Applied C and C++ Programming

Alice E. Fischer
David W. Eggert
University of New Haven

Michael J. Fischer
Yale University

August 2018

Part VI

Developing Sophistication

Chapter 19

Recursion

This chapter presents **recursion**, a fundamental technique in which a function solves a problem by dividing it into two or more sections, calling itself to solve each portion of the original problem, continuing the self-calls until a subproblem's solution is trivial, and then building a total solution out of the partial solutions. We examine how to apply recursion to two familiar problems: searching and sorting.

Prior to this, as an aid to understanding how recursion works, we briefly survey C/C++ storage classes, giving a detailed description of how the `auto` storage class is used to implement recursive parameter passing.

19.1 Storage Classes

Every variable has a storage-class attribute in addition to its data type and perhaps a type qualifier like `const` or `volatile` as well. The **storage class** determines when and where a variable is allocated in memory, when it is initialized, how long it persists, and which functions may use it. There are three useful storage classes in C++: automatic, `extern`, and `static`. Automatic is the default storage class for all local variables and parameters. `Static` is used to create a variable with a lifetime longer than the block that declares it. `Extern` is the default for global variables. It makes a global variable in one module visible to other program modules. `Extern` is rarely needed if good OO design is used.

19.1.1 Automatic Storage

The automatic storage class is used in C/C++ for function parameters. Local variables within functions also are automatic by default, but can be declared `static`, as discussed in the next section. Whenever you call a function, memory for every automatic variable is allocated in an area called the **run-time stack**, with other function call information. This memory is deallocated when the function returns. The stack memory is managed automatically and efficiently by the C/C++ system.

Automatic is the default storage class for local variables because it makes the most efficient use of computer memory. Good programmers use automatic variables (rather than `static` local variables or global variables) wherever possible, because their lifetime is brief and their scope is restricted to one function. They provide an effective way to isolate the actions of each program unit from the others. Unintentional interference between functions is minimized, making it much easier to debug a large program. On the other hand, the constant allocation and deallocation of storage adds slightly to the execution time. Because it is done quite efficiently, the time wasted is minimal and the advantages of the added modularity are great. The characteristics of automatic memory are discussed in greater detail in an upcoming section on the run-time stack, but briefly automatic memory is

- **Allocated** on the run-time stack.
- **Initialized** during the function call.

- **Visible** only to the statements in the body of the declaring function.
- **Deallocated** during the function-return process. Local variables are freed by the called function and parameter storage is freed by the caller, both under the control of the C/C++ run-time system.

19.1.2 Static Storage

The name *static* is derived from the word *stay;* it is used because `static` variables stay around from the beginning to the end of execution. In older computer languages, all variables were `static`. In C/C++, local variables are automatic by default but can be declared `static`. There are two purposes for `static` local variables: to remember the state of the subprogram from the end of one execution to the beginning of the next, and to increase time and space efficiency when a function uses a table of constants. The characteristics of `static` local memory are

- It is allocated in a portion of computer memory separate from the run-time stack, which we call the **static allocation area**. In many implementations, this immediately follows the memory block used for the program's code.
- It is initialized when the program is loaded into memory. Initializers must be constants or constant expressions; they cannot refer to non-`static` variables, even ones with a `const` qualifier.
- It is deallocated when the program exits and returns to the system.
- It is visible to the statements in the body of the function in which the `static` variable is declared.

Global variables default to `extern` but also may be declared `static`.[1] A `static` global variable is like a `static` local variable in almost every way. It is declared at the top of the source-code file and visible to all functions in that same file.[2]

The state of a function. In some complex situations, a function must remember a piece of data from one call to the next. For example, a function that assigns consecutive identification numbers to a series of manufactured objects would need to know the last number assigned before it could assign the next. We say that this last number is part of the **state** of the system. Since automatic variables come and go as function calls are begun and completed, they cannot be used to store state information. Global variables can be used, but doing so is foolish because these variables can be seen and changed by other parts of the program. A `static` local variable is an ideal way to save state information because it is both private and persistent.

Tables of constants. Normally, the time consumed by the allocating and initializing of automatic variables is minimal. It is not considered a waste because this allocation method guarantees that every function starts its computations with a clean slate. However, reinitializing a **table of constants** every time you enter a function that uses them can consume appreciable execution time. It also requires extra space, because the program code must contain a copy of the entire initializer for the table, which it uses to reinitialize the local automatic table every time the function is called. If the table truly is constant, there is certainly no need to keep recopying it.

We could solve this problem by declaring the table as a global constant. However, this is not really a good solution if the table is used by only one function. A general principle is that things used together should be written together in a program. If possible, a table should be defined by the function that uses it, not at the top of the program file, which may be several pages away. A better way to eliminate the wasted time and space is to declare the table inside the function with a `static` storage class. It will be visible only within the defining function and will not be re-created every time the function is called. Space will be allocated once (before program execution begins) and initialized at that time.

[1] This makes a difference only in a multimodule application where the modules are compiled separately and linked before execution.

[2] The `static` variable is visible to functions from any included files if the `#include` command follows the static declaration.

19.1.3 External Storage (Advanced Topic)

A variable or function of the `extern` storage class is made known to the linker explicitly. This becomes useful when an application is so large that it is broken up into modules, often written by different people. Each module is a set of types, functions, and data objects in a separate source-code file. These files are compiled separately and joined into a single working unit by the linker. In such a program, many objects and functions are defined and used only within one subsystem. Others are defined in one module and used by others. They must be made known to the linker, so that it can make the necessary connections. This is known as **external linkage**.

Functions, as well as global constants and variables, are labeled `extern` by default. These symbols are made known to the linker unless they are explicitly declared to be `static`. A `static` function or object is visible only within the code module that defines it.

Rules for `extern` declarations. Developing large, multi-module programs is beyond the scope of this text. The following details are provided for general knowledge and as a basis for the small multi-module system developed in Chapter 20.3.

An `extern` symbol must be declared in both the code file that supplies it and the code files that use it. One declaration defines the object and one or more declarations in other modules may import it. These two kinds of declarations are subtly different in several ways:

Creation. A data object declaration with an initializer is the defining occurrence, whether or not the keyword `extern` is present. A function prototype is a defining occurrence but only if the function definition appears in the same code module.

Allocation and initialization. The defining occurrence is allocated and initialized when its code module is loaded into memory. Initializers must be constants or constant expressions.

Deallocation. External symbols are global in nature, therefore they are freed when the program exits and returns to the system.

Visibility. The defining occurrence is visible in its own code module and any other code module that contains a matching declaration with the keyword `extern`. The importing occurrence must not include an initializer (if it is a data object) or a function definition (if it is a prototype).

19.2 The Run-Time Stack (Advanced Topic)

During a function call, the caller passes arguments that will be stored in the called-function's parameter variables, and these are used by the called-function in its computations. Static storage cannot be used for **parameters** because recursion requires storage for each parameter in the stack-area created for each call on the function. When a function calls itself, two or more sets of its argument values must coexist. At compile time, the translator cannot know how many times a recursive function will call itself, so it cannot know how much storage will be needed for all instances of its parameters. For this reason, C/C++ allocates space for parameters and local variables in a storage area that can grow dynamically as needed.

In C/C++, as in most modern languages, a data structure called the *execution stack*, *run-time stack*, or simply, *stack* is used to store one *stack frame* for each function that has been called, but has not yet returned, since execution began. In this section, we look closely at the mechanism by which a function actually receives and stores its parameters.

19.2.1 Stack Frames

A function's **stack frame**, or **activation record**, holds its parameters, return address, local variables, and a pointer to the prior stack frame. In it, there is one variable for each parameter listed in the function

header. Before control is transferred from the caller to the function, the argument values written as part of the function call are computed and copied into the parameter variables in the new stack frame, as follows:

1. Memory is allocated in the stack frame for the parameters.
2. The argument expressions in the call are evaluated and the resulting values are written in order on the stack. If the type of an argument does not match the type of the corresponding parameter in the function's prototype, the argument value is converted to the required type.
3. Then control is transferred to the subprogram with a jump-to-subroutine machine instruction. One of the actions of this instruction is to store the return address in the stack.
4. When control enters the subprogram, it associates the values in the stack frame, in order, with the parameter names in the function declaration. More stack storage is then allocated and initialized for the local automatic variables (storage for `static` variables was allocated at load time).
5. Finally, control is transferred to the first line of code in the subprogram. Execution of the subprogram continues until a `return` statement is reached. If there is no `return` statement, execution continues until the last line of code is finished.

When execution of the function ends, the result is returned and storage is freed as follows:

1. The result of the `return` expression is converted (if necessary) to the return type declared by the function prototype. The return value then is stored where the caller expects to find it.
2. The local automatic variable storage area is freed. (Local `static` variables are not freed until the program terminates.)
3. Control returns to the caller at the **return address** that was stored in the stack frame during the function call.
4. The caller picks up the return value from its prearranged location and frees the stack storage area occupied by argument values.
5. Execution continues in the calling program at the machine instruction after the jump-to-subroutine instruction.

19.2.2 Stack Diagrams and Program Traces

A **stack diagram** is a picture of the stack frames that exist at some given time during execution. We use stack diagrams to help understand what does and does not happen with function calls when we manually trace the execution of a program. They are particularly helpful in understanding recursion. Figure 19.1 shows a generic stack diagram. Figure 19.2 shows a more specific instance of a stack, which can be referred to during the following discussion.

To **trace** the operation of **a program**, start by drawing an area for global and `static` variables. Record the names and initial values of these variables. Set aside a second area for the program's output to simulate the screen. Begin the stack diagram at the bottom of the page with a frame for `main()` (new frames will be added above this one). Record the names and initial values of the variables of `main()` in this frame. Make three columns for each variable, giving its type, its name, and the initial value (or a `?` if it is not initialized). Leave room to write in new values that might be assigned later.

Finally, start with the first line of code in `main()` and progress from line to line through the program code. Emulate whatever action each line calls for. If the line is an output command, write the output in your screen area. If the line is an assignment or input statement, change the value of the corresponding variable in the rightmost column of your stack diagram.

If the line is a function call, mark the position of the line in the calling program so that you can return to that spot. Add a frame to your stack diagram. Label the new frame with the function's name and record the argument values in it. Label these with the parameter names and add the local variables and their values. Then trace the code of the function until you come to the last line or a `return` statement. Write the

The diagram on the left shows the arrangement of frames on the stack; the frame for `main()` is allocated first; a frame for each other function is added when the function is called and removed when it returns. The diagram on the right shows a more detailed view of the components of one stack frame.

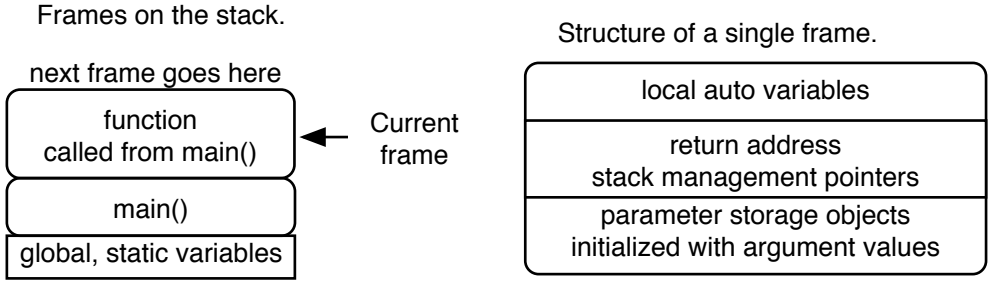


Figure 19.1. The run-time stack.

function's return value under its name on the left side of the stack frame and draw a diagonal line through the function's stack frame to indicate that it has been freed.

For example, Figure 19.2 shows the stack at two times during the execution of the gas pressure program from Figure 9.5. The diagram on the left shows the stack during the process of returning from the `ideal()` function. In the next program step, the result will be stored in the variable `pres` of `main()` and the stack frame for `ideal()` will be freed. Later, `vander()` is called. The diagram on the right shows the stack when `vander()` is ready to return. The stack frame for `ideal()` is gone (it would be crossed out in your diagram), and a new stack frame for `vander()` is in its place.

19.3 Iteration and Recursion

In Chapter 5.8 we introduced a number guessing game that asked the human user to guess a secret number in the range 1...1000. If the user plays this game wisely, each guess will be halfway between the smallest and largest remaining possible values. By dividing the remaining interval in half each time and discarding one half, the secret number will eventually be the only possibility left. The player is able to halve the unexplored territory each time because, on each step, we tell the user which subinterval, left or right, contains the target

These diagrams are pictures of the stack for the gas pressure program developed in Figure 9.5. The values shown for `temp`, `m`, and `vol` were entered by the user. On the left, the stack is shown during a call on the `ideal()` function, just before the function's `return` statement occurs. On the right, the stack is shown just before the return from `vander()`. The gray areas in each stack frame contain the function's return address and information used by the system for stack management.

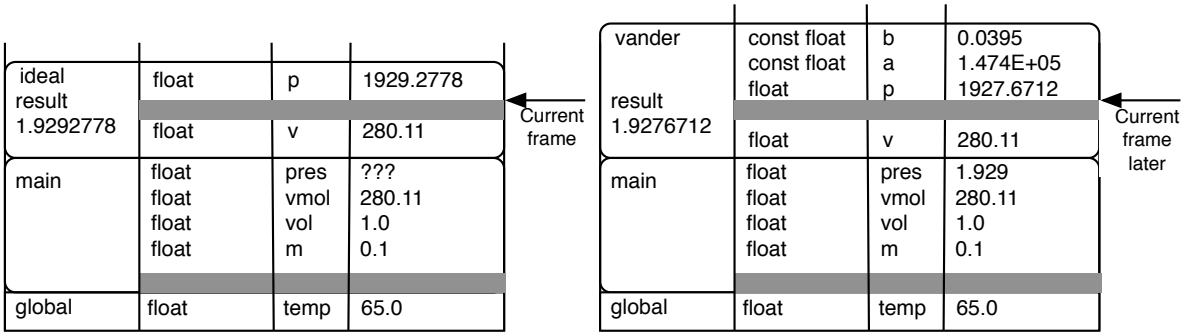


Figure 19.2. The run-time stack for the gas pressure program.

number. This strategy is a form of **binary search**, which we formalize later in this chapter.

The binary search algorithm is an example of a general problem-solving approach called **divide and conquer**. If the problem is too difficult to solve when it is first presented, simplify it and try again. If this can be repeated, eventually the problem will become manageable. A loop, or **iteration**, is one way to repeat a process; recursion is another. We review the essentials of algorithms that iterate, then show how the problem simplification strategy is reflected in recursive processes.

19.3.1 The Nature of Iteration

Every loop consists of a control section and a body. The control section performs initialization and contains an expression, called the *loop condition*, that is tested before, during, or after every iteration. The loop ends when this condition has the value `false`. If we use a loop, something must change on each iteration (and that change must affect the loop condition) or the process will never end. We have seen infinite loops that fail to increment the loop variable or read a new input value; they simply repeat the same statements again and again until the user kills the process. Every programmer occasionally writes such a loop by mistake. In loops that are not controlled by the user's input, each iteration must make the remaining job smaller; that is, each trip around the loop must perform one step of a finite process, leaving one fewer step to do. Often progress is measured by the value of a variable that is incremented or decremented on each iteration, until a predetermined goal is reached.

19.3.2 The Nature of Recursion

A *recursive function* does its task by calling upon itself, not by going around a loop. Like a loop, a recursive function must reduce the size of the remaining task on every call, or the process will become mired in an infinite repetition of the same process step; this is called **infinite recursion**. Eventually, an infinite recursion will be detected by the system and the process aborted with a comment such as **stack overflow**, due to the allocation of too many stack frames or too much memory.

When one function calls another, argument values are passed into the subprogram, local storage is allocated on the stack for them, and an answer eventually is passed back to the caller. The same is true when a recursive function calls itself. The stack provides storage for parameters and local variables for each call on a function. We say that a **call is active** until the called function returns and its frame is discarded. Normally, there are several active functions at once, because one function can call a second one, and the second can call a third, and so forth. Therefore, at any given time, many frames can be on the stack. The computer always is executing the most recently called function, so it uses the most recent frame. When the task of a function is complete, its stack frame is eliminated and it returns control to its caller. The caller, which had been waiting for the subprogram to finish, resumes its own task, using its own stack frame, which is now on top of the stack. A recursive function calls itself, so the caller and the subprogram are two activations of the same function.

If a recursive function calls itself five times, six stack frames will exist simultaneously for it, each holding the parameters for one of the active calls. Each time one of the calls returns, its stack frame is discarded and control goes back to the prior invocation. That is, when the fifth active call returns, control goes back to the fourth active call. The fourth incarnation then resumes its work at the spot just after the recursive call. This process proceeds smoothly, just as with ordinary function calls, so long as each call *specifies a smaller task* than the previous call. In theory, if the job does not get smaller, the recursions will continue endlessly. Practically speaking, though, a recursion is unlikely to last forever because each stack frame requires memory, thereby making the stack longer. Eventually, all of the available memory is used up and the stack becomes too large to fit into memory. At this time, the system will report that the stack has overflowed and abort the user's process.

As with loops, every recursion must contain a condition used to determine whether to continue with more recursive calls or return to its caller. This condition often is referred to as the **base case**, because the action to take (usually to return) is basic and straightforward. Frequently, the entire body of a recursive function is just a single `if...else` statement, where the `if` contains the termination condition, the `true`

clause contains the `return` statement, and the `else` clause contains the recursive call. The simplest and most common recursive strategy is

- Base cases: The function checks the first base case and returns if it is satisfied. If needed, it checks the second, third, and further base cases and returns if any one of them is satisfied.
- Recursive step: If no base case is satisfied, the function calls itself recursively with arguments that exclude part of the original problem. When the recursive call returns, the function processes the answer, if necessary, and returns to the caller.

The program in Figure 19.3 applies this recursive strategy to the task of finding the minimum value in a list of data items. On each recursive step, the function processes the first item on the list and calls itself recursively to process the rest of the list. Thus, each step reduces the size of the task by one item. This pattern is typical of recursion applied to a list or array of data items, and is called a **simple list recursion**.

19.3.3 Tail Recursion

If the recursive call is the last action taken before the function returns, we say it is a **tail recursion**. Once the returns start in a tail recursion, all the calls return, one after another, and all the stack frames are discarded one by one until control reaches the original caller. The value computed during the last call is sent back through each successive stack frame and eventually returned to the original caller. The binary search function in Figure **??** is tail recursive; the results of the recursive calls are returned immediately without further processing. Any tail recursive algorithm can be implemented easily, and more efficiently, as a loop. The bisection program in Chapter 11 that finds the roots of an equation is a form of binary search implemented using a `do...while` loop.

However, not all recursive algorithms are tail recursive. For example, the `array_min()` function in Figure 19.3 is not tail recursive because the result of the recursive call is compared to another value before returning. Although we can easily write an iterative algorithm to find the minimum value in a list, many recursive algorithms cannot be rewritten easily as iterative programs; they are essentially recursive and have no simple, natural, iterative form. The last program in this chapter, quicksort, falls into this category. It is one of the most important and easily understood of these inherently recursive algorithms.

19.4 A Simple Example of Recursion

The program in Figure 19.3 uses recursion to find the minumum value in an array of values. Compare it to the iterative maximum-finding program in Figure 10.24. The iterative solution is better; it is just as simple and much more efficient. However, the recursive solution is useful for illustrating how recursion works and developing some intuition for it. Later examples will demonstrate that recursive solutions can be easier to develop than iterative versions in some cases.

Notes on Figure 19.3: A recursive program to find a minimum.

Second box, Figure 19.3: the recursive descent. We want to find the minimum value in a set of values. But that requires some work, so we use a strategy of shrinking the size of the problem. At each step, we defer the action of examining the first value in the set (`a[0]`) and turn our attention to finding the minimum of the rest (`a[1]` to `a[n-1]`). If we do this recursively, we put aside the values one at a time, until only one value remains. Once we reach a set of one, we know (trivially) that this value is the minimum, so we can simply return it to the caller. This occurs halfway through the entire process. At this point, we have progressed, recursively, to the end of the array, and reached the base case of the recursion. Given the sample `data` array, the program output up until this point is

```
Find minimum in an array of 15 positive integers.
Entering: n=15  head=19
Entering: n=14  head=17
Entering: n=13  head= 2
```

```
Entering: n=12  head=43
Entering: n=11  head=47
Entering: n=10  head= 5
Entering: n= 9  head=37
Entering: n= 8  head=23
Entering: n= 7  head= 3
Entering: n= 6  head=41
Entering: n= 5  head=29
Entering: n= 4  head=31
Entering: n= 3  head= 7
Entering: n= 2  head=11
---- Base case ----
```

```
#include <iomanip>
using namespace std;
#define N 15
int arrayMin( int a[], int n );

int main( void )
{
    int data[N] = { 19, 17, 2, 43, 47, 5, 37, 23, 3, 41, 29, 31, 7, 11, 13 };
    int answer;

    cout <<"\n Find minimum in an array of " <<N <<" positive integers.\n";
    answer = arrayMin( data, N );
    cout <<"\n Minimum = " <<answer <<"\n\n";
}

// ------------------------------------------------------------------
int arrayMin( int a[], int n )
{
    int tailmin, answer;
    cout <<" Entering:  n=" <<setw(2) <<right <<n
         <<" head= " <<a[0] <<"\n";
    // Base case ---------------------------------------------------
    if (n == 1) {
        cout <<" ---- Base case ----\n";
        answer = a[0];
    }

    // Recursive step ----------------------------------------------
    else {
        tailmin = arrayMin( &a[1], n-1 );
        if (a[0] < tailmin) answer = a[0];
        else answer = tailmin;
    }
    cout <<" Leaving:  n=" <<setw(2) <<right <<n <<" head=" <<setw(2)
         <<a[0] <<" returning=" <<setw(2) <<answer <<"\n";
    return answer;
}
```

Figure 19.3. A recursive program to find a minimum.

Although we always print `a[0]`, it is not the same as `data[0]`, except on the very first call; it actually is the initial value of the portion of the array that is the argument for the current recursive call. During the processing, we repeatedly peel off one element and recurse. From the output, we can see that, after executing the first `printf()` statement 14 times, we have not yet reached the second `printf()` statement because of the recursions. Finally, `n` is reduced to 1 and we reach the base case.

First box, Figure 19.3: the base case. When writing a recursive function that operates on an array, the base case usually happens when one item is left in the array. In this example, given a set of one element, that element is the minimum value in the set. Therefore, we simply return it. The `true` clause of the `if` statement contains a diagnostic `printf()` statement; it is executed midway through the overall process, after the recursive descent ends and before the returns begin. If we eliminate this `printf()` statement, the base case becomes very simple:

```
if (n == 1)   return a[0];
```

The diagnostic output from the base-case call shows us that we have reached the end of the array:

```
Entering: n= 1  head=13
---- Base case ----
Leaving:  n= 1  head=13  returning=13
```

Now we begin returning. The stack frame of the returning function is deallocated and the stack frame of the caller, which had been suspended, again becomes current. When execution resumes, we are back in the context from which the call was made. When we return from the base case, we go back to the context in which there were two values in the set.

Second box revisited, Figure 19.3: ascent from the recursion. At each step, we compare the value returned by the recursive call to `a[0]`. (Remember that `a[0]` is a different slot of the array on each recursion.) We return the smaller of these two values to the next level. Thus, at each level, we return with the minimum value of the tail end of the array and that tail grows by one element in length each time we return. When we get all the way back to the original call, we will have the minimum value of the entire array, which is printed by `main()`:

```
 ---- Base case ----
Leaving:   n= 1   head=13   returning=13
Leaving:   n= 2   head=11   returning=11
Leaving:   n= 3   head= 7   returning= 7
Leaving:   n= 4   head=31   returning= 7
Leaving:   n= 5   head=29   returning= 7
Leaving:   n= 6   head=41   returning= 7
Leaving:   n= 7   head= 3   returning= 3
Leaving:   n= 8   head=23   returning= 3
Leaving:   n= 9   head=37   returning= 3
Leaving:   n=10   head= 5   returning= 3
Leaving:   n=11   head=47   returning= 3
Leaving:   n=12   head=43   returning= 3
Leaving:   n=13   head= 2   returning= 2
Leaving:   n=14   head=17   returning= 2
Leaving:   n=15   head=19   returning= 2

Minimum = 2
```

19.5 A More Complex Example: Binary Search

Searching an arbitrary array for the position of a key element is a problem that has only one kind of general solution, the sequential search. If the data in the array are in some unknown order, we must compare the

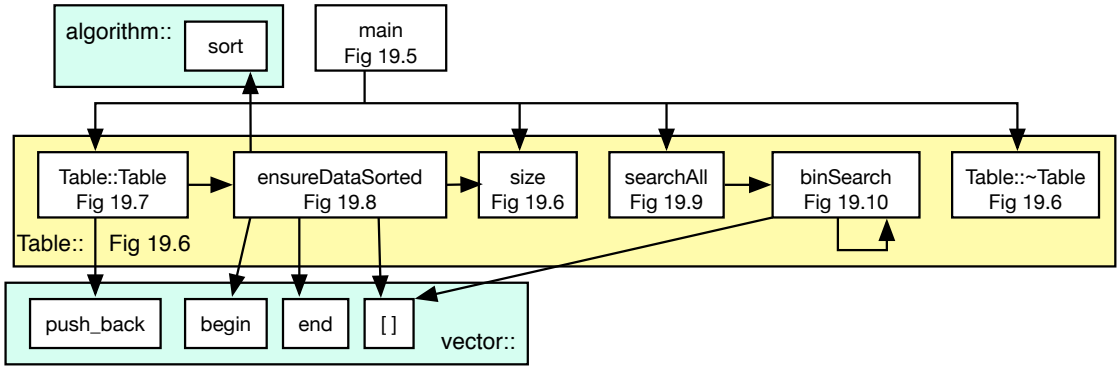


Figure 19.4. Call chart for binary search.

key to every item in the array before we know that the key value is not one of the array elements. Sequential search works well for applications like the Chauvenet table, presented in Chapter 10, where only a few values are looked through. For an extensive data set such as a telephone book or an electronic parts catalog, sequential search is painfully slow and completely inadequate. Therefore, large databases are organized or indexed so that rapid retrieval is possible. One such way to organize data is simply to sort it according to a key value and store the sorted data in an array. We have seen algorithms for the selection and insertion sort so far. The quicksort algorithm will be discussed in the next section.

Binary search is an algorithm that allows us to take advantage of the fact that the data in an array are sorted. (We simplify the following discussion by assuming that it is in ascending order. One of the exercises examines descending order.) The search can be implemented easily using either iteration or recursion. Both formulations are straightforward. The iterative version is a little more efficient, while the recursive version is a little shorter and easier to write. We will use the program in this section to illustrate both recursion and the binary search algorithm. The call chart in Figure 19.4 summarizes the overall structure of this program and where each piece can be found. Figures 19.5 through **??** contain a program that reads a sorted data file into an array, then searches for values requested by the user. We start with a `main()` program and its utility functions, then look at the recursive function last.

Notes on Figure 19.4. Call chart for binary search.
In the diagram, the yellow box surrounds functions defined by the Table class. The blue-green boxes show functions from the system libraries algorithm and vector. The circular arrow represents the recursive call on `binSearch()`.

Notes on Figure 19.5. The binary search main program.

First box, Figure 19.3: The file name.

- This program asks the user to enter the name of the data file.
- The function `getline()` reads an input of indefinite length and stores it in a C++ string variable. The string variable will grow as long as necessary to contain all the input up to a newline.

Second box, Figure 19.5: Entering the OO world.

- A always, the job of main is to instantiate one Table object and call its primary function to search the data for key values entered by the user.
- We create the Table object to use the filename just entered. The constructor will open and read the file, and store the contents in a `vector`. The vector will grow as long as necessary to store all the data in the file.
- It is always a good idea for `main()` to display information for the user at the end of every major step of its process.
- Finally, now that the data is ready to use, we call `searchAll`, the primary function in the Table class.

This `main()` program uses the class defined in Figure 19.6 and Figure 19.7

```
#include "table.hpp"
// ------------------------------------------------------------------------
int main( void )
{
    string inname;

    cout <<"\n Find a key value in an array of integers, 0 ...  1000.\n"
        <<" Enter name of data file to be searched:  ";
    getline( cin, inname );

    Table t( inname );
    cout <<" " <<t.size() <<" data items read; ready to search.\n";
    t.searchAll();

    return 0;
}
```

Figure 19.5. The binary search main program.

Sample output from `main()`***:***

```
Find a key value in an array of integers, 0 ... 1000.
Enter name of data file to be searched: binserch.dat.txt
17 data items read; ready to search.
```

Notes on Figure 19.6. The Table class.

First box, Figure 19.6: The vector.

- A `vector` is a "smart" array: it starts empty, at a default length, and grows repeatedly by doubling its length each time. Growth happens when the current array is full and your program tries to put another data item into it. To use `vector`, `#include <vector>`.
- The self-adjusting nature of a vector is especially useful for input, if you do not know how much data to expect.

noindent This class is instantiated by the `main()` program in Figure 19.5.

```
#include "tools.hpp"
#define NOT_FOUND -1

class Table {
private:
    vector<int> data;

    void ensureDataSorted();
    int binSearch( int left, int right, int key );
public:
    Table( string inname );
    ~Table(){}
    void searchAll(); int size(){ return data.size(); }

};
```

Figure 19.6. The Table class declaration.

```
Table:: Table( string inname ) {
    int input;
    ifstream infile( inname );
    if (!infile.is_open()) fatal( " Cannot open input file %s", inname.data() );

    for( ;; ){
       infile >> input;
       if( ! infile.good() ) break;
       data.push_back( input );
    }

    ensureDataSorted();
}
```

Figure 19.7. The Table constructor.

Second box, Figure 19.6: Private functions.
Neither one of these functions is ever called from outside the class.

- `ensureDataSorted()` is called from the Table constructor if the input file was not sorted.
- `binSearch()` is the recursive search function, called from `searchAll()` to search for a series of inputs in the sorted file.

Third box, Figure 19.6: Public functions.
These four functions are all called from `main()`.

- The Table constructor sets up the sorted data array, ready to search.
- `searchAll()` does the actual search.
- The Table destructor is called implicitly from `main()` when the program finishes. Note that it does nothing, because the Table functions did not create any dynamic memory.
- The destructor and the size() function are both short inline functions. An inline function is more efficient at run-time than an ordinary function.
- `size()` returns the answer from calling `vector::size()`. "Wrapper" functions like this are often used with data structures defined by the C++ libraries.

Notes on Figure 19.7: The Table constructor.
This program uses data from the keyboard to open a data file. The data is supposed to be in ascending sorted order. Therefore, reading the input really is a three-step process: (1) identify and open the file, (2) read the information, and (3) verify that it is sorted in ascending order.

First box, Figure 19.7: Opening the input stream.

- The data file name was read in `main()` and has been passed to the constructor as a parameter. The first line opens the stream.
- It is always necessary to check whether a file is properly open. In this case, we abort if it is not.
- The function `fatal()` is used to print an error comment. The The first argument is a C-style format. The second is a c-string, as required by the `%s` in the format,
- We can extract the C-string from the C++ string by using the `.data()` function or the `.c_str()` function.

This function is called from the Table constructor in Figure 19.7

```
void Table::
ensureDataSorted() {
    for (int k = 1; k < data.size(); ++k) {
        if (data[k-1] > data[k]) {
            sort( data.begin(), data.end() );
            return;
        }
    }
}
```

Figure 19.8. Ensuring that the data is sorted.

- Here is the error report that resulted from entering an incorrect file name:

```
Find a key value in an array of integers, 0 ... 1000.
Enter name of data file to be searched: bins.txt
Cannot open input file: bins.txt
Error exit; aborting.
```

Second box, Figure 19.7: Reading the data.

- When using `>>` to do input, a program must test for end-of-file *after* reading some data, not *before*. So we use the indefinite for loop, not a while loop.
- Inside the for loop there is an `if...break` that tests for end-of-file and ends the loop when that happens.
- The `if...break` will also end the loop if there was a hardware read error or the file contains non-numeric data.
- Once read, each input number is stored in the next available slot of the vector named `data`.

Third box, Figure 19.7: A precondition.

- A binary search program cannot function properly unless the data in the array are sorted in the correct order, in this case, ascending[3] order.
- Rather than take it on faith that the numbers are sorted, we call the function in Figure 19.8 to check the order and, if an error is found, sort the data.

Notes on Figure 19.8: Ensuring that the data is sorted. One of the principles of OO design is that every class should take care of its own emergencies. In this case, it means that we should make sure the data in the array is sorted before trying to use binary search on it.

Outer box, Figure 19.8: Check the whole array.
Note that this loop starts by comparing the contents of slots 0 and 1. We cannot use a for-each loop here because only $size - 1$ comparisons are required to sort $size$ numbers.

Middle box, Figure 19.8: Test the order of two items. If two adjacent items are in ascending order, all is well and there is nothing to do. That is why there is no `else` clause for thie `if`.

Inner box, Figure 19.8: Sort if necessary. If any two adjacent items are not in ascending order, the array is not sorted. In that case, we call the `sort()` function from the algorithms library. You can do this with any vector. `.begin()` and `.end()` are defined as iterators that point to the first thing in the vector and the first address that is past the end of the data in the vector.

[3]Technically, we require non-descending order.

This function is called from the main program in Figure 19.5

```
void Table::
searchAll() {
    int slot, begin, end, key;
    cout <<"\ Enter numbers to search for; period to quit.\n";
    for(;;) {
        cout <<" What number do you wish to find?  ";
        cin >> key;
        if (!cin.good()) break;

        cout <<" " <<key <<endl;

        slot = binSearch( 0, data.size(), key );
        if (slot == NOT_FOUND) cout <<" is not in table.\n\n";
        else cout <<" was found in slot " <<slot <<".\n\n";
    }
}
```

Figure 19.9. Searching for numbers.

Notes on Figure 19.9: Interaction with the user.
It is often not necessary to give a full prompt for every user input. In this case, we prompt at the beginning and not every time around the input loop.

First box, Figure 19.9: Prompt once for all.
The input variable, `key` is declared to be an `int`. Therefore, when you write `cin >> key`, the system expects the user to type in a number.

Second box, Figure 19.9: Search for another?

- The prompt at the top tells the user to enter numbers (plural) and to enter a '.' to quit. Actually, since we are doing integer input, *any* non-digit will cause a stream error (faulty input conversion).
- Thus, you could enter a period, as instructed, or enter a letter or special character .... Any of them will cause the stream to set an error flag.
- The third line in this box tests for a stream error. The same test can be used to find an end-of-file, but here we are actually looking for a conversion error.

Third box, Figure 19.9: Debugging technique.
When trying to debug a loop or a recursion it is always a good idea to put a printout in your code that will help you track the progress of the algorithm. This debugging printout echoes the user's input, a necessary step to be sure that the data is being read correctly.

Fourth box, Figure 19.9: The search.

- The actual recursive search is done in `binSearch()`. This function sets up the search, receives the found/not found answer, and displays it for the user.
- In the first line, we call `binSearch()`. The third parameter is the number we are searching for.
- The first and second subscripts are integer subscripts because we want to do arithmetic with them: 0 is the subscript of the first data item in the array.
- `data.size()` is the number of data values in the array and also an *offboard* subscript for the end of the data. That is, `data.size()` is the subscript for the first slot that does not contain data.

This recursive function is called from `searchAll()` in Figure 19.9. Diagrams of the operation start in Figure 19.11.

```
int Table::
binSearch( int left, int right, int key ) {
    int mid = left + (right-left)/2;    // Compute middle of search interval
    cout <<" left=" <<left <<" mid=" <<mid <<" right=" <<right <<endl;

    // Base cases:  (1) we found it or (2) it's not there
    if (left >= right) return NOT_FOUND;
    if (data[mid] == key) return mid;

    // Recursion:  search a smaller array
    if (key < data[mid]) return binSearch( left, mid, key );
    else                 return binSearch( mid + 1, right, key );
}
```

Figure 19.10. The binary search function.

Notes on Figure 19.10: The binary search function.
We search for the key value among the `n` values in the data array, which are sorted in ascending order.

First box, Figure 19.10: Split the array.

- On each recursion, the left and right subscripts are closer to each other. To find the subscript of the middle of the array, we use `right - left` to calculate how far apart they are, then add half the distance to `left` to get the middle.
- This calculation uses integer arithmetic. If there are an odd number of things, the answer is rounded down.
- Debugging a recursive function can be a challenge unless the programmer has enough information to track the progress of the recursion. To find and eliminate an error, the programmer needs to know the actual arguments in each recursive call. For this purpose, while debugging this program, the programmer inserted an output statement in this function just after the initialization of `mid`.

```
int mid = left + (right-left)/2;
cout <<" left=" <<left <<" mid=" <<mid <<" right=" <<right <<"\n";
```

Second box, Figure 19.10: Check the base cases.
There are two base cases:

- We can guarantee that the recursion will eventually end by testing and eliminating the item at the split point on every call. This guarantees that each successive recursive call will be searching a shorter list of possibilities.
- If the left and right subscripts have met and passed each other in the middle, the key value is not in the array and we return an error code. The symbolic name for this code was defined in the Table header file.
- If the value at `mid` matches the key value, we return the subscript at which the key was found.
- The order of these two comparisons *does* matter. It is possible for `left, right`, and `mid` to be all the same subscript. This happens when the remaining unsearched portion of the array is empty, but also if the array was empty when the search began. If there is no data at all in the array, or if `left, right`, and `mid` are all offboard (past the end of the data), it is an error to test the data at `mid`. So we must test whether data exists first.

Third box, Figure 19.10: Recurse.

- First, we test whether the key is in the bottom or the top half of the remaining area. Then we recurse on one of the halves, never both. We already checked the `mid` value, so that is excluded from both recursions.

Output from the binary search test run with debugging printouts:

```
Enter numbers to search for; period to quit.
What number do you wish to find? 345
345
left=0  mid=8  right=17
left=0  mid=4  right=8
left=5  mid=6  right=8
left=5  mid=5  right=6
left=6  mid=6  right=6
is not in table.

What number do you wish to find? 387
387
left=0  mid=8  right=17
left=0  mid=4  right=8
left=5  mid=6  right=8
was found in slot 6.

What number do you wish to find? 0
0
left=0  mid=8  right=17
left=0  mid=4  right=8
left=0  mid=2  right=4
left=0  mid=1  right=2
left=0  mid=0  right=1
left=0  mid=0  right=0
is not in table.

What number do you wish to find? 967
967
left=0  mid=8  right=17
left=9  mid=13  right=17
left=14  mid=15  right=17
left=16  mid=16  right=17
was found in slot 16.
```

Output from the binary search test run without debugging printouts:

```
Enter numbers to search for; period to quit.
What number do you wish to find? 345
345 is not in table.

What number do you wish to find? 387
387 was found in slot 6.

What number do you wish to find? .
```

We are prepared to begin a binary search of the array `data`, shown below, looking for the key value 270. The `left`, `right`, and `mid` subscripts are shown as they would be after making the first call at the beginning of this search.

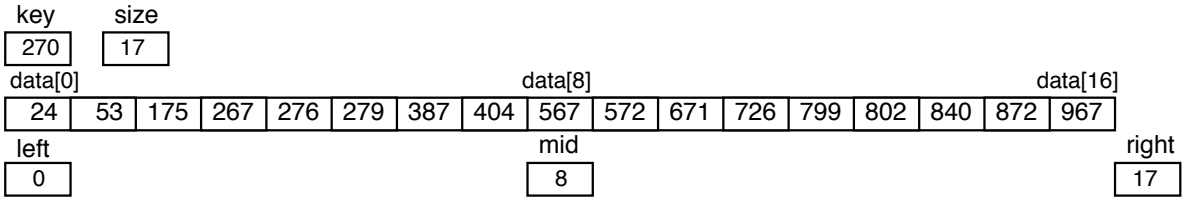


Figure 19.11. Diagrams of a Binary Search: Ready to begin.

Notes on Figure 19.11: Ready to begin.

- Initially `left = 0`, `right =` the number of items in the array, and `mid` is halfway between them.
- Note that all the data slots are colored white. They will become gray when they have been eliminated from scope of the the remaining search.

Notes on Figure 19.12: After the first test.

- After comparing the key value 270 to the value of `data[mid] = 567`, the program knows that the key should be in the left half.
- `bin_search()` is called recursively to search an interval with the same left end but with `right = mid`, since `mid` has been checked and `right` is supposed to be offboard.

Notes on Figure 19.13: After the second test.

- After comparing the key value 270 to the value of `data[mid] = 276`, the program calls `bin_search()` recursively to search an interval with the same left end but with `right = 4`.

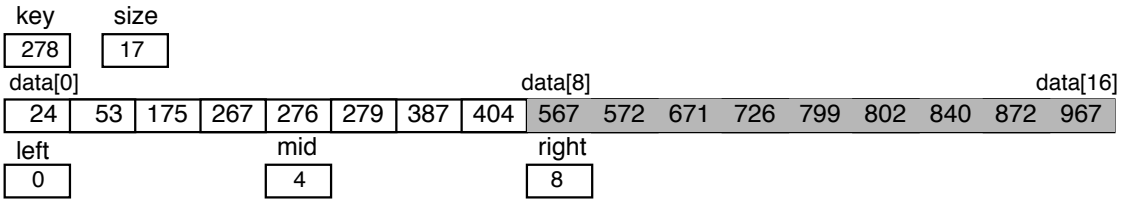


Figure 19.12. Second active call.

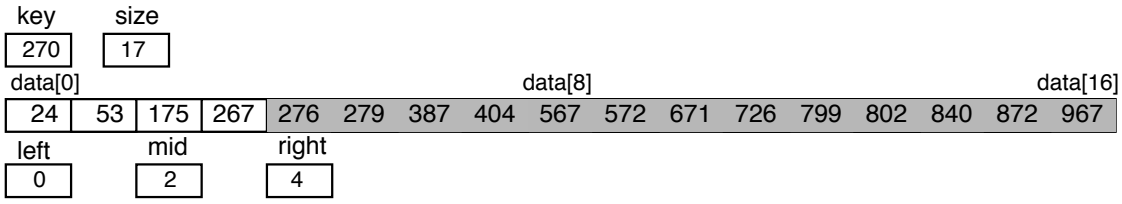


Figure 19.13. Third active call.

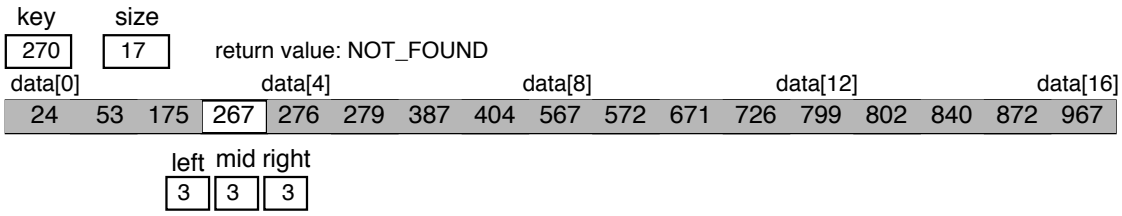


Figure 19.14. Fourth active call.

Notes on Figure 19.14: After the third test.

- On this call, the left end of the remaining interval has been eliminated. The new left end is `mid+1 = 3` and `right` is still 4, so there is one thing left to search.

  The key is 270 does not match the value in that slot 3, so the program returns the failure code.

19.6 Quicksort

Even though computers become faster and more powerful each year, efficiency still matters when dealing with large quantities of data. This is especially true when large data sets must be sorted. Some sorting algorithms (selection sort, insertion sort) take an amount of time proportional to the square of the number of items being sorted and so are only useful on very short arrays (say, 10 items or fewer).[4] Even the fastest computer will take a long time to sort 10,000 items by one of these methods. The fastest known sorting algorithm, **radix sort**, takes an amount of time that is directly proportional to the number of items, but it requires considerable effort to set up the necessary data structures before sorting starts. Therefore, it is useful only when sorting very long arrays of data. In contrast, the **quicksort** algorithm is one of the fastest ways to sort an array containing a moderate number of data items (up to millions).

[4]The theory behind this is beyond the scope of this book.

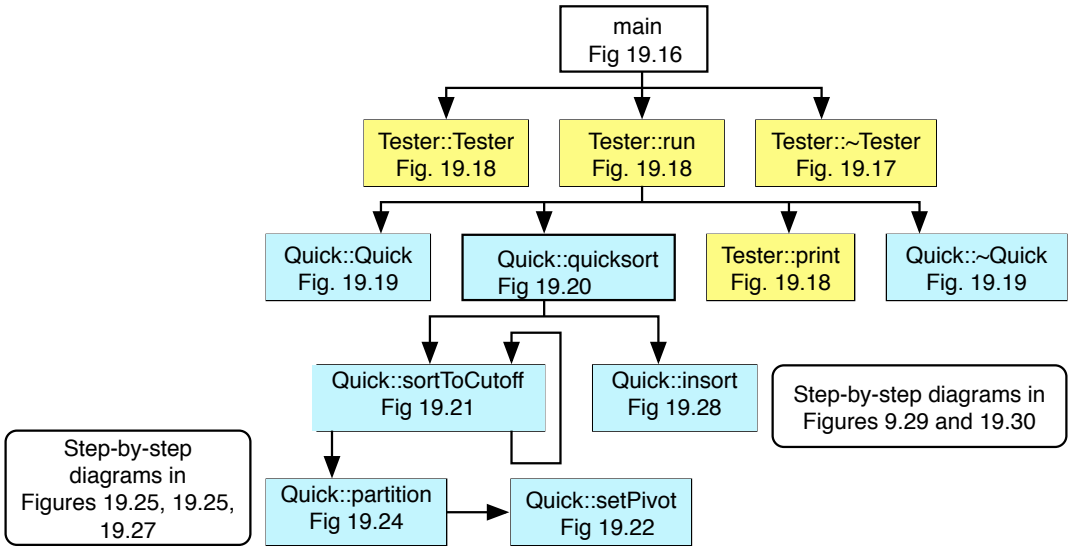


Figure 19.15. Call graph for the quicksort program.

This `main()` program uses the `Tester` class in Figures 19.17 and 19.18.

```
#include "tools.hpp"
#include "Tester.hpp"

#define LENGTH 1000000

// ------------------------------------------------------------------------
int main(void) {
    banner();
    Tester( LENGTH ).run();
    bye();
}
```

Figure 19.16. The main program for `quicksort()`.

Quicksort is an example of the strategy of divide and conquer: Divide the problem repeatedly into smaller, simpler tasks until each portion is easy and quick to solve (conquer). If this can be done in such a way that the collection of small problems actually is equivalent to the original one, then after solving all of the small problems, the solution to the large one can be found. In particular, the quicksort algorithm works by splitting an array of items into two portions so that everything in the leftmost portion is smaller than everything in the rightmost portion, then it recursively works on each section. At each recursive step, the remaining part of the array is split again. This splitting continues until each segment of the array is very small (often one or two elements) and can be sorted directly. After all of the small segments have been sorted, the entire array turns out to be sorted as well, because every step put smaller things to the left of larger things.

There have been many implementations of quicksort, some very fast and others not so fast. In general, if an array of items is in random order, any of the quicksort implementations will sort it faster than a simple sort (selection, insertion, etc.). The version of quicksort presented here is especially efficient and one of the best, because its innermost loops are very short and simple. Further optimizations that can improve the efficiency of this algorithm have been omitted from the code in this section for the sake of clarity. However, two possible improvements are mentioned at the end of the section and explored in the exercises.

As an aid to following this example, a call graph is given in Figure 19.15. It includes references to the figures in which each function is defined or its actions are illustrated. We begin the discussion with the `main()` program in Figure 19.16, which calls the `quicksort()` function in Figure 19.20.

Notes on Figure 19.16: The main program for `quicksort()`.
This is a typical C++ main program: it creates a `Tester` object and calls its run function. The calls on `banner()` and `bye()` let the user track the progress.

The #define at the top makes it easy for us to test the program with any size data set. This is a very fast implementation of the algorithm. Given the amazing speed of today's computers, we need to sort a million numbers to even notice or measure the execution time.

Notes on Figure 19.17. Tester generates data for `quicksort()`.

First box, Figure 19.17: data members.
We need only two data members: a dynamic array and the length of that array. The array is allocated dynamically so that we can test the quicksort with data sets of varied sizes. The length of the array is #defined at the top of main and passed as a parameter to the Tester constructor.

Second box, Figure 19.17: Create the data, sort it, and output it.

This function is called from the `main()` program in Figure 19.16. It generates the required amount of random data and stores it in the data array to be sorted. Then it creates a Quick object to do the sorting.

```
#pragma once

class Tester {
private:
    int length;
    int* data;

public:
    Tester( int len );
    ~Tester() { delete [] data; }
    void run();
    void print() const;
};
```

Figure 19.17. Tester generates data for `quicksort()`.

- The `Tester` constructor generates random data for the test. It allocates a long dynamic array for the data, so it needs to receive the desired array length from `main()`. See the first box in Figure 19.18
- The `Tester` destructor must deallocate the dynamic array. That is a simple one-line task, so the function is declared inline in the .hpp file.
- The `run()` function calls the quicksort function. See the second box in Figure 19.18
- The `print()` function outputs the results to a file, not to the screen, because we expect the results to be voluminous. It is declared as a `const` function because a `print()` function should never modify its object. See the third box in Figure 19.18

Notes on Figure 19.18. Implementation for the Tester class.

First box, Figure 19.18: the constructor.

- The `Tester` constructor uses ctors (constructor initializers) to initialize the two data members of the new object. The ctor list starts with a colon after the parameter list of the constructor and before the open brace at the beginning of the body. The ctors are always executed before the body of the method.
- The first ctor, `length(len)` stores the parameter named len into the data member named length.
- The second ctor, `data(new int[len])` allocates a long dynamic array for the data, and stores the pointer into the class member named `data`.
- The body of the constructor is a loop that generates random numbers and stores them in the data array.

Second box, Figure 19.18: Test the quicksort. The `run()` function prints user information, calls the `quicksort()` function, then writes the sorted data to a file by calling `print()`.

Third box, Figure 19.18: Print to a file.

- Lines 1, 2, and 4 of the `print()` function are needed to open a file for the output, verify it, and close it.
- Line 3 is a simple loop to do the output. Note that is is very much like the loop that generated the data.

Notes on Figure 19.19: the Quick class.

First box, Figure 19.19: the data members. To sort, we need to know the where the data begins and how much data we have. These are passed into the constructor by the `Tester::run()` and stored in member variables. They are accessed by every function in the class.

Third box, Figure 19.19: the public class interface.

- There are only two public functions: the `Quick` constructor and `quicksort()`. All other work is done by private helper functions.
- The constructor is an inline function with no body. Its uses ctors to initialize the two data members. The ctors must be given in the same order as the definitions of the class members.

Second box, Figure 19.19: the private functions.

- `setPivot()`: On each recursion, a quicksort algorithm chooses a pivot value and puts it in place in the array.
- `partition()`: Then the `partition()` function separates the large and small data items. Items that are smaller than the pivot are moved to the low-subscript end of the array. Items that are larger than the pivot are moved to the high-subscript end. Items equal to the pivot might end up in either portion. After

---

This function is instantiated by the `main()` program in Figure 19.16. It generates the required amount of random data and stores it in the data array to be sorted. Then it creates a Quick object to do the sorting.

```
#include "Tester.hpp"
#include "quick.hpp"
#include "tools.hpp"

// Allocate and fill data array with random non-negative integers.
Tester::
Tester( int len ) :  length(len), data(new int[len]) {
    for (int k=0; k<length; k++) data[k] = rand();
}

// Run test of quicksort on random data array ------------------------
void Tester::
run() {
    cout << "\ Quicksort program, cutoff=" << CUTOFF << ".";
    cout << " " << length << " data items generated; ready to sort.\n";
    Quick( length, data ).quicksort();
    cout << " Data sorted; writing to output stream.\";
    print();
}

// Print array values to selected stream.  ---------------------------
void Tester::
print() const {
    ofstream out("sorted.txt");
    if(!out.is_open()) fatal( "Cannot open output stream %s", "sorted.txt" );
    for (int k=0; k<length; k++) out << data[k] << endl;
    out.close();
}
```

Figure 19.18. Implementation for the Tester class.

```
#pragma once
#include "tools.hpp"

#define CUTOFF 20

typedef int BT; class Quick {
private:
    BT* data;   // Array containing data to be sorted.
    int n;      // Number of items to be sorted, initially

    // Prototypes of private, non-inline functions.
    void insort(BT* begin, BT* end);
    BT setPivot(BT* begin, BT* end);
    BT* partition(BT* begin, BT* end);
    void sortToCutoff(BT* begin, BT* end);

public:
    Quick( int n, BT* d ) :  data(d), n(n) {}
    void quicksort();

};
```

Figure 19.19. The Quick class.

examining all array elements, and moving them to the right array area, the pivot value is placed between the two areas.

- `sortToCutoff()` is the actual recursive sorting function. The recursion ends when the number of data items remaining to be sorted no longer justifies the overhead inherent in doing recursion. This leaves the data items almost, but not quite, sorted.
- `insort()` is an insertion sort that is used after the cutoff to adjust the positions of the almost-sorted items.

Notes Figure 19.20: The quicksort() function.
This function is called from the Tester::run() function in Figure 19.18. In turn, it calls the functions in Figure 19.21 and 19.28.

- The quicksort algorithm is much faster than any double-loop sorting algorithm if the number of things to be sorted is large. However, insertion sort is faster if there are only a few things to sort.
- In the code, there are two function calls. The first calls a recursive function, `sortToCutoff()` to subdivide the data set into a series of 1 to CUTOFF chunks. When it finishes, all of the data values in each chunk

```
#include "quick.hpp"

void Quick::
quicksort() {
    sortToCutoff(&data[0], &data[n]); // Recursively sort down to CUTOFF-sized blocks.
    insort(&data[0], &data[n]);       // Use insertion sort to finish the sorting
}
```

Figure 19.20. The quicksort() function.

```
// Recursively sort down to CUTOFF-sized blocks. -----
void Quick::
sortToCutoff(BT* begin, BT* end) {
    if (end - begin > CUTOFF) {
        BT* split = partition( begin, end );
        sortToCutoff( begin, split );
        sortToCutoff( split+1, end );
    }

    // else there is nothing more to do.  Just return.
}
```

Figure 19.21. sortToCutoff ()

are smaller than all the data values in the next chunk. That is, the array is mostly sorted.

- Then `insort` (an ordinary insertion sort) is called to finish sorting the items within each chunk. A CUTOFF of 20 to 25 was determined experimentally. Machine architecture and OS cacheing strategies make this larger or smaller on different systems.

Notes Figure 19.21: sortToCutoff().

First box, Figure 19.21: `sortToCutoff()`

- This function is called with a pointer to the beginning and end of the portion of the array that will be sorted.
- Subtracting the beginning pointer from the end pointer uses pointer arithmetic. The result is the number of items in this part of the array. If it is > the CUTOFF, processing continues, otherwise the call returns without doing anything and the recursion ends.
- If there are still > CUTOFF items to sort, we call the `partition()` function to divide the data into to blocks.
- When `partition()` finishes, all the data in the left-hand block will be <= *pivot*, all the data in the right-hand block will be >= *pivot*, and the pivot value will be stored in the array between the two blocks.
- The pivot value is in its final, sorted, position, and the remaining data items are split into two blocks that can be sorted independently and will not mixed or merged later.
- At this point we call `sortToCutoff()` recursively, twice, to sort the lower and upper blocks.

Notes on Figure ??: Setting the pivot.

There are many implementations of the general quicksort strategy – some are very fast, some are much slower. This implementation is one of the fastest. The most important improvements are in the `setPivot()` and `partition()` functions.

First box, Figure ??: Identify 3 candidates for being the pivot.

- We want the pivot be the median of 3 data values, chosen from different parts of the data array. This helps to avoid worst-case behavior.
- For two of the three data values, we use the first and last in the array (∗*begin* and ∗*last*). Using the first and last values avoids worst case behavior when the data is nearly sorted but ends with some unsorted values.
- The third value is taken from the middle of the array (∗*mid*). If the array is nearly sorted, this value should approximate the actual median value.

This function is in the file quick.cpp. It is called from `sortToCutoff()` in Figure 19.21.

```
#include "quick.hpp"
//-----------------------------------------------------------------------------
// Choose a pivot value, use the median of the first, last, and middle values.
// Place the median at the beginning of the array (*begin).
// Put the maximum value in *last and the minimum in *mid.

BT Quick::
setPivot(BT* begin, BT* end) {
    BT* last = end - 1;                       // last is onboard, end is offboard.
    BT* mid = begin + (last - begin) / 2;     // Find the middle.

    if (*mid   > *last)  swap(*mid,   *last); // Swap larger into end position.
    if (*begin > *last)  swap(*begin, *last); // Maximum item is now in place.
    if (*begin < *mid)   swap(*begin, *mid);  // Put median in slot at beginning.

    return *begin;
}
```

Figure 19.22. Setting the pivot.

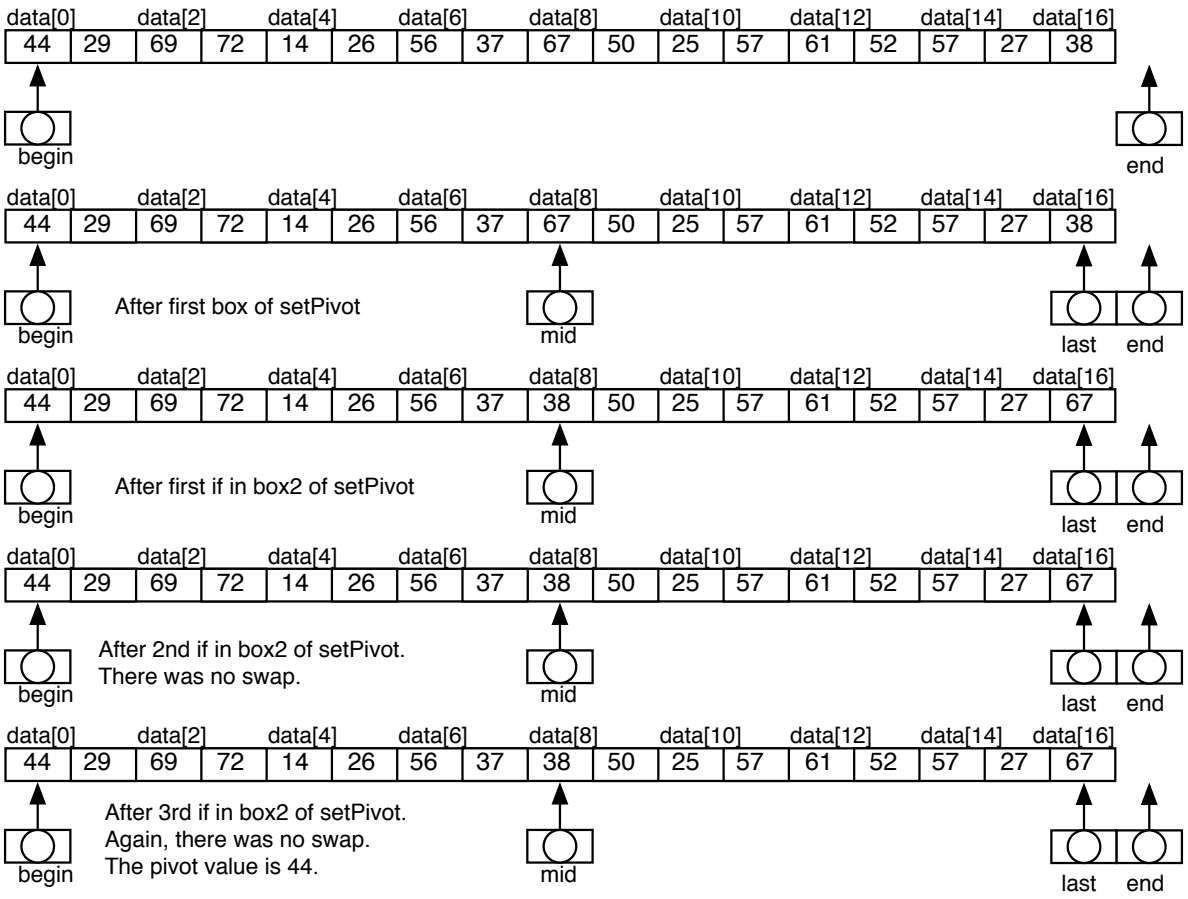


Figure 19.23. A diagram of setPivot().

Second box, Figure ?? set up sentinel values for the partitioning scans.

- During the partition step, we want the pivot to be at the beginning of the part of the array we are sorting.
- We use three `if` statements to compare $*begin$, $*mid$, and $*last$ to find the median. In the process, we also move the largest to the right end of the array and move the pivot to the left end.
- After the second `if` statement, we know the largest value of the three is in the last array slot. The third `if` compares the other two values and puts the median value into slot `begin` to be used as the pivot.
- The pivot and the maximum value are now at the two ends of the array. These two values act as sentinels when the array is scanned. The ensure that the scanners do not "fall off" the ends of the array.
- Without a sentinel, the inner loop must compare two data items *and* check whether it has reached the end of the array. Using a sentinel makes the second check unnecessary and significantly speeds up the loop.
- The sentinel on the right end of the array can be any value `>= pivot`, because this will end the comparison loop for the scanner named `luke`, that starts at the left and moves to the right. (Luke likes little things and throws the big things out of his area.)
- The sentinel on the left end of the array is a copy of the pivot value, because this will end the comparison loop for the scanner named `rose`, that starts at the right and moves to the left. (Rose relishes robust things and throws the little things out of her area.)

Third box, Figure ??: returning the pivot.
This function is called from `partition` and returns the value of the pivot to `partition`, where it is stored in a local variable named `pivot`.

Notes on Figure 19.23: The setPivot() function.
In this diagram, we walk through each step of the setPivot() function the first time `quicksort()` is called on the array shown.

- First line: note that `end` is an off-board pointer and the last data is at slot `end-1`.
- Second line: using integer arithmetic, $(17 - 0)/2 = 8$ so slot 8 is the middle of the array.
- Now we use three compare-then-swap instructions to find the median of three numbers, 44, 67, and 38.
- Third line: We compare the mid and last values (67 and 38) and put the larger (67) in the last slot.
- Fourth line: Now we compare the first and last values (44 and 67). Since the larger value (67) is already in the last slot, no swap is needed. This happens half of the time.
- Now we know that the last slot contains the maximum, but we do not yet know whether the minimum is in slot `begin` or the middle slot.
- Fifth line: So we compare those values and swap, if necessary, to put the median value in slot `begin`. In this case, no swap is needed.
- The value now at the beginning of the array will be the pivot value for the partition step.

Notes on Figure 19.24: The `partition()` function, the heart of quicksort().
Most implementations of partition use the variable names `i` and `j`. However, we use the variable names `luke` and `rose` because the reader is less likely to get mixed up about which scanner is moving in which direction.

First box, Figure 19.24: set the pointers and the pivot.

- Most of the execution time of `quicksort()` is spent in the `partition()` function. This implementation is very fast because the two partitioning loops do very little work.
- To speed execution, we use pointers instead of subscripts. We set pointers named `luke` and `rose` to slot `begin` of the array and to the last element in the array.
- After calling `setPivot()`, both of these array positions will contain sentinel values that have already been checked during the pivot computation. Note that this call also returns the pivot value, which is stored in the variable `pivot`.

```
#include "quick.hpp"

// Move smaller items to left end of the array, larger ones to the right end.
BT* Quick::
partition(BT* begin, BT* end) {     // end is an off-board end pointer.
    BT* luke = begin;                    // Luke likes little things.
    BT* rose = end - 1;                  // Rose starts on-board at the end.
                                         // Rose likes big, robust things.
    BT pivot = setPivot(luke, rose);     // Now pivot element is in *begin

    for (;;) {       // Loop until Luke meets Rose somewhere in middle of array.
        while (*++luke < pivot);         // stops at begin bith thing, or *last
        while (*--rose > pivot);         // stops at 1st little thing, or *begin

        if (luke >= rose) break;         // L and R have met in the middle.
        swap ( *rose, *luke);            // Little goes on left, big on right.
    }
    // Now, left of rose all is < pivot and right of Rose, all is > pivot.
    *begin = *rose, *rose = pivot;       // So swap pivot to rose's position.
    return rose;                         // Return subscript of split-point.
}
```

Figure 19.24. The `partition()` function, the heart of quicksort().

- So `pivot` holds a copy of the value in slot `begin` of the array. Using the copy speeds up the comparisons during the inner scan loops because no dereference or subscript operation is needed.

Second box, Figure 19.24: scanning.

- The heart of a quicksort is the **partition step**, which splits the array into two parts, separated by one item at the split point. The split is done in such a way that everything in the left section is smaller than or equal to the value at the split point, and the split value is smaller than or equal to everything in the right section.
- In most implementations, the task is performed by the function named `partition()` (Figure 19.24). The value returned by `partition()` is a pointer to the split position. This state is shown in the example in Figure 19.22. This slot is gray to indicate that the value stored there is in its final, sorted position.
- Before every iteration, both scanners are pointing at values that have already been compared. So the first action is to increment or decrement the scanner.
- The two scanning loops in the second box have no loop body. The increments that are part of the while-test do all the necessary work. Because there are sentinels on the ends of the array, we do not need to test for the end of the array: the sentinel will stop the loop.
- When `luke` stops scanning, he is pointing at a data that is too big to be in his part of the array. When `rose` stops, she is pointing at a data value that is too small for her.

Third box, Figure 19.24: partitioning.

- When control gets to the third box, there are two possibilities and two possible next actions.
  - Luke is still to the left of rose, so they have not yet looked at all the data in the array. They swap data items and continue with the scan.
  - They have crossed, or are sitting on the same array slot. This means that they have examined all the data in the array and they are done. So they break out of the outer loop.

Fourth box, Figure 19.24: put the pivot in place.

- When `luke` and `rose` cross, `rose` is pointing at an item that is acceptable on the left (small value) end of the array.
- This value is swapped with the pivot value, which is in slot `begin`.
- Now the pivot value is in its final resting place and the data items have been partitioned into smaller and larger sets. Partition returns `rose`, a pointer to the split-point. Then `sortToCutoff()` uses that pointer to calculate parameters for the next pair of recursions.
- If the data set has several copies of the pivot value, some copies are likely to end up in both halves of the array. That is OK. The algorithm still works.
- If the scan stops with `luke` and `rose` pointing at the same slot, that slot contains a copy of the pivot. That is OK.

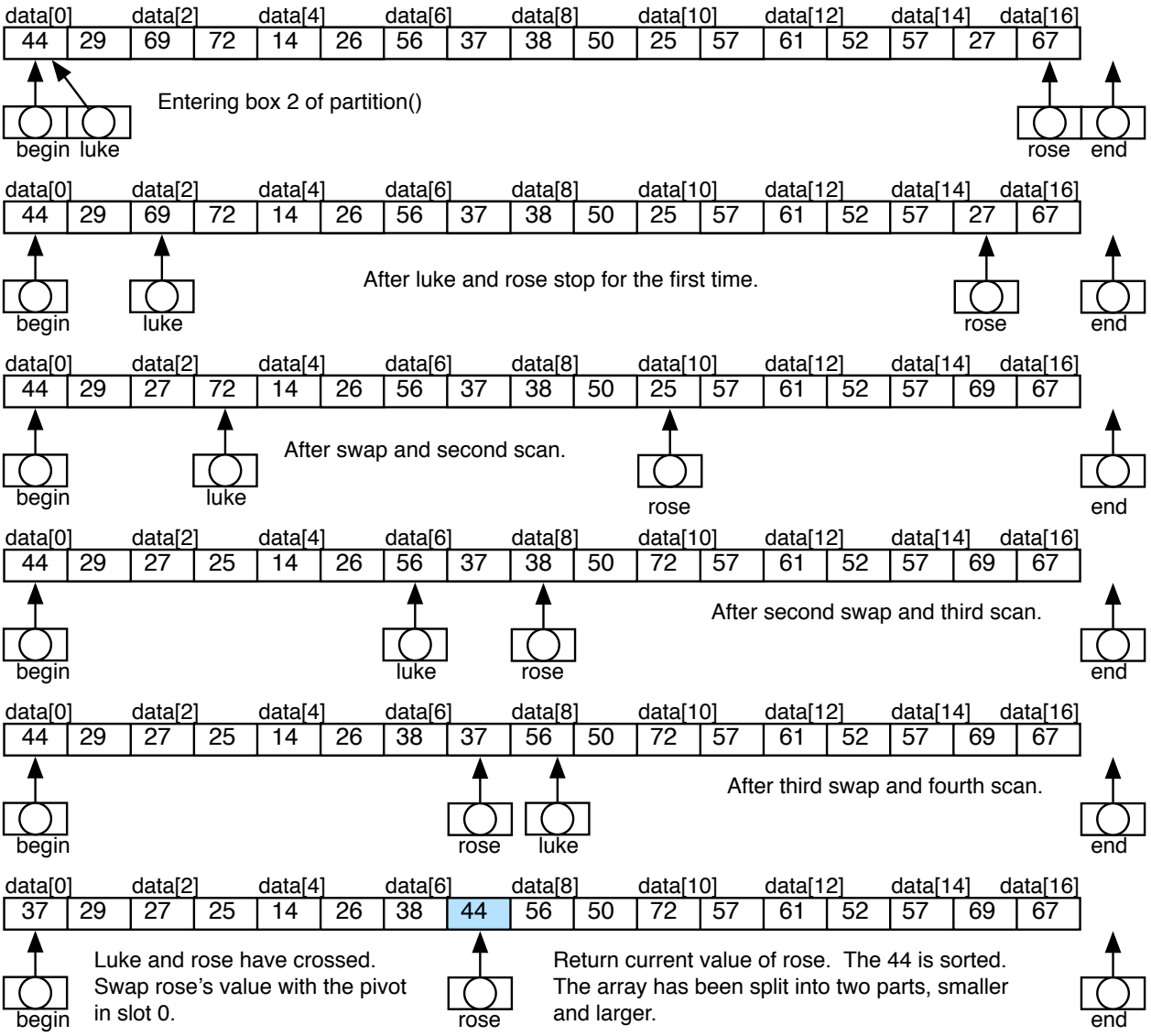
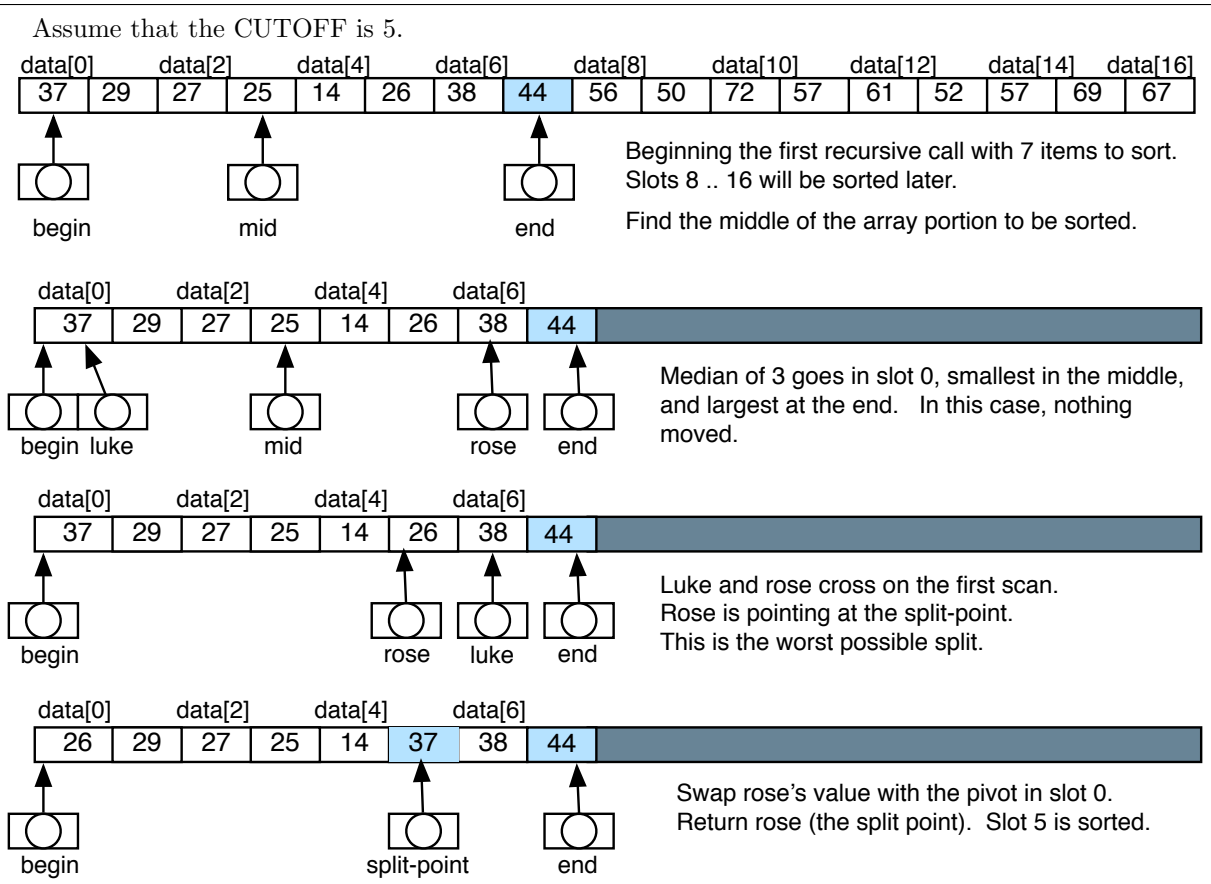


Figure 19.25. The first pass through partition.

- Every recursive call on `sortToCutoff()` starts with one big block of data items and separates them into two smaller blocks separated by a sorted pivot value. The most desirable division, of course, is an even split, which rarely happens with random data. That is OK.
- It is possible for the split point to be anywhere in the array interval, except for slot `begin` and the last slot. The split point cannot be there because we put a value `>=pivot` at the right end, and we know that a value `<=pivot` will end up at the right end. That is OK. This might cause an additional recursion, but the algorithm will still work.
- When the last recursive call on `sortToCutoff()` returns, the data is approximately sorted. One trip through insertion sort fixes the little problems that remain.

Notes Figure 19.28: Finish sorting using insertion sort.
This function is called by `quicksort()`, Figure 19.20.

- We improve the efficiency of quicksort by terminating the recursions before the data is fully sorted, when the number of items left to sort is small enough so that insertion sort is faster. This is because every recursive call builds (and later tears down) a stack frame, while loops simply increment a counter in the current stack frame.



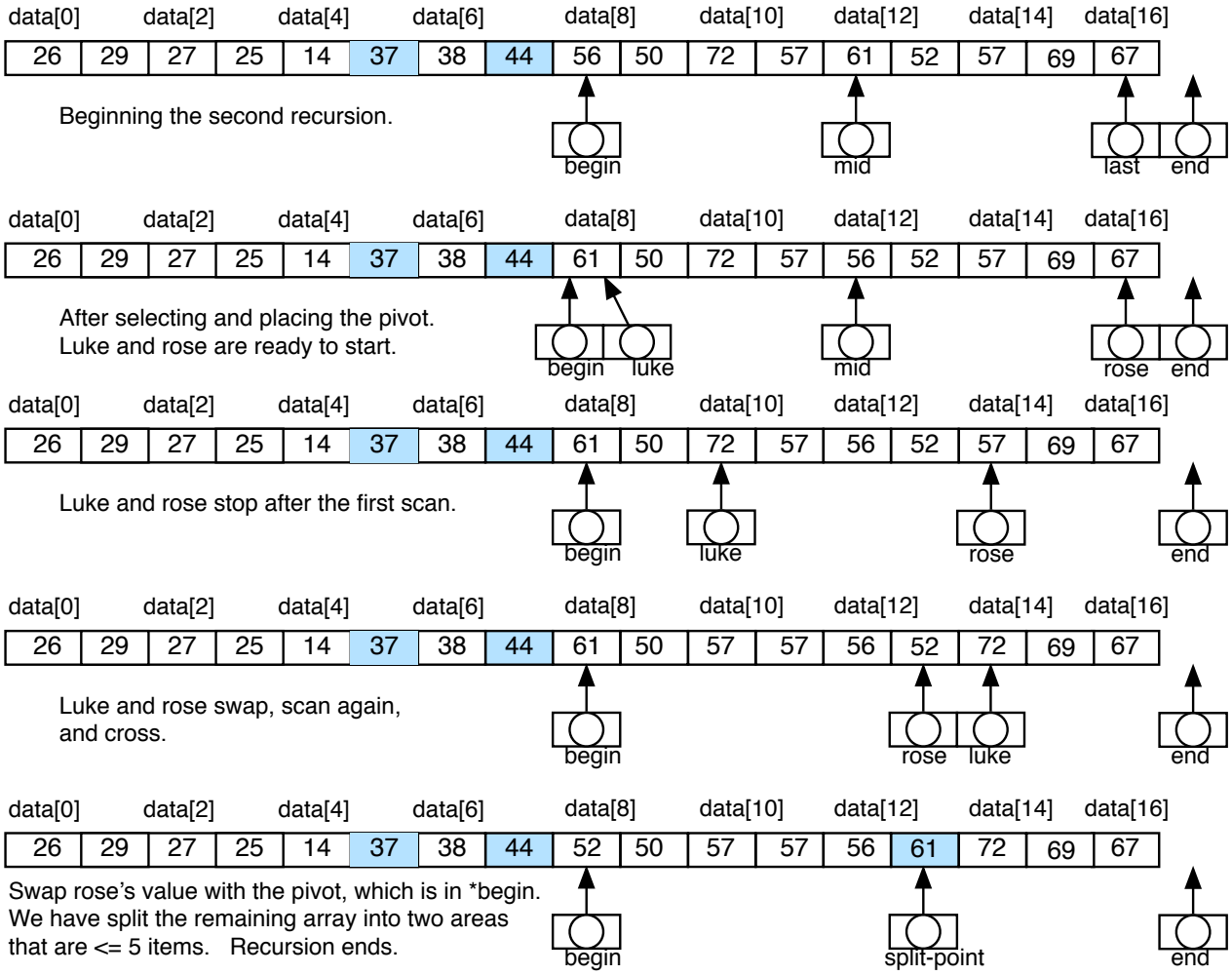
The areas to both left and right of the split point now contains five or fewer slots. Recursion ends because five is the recursion cutoff. The left end of the array is now nearly sorted and we start the recursive calls on the right end of the array.

Figure 19.26. The first recursive call.

- The best cutoff may be different for different platforms. For old single processor machines, the threshold was 8 to 10 items.
- For modern multi-core machines with large on-board caches, the threshold seems to be 20 to 30 items.
- We can sort a small set of items (below the threshold) fastest using an insertion sort.
- By doing so, we eliminating the overhead of recursion and shorten sorting time. However, adding the insertion sort does make the code longer.

First box, Figure 19.28: Set up three pointers for sorting.

- `fence` is the subscript of the last unsorted data item. Items in subscripts greater than the fence are sorted. We set it to subscript $N - 1$, since one thing is always sorted.
- If `fence < begin`, there is nothing left to sort and we end the sort loop.
- Otherwise, we set `newcomer = *fence`, the data at the fence.
- Now set `hole = fence` because we have copied the data and it is OK to move some other data item into that hole.



After one recursive call, there are five or fewer items on both sides of the split-point and recursion ends. The entire array is now approximately sorted. We end the recursions and use insertion sort to finish.

Figure 19.27. Finishing the recursions.

```
// Use insertion sort to finish the sorting ----------------------------
void Quick::
insort( BT* begin, BT* end ) {
    BT* hole;         // currently empty location
    BT* k;            // location currently being compared to newcomer
    BT* fence;        // last location in unsorted part
    int newcomer;     // element being inserted into sorted part

    for (fence = end - 2; fence >= begin; fence--) {
        hole = fence;
        newcomer = *hole;
        for (k = hole+1; k != end && newcomer > *k; ) {
            // Move data from slot k into the hole and ++ both indices
            *hole++ = *k++;
        }
        *hole = newcomer;
    }
}
```

Figure 19.28. Finish sorting using insertion sort.

- Finally, a pointer `k` is set to the array position just to the left of the hole.
- We are now ready to begin the outer loop of the sort.

Second box, Figure 19.28: The double-loop insertion sort.

- In the beginning, the last item in the array is considered sorted and the second-to-last item is the first newcomer.
- To sort N items, insertion sort makes $N - 1$ passes over the sorted part of the array. Each pass inserts one new item (the newcomer) into the sorted part.
- A pass might require only one comparison, or it might continue to the end of the array. On the average, it goes half-way to the end.
- After each trip through the inner loop, the newcomer is stored in the hole, wherever that might be.
- Note that this uses one assignment per data-move. This is much better than doing a swap, which uses three assignments.
- Control now goes back to the top of the loop, where the fence is decremented and the sorted portion of the array is lengthened by 1. Then `*fence` is copied into `newcomer` and `hole` is set `=fence`.

Inner box, Figure 19.28: the insertion loop.

- We insert a newcomer by moving data to the left in the array and moving the hole right, until the hole is where the newcomer belongs.
- The code is concise to the extreme: `*hole++ = *k++;`
  - This is the entire action of the loop!
  - `*hole = *k` moves the data from subscript `k` to subscript `hole`.
  - The two pointers progress in tandem. They are POST incremented, that is, they are incremented after the data is moved.
  - We really do not need two pointers here. However, using `k` eliminates the need to be constantly computing `hole-1`.
- The loop ends when the scanning pointer, `k`, reaches the end of the array or finds the insertion spot for the newcomer.
- When the insertion slot is found, the newcomer is stored in the hole.

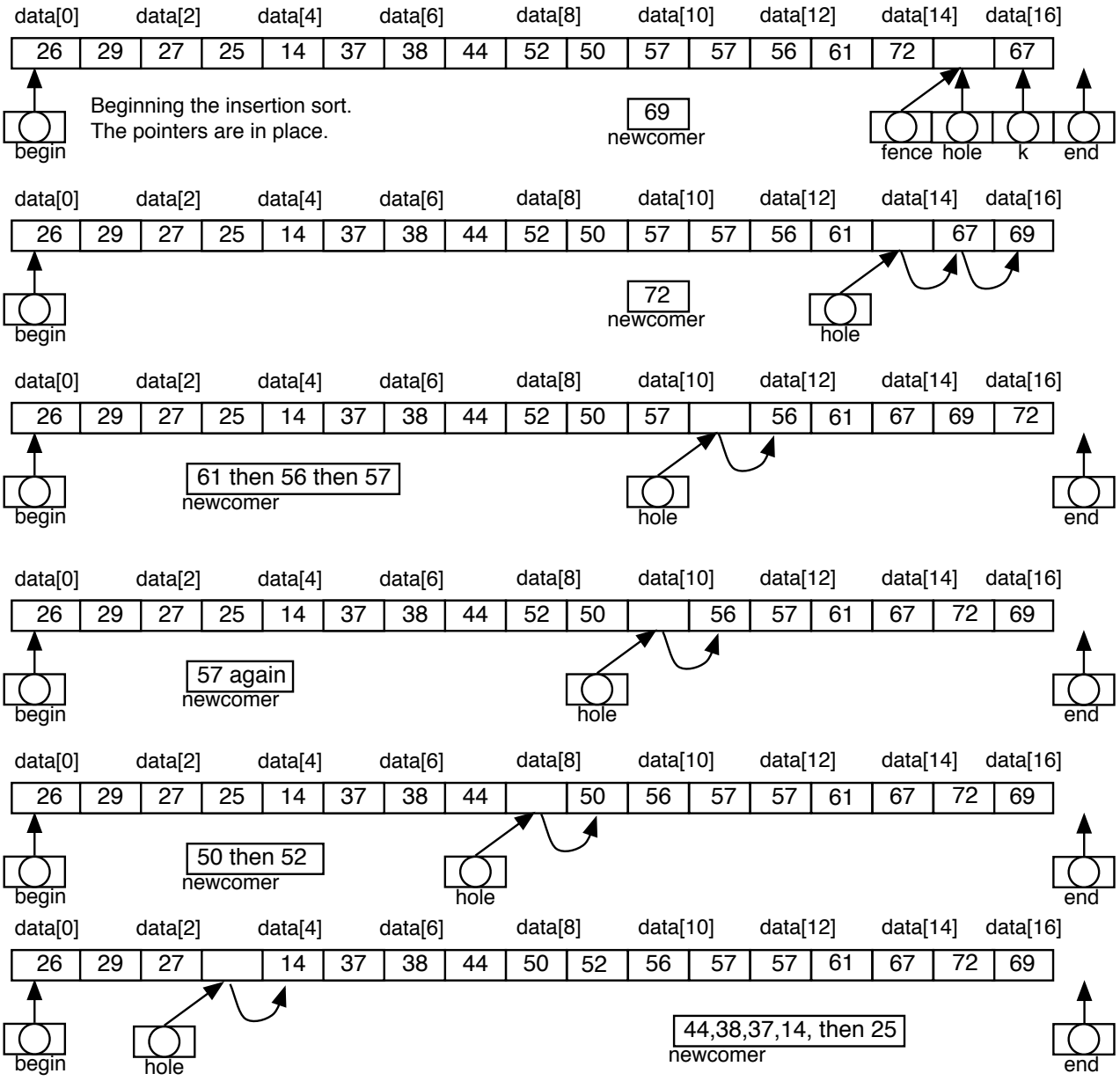


Figure 19.29. Insertion sort, first part.

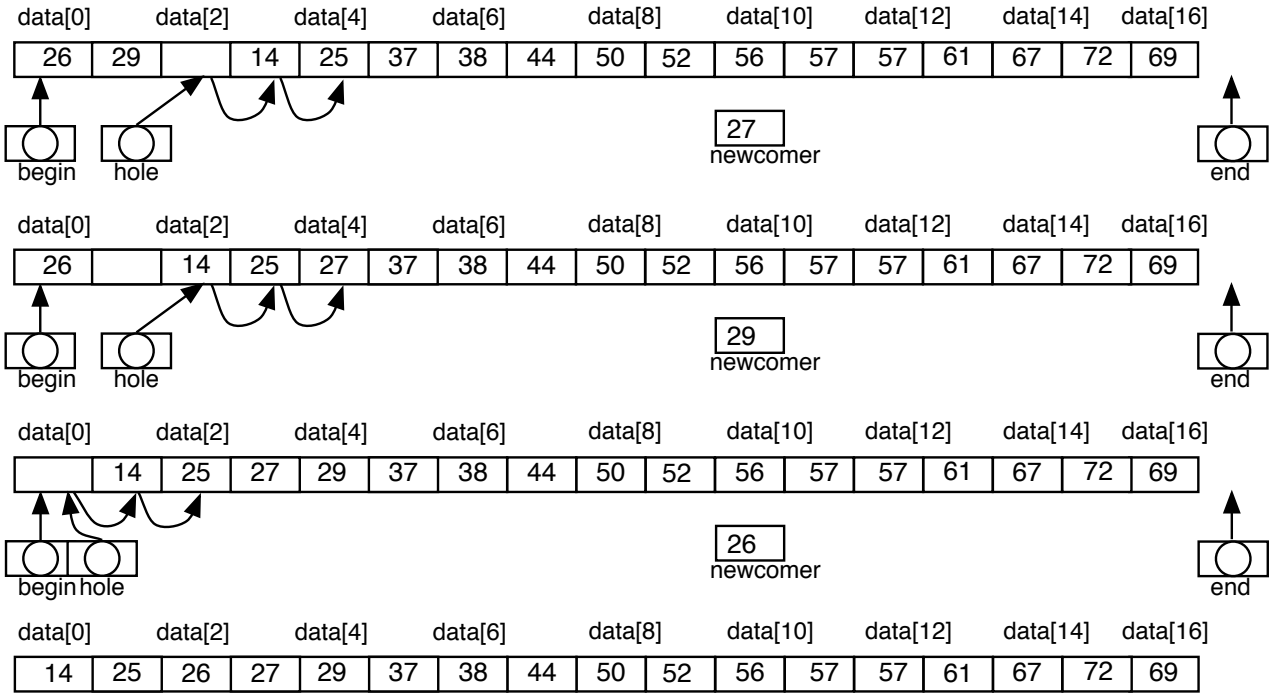


Figure 19.30. Insertion sort, second part.

19.6.1 Possible Improvements

The specific implementation just presented has been compared extensively to other variants of quicksort. It performs better in timed tests than everything else we have tried, for arrays holding tens to millions of numbers. We know of one possible significant improvement.

Execution time can be saved by using only one recursive call in `sortToCutoff`. The function would then be programmed as a loop, and instead of the second recursion, the program would assign the parameter values to local variables and go around the loop again. The loop would end when the data size was reduced to the CUTOFF.

19.7 What You Should Remember

19.7.1 Major Concepts

Storage class. C/C++ supports three useful storage classes: `auto`, `static`, and `extern`. Each storage class is associated with a different way of allocating and initializing storage, as well as different rules for scope and visibility.

- `auto` *storage.* Parameters and most local variables have storage class `auto` and are stored on the run-time stack. These objects are allocated and initialized each time the enclosing function is called and deallocated when the function completes.
- `static` *storage.* Global variables and `static` local variables are allocated and initialized when the program is loaded into computer memory and remain until the program terminates. The `static` variables can be used to store the state of a function between calls on that function. They also provide an efficient way to store a constant table that is local to a function.

- `extern` *variables.* Large applications composed of multiple code modules sometimes have `extern` variables. These are defined and initialized in one module and used in others. They remain active until the program terminates. Functions are labeled `extern` by default, but may be declared to be `static` if their scope should be restricted to one code module.

Run-time stack. The state of a program is kept on the run-time stack. Every active function has a stack frame that contains its parameters, local variables, and a return address. All aspects of a function call are handled through communication via the stack. Drawing stack diagrams to use while performing a program trace is one method of debugging your software.

Recursion. A recursive function calls itself. Using the recursive divide-and-conquer technique, we can solve a problem by dividing it into two or more simpler problems and applying the same technique again to these simpler tasks. Eventually, the subproblems become simple enough to be solved directly and the recursive descent ends. The solution to the original problem then is composed from the solutions to the simpler parts. A storage area that can grow dynamically, such as the run-time stack, is necessary to implement recursion, because multiple activation records for the same function must exist simultaneously.

Binary search. Binary search is the fastest way to search a sorted list using a single processor. It can be implemented using either tail recursion or iteration. The strategy is to split the list of possibilities in half, based on the relative position of the key to the middle item in the remaining search range. This is done repeatedly, until we either find the desired item or discover that it is not in the list.

Quicksort. This is one of the best algorithms for sorting a moderate number (up to millions) of items. The algorithm uses a recursive divide-and-conquer strategy, sorting the data by repeatedly partitioning it into a set of small items on one end of the array and a set of large items on the other end, then recursively sorting both sets. Each partition step leaves one more item in its final sorted position between the sets of smaller and larger items.

19.7.2 Programming Style

- Streamline `main()`. A main program in any object-oriented language should display a greeting, create an object, call its primary function, and contain termination code. All file handling and computation should be done by class functions. This improves the modularity and overall readability of the program.
- Use private class members to model real-world objects and to store information that is calculated by one class function and used by others.
- Use local variables to store data that is used in only one function.
- Do not use global variables!
- Use `static` local variables (not globals) to implement constant tables and remember state information from one call on a function to the next. Keep the constants that belong to a function within it if possible, otherwise, define them as class members.
- Failing to check for errors at every possible stage is irresponsible. This may mean checking for invalid input parameters, checking the return values of functions, and looking for invalid operations like dividing by 0. Try to think of all the special cases your program may deal with and devise some sort of response.
- Eventually, it is important to write efficient code. However, it is more important to provide good user-information and even more important to write code that works properly. Many people spend too much time worrying about a program's efficiency and too little time worrying about getting a correct answer or appropriate error comments under all conditions.

- In general, if a problem can be solved using either iteration or tail recursion, a loop will be more efficient because it does not incur the overhead present in multiple function calls. In many situations, the recursive solution may be the most natural to write. It then can be converted to an iterative form if speed is paramount. However, for algorithms that follow a divide-and-conquer strategy, there is no easy way to avoid recursion.
- Certain functions make assumptions about the nature of the input parameters. For instance, the binary search function assumes that the data are sorted in ascending order. If you write such a function, you must write a comment explaining the precondition that you rely upon. Before your code calls such a function, you should validate such assumptions.
- Program efficiency can be improved most by looking at the amount of work done in the innermost loops. The less unnecessary work done in a loop that is executed many times, the faster the program will run. For instance, removing a test from the scanning loops of `partition()`, by using a sentinel value, makes the quicksort explained here very efficient.
- Don't use an inefficient tool if there is no need. For example, if the data array already is sorted, use a binary search rather than a linear sequential search. And, for moderate-sized data sets, use quicksort rather than insertion sort or selection sort. For small data sets, insertion sort usually is best.
- Reuse previously debugged code whenever you can.

19.7.3 Sticky Points and Common Errors

Recursion. Recursion is a technique that can be difficult to understand at first. However, once programmers understand it, they wonder what the fuss was about. To make it easier to trace a recursive program, pretend that, each time the recursive function calls itself, the new invocation has the same name but with a number appended. By having names with numbers you may be able to keep track more easily of what is going on.

Infinite recursions. To avoid an infinite recursive descent, a recursive algorithm must reduce the "size" of the remaining problem on every recursive step. Degenerate and basic cases must be identified and handled properly to ensure that the stopping condition eventually will become true. Just as it can be tricky to get the limits of a loop correct, a common error with recursion is to stop either one step too early or one too late.

Storage types. Probably the trickiest storage class to use is `static`. Mixing up `static` and `extern` can cause linker errors. Confusing `static` with `auto` may cause state information to be lost or extra copies of constants to be stored in memory.

19.7.4 New and Revisited Vocabulary

These are the most important terms and concepts discussed in this chapter:

storage class	run-time stack	recursive descent
`auto`	stack frame	infinite recursion
`extern`	activation record	stack overflow
`static`	parameters	list recursion
static visibility	local variables	tail recursion
static allocation area	return address	divide and conquer
state	stack diagram	binary search
initialization time	active call	split point
table of constants	iteration	quicksort
external linkage	recursion	pivot
program trace	base case(s)	partition step
	recursive step	

19.8 Exercises

19.8.1 Self-Test Exercises

1. Describe the differences between `extern` and `static` storage classes. Discuss the differences between `static` and `auto`. Try to compare `extern` and `auto` as well.
2. Sort this data set by hand into ascending order: 44, 56, 23, 14, 9, 21, 31, 8, 19, 14
   Then show how the values of `left`, `right`, and `mid` change when using the binary search algorithm to look for the value 23. Repeat this for the value 22. Use a stack diagram to show the changing parameter values for each call.
3. The following program computes the sum of the first N numbers in the array `primes`. Trace the execution of this program, showing the stack frame created for each recursive call, the values stored in the stack frame, and the value returned.

```
#include <stdio.h>

#define N 5
int sum( int ar[], int n );

int main( void )
{
    int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 };
    int answer;

    printf( "\nCalculating sum of first %i primes\n", N );
    answer = sum( primes, N );
    printf( "Sum = %i\n\n", answer );
}

// ---------------------------------------------------------------
int sum( int a[], int n ) {
    if (n == 1) return a[0];
    else return a[0] + sum( &a[1], n-1 );
}
```

4. The binary search function in Figure **??** has three `if` statements. These three statements could be arranged in six different orders. Say whether or not the recursion will work and terminate properly for each of the five other possibilities and explain why.
5. Take the sorted data set in Figure 19.11 and scramble it up by hand. Then show the sequence of calls on `quick()` when using the quicksort algorithm to sort the data back into ascending order. Use a stack diagram to show the changing parameter values for each call. Assume a cutoff of 3.
6. The quicksort algorithm in this chapter reorders the left half of the array, then the right half. If the order of these two recursive calls is changed, will the algorithm still work? If so, prove it. If not, explain why not.
7. The following function iteratively computes the product of the first `n` elements of the array `a`. Write a recursive version of this function. Model your solution after the `sum()` function shown in exercise 4.

```
long int product( int a[], int n )
{
    long int product = 1;
    for (int k = 0; k < n; k++) product *= a[k];
    return product;
}
```

19.8.2 Using Pencil and Paper

1. Consider changing the binary search algorithm in Figure **??** to work on a data set in descending order. Indicate what lines need to be changed and how.
2. Sort this data set by hand into descending order: 19, 17, 2, 43, 47, 5, 37, 23, 41, 3, 29, 31, 7, 11, 13. Then show how the value of `mid` changes when using the new binary search algorithm developed in the previous exercise. Look first for the value 3, then for the value 4. Use a stack diagram to show the changing parameter values for each call.

3. Consider changing the quicksort algorithm in this chapter so that the data set produced is in descending order. Indicate what lines need to be changed and how.
4. Scramble the sorted data set in Figure 19.11 by hand. Then show the sequence of calls on `quick()` when using the new quicksort algorithm developed in the previous exercise to sort the data into descending order. Use a stack diagram to show the changing parameter values for each call.
5. Write a recursive function that computes the value of `n!`. The definition of factorial was given in Chapter 7, Section 8.2.1.
6. Write a recursive function that computes the value of `n!`. The definition of factorial was given in Chapter 7, Section 8.2.1.
7. The following program does some string manipulation. Trace its execution, showing the stack frame created for each recursive call, the values stored in the stack frame, and the value returned.

```
#include <stdio.h>
#define N 3

char* behead( char* s, int n );

int main( void )
{
    char word[] = "distrust";
    char* answer;

    printf( "\nBeheading the victim:  %s?\n", word );
    answer = behead( word, N );
    printf( "Now:  %s!\n\n", answer );
}

// ---------------------------------------------------------------
char* behead( char* s, int n )
{
    if (n == 0) return s;
    else return behead( s+1, n-1 );
}
```

8. Write an iterative version of the `bin_search()` function in Figure **??**. In many ways, the result will be similar to the `go_find()` function in Figure 12.19, which searches for the root of a function using the bisection method.

19.8.3 Using the Computer

1. Running sum and difference.

   Write a recursive function, `sigma()`, that will return the result of an alternating sum and difference of the values in an array of `double`s. The first, third, fifth, and so forth calls on the function should add the next array element to the total. The second, fourth, sixth, and so on calls should subtract the next array element from the total. Write a `main()` function to read in values from a user-specified data file, compute the alternating sum and difference, and print the result. Assume no more than 1,000 inputs will be processed. Hint: During debugging, print the parameter values of the recursive function and print its result just before the `return` statement.

2. Fibonacci sequence.

   Write a short program that computes numbers in the Fibonacci sequence. This sequence is defined such that `fib(1) = 0` and `fib(2) = 1` and, for any other number in the sequence, `fib(n) = fib(n-1) + fib(n-2)`. Write a recursive function to compute the nth number in the sequence. Call this function from your main program. Use functions from the `time` library to determine how many seconds it takes to compute the 10th, 20th, and the 30th numbers in the sequence and output these times. Why does it take so long to compute the later numbers in the sequence? (See Chapter 15, computer exercise 2, for instructions on using the `time` library.)

3. Reversal.

   Write a short program that inputs a word or phrase of unknown length from the user, echoes it, and then prints it backward, exactly under the original. If the phrase is a *palindrome*, it will read the same forward

as backward. Use a recursive function to peel off the letters and store them in a second array in reverse order. After reversing the string, compare it to the original and display the message `Palindrome` if the string is the same forward as backward. Otherwise, display `No palindrome`. For example, *pan a nap* is a palindrome; *banana* is not. Hint: The `behead()` function (given in an earlier exercise) may help you.

4. Two piles.

   Write a short program that separates negative and positive numbers, printing negative numbers on the first line and positive ones on the second line. Write a brief main program to prompt the user for a number N between 5 and 50. Then read N numbers into an array and call your recursive function (described next) to separate them. Write a recursive function named `neg_first()` to process the array. If an element is negative, the function should print it before making the next recursive call. If it is positive, the function should call itself recursively to print the other numbers and then print this number afterward.

5. Proportional search.

   Modify the `bin_search()` function in Figure **??** as follows. Rather than always picking the value in the middle of the remaining array, assume that the data in the array are distributed uniformly between the smallest and largest values, and let the next index position be determined proportionally, using this formula:

$$\text{index} = \frac{\text{key} - \text{data}_{\text{first}}}{\text{data}_{\text{last}} - \text{data}_{\text{first}}} \times (\text{last} - \text{first}) + \text{first}$$

   After choosing an index and testing the corresponding value, continue the search as before. Compare the performance of this algorithm with the old binary search algorithm by searching for values in the example data set used in this chapter and noting how many calls each version makes. Which is better? Why?

6. The median value.

   The brute force method of finding the median value in a set is to sort the set first, then select the value in the middle. But sorting is a slow process. A more efficient algorithm, related to both quicksort and binary search, works as follows.

   (a) Let `N` be the number of values in the set, and let those values be stored in an array, `A`. If we sort the values in `A`, then by definition, the median value will be in slot `m = N/2`.
   (b) Perform the quicksort partitioning process on `A` once using the function in Figure 19.24. Store the return value (a pointer to the split point) in `p`.
   (c) Use pointer subtraction to compute `s`, the subscript corresponding to `p`. If `s==m`, we are done; we have found the median and it is in slot `m`. Everything to its left is smaller (or equal), everything to its right is larger, and `m` is in the middle of the array. If `s` is less than `m`, we know the median *is not* in the leftmost partition. Actually, it now is `m-s` positions from the left end of the right partition. So repeat the partition process with the right portion of the array starting at position `s+1` and reset `m = m-s`. If `s` is greater than `m`, we know the median *is* in the leftmost partition and still is `m` slots from the left end of the partition. So repeat the partition process beginning with the left end of the array and ending at position `s-1`.
   (d) Return the value in slot `m` after it has been found.

   Write a program that uses this algorithm to find the median value in a user-specified file containing an unknown quantity of real numbers.

7. A path through a maze.

   Assume you have the following type definition:

```
typedef bool matrix[N][N];
```

   where `N` is a tt `#`defined constant. Also assume a variable of this type is initialized with a connected trail of elements containing the value `true`, going from a source element somewhere in the middle to a border element.

(a) Write a function, `print_trail()`, that will print out the row and column positions of the elements in this trail, from border to source, recursively. This function should have five parameters. The first is `grid`, a variable of type `matrix`. The next two are `row` and `col`; these are the subscripts of the current element of the trail. The last two are `old_row` and `old_col`, the subscripts of the preceding element. The initial call to `print_trail()` should have `row` and `col` set to the point at the beginning of the trail in the middle of `grid`, while `old_row` and `old_col` are both -1. To follow the trail, we can go up, down, right, or left. Check the four directions, looking for an element with a value `true` and making sure not to pick the one with the location `[old_row][old_col]`. Then recursively call the `print_trail()` function to follow the rest of the trail from that position on. When a border element is reached, the recursion ends. This is the base case. As the function finishes and begins returning, it should print the location of the current element. This will print the trail elements in order from the border position to the middle position.

(b) Write a program that will read an N-by-N maze matrix of ones and zeros from a user-specified file to initialize `grid`. From the next (and last) line of the file, read the row and column indexes at which the trail starts. Finally, call `print_trail()` with the starting position of the trail and have it print the trail.

Chapter 20

Command Line Arguments

This chapter introduces three techniques First, we present command-line arguments, which bring directives into a main function from the operating system. Command-line arguments can be used instead of interactive or file input when executing a program from a command shell or from some IDE's.

We also introduce functions with a variable number of parameters and show how to use them to unify related function methods.

20.1 Command-Line Arguments

In the examples presented so far, all communication between the user and the program has been either by interactive query and response or through data stored in a file. Both are useful and powerful ways to control a program and the only ways supported by some limited systems. However, most C/C++ systems provide a way to compile a program, link in the libraries, and store an executable form for later use. Such executable files can be started from the operating system's command shell without entering the compiler's development environment. When this option is available, the operating system's command line offers one way for the user to convey control information to a program.

The **command line** is an interface between the user and the operating system. Simple information such as the number of data items to process, the name of a file, or the setting of a control switch often is passed to a program through this mechanism. A command starts with the name of the application to execute, followed by an appropriate number of additional pieces of information, each separated by spaces. Some kinds of information on the command line may be intercepted and altered or used by the system itself; for example, wild-card patterns are expanded and file redirection commands are used. The rest of the information on the line is parsed into strings and delivered to the program in the form of a **command-line argument** vector.

20.1.1 The Argument Vector

To use command-line arguments, the programmer must declare parameters for `main()` to receive the following two arguments Figure 20.1 shows an example of a command line and the argument data structure that `main()` receives because of it. This command line is for a Unix system; the ~> is the Unix system prompt.

1. The **argument count**, an integer, customarily is named `argc`. This is the number of items, including the name of the program, that are on the command line after the operating system processes wild cards and redirection. Whitespace generally separates each item from the next. Certain grouping characters, such as quotation marks, often can be used to group separate words into one item.
2. The **argument vector**, an array of strings, customarily is named `argv`. Each element in the array is a pointer to a string containing one of the items the user wrote on the command line. (Note that this is the same data structure used for menus.) Element 0 points to the program name; the remaining elements are stored in the vector in the order they were typed.

20.1.2 Decoding Arguments

The first argument (`argv[0]`) is the first thing typed on the command line: the name of the executable program. This is used for making meaningful error comments. The other arguments can be arbitrary strings

Given the command line below, we illustrate the argument data structure delivered to `main()` by the command-line processor on a Unix system.

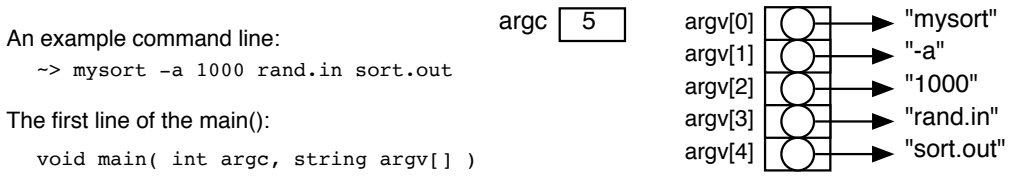


Figure 20.1. The argument vector.

or numbers; quotes are needed if a string has embedded spaces. Arguments have whatever sort of meaning the programmer devises. They have no preset meaning or order, other than that set up by the programmer. No hidden mechanism uses the command-line arguments to control a program. Customs have evolved that shed light on those issues, but those are only customs, not rules.

A program must decode and interpret the arguments itself and is well advised to check for all sorts of errors in the process. Potential errors include misspelled file names, missing or out-of-order arguments, and meaningless control switch settings. In this section, we show how a variety of command-line arguments can be decoded and used to affect the operation of a program. The program that starts in Figure 20.2 allows the user to enter an assortment of information on the command line, including a number, two file names, and (optionally) a control switch. The Sorter class is in Figures 20.3 and Figure **??** The insertion sort program here is a generalized version of the one in Figure 19.28.

Notes on Figure 20.2. Using command-line arguments.
This application reads a file of numbers and sorts them. The command line is used to specify the sorting order, either ascending or descending, the number of items to sort, and the names of the input and output files. These parameters must be in the order given; the sorting order is optional

First box, Figure 20.2: The `#includes`**.** The file `tools.hpp` includes all of the C and C++ header files that a program at this level is likely to need. We `#incude` it here instead of including a lot of other header files, and also to give access to the functions `banner()` and `bye()` are used in main to display information for the user at run time.

```
#include "tools.hpp
#include "sorter.hpp"

// ------------------------------------------------------------
int main( int argc, string argv[] ) {
    banner();
    cout <<"\nCommand Line Demo Program.\n";
        Sorter s( argc, argv );
        s.sort();
    bye();

}
```

Figure 20.2. Using command-line arguments.

Second box, Figure 20.2 (outer): The `main()` function. This function in Figure 20.2 is a typical `main()` program for an OO application. It prints identifying user-information at the beginning, and prints a termination comment at the end. The functions `banner()` and `bye()` are defined in tools.cpp.

Second box, Figure 20.2 (inner): Entering the OO world. Between those actions, `main()` creates one object, a Sorter. The two command-line arguments received by `main()` must be passed as arguments to the Sorter constructor. Then `main()` calls `s.sort()` to do the work, and `s.printData()` to write the sorted data to a file.

Notes on Figure 20.3. The Sorter class.

First box, Figure 20.3:
We `#incude` the tools header file directly or indirectly in every other .hpp file.

Second box, Figure 20.3: The `enum` declaration.

- Enum types and enum constants are used to make code easier to read and understand. Here, we need to know whether to sort the input into ascending or descending order. We could use a boolean or integer constant, but and enum constant is better because it is not cryptic. The words "ascending" and "descending" clearly announces their meanings.
- The string array is used to create meaningful output when the enum constants are used.
- These are in the private area of the class because no other part of the program needs them.

Third box, Figure 20.3: The data members.

- The first four data members, `n, max, data` and `end` store the actual data and essential information about the data array: how full it is, how long it is, where it begins, and where it ends. They are initialized by the class constructor and the `getData()` function.
- The variable `order` is initialized by the constructor after decoding one of the command-line arguments. It is used by `sort()`.
- The last two data members are the input and output streams. They are opened by the constructor using the file names on the command line, and are use by `getData()` and `printData()`.

Fourth box, Figure 20.3: The private function.

- A function should be declared `private` when it is a helper function for some public class method.
- `getData()` is used by the destructor after the sort is finished. It is not intended for use by `main()`.
- `printData()` is used by the constructor after it finishes processing the command line. It is not intended for use by `main()`. It has a stream parameter so that it can be used for two purposes:
  - During debugging, to write the partly sorted data to the screen.
  - After sorting, to write the sorted data to the output file.

Fifth box, Figure 20.3: The public functions.

- The constructor and destructor for any class must be used by some function outside the class. Therefore, they are public functions.
- The constructor is defined in sorter.cpp because it is a long, complex method. However, the destructor is one line and is defined fully here. This is known as an `inline` function. This method writes the sorted data to the output file then frees the dynamic memory.
- `sort()` is called from `main()`. All it does is to sort the data that is already in the array.

This class is instantiated from Figure 20.2. It calls the functions in Figures **??**, **??**, **??**, and 20.8.

```
#include "tools.hpp"

class Sorter {
private:
    enum OrderT ascend, descend ;
    string labels[2] =  "ascending", "descending" ;

    int n;          // Actual number of data items read.
    int max;        // Number of data items expected.
    float* data;    // Dynamic array to store the data.
    float* end;     // Offboard end-data pointer.
    OrderT order;   // ascending or descending
    ifstream ins;
    ofstream outs;

    void getData();
    void printData( ostream& out);
 public:
    Sorter( int argc, char* argv[] );
    ~Sorter(){ printData( outs ); delete[] data; }
    void sort();
};
```

Figure 20.3. The Sorter class.

Notes on Figure 20.4. The Sorter Constructor.

As with any constructor, the goal is to set up the data structure(s) that will be used by the rest of the Sorter class. In this case, information needed to do the setup comes from the command line and a file.

In this application, the primary inputs and outputs come from or go to files. Little or nothing needs to be displayed on the screen. However, it is disconcerting for a user to run a program and see nothing! Thus, we provide feedback at every stage of the setup process.

First box: The .cpp file for every class must `#incude` its own .hpp file.

Second box: Check the number of args on the command line.

- The "-a" or "-d" switch is optional. There are also four required parameters: the name of the executable file, names for input and output files, and the number of values to read in and sort. Thus, there must be either 4 or 5 command-line arguments.
- We check whether `argc` is between 4 and 5. If not, there is an error somewhere. The customary response to this kind of error is a usage comment that lists the required and optional parameters (in square brackets).
- The function `fatal()` is in tools.cpp. It displays an error comment on the `cerr` stream, closes any open files, and aborts execution by calling `exit(1)`. (The '1' is a code for abnormal termination.) The syntax for using `fatal()` is just like the syntax for using `printf()`. As part of the error comment you may print out one or more variables.
- This program calls `fatal()` to handle all of the errors it detects.

Third and sixth boxes: Determine the sorting order

- The "-a" or "-d" switch is optional. If it is present, there must be 5 arguments on the command line. If there are only 4 args, then "-a" is the default, and we set `order` = the default enum constant `ascend`.
- If there are 5 args, we must decode the optional switch. There are three possibilities that we test using C's strcmp():

  - The argument matches '-d', so we set `order = descend`.
  - The argument matches '-a', so we set `order = ascend`.
  - It is something else, so we call `fatal()` and abort.
- This information is echoed back to the user in box 6.

Fourth box, Figure 20.4: Open and verify the streams.

- The two file names are the second and third required arguments. We open these streams.
- Whenever a stream is opened, it *must* be checked for validity. Using `streamName.is_open()` is the modern way to do this task.
- If the stream is not open, it is important to supply the name of the missing file as part of the error comment. Note how we use `fatal()` to do so.

This program calls `getData()` from Figure **??**, and `fatal()` from Figure 20.8.

```
#include "sorter.hpp"

// ------------------------------------------------------------------------------
// Decode and process the command-line arguments.
Sorter::
Sorter( int argc, char* argv[] ) {
    char* last;               // To test for proper number conversion.
    int start = argc-3;       // Subscript of the first of the 3 required arguments.
    if (argc < 4 || argc > 5)
        fatal( "\nUsage:  %s [-ad] num infile outfile", argv[0] );

    // ---------------------- Pick up and check sort order (optional parameter).
    if (argc == 5) {
        if (0 == strcmp( argv[1], "-d" ) ) order = descend;
        else if (0 == strcmp( argv[1], "-a")) order = ascend;
        else fatal( "\nSort order %s undefined.", argv[1] );
    }
    else order = ascend;       // Default if switch is omitted.

    // ------------------------------------------------------------ Open streams
    ins.open( argv[start + 1] );
    if (!ins.is_open()) fatal("\nCannot open %s for reading.\n", argv[start+1]);
    outs.open( argv[start + 2] );
    if (!outs.is_open()) fatal("\nCannot open %s for writing.\n", argv[start+2]);

    // ---------- Convert size from string to numeric format and allocate memory.
    max = strtol( argv[start], &last, 10 );
    if (*last != '\0')    fatal( "\nError:  %s is not an integer.\n", argv[start] );
    if (max < 2)          fatal( "\nError:  %i is too few items to sort.\n", n );
    data = new float[max];
    getData();

    cout << "Data will be sorted in " <<labels[order] <<" order\n";
}
```

Figure 20.4. The Sorter Constructor.

Fifth box, Figure 20.4: A number on the command line.

- If a command-line argument is a number, it must be converted from a string of digits to a binary number before use. This is not done automatically by the system, as `cin >> n` would do for keyboard input. However, the C `stdlib` library provides the function `strtol()` for this purpose.
- `strtol()` stands for string-to-long. It converts a null-terminated array of digits to a `long int`. Its first argument is the string to be converted and the third is the conversion base (10 for decimal, 16 for hexadecimal, etc.). The second argument is the address of a `char*` that will be set by `strtol()` to the first character in the input string that is not a legal digit and was therefore not used in the conversion process. The result of the conversion is stored in `max`
- In this context, the entire string should be a single number and the `last` pointer should be left pointing at the null terminator at the end of the string. After attempting to convert the number, which is always the third-to-last argument, we check to be sure the string was convertible and, if it was, that the result is reasonable (at least two items).
- Here are the results (with several lines omitted) of two faulty attempts at specifying the number of items:

```
~> comline 3a rand.in sort.out
Error: 3a is not an integer.
--------------------------------------
~> comline a3 rand.in sort.out
Error: a3 is not an integer.
```

- If the number conversion was correct, the result is stored in `max` and used to allocate memory for the data array. Then the input function, `getData()` is called to fill the array with data.

Notes on Figure 20.5: Reading the data.
This function reads data items from the input file specified on the command line. Reading ends when the specified number of data items have been read, even if more data is in the file. Reading will end earlier of there is less data in the file than the expected maximum.

First box, Figure 20.5: Mark the beginning and end. The input loop used here works with pointers, not subscripts. To prepare for the loop, we set two pointers: `cursor` starts at array slot 0 and traverses the array until it meets `end`, which points to the first memory location after the end of the array.

Second box, Figure 20.5: The input loop.

- Incrementing `cursor` (which points to one slot of an array) moves the `cursor` on to the next array slot. The loop ends when `cursor` points at the same location as `end`.
- The statement `ins >> *cursor` reads one float value from the open input file and stores it in the slot under the cursor.
- It is necessary to check for end-of-file *after* every read. It is not correct to check before the read. That will, in general, cause the last line to be read twice.
- It is also necessary to check for read errors after every read. There are a variety of things an application might do about a read error. In this demo program, we do the simplest thing, that is, stop reading and process whatever data we already have.
- A stream is "good" after a read if data was read, converted, and stored in memory correctly. Any kind of failure (hardware, number conversion, eof) will set a flag in the stream's object and that will cause the stream to be not good.

Third box Figure 20.5: How many items were read?

- When two pointers point at slots in the same array, subtracting the leftmost from the rightmost will tell you how many slots lie between the pointers.
- Since `data` points at the slot 0 of the array and `cursor` points at the first un-filled slot, the difference `cursor - data` tells us the number of data items that were read and stored in the array. We save this number in a class member, `n` so that the `sort()` and `printData()` functions will know what part of the array to process.
- The number of data items is used to set `end` to the first array slot not occupied by data. If all expected data was read properly, this will be the first slot after the array ends. If not, it will be the first slot that does not contain valid data. This end pointer is used by the `sort()` and `printData()` functions.

```
void Sorter::
getData() {
    float* cursor = data;        // Set cursor to beginning of data array.
    end = data + max;            // end is an off-board end-marker.

    for( ; cursor<end; ++cursor) {
        ins >> *cursor;
        if( !ins.good() ) break; // Stop loop for error or for end of file.
    }

    n = cursor - data;           // n is the actual # of items read.
    end = data + n;              // an off-board sentinel pointer

    cout << n << " input values were read and stored.\n";
}
```

Figure 20.5. Reading the data.

Notes on Figure 20.6: Printing the sorted data.
This function has a stream parameter so that it can send output either to the screen or to a file.

First box: The `cursor` is reset to the beginning of the array (slot 0). It will traverse the array until it meets `end`, which was set by the `getData()` function.

Second box, Figure 20.6: the loop.
- This loop is like the one in `getData()`, with one important difference: the `end` pointer might be set differently.
- Before reading data into an array, `end` must be set to mark the end of the memory locations allocated for the array. Before processing or printing, it must be set to mark the end of the data that has been stored there.

Notes on Figure 20.7: Sort in ascending or descending order.
An insertion sort is a nested-loop sort where the outer loop is executed a fixed number of times (n-1). The inner loop starts with one iteration and adds one more iteration each time through the outer loop.

First box, Figure 20.7: The pointers!
- `fence` is the pointer that controls the outer loop.
- `hole` is the pointer that controls the inner loop and the array slot that does not contain important data.
- `newcomer` is the data that was moved out of the hole at the beginning of the inner loop.

```
// Print array values, one per line, to selected stream.
void Sorter::
printData( ostream& out ) {
    float* cursor = data;        // Set cursor to beginning of data array.

    for( ; cursor < end; ++cursor) out << *cursor <<endl;
}
```

Figure 20.6. Printing the sorted data.

```
void Sorter::
sort() {
    float* hole;          // currently empty location
    float* k;             // location currently being compared to newcomer
    float* fence;         // last location in unsorted part
    float newcomer;       // Data value being inserted.

    // Insert n-1 items into the portion of array after fence, which is sorted.
    for (fence = end - 2; fence >= data; fence--) {
        // Pick up next item and insert into sorted portion of array.
        hole = fence;
        newcomer = *hole;
        // cout <<newcomer <<" is newcomer \n";
        k = hole + 1;
        for (; k != end; ) {
            // cout <<*k <<" is *k.\n";
            if (order == ascend && newcomer <= *k )
                break;              // Insertion slot found...
            else if (order == descend && newcomer >= *k)
                break;              // ....  so leave loop.
            else *hole++ = *k++;    // else move item back one slot.
        }
        *hole = newcomer;
        // printData(cout);
    }
}
```

Figure 20.7. Sort in ascending or descending order.

- `k` is always positioned just to the right of hole. We use `k` to avoid writing `hole+1` again and again. The +1's become rapidly confusing.

Outer box, Figure 20.7: The Outer loop of Insertion sort.

- An set of 1 thing is always sorted. So we start `fence` at the 2nd-last data value in the array. `fence` moves backward toward the head of the array. The loop ends when fence fall off the left end of the data array.
- On each iteration, the newcomer is the item under `fence`, and `hole = fence`.
- The inner loop inserts the newcomer into the part of the array to the right of the fence.
- The output statement that is commented out was used during debugging to track the progress of the process.

First inner box, Figure 20.7: The inner loop of Insertion sort.

- We set `k = hole+1` because we will be using both slots repeatedly as we move data from slot k into the hole.
- The output statement that is commented out was used during debugging to track the progress of the algorithm.
- On each iteration of the inner loop, the data in slot `k` (`*k`) moves one slot to the left in the array `*hole`, then both `k` and `hole` are incremented to move them to the next pair of slots to the right.
- This movement ends when we get to the proper insertion place for the newcomer, and break out of the loop. The "proper" slot depends on the sort order.

```
void fatal(const char* format, ...) {
    va_list vargs;         // optional arguments
    va_start( vargs, format );
    vfprintf( stderr, format, vargs );
    fprintf( stderr, "\nError exit; aborting.\n" );
    exit( 1 );
}
```

Figure 20.8. The `fatal()` function.

- At the end of the inner loop, the newcomer is stored in its proper position.

Innermost box, Figure 20.7: Use the right operator for the desired sort order.
We need a <= comparison to sort in ascending order and a >= comparison to sort in descending order.

20.2 Functions with Variable Numbers of Arguments

C actually provides a library, `stdarg`, to make it possible to write functions that have an indefinite number of parameters of any possible combination of types. The argument list of such a function must start with one (or more) argument that specifies how the other arguments are to be used.

For example, consider `printf()`. A call on `printf()` starts with a format and the format has one percent-sign for each expression on the rest of the list. Thus, the format tells the compiler and run-time system what to do with the remaining arguments.

The `stdarg` library allows us to write functions that work like `printf()`. We call such a function a *varargs* function. In this section, we define a varargs function, `fatal()`, to print error messages and abort execution. The new `fatal()` function accepts exactly the same set of parameters as `printf()`: a format string and a list of variables to output. It uses methods in the `stdarg` library to pass these parameters on to a library function named `vfprintf()`. When printing is done, `fatal()`,terminates execution properly.

Notes on Figure 20.8: The fatal() function.
The `fatal()` function is used after an error has been detected and further progress is impossible. It takes a format argument (like `printf()`) followed by any number of data arguments. It formats and prints an error message, using the data arguments. Then it calls exit(), which flushes the output stream buffers, closes all open streams, and aborts cleanly.

First box: the varargs prototype. The parameter list of a varargs function must start with at least one ordinary (required) parameter. Following that, all of the optional parameters are replaced by three dots. This tells the compiler to pack the actual arguments into a data structure that can be passed around from function to function.

Second box: capturing the argument list. A function declared with `...` must declare a variable of type `va_list` and must initialize it, as shown, by calling `va_start()`. The second parameter to vastart is the name of the last required parameter. After this call, the variable `vargs` is initialized to point at an array of arguments suitable for processing by other varargs functions.

Third box: using the argument list. This line prints the error message. The function `vfprintf` expects three arguments: an open output stream, a printf-style format, and a `va_list` variable. It works exactly like `fprintf()` except that the many arguments of `fprintf()` are replaced by the single list-argument.

Fourth box: graceful termination.

- We wish to accomplish two things here:
  - Provide good information to the user.
  - End texecution cleanly.
- The `exit()` function flushes all output buffers and closes all open streams. Then it returns to the operating system with the error code 1, which means abnormal termination.

20.3 Modular Organization

A well-designed large program is built in several modules, with a `main()` program at the top level that calls functions from other modules. Each class forms a module. When the application is designed, the purpose of each class is specified, as are the ways each class can interact with the others. Each then can be stored in a separate file and developed by a different member of the development team. The modules are composed of programmer-defined classes with data and function members. Header files are used to keep the interface between the modules consistent and permit functions in one module to call functions in another. Source files contain the actual code (except for inline functions). We further explore how the techniques presented so far extend into creating larger and more protected object-oriented applications.

20.3.1 File Management Issues with Modular Construction

Writing, compiling, and running a simple one-module program is very much the same in any system environment. The programmer creates a source file, then either types a compile command or selects a "compile" option from a menu. If compilation is successful, an execute or run command is given either automatically, or from the command line or by selecting a menu option.

Writing and running a large program is much more complex and the process differs from system to system and from one IDE to another. The design of a large program often includes several programmer-defined modules that perform different phases of the overall task, and these may be written by different people. Each module, in turn, is a set of related definitions and functions, written in separate source code and header files, and compiled into separate object files. The object files are linked together, along with the system libraries, to create an integrated executable program. In this section, we discuss some general principles and guidelines for **modular** program **design** and explain how to build a **multimodule program** in a Unix environment.

Organizing and managing the files of a modular application and creating an executable program from them raises a group of problems related to efficiency, completeness, and consistency:

- It is undesirable to recompile an entire large application every time a change is made in one small part of it. During debugging, modules are normally changed one at a time, as errors are found. We must be able to avoid recompiling the other modules.
- We must be able to compile a module without linking it to anything, so that a debugged module can be stored much like a library.
- We must be able to link programs to object-code modules that were previously compiled, whether the module was produced locally or is part of a library package for which we do not have the source code.
- We must have access to the header files needed by each code module, whether these are our own or part of a library package.
- We must keep track of which object files and libraries are needed and make them all available to the linker at the appropriate time.
- We must avoid including the same header file twice in one module or the same source-code file twice in the linked program. The system linker will not work if it encounters a symbol that is defined twice . . . even if the declarations are identical
- All modules must agree on the prototypes for shared functions, the values of common constants, the details of shared type definitions, and the names of included header files.

- We must have a way to determine whether a module is properly compiled and up to date. We must ensure that we use the most recent version of each module. A systematic way to do this is vital if the modules are being developed independently.
- If more than one programmer is working on the project, it is important to use a *version control system* to ensure that everyone works on the most recent version and that diverging changes are not introduced into the code base.
- We should have a way to test individual modules independently.

A variety of techniques have been developed to address the problems of **code management**: Effective use of subdirectories, collaboration sites, system search paths, revision control systems, makefiles, project files, header files, compiler options, and preprocessor commands, just to mention a few. These tools make it easier and faster to create a large program and much faster to debug it. The amazing large systems that we use daily could not have been developed using the tools and methods programmers had in 1970.

Different operating system environments provide alternate ways to organize files and perform the compilation tasks; the programmer must learn how each local environment works. We now discuss a few of these techniques.

Files and folders. Four kinds of modules might be used as part of a project:

1. *System libraries* include both the standard C or C++ libraries and compiler-specific packages for graphics, sound, and the like.
2. Local, user-defined library modules include both personal libraries and modules that are shared among members of a group or employees of a company.
3. The programmer's main module containing the function `main()`.
4. Other modules that are defined to support this particular application.

To keep all of these parts organized and avoid conflict with the parts of other programs, a subdirectory, or folder, should be established for each multimodule project. This folder will contain all of the user's code modules, header files, relevant input data and output files, and ideally, a document file that explains how to use the program. Very large projects may have a subdirectory for each module, especially if modules are being developed by different people. Depending on your system, the relevant programmer-defined library modules and header files also may go in this subdirectory. Alternatively, it may be more convenient to put local library files in a more central directory that is accessible to other projects. In this case, these files are accessed by setting up appropriate search paths. These will be in addition to the standard search paths that the compiler uses to locate system libraries and system header files.

Header files and source code files. The best way to work with a large application is to break it up into subsystems, then implement each subsystem as a separate code module. Each one is likely to have functions and objects for internal use only, and others that are **exported** and used by other modules. The **module interface** consists of the set of public definitions and function prototypes.

In C we organize each program module by splitting it into two sections: the internal code and the interface. The interface portion becomes a **header file** (usually with a `.h` extension). It should include the headers of the modules it depends on. The code portion becomes a **source code file** (with a `.c` extension), which must `#include` its own header file.

In C++ we organize each program module by splitting it slightly differently: the class declaration and all the `#include` commands are in the header file (with a `.hpp` or `.h` extension). This class declaration gives full definitions of very short functions (one liners). The rest of the function definitions go into the `.cpp` file, which must include its own header file.

In both C and C++, any module that uses things defined by another module must also `#include` the header file of the other module.

When we `#include` a header file in a program, the type definitions and prototypes are copied into the program. This allows the compiler to verify and set up calls on the imported functions. The actual code of those functions is *not* copied into the importing file. The code of the two files will not be connected until all modules are linked together, later.

For example, when we include `stdio.h`, we do not include the source code for the whole library (which is massively large). Rather, we include the prototypes for the library and a variety of type definitions and constants. This is the information needed to compile our calls on the library functions. Only our own code is compiled, not the library's, which saves much compilation time during a long program's development.

You should never `#include` a source code file in another module. Beginners make this error before they learn how to use an IDE and projects properly. It may work for very simple programs. However, it is not considered a good practice because it increases the possibility that functions in one code module might inadvertently interact with parts of another in unanticipated and damaging ways.

In a properly constructed modular program, the code portion of each module is compiled to produce an object code module (usually having a `.o` extension), which later is linked with the object modules of the libraries and other user modules to form a **load module,** or executable program. During the linking stage, all calls on imported functions are linked to the appropriate function in the exporting module, making a seamless and connected whole, as if the library source code actually was part of that module. The linking operation will fail if the linker cannot find a module, among the set provided, that defines each function used. It also fails if two definitions of that function are provided. For this reason, it is very important to avoid including two modules that define the same thing.

Include-guards avoid header file duplication.

A header file may be included in several modules (normally, at least two), and it may be included by other header files. This can become a problem because a module will not compile if it includes the same header file twice. In small programs, one can often deal with this double-inclusion problem by being extremely careful. However, in a large program, the inclusion relationships can become quite complex. It is simply not worth the time and attention to be careful about what gets included where. The C/C++ preprocessor provides two tools to eliminate the multiple-inclusion problem. The easiest solution is to write

```
#pragma once
```

on the first line of the header file. This gives the compiler guidance that the contents of the file must not be included multiple times. Using `#pragma once` is convenient and easy. It works on all the systems personally known to the author, but it is *not* guaranteed by the C/C++ standard to work on all systems. Programmers using those older systems must fall back on the old way to control multiple inclusion, using `#ifndef`.

The first two commands and the last line in a header file should have the form:

```
#ifndef MYMODULE
#define MYMODULE
...
#endif
```

Following the `#pragma once`, or between the `#define` and the `#endif`, are the lines that form the interface for the module. All header files should be guarded by preprocessor commands in one way or the other. The symbol named in the `#ifndef` command is arbitrary, but it should be unique and berelated to the name of the module.

A `#ifndef` command is the beginning of a **conditional compilation** block that ends with the matching `#endif`. When the preprocessor phase of the compiler encounters the command `#ifndef MYMODULE`, it looks for `MYMODULE` in its table of defined symbols. If present, everything following the command is skipped up to the matching `#endif`, thereby ensuring that the following declarations are not included a second time. If `MYMODULE` was not previously defined, this is the first inclusion of the file. The compiler enters the block and immediately defines `MYMODULE`, preventing future double inclusion. Then it processes the other definitions and declarations in the file normally.

20.3.2 Building a Multimodule Program

In developing a multi-module project, it is important to have some way to specify the list of parts necessary to build that project. Also, the programmer needs a tool to help keep the parts consistent and collected in one place. In the pre-IDE days, this was done by writing a Unix *makefile*. Now, we can do it by creating a new project within the IDE and adding files to that project. Some IDEs then create a traditional makefile for you. The project file or makefile is like a top-level directory of components. It is the tool used by the IDE to guarantee that the most up-to-date version of each source file is compiled or linked each time.

Using an IDE. An IDE (Integrated Development Environment) for C/C++ does this job and also provides a structured text editor that "knows about" the syntax of C/C++. Other parts of the IDE supply an interface to the compiler that displays error comments, a run-time console, and a project-management system. In the Windows world, Eclipse is probably the best choice. Visual Studio is commonly used but has serious problems: it permits a variety of small errors that cause the code to not compile on a standard compiler. Simple IDE's such as CodeBlocks are simply not adequate beyond the first programming course. For Mac users, there is XCode. Many Linux programmers use KDE or Eclipse.

A new project. When you ask your IDE to create a new project, the IDE will create a project folder for you, and inside it, various files and subdirectories that will be used by the IDE. Be sure this directory is created in your own home directory, not in some hidden system directory.

Many IDEs also create a code file containing a skeletal main program. This main program may be fine, as it is. However, each IDE has its own peculiarities. For example, Visual Studio puts an unnecessary line in the file that is incompatible with standard C/C++, so that line must be removed: `#include <stdafx.h>`.

If you are importing existing files from other directories, you should first copy or move those files into the project directory. Then use the IDE's menus to "add" the files to your project. When you click on the "Build" option, everything you have "added" will be compiled and the results will be linked together to form an executable program.

After importing copies of all the code files you wish to reuse, the skeletal main program is either fleshed out to form an actual main program or replaced (using copy and paste). Write a few lines of main, then compile and test the project. Repeat until done. As you develop additional modules, continue this process: write a couple of functions, then compile and test. Avoid trying to integrate large amounts of new code at once.

20.3.3 Using a makefile.

A makefile lists the parts necessary to build a project and the relationships among those parts. It lets you define the compiler settings that should be used.

The project directory. Establish a directory for this project. Everything related to the project will go into this directory: code, data, documentation, and the results of compilation. DO NOT mix all your programs together in the same flat directory! Copy files you wish to reuse from other projects into this directory.

Entering the code. To use a makefile, you need a separate text editor: any one will do. Use whichever editor you like best, or use the editor that is part of your favored IDE. Open a new file and start typing. Use the same text editor to create your makefile (directions follow) and your data files. Work incrementally: code a bit, then compile and test.

Compiling and linking. In a typical Unix environment, the compilation and linking process is automated by a makefile. The steps involved are explicit and therefore easy to demonstrate. Analogous things happen within an IDE, under the control of options that can be set by the programmer. However, in an IDE, they are hidden and difficult to demonstrate.

In many systems, compilers can be called directly from a command line. For instance, suppose we have a source code file, `lab1.cpp`, that we wish to **compile and link** with the standard C++ library functions. The traditional Unix command to accomplish this is

```
~> c++ -o lab1 lab1.cpp
```

The command starts with the name of the compiler (typically `c++` or `g++`). Following that is a `-o` switch and an accompanying name (`lab1`) that will be given to the executable program. The last argument is the name of the source code file. When this command is executed, the compiler is called to translate `lab1.cpp`. If there are no fatal errors, an intermediate object file, `lab1.o`, is produced. Then the linker is called to join this object file with the object files for the standard library functions. If linking also is successful, the executable file, `lab1`, is produced. To run the program, the name of the file simply is typed at the command-line prompt:

```
~> lab1
```

This diagram represents a game application with three modules (`game`, `board`, and `tools`). Rectangles represent the programmer's files, shaded oval boxes are executions of programs (the compiler, the linker, and the user's finished application). Arrows represent dependencies and the flow of information.

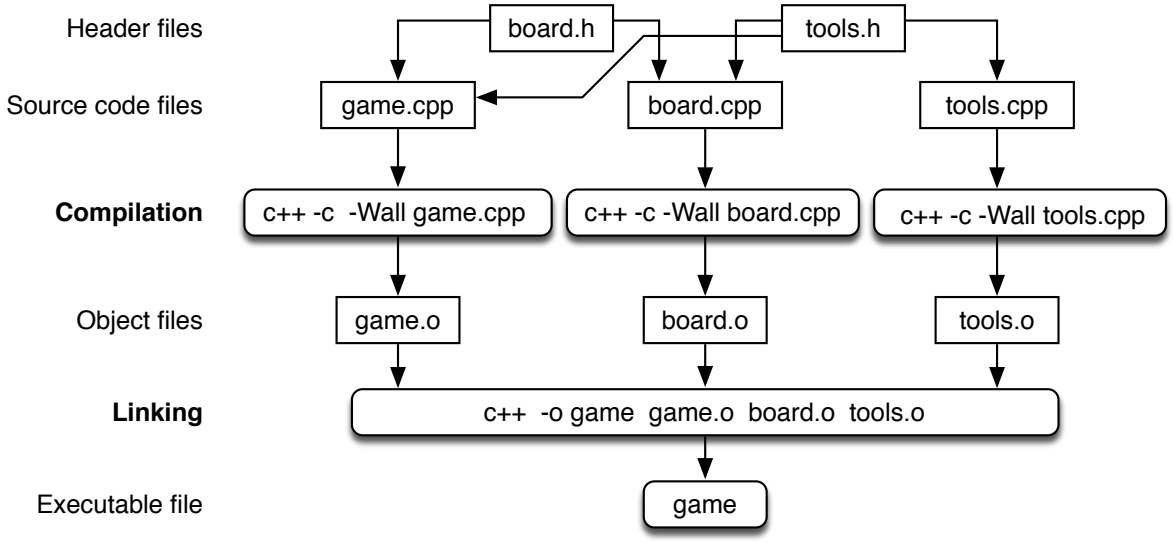


Figure 20.9. Steps in building a program.

Compilers offer many options, in addition to `-o`, which can be very useful. A complete list of **compiler options** is too numerous to give here, but we describe a few commonly used options. Here are two that you may need:

- The programs in this book were compiled under the C++ 11 standard. Some compilers default to an older standard. To use C++ 11 I must write a switch on the command line: `-std=c++11`
- Many compilers produce helpful warning comments when the `-W` switch is used to turn on "all" error checking: `-Wall`

Thus, to compile the program named `lab1.cpp`, I might write this command:

```
~> c++ -Wall -std=c++11 -o lab1 lab1.cpp
```

Separate compilation and linking. A single-module program generally is compiled and linked in one step. However, in a large multimodule program, each module is compiled separately, as it is developed, and linking is done as a final step. To **compile only** (without linking), the `-c` switch is used instead of `-o`. For example, suppose a game program has two user-defined modules, `game` and `board`, and also calls functions from the `tools` library. The three separate compilation commands would be

```
~> c++ -c -Wall game.c
~> c++ -c -Wall board.c
~> c++ -c -Wall tools.c
```

The upper part of the diagram in Figure 20.9 illustrates the relationships among the code files, header files, and the three compilation commands. The direct input to each step is the `.cpp` file that is listed in the c++ command. The `.h` files are indirectly included by each `.cpp` file and so are not listed explicitly in the command.

If there are no errors in the three source code modules, the result of the compilation commands will be three object files: `game.o`, `board.o`, and `tools.o`. These three files then can be linked with functions from the standard library by using the command

```
~> c++ -o game  game.o board.o tools.o
```

This step is illustrated by the bottom part of the diagram; the result is an executable program, `game`, that can be run from the command line by typing its name:

```
~> game
```

Dependencies. The arrows in Figure 20.9 show how the object and executable files for an application **depend** on various combinations of source code and header files. For example, `game.o` depends directly on `game.cpp` and through it, indirectly, on `board.h` and `tools.h`. Similarly, `tools.o` depends on `tools.cpp` and `tools.h`. The executable program, which directly depends on all three `.o` files, indirectly depends on all five source code and header files. If one of these five files is changed, every file that depends on it becomes an **obsolete file** and needs to be rebuilt. Therefore, if we change `game.cpp`, `game.o` and `game` need to be regenerated. And if we change `tools.h`, all three object files and the executable program need to be rebuilt. The `.cpp` files need not be changed because they always include the version of the `.h` files current at compilation time. Keeping all the dependent files updated is important but can be tricky, especially if an application is large and has dozens of modules. This is the kind of detailed bookkeeping that human beings find difficult but is easy for a computer. Clearly, an automated system is needed.

A makefile automates the process. An IDE provides automated support for making an "application" or "project." The programmer names the project and lists the source code files and any special libraries that compose it. Once the project has been defined, the programmer typically selects a menu option labeled `MAKE` or `BUILD` to compile and link the project. Unix systems provide a "make" facility that is less user-friendly but more flexible and much more portable. The programmer creates a file of compilation and linking commands, called a **makefile.** In any system, the project file or makefile represents the application as a whole and encodes the dependency information and compilation commands shown in Figure 20.9.

A **make facility** should provide several important services:

- It should allow the programmer to list the modules and libraries that compose the application and specify how to produce and use each one.
- For each module, it should allow the programmer to list the files on which it depends. In some systems, this list is automatically derived from the source code.
- When given the command `make`, the facility should generate the executable program. Initially, this means it must compile every module and link them together.
- When a source code module is changed, the corresponding object module becomes obsolete. The next time a `make` command is given, it should be recompiled.
- When a header file is changed, the object files for all dependent modules become obsolete. The next time a `make` command is given, they should be recompiled.
- When a module is recompiled, the any executable program that uses it becomes obsolete, so the application should be relinked.

Multiple targets. A *target* is usually the name of a file that is generated when a program is built. Executable files and object files are examples. A target can also be the name of an action to carry out, such as 'clean'.

A single makefile may specify several *targets*. For example, the primary target (the first one listed) may be to compile the program in the normal way, ready for deployment. A second target might compile the program in debug mode, or compile a closely related program that shares the directory. A third target (normally named "clean") might be provided to delete all compiler output and force all modules to be recompiled on the next build.

To use a makefile to build the primary target, you simply say "make". To build any other target defined by that makefile, write the name of the target after the "make":

```
~> make
~> make clean
```

20.3.4 A Unix makefile.

A makefile contains a set of commands to compile and link an application, plus other information used to automate the process, maintain consistency, and avoid unnecessary work. A system like the Unix make facility could be implemented for any operating system. Here, though, the information we give is specific to C/C++ and Unix and systems derived from Unix.

```
# ------------------- Makefile for the game application
OBJS = game.o board.o tools.o
CXXFLAGS = -Wall -std=c++14 -O1
# CXX is predefined to the default C++ compiler on your machine.
# On a Mac, CXX = clang++.  On Linux, CXX = g++.  Both define c++ also.

# ------------------------------------ Linking command
game:  $(OBJS)
    $(CXX) -o game $(OBJS)

# --------------------------------- Compilation commands
game.o:  game.cpp board.h tools.h
    $(CXX) -c $(CXXFLAGS) game.cpp

board.o:  board.cpp board.h tools.h
    $(CXX) -c $(CXXFLAGS) board.cpp

tools.o:  tools.cpp tools.h
    $(CXX) -c $(CXXFLAGS) tools.cpp

# ---------------------------- Optional cleanup command
clean:
    rm -f $(OBJS) game
```

Figure 20.10. A Unix makefile for game.

To explain the meaning and syntax for some basic commands in a makefile that accomplishes these services, we examine a makefile, shown in Figure 20.10,[1] for the hypothetical `game` program. We discuss the fundamental parts of a simple makefile.[2]

Notes on Figure 20.10: A Unix makefile for game.
The first line in this makefile is a comment; note that a comment is limited to one line and starts with a `#` character, not C++ comment marks. This makefile has a comment in the first box and at the beginning of each section.

First box: symbol definitions.

- A major goal of the make facility is to maintain consistency across all modules in the application. Another is to avoid omitting some essential file name or switch from one of the commands. To achieve this goal, a make facility lets us define symbolic names for phrases that must be used more than once in the set of compilation and linking commands. In this box, we define two symbols, `OBJS` and `CXXFLAGS`. The symbol `CXX` is defined by the system to be the default local C++ compiler.
- Three "flags" or "switches" are defined on this line and are recommended for use in this class:
  - `-Wall` tells the compiler to provide the highest level of warnings.
  - `-std=c++14` tells the make system to use the 2014 version of the C++ standard.
  - `-O1` tells the compiler to perform level-1 optimization. This does some flow analysis, and the comments provided may be helpful in debugging logic errors. (Note: this is a letter O, not a numeral 0.)
- These symbols are treated by the make facility in much the same way that a `const` variable is treated in C. They are defined by giving a symbol name followed by an equal sign and a defining phrase.

[1] All Unix-like systems provide similar make facilities; the particular one discussed here is the Gnu make program, from the Free Software Foundation.

[2] A full discussion of makefile capabilities is beyond the scope of this text, including archiving commands and ways to reduce the number of compilation commands that must be written.

- We define `OBJS` to stand for the list of object files that form the application. We want a symbol to stand for this list because we use the list three times in the makefile and want to be sure the three copies are identical.
- When compiling a series of modules, we should be consistent about the compiler options used. To ensure consistency, we define a symbol, `CXXFLAGS`, as the list of compiler options we want to use for the application. In this example, we use only the error warning option, `-Wall`. Note: the `XX` stands for ++, which cannot be part of a name in this language.

Second, third, and fourth boxes: dependencies and rules.
These two boxes contain pairs of lines, which we will call **rules.** Each rule consists of a dependency declaration followed by a command used to compile, link, or clean up. The dependency declaration on the first line of each pair encodes the arrows in Figure 20.9. It is used by the make facility to determine whether a dependent file is current or obsolete. (A file becomes obsolete whenever a file above it in the dependency tree is changed.)

The second line of each rule is indented with a tab character. It must be a tab, not spaces. This line gives the Unix command that should be executed in order to make the target (the left side of first line) out of the code file and header files on the right.

Second box: the linking rule.
- The primary target should be the first target in the makefile, so its rule is the first rule in the makefile.
- The first line of this rule starts with the name that will be given to the finished application. Following that is a colon and a list of all the object files on which the application directly depends. In this example, we name the three object files by referring to the symbol `OBJS`, defined in the first box.
- The second line is a linking command. This command says that the object files listed in the first box must be linked with the standard C++ libraries to form the `game` application.
- The `$(...)` notation indicates that the symbol in the parentheses has been defined at the top of the makefile and the symbol's definition should be substituted for the `$(...)` unit. This process is almost identical to the way `#define` commands are used in C. After substitution, these lines would say:

```
game:  game.o board.o tools.o
    c++ -o game game.o board.o tools.o
```

- Note: The first keystroke on the line of any compilation or linking command in a makefile must be the tab character. The make facility will not work if spaces are used to indent the `c++` command or if it is not indented.

Third box: the compilation commands.
- Since the application depends on three object files, there are three rules here that describe how to build them. The first line in each rule lists a set of dependencies: one source code file and all the programmer-defined header files that it includes.
- The second line in each rule is a compilation-only command. All three of these use the list of options defined for `CXXFLAGS` in the first box. After substituting the flags for the `$(...)`, the first command would say

```
game.o:  game.c board.h tools.h
    c++ -c -Wall -std=c++14 -O1game.c
```

- Using these switches, the compiler will use a low level of optimization (rather than none), give all warnings (rather than just fatal error notices), and use the C++ 2014 standard (instead of an older version of the language).

Fourth box: the cleanup command.
- This command automates the job of deleting all the object files and the executable file. You can have a makefile without it, but that is not a good idea.
- The phrase `rm -f` is the Unix command to remove (delete) a list of files. The `-f` flag says that the system should do it without prompting the user for verification on each file in the list. If a file is not there, the system also should not complain. We write `$(OBJS)` as an argument to delete all the object files listed in the first box. Then we add the name given to the executable file in the second box.

- The first line of the rule gives a name by which this operation can be identified. There are no prerequisites to doing the deletion command, so the dependency list is left empty. It is helpful to have a cleanup command when the programmer wishes to archive the source modules or when the file creation times indicate that the executable file is newer than any of the files on which it depends, but the programmer wishes to recompile from scratch anyway. (This can happen for a variety of reasons, such as changing the list of compiler options in `CXXFLAGS`.)
- To use the cleanup command, type

```
~> make clean
```

Using the makefile.

For the makefile-beginner, only one makefile should be in a directory, and the files named in its dependency lists should be in the same directory. If this is true, we build the primary target by typing

```
~> make
```

The `make` program will look in the current default directory for a file named `makefile` and open it. Unless a specific rule name (such as `make clean`) has been given, it starts at the top of the makefile, interprets the definitions, and carries out the commands. It focuses on the first rule, which should be the linking rule. The name at the left side of its dependency line will be the name of the finished executable file. If that file does not exist or if it is older than one of the files on the dependency list, the linking should be redone. First, however, `make` must check whether the object files are all up-to-date.

If an object file does not exist or is obsolete, the compilation command on the second line of its rule will be used to compile the file. Compilation is skipped when the object code file is newer than the source code and header files on its dependency list. When all the required object files are up to date, the linking command in the first rule finally is executed.