Programming*, published by Addison-Wesley, Reading, Mass.

Three volumes have appeared to date:

1. *Fundamental Algorithms*, second edition, 1973, 634 pages.
2. *Seminumerical Algorithms*, second edition, 1980, 700 pages.
3. *Sorting and Searching*, 1973, 722 pages.

In future chapters we shall often give references to this series of books, and for convenience we shall do so by specifying only the name KNUTH together with the volume and page numbers. The algorithms are written both in English and in an assembler language, where KNUTH calculates detailed counts of operations to compare various algorithms.

A detailed description of the standard library in C++ occupies a large part of the following important reference:

BJARNE STROUSTRUP, *The C++ Programming Language*, third edition, Addison-Wesley, Reading, Mass., 1997.

The Polish notation is so natural and useful that one might expect its discovery to be hundreds of years ago. It may be surprising to note that it is a discovery of the twentieth century:

JAN ŁUKASIEWICZ, *Elementy Logiki Matematyczny*, Warsaw, 1929; English translation: *Elements of Mathematical Logic*, Pergamon Press, 1963.