Applied C and C++ Programming

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

Michael J. Fischer
Yale University

August 2018

Part V

Pointers

Chapter 16

Dynamic Arrays

In this chapter, we introduce two important parts of the C++ language: allocating arrays dynamically and using pointers to process them. We show how pointers and subscripts are alternative ways to process an array, how an array of unpredictable length can be allocated at run time, and how dynamic allocation can be used to remove limitations from algorithms that work with arrays. Destructors are introduced for managing the dynamic memory.

Finally, two basic sorting algorithms are presented and used to illustrate all the other techniques introduced in the chapter.

16.1 Operator Definitions

In C++, a programmer may define additional methods for any operator defined by the C precedence table[1]. We call these definitions operator extensions, and they are a very powerful tool for making it possible to write C++ code that looks like the formulas you would see in a textbook on mathematics or physics. For example, it is common to define the arithmetic operators on the class Complex.

An operator definition must have the same number of parameters as the original C operator, and it will have the same precedence and associativity. If it also has the same purpose and general meaning as the original, we call it an operator extension. For example, if the definition of + on Complex performs complex addition, then the new operator is an extension of the original.

If the new operator has a different meaning, it is generally called an operator overload. For instance, if you define + to mean string concatenation, that is an overload, because now + has two unrelated meanings. Operator overloads should normally be avoided, although some are built into C++, for example << and >> for I/O and + for strings.

Operator extensions should be used carefully, and only to allow a new class to "behave like" the programmer expects it to behave. For example, in this chapter we define a class, Flex, which implements an array that can grow longer if needed to store an unpredictable amount of information. Because this class is used to replace simple C arrays, programmers want it to behave like an array – they want to be able to access the data inside the Flex by using subscript. So the Flex class *extends* the subscript operator, and delegates the subscripting operation to the simple dynamic array *inside* the Flex array. You can see the prototype for the new subscript function in Figure 16.11 and the actual method definition in Figure 16.12.

16.2 Pointers—Old and New Ideas

This section collects, reviews, and elaborates on the material concerning pointers introduced in earlier Chapters, and extends it to show new ways pointers may be used with arrays. In many ways, a pointer in

[1] A complete treatment of this topic is beyond the scope of this book. However, in the next few chapters, we need to be able to extend the subscript operator, so that one special operator extension will be presented.

We declare an array and show how to set a pointer to its first slot or to an interior slot.

```
double z[3] = {0.0, 1.0, 0.0};
double* zp = z;
double* wp = &z[2];
```

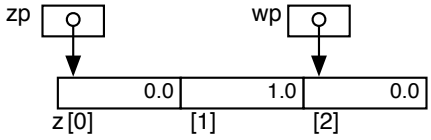


Figure 16.1. Pointing at an array.

C++ is like a pronoun in English. Both can be attached, or bound, to an object and changed later to refer to a different object.

16.2.1 Pointer Declarations and Initialization

A pointer variable is created when we use an asterisk after a type name in a declaration. The declaration `int k`, with no `*`, creates an `int` variable, but `int* p` creates a pointer variable that can refer to any `int`. The type named is called the **base type** of the pointer, and the pointer can point meaningfully only at variables of that type.

The **referent** of a pointer (the address of an object of a matching type) can be set either by initialization or by assignment. Figures **??** through **??** give examples of declarations of several types of pointers. In the following paragraphs, we briefly summarize the syntax and meaning of these pointer assignments.

Pointing at an array. In C and in C++, an array is a sequence of variables that have the same type and are stored consecutively and contiguously in memory. When we point at an array, we actually point only at one of the elements (slots) in that array, not at the entire object. However, pointing at one slot gives us access to all the other slots that precede and follow it. Figure 16.1 illustrates pointing to an array.

To point at a single slot in an array, we use the array name with both an ampersand and a subscript, as in `wp = &z[2]`. To set a pointer to the beginning of an array, we could write `zp = &z[0]` or, more simply, omit both the ampersand and subscript and write `zp = z`. We normally use the second form. This simpler syntax works because, in C and in C++, the name of an array is translated as a pointer to the first slot of that array.[2] Similarly, there are two ways to set a pointer to the third array element: `zp = &z[2]`, as just discussed, and `zp = z+2`, which is discussed in a later section.

16.2.2 Using Pointers

To use pointers skillfully, the meanings of several pointer operations must be understood. These include subscript, direct and indirect reference, direct and indirect assignment, input and output through pointers, and pointer arithmetic. The following paragraphs review or present these operations.

Pointers with subscripts. Syntactically, there is no difference between using an array name and a pointer to an array. If the referent of pointer `p` is one of the slots within an array, then `p` can be used with a subscript to refer to the other slots of that array. For example, in Figure 16.1, the pointer `zp` refers to array element `z[0]`, so using a subscript with `zp` means the same thing as using a subscript with `z`, and `&zp[1]` means the same thing as `&z[1]`. The subscript used with a pointer is interpreted relative to the pointer's referent. In Figure 16.1, pointer `wp` refers to `z[2]`, so `wp[0]` means the same thing as `z[2]`, and `wp[1]` would be the slot after the end of the diagrammed array. The ability to use a **pointer with a subscript** is important because array arguments are passed by address. Within a function, the parameter is a pointer to the beginning of the array argument. The notational equivalence of arrays and pointers lets us use subscripts with array parameters.

[2]This nonuniform syntax causes confusion for many people. It permits an efficient implementation with an economy of notation at the expense of clarity. When C was a new language, efficiency was a major concern, and the C language developers expected only experts to use it.

An indirect reference accesses the value of the pointer's referent. These declarations create two `double` variables and two objects of class `PointT`. The assignments change the values marked by a large X.

```
double w = 16.2;
double* wp = &w;
double x;
PointT corner = {1.5, -2.0};
PointT* cp = &corner;
PointT c2;

x = *wp;   // Refers to a double.
c2 = *cp;  // Refers to an object.
w = cp->x; // Refers to a member.
```

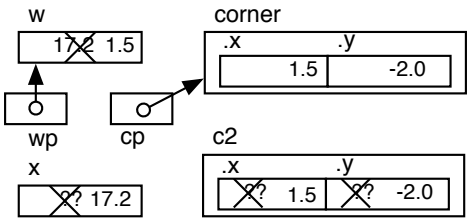


Figure 16.2. Indirect reference.

Indirect reference. Two operators in C++ dereference pointers: the asterisk (`*`) and the arrow (`->`). The expression `*p` stands for the referent of `p`, the same way a pronoun stands for a noun. We say that we can use `p` to reference `k` *indirectly*. If `p` points at `k`, then `m = *p` means the same thing as `m = k`. Longer expressions also can be written; anywhere that you might write a variable name, you can write an asterisk and the name of a pointer that refers to the variable. For example, the expression `m = *p + 2` adds 2 to the value of `k` (the referent of `p`) and stores the result in `m`. Figure 16.2 shows two more examples of this indirect referencing.

The arrow operator, `->`, is used only for pointers that refer to objects. It gives a convenient way to dereference the pointer and select a member of the object in one operation. Therefore, in Figure 16.2, the expression `cp->x` would mean the same thing as `(*cp).x`; namely, "dereference the pointer `cp` and select the member named `x` from the object that is the referent of `cp`." Programmers should use the "`cp ->`" notation rather than the clumsier "(*cp)." notation. (When the expression is written with the asterisk, parentheses are necessary because of the higher precedence of the `.` operator.) Pointers often are used to process arrays of structures; in such programs, the arrow operator is particularly convenient.

Direct reference. Like any other variable, we can simply refer to the value of a pointer, getting an address. This address can be passed as an argument to a function, stored in another variable, and so forth. For example, the assignment `p1 = p2;` copies the address from `p2` into `p1` so that both point at the same referent.

Direct and indirect assignment. C++ permits assignment between any two simple variables, objects, or pointers that have the same type. The meaning is the same for any type: The value of the expression on the right is copied into the storage area for the object on the left. With pointers, an assignment can be either direct, as in `p2 = ??`, or indirect, as in `*p2 = ??`. A direct assignment changes the value stored in the pointer and makes the pointer refer to a different variable. In contrast, an **indirect assignment**, which is written with an asterisk in front of the pointer name, changes the value of the underlying variable. For example, if `p` points at `k`, then `*p = n` means the same thing as `k = n`. Figure 16.3 gives an example of each of these assignments.

Input and output. Finally, let us look at the use of pointers with `>>` and `<<`. This is illustrated in Figure 16.4, where the pointer `ap` refers to `a[0]`. We use `ap` to read data indirectly into `a[0]`, then we read input directly into `a[1]`. Similarly, we print the first input indirectly, the second one directly.

Printing a pointer. The memory address contained in a pointer also can be printed and will appear in hex: : `cout <<ap`. This technique can be useful when debugging a pointer program.

Indirect assignment changes the value stored in the pointer's referent; direct assignment changes the pointer itself.

```
int k;
int* p;

p = &k;   // Direct assignment.
*p = 17;  // Indirect assignment.
```

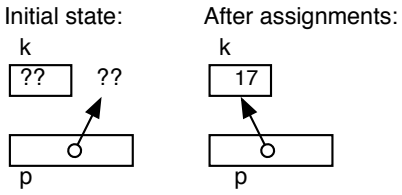


Figure 16.3. Direct and indirect assignment.

16.2.3 Pointer Arithmetic and Logic

Some, but not all, of the operations defined for numbers also are defined for pointers. Addition, subtraction, comparison, increment, and decrement are defined because these have reasonable and useful interpretations with pointers. Division, multiplication, modulo, and the logical operators are not defined for pointers because they have no meaning.

Addition and subtraction with pointers. All **pointer arithmetic** relates in one way or another to the use of pointers to process arrays. If a pointer points at some element of an array, then adding 1 to it makes it refer to the next array slot, while subtracting 1 moves a pointer to the prior slot. This is true of all arrays, not just arrays of 1-byte characters. A pointer and the array it points into must have the same base type, and the size of that base type is factored in when pointer arithmetic is performed. In Figure 16.5, `ip1` points at the beginning of the array, which has memory address 100. (The actual content of the pointer variable is the address 100.) Similarly, `ip2` points at the fourth slot of the array, which has memory address 106.

Adding `n` to a pointer creates another pointer `n` slots further along the array. In Figure 16.5, we set `ip1` to point at the beginning of the array `ages`. So `ip1 + 1` refers to the second array slot, at memory address 102. Now we can address any slot in the array by adding an integer to `ip1` or `ages`. As long as the integer is not negative and is smaller than the length of the array, the result always will be a pointer to a valid array slot.[3] For example, `ages[5]` is the last slot in the array. So, `ip1+5` refers to the same location as `ages[5]`.[4] Warning: The result of a pointer addition is an address, not a data value Do not use pointer arithmetic instead of a subscript for normal processing. Use it only to set pointers.

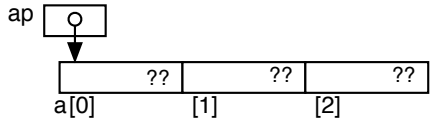
[3]Any address resulting from pointer arithmetic refers to the beginning (not the middle) of an array slot because C adds or subtracts a multiple of the size of the base type of the array.

[4]Both also are synonyms for `ages+5`, because the name of an array is translated as a pointer to slot 0 of the array.

Two declarations are given on the left, resulting in the memory layout shown. On the right are calls on `cin >>` and `cout <<` and their results.

```
double a[3];
double* ap = a;
```

Initial state:




```
cin >>*ap >>a[1];
cout << fixed <<setprecision(2);
cout <<"Input was " <<*ap <<", " <<a[1] <<endl;
```

After I/O:

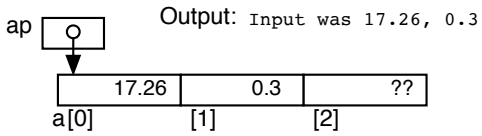
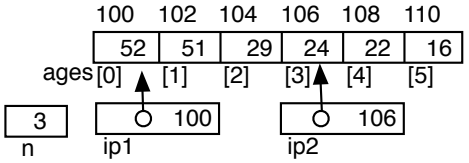


Figure 16.4. Input and output through pointers.

Pointers can be set to refer to an element in an array using the array name and the element's index value. In the diagram, subscripts are given below the array and memory addresses above it.

```
short int ages[6] = {52,51,29,24,22,16};
int* ip1 = ages;
int* ip2 = ages + 3;
int n = ip2 - ip1;
```



If two pointers point at elements in the same array, subtracting one from the other tells you how many array slots are between them.

Figure 16.5. Pointer addition and subtraction.

If two pointers point at slots in the same array, subtracting one from the other gives the number of array slots between them. In Figure 16.5, `ip2 - ip1` is 3. The result of the subtraction `ip2 - ip1` is 3, not 6, because all pointer arithmetic is done in terms of array slots, not the underlying memory addresses.

Pointer subtraction is routinely used to find out how may data values have been input and stored in an array. If `head` is a pointer to slot 0 of the array and `last` is a pointer to the first unfilled array slot, then `last - head` is the number of values stored in the array.

Increment and decrement with pointers. Sometimes, pointers, rather than integer counters, are used to control loops. The increment and decrement operators make pointer loops easy and convenient. Increment moves a pointer to the next slot of an array; decrement moves it to the prior slot. For example, in Figure 16.6, the pointer `ip1` is initialized to the beginning of the array `ages`, then it is incremented. After the increment, it points to `ages[1]`.

Pointer comparisons. All six comparison operators are defined for use with two pointers that refer to the same data type. Two pointers are equal (`==`) if they point to the same slot, unequal otherwise. In Figure 16.7, `(ip1 == ages)` is true but `(ip1 == ip2)` is false. In general, a pointer `ip1` is less than a pointer `ip2` if the referent of `ip1` is a slot with a lower subscript than that of the referent of `ip2`. Pointer arithmetic can be used in these **pointer comparisons**. In the diagram, `(ip1 < ip2)` is true, as is `(ip2 < ip1+5)`.

A pointer can traverse the elements of an array using pointer arithmetic. In the diagram, subscripts are given below each array and memory addresses above it.

```
int* ip1 = ages;
ip1++;
```

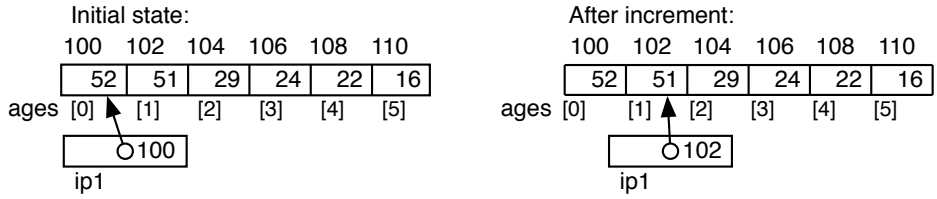


Figure 16.6. Incrementing a pointer.

Comparing pointers or addresses is just like comparing integers.

```
short int ages[6] = {52,51,29,24,22,16};
short int* ip1 = ages;
short int* ip2 = ages + 3;
short int* ipend = ages + 6;
if (ip1 == ip2) ...    // false
if (ip1 > ip2) ...     // false
if (ip2 < ipend) ...   // true
```

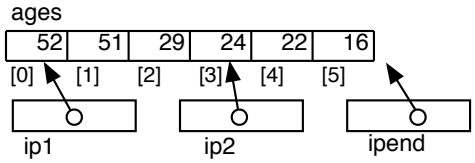


Figure 16.7. Pointer comparisons.

16.2.4 Using Pointers with Arrays

There is a very close relationship between pointers and arrays: A pointer to an array can be used with a subscript, just like the array itself. The difference between a pointer to an array and the name of an array is that the latter refers to a particular area of storage, while the former acts like a pronoun that can refer to any array or any part of an array. In this section, we show how pointers can be used to process an entire array.

A major use of pointers in C++ is sequential processing of arrays, especially dynamically allocated arrays. Three kinds of pointers are commonly used for this purpose: head pointers, scanning pointers, and tail pointers. A **head pointer** stores the address of the beginning of the array that is returned by `new` when an array is allocated dynamically. The name of an array that is declared with dimensions can be treated like a head pointer in most contexts. A **tail pointer** is initialized to the end of an array. Both head and tail pointers, after they are set, should remain constant during their useful lifetime, and the head pointer is used to free the storage. In contrast, a **scanning pointer**, or **cursor**, usually starts at the head of an array, points at each array slot in turn, and finishes at the end of the array.

We can use pointer arithmetic to cycle through the elements of an array with great efficiency and simplicity. The scanning pointer initially is set to the head of the array, as shown in Figure 16.8. The tail pointer, often called a *sentinel*, is used in the test that terminates the loop. It points either at the last slot in the array (**on-board sentinel**) or at the first location past the end of the array (**off-board sentinel**). During scanning, all unprocessed array elements lie between the cursor and the sentinel. To process the entire array, we use a **scanning loop** to increment the `cursor` value so that it scans across the array, dereferencing it (`*cursor`) to access each array element in turn. We leave the loop when the cursor reaches the end of the array; that is, when it surpasses an on-board sentinel or bumps into an off-board sentinel. Figure 16.9 shows such a loop that will print the contents of an array. The upper diagram shows the positions of the pointers after the first pass through the loop. The lower diagram shows the positions of the pointers after exiting from the loop.

Measuring progress with pointer subtraction. We can use pointer arithmetic to calculate how much of an array has been processed and how much remains. Remember that, if `pp1` and `pp2` are pointers and

Suppose we start with the declarations on the left. Array `ages` contains six integers. A cursor is initialized to the head of `ages` and an off-board sentinel is set to point at its end.

```
#define N  6
short int ages[N] = {52,51,29,24,22,16};
short int* cursor = ages;
short int* offBoard = ages + N;
```

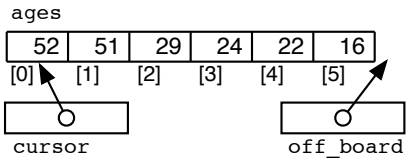
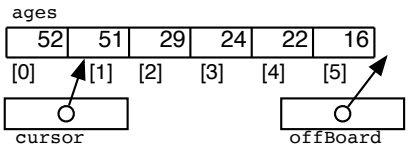


Figure 16.8. An array with a cursor and a sentinel.

Starting with the declarations in Figure 16.8, we write a loop to process the `ages` array:

```
for (cursor = ages; cursor < off_board; ++cursor) {
    cout <<*cursor ;
}
```

After the first pass through the loop, the output is: `52`



After exiting from the loop, the output is: `52 51 29 24 22 16`

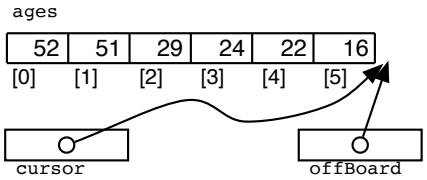


Figure 16.9. An increment loop.

both are pointing at the same array, then `pp2 - pp1` tells us how many array slots lie between `pp1` and `pp2` (including one end of the range). Remember also that an unsubscripted array name is translated as a pointer to the beginning of the array. Therefore, if we are processing an array sequentially, we can calculate how many array slots already have been processed by subtracting the array name from a cursor pointing to the first unprocessed slot. For example, consider the loop in Figure 16.9. At the beginning of the loop, `cursor-ages == 0`; after the first pass through the loop, `cursor-ages == 1` because the cursor has been incremented and one data value has been printed. Similarly, subtracting the cursor from a sentinel tells us how many array slots are left to process between the cursor and the end of the array. For example, after the increment operation in Figure 16.9, `offBoard - cursor` is `5`. This information can be useful in any program that processes an array using pointers.

Subscripts vs. pointers. Programmers converting to C or C++ from other languages often prefer to use subscripts. However, experienced C programmers use pointers more often, because once mastered, they are simple and convenient to use. Many applications of arrays use the array elements sequentially, visiting the slots in either ascending or descending order. For these situations, pointers provide a more concise and more efficient way to code the application. Every time a subscript is used, the corresponding memory address must be computed. To do so, the compiler generates code to multiply the subscript by the size of the base type, then adds the result to the base address of the array. In contrast, when you increment a pointer, the compiler generates code to add the size of the base type to the contents of the pointer. No multiplication is needed, just one addition. Also, once a pointer has been set to refer to a desired location, its value can be used many times without further computations. So, overall, sequential processing is more efficient with pointers than with subscripts. Therefore, experienced programmers often prefer using pointers when an array is to be processed sequentially but use subscripts for nonsequential access to array elements or for processing parallel arrays.

16.3 Dynamic Memory Allocation

When an array is declared with a constant length, like the `dataList` array in Figure 10.29, a C or C++ translator calculates the size of the array at compile time. At run time, the predetermined amount of storage is allocated and the program must work within fixed array boundaries. In many applications, though, the amount of data to be processed and, therefore, the length of an array to store the data are not known ahead of time. A sort program is a good example of an application that may operate on a small or large amount of data; the operation of the program is not tied to any particular amount of data. However, defining a maximum array length at compile time limits the usefulness of a sort program to data sets smaller than that maximum.

We can eliminate this artificial restriction by using pointers and **dynamic memory allocation**. We can write a program that can sort any amount of data that will fit into the memory of the computer. If the initial amount of memory is inadequate, the array is resized to be able to contain the entire data set.

This makes the program much more flexible than one with a `#define`d array length. Image processing and graphics applications profit from dynamically allocated memory, because images and graphical objects come in many sizes, from small to large, but the processing methods remain the same.

Both C and C++ support a set of functions for creating and handling dynamically allocated memory. When given the required size of a memory area, these functions interact with the operating system and ask it to reserve an additional block of memory for the program's use. The beginning address of this block is returned to the program as a pointer value that can be saved and later used to access the memory. The dynamic-memory functions for C++ are listed below and described in more detail in the subsections that follow. The corresponding functions in C are listed in Appendix E.

Dynamic memory allocation functions. These functions are defined in C++.

- `new TypeName;`
  Allocate a block of memory large enough to store this type and some bookkeeping information. Initialize the memory using the TypeName constructor. Return a pointer to the beginning of the initialized object[5].
- `new TypeName[n];`
  Allocate a block of memory large enough to store an array of n objects of this type, plus some bookkeeping information. Initialize the memory by using the TypeName default constructor n times. Return a pointer to the beginning of the array. The array length, n must be stored as part of this allocation because it is needed later to free the memory.
- `delete pointerName;`
  Recycle the memory block pointed at by the pointerName[6]. Return it to the operating system for possible future use. A block that was created using `new` should be deleted when it no longer is needed by the program. Objects that were created by declarations must never be manually deleted. (Deletion for local variables happens automatically when the variable goes out of scope.)
- `delete[] pointerName;`
  Recycle the entire memory block pointed at by the pointerName. First, however, run the destructor for the base type of the array n times to free all memory attached to the objects stored in the array.

16.3.1 Simple Memory Allocation

When a programmer cannot predict the amount of memory that will be needed by a program, the `new` can be used at run time to allocate the desired number of bytes. For example, we can use `LumberT` from Chapter 13:

[5] New is analogous to malloc and calloc in C
[6] Delete is analogous to free in C

This is the conceptual model of what happens when you call `new`.

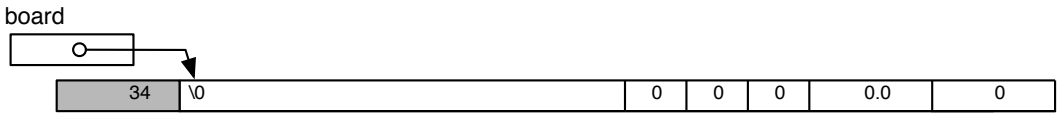
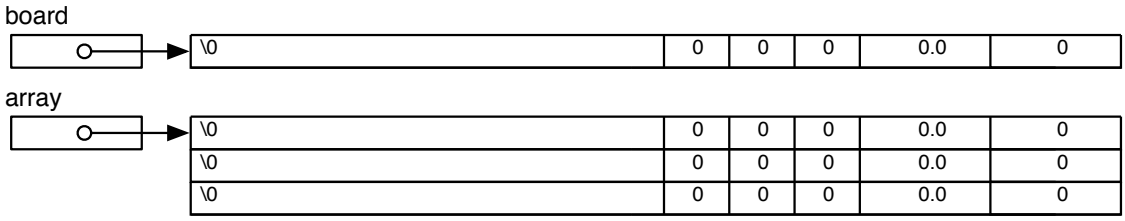


Figure 16.10. Allocating new memory.

```
LumberT* board = new LumberT;                // Allocate one object.
LumberT* array = new LumberT [3];            // Allocate three objects.
```

Before returning, the default constructor for LumberT will be called to initialize the objects. Conceptually, this code allocates memory areas like the ones in Figure 16.10



Actually, the allocation is somewhat more complex, although the programmer rarely needs to know about it, and the implementation is not covered by the standard. The truth is, some additional storage is allocated to allow the system to manage the dynamic memory. The area is, at least, the size of a long integer, and the location is normally immediately preceding the first byte of the new object. The gray area in the diagram represents these additional bytes that the C++ system sets aside; it stores the total size of the allocated block (the size of a `size_t` value plus the size of the white area). The importance of these bytes becomes clear when we discuss `delete`.

In old `C`, it was necessary to check for the success of any request for dynamic allocation. In C++, that is not necessary and not appropriate If the computer does not have enough memory to satisfy the request for a new allocation, a `bad_alloc` exception will be thrown. Unless an exception handler is written as part of the code, this will terminate execution.

Freeing Dynamic Memory

In many applications, memory requirements grow and shrink repeatedly during execution. A program may request several chunks of memory to accommodate the data during one phase then, after using the memory, have no future need for it. Memory use and, sometimes, execution speed are made more efficient by recycling memory; that is, returning it to the system to be reused for some other purpose. Dynamically allocated memory can be recycled by calling `delete`, which frees a block of memory by returning it to the system's memory manager. The number of bytes in the gray area at the beginning of each allocated block determines how much memory is freed. The use of `delete` is illustrated in the `Flex` class in Figure 16.11.

While each program is responsible for recycling its own obsolete memory blocks, a few warnings are in order. A block should be deleted only once; a second attempt to delete the same block is an error. Similarly, we use `delete` only to recycle memory areas created by `new`. Its use with a pointer to any other memory area is an error. Another common mistake, described next, is to attempt to use a block after it has been deleted. These are serious errors that cannot be detected by the compiler and may cause a variety of unpredictable results at run time.

Dangling Pointers

A **dangling pointer** is one whose referent has been reclaimed by the system. Any attempt to use a dangling pointer is wrong. Typically, this happens because multiple pointers often point at the same memory block. When a block is first allocated, only one pointer points to it. However, that pointer might be copied several times as it is passed into and out of functions and stored in data structures. If one copy of the pointer is used to free the memory block, all other copies of that pointer become dangling references. A dangling reference may seem to work at first, until the block is reallocated for another purpose. After that, two parts of the program will simultaneously try to use the same storage and the contents of that location become unpredictable.

To avoid the problems caused by dangling pointers, the programmer must have a clear idea about which class, and which variable within the class, is responsible for keeping the "master pointer" to each dynamic object, and later, for freeing the dynamic memory.

Memory Leaks

If you do not explicitly free the dynamically allocated memory, it will be returned to the system's memory manager when the program completes. So, forgetting to perform a `delete` operation is not as damaging as freeing the same block twice.

However, some programs are intended to run for hours or days at a time without being restarted. In such programs, it is much more important to free all dynamic memory when its useful lifetime is over. The term *memory leak* is used to describe memory that should have been recycled but was not. Memory leaks in major commercial software systems are common. The symptoms are a gradual slowdown in system performance and, eventually, a system "lock up" or crash.

Thus, it is important for programmers to learn how to free memory that is no longer is needed, and it is always good programming style to do so. This is even true when the memory is used for a short time by only one function. Functions often are reused in a new context, they always should clean up after themselves.

Using Dynamic Arrays

A pointer to a dynamic array can be used with a subscript, just like the array itself, as shown in this code fragment below that allocates space for `n` long integers, then reads that many numbers from `cin`:

```
lptr = new long[n];
for (int k = 0; k < n; ++k) cin >> lptr[k] );
```

The close relationship between an array pointer and an unsubscripted array name makes it very easy to take an application written for ordinary arrays and convert it to use dynamic arrays. As an example, consider the selection sort program from Figure 10.29. This program consists of `main()` and four other functions and uses an array whose length is defined at compile time. To allow this program to use dynamic allocation in C , only six lines in the main program need be changed; namely, the lines that determine the length of the array and allocate it. None of the function definitions or prototypes need to be modified. We will revisit the selection sort program, written in C++ later in this chapter after presenting dynamic arrays that can grow, as needed.

16.4 Arrays that Grow

When we declare an array length in a program, we then must write code to guard that array and ensure that no part of the program walks off the end of the array and stores information into adjacent memory cells. Array-based programs become longer and more complex because of these efforts. The situation is worst when allocating an array to hold input. How do we guess how much input there will be? How do we detect when there is not enough space, and what do we do if there is more input?

Using `new` to allocate arrays at run time is the first step in removing restrictions on the length of an array. However, that array space still must be allocated before the data are read and before we know how many data items actually exist. A dynamic array still cannot accommodate an amount of data greater than expected.

Flex Arrays. To remove the size restrictions, we need to be able to *resize the data array.* We want to replace the existing (too small) memory block by a larger one that contains the same data. To do this we define a class that contains the dynamic array, the two integers needed to manage it, and functions to handle resizing and related needs. The Flex class needs three data members:

- The dynamic array.
- The current capacity of the array, *max* (the number of slots in it).
- The number of data items currently stored in the array, n

At least two function members are needed. The first is `push_back()`, which puts data into the first unoccupied slot in array. The second is `grow()`, which actually does the resizing. The `push_back()` method has two steps:

- Check whether there is currently space in the array for another data item. If not, call `grow()`.
- Whether or not growth happened, put the additional data item into the first empty slot in the array, and increment n.

The `grow()` method has several steps:

- Use a temporary variable to point at the existing array.
- Double *max*. By doubling the array capacity each time, we guarantee that the time required to reallocate and copy data will always be acceptable, and that the total number of data items copied in the lifetime of the array will not exceed the current length of the array.
- Allocate a new memory area with the new *max* capacity.
- Copy n data objects from the old memory into the new area.
- Delete the old memory area.

Finally, C++ allows a program to define the subscript operator for that class. Doing so allows the programmer to treat the Flex array as if it were an ordinary array.

16.4.1 The Flex Class

The C++ language supports a sophisticated and generic growing array type named `vector`. A practicing programmer would use `vector` when storage needs are unpredictable. This section presents a much simpler class named `Flex` that works the same way as `vector` but is easier to study. By learning how `Flex` works, the student learns how `vector` works.

Notes on Figures 16.11 and 16.12: Flex: a growing array. This is the header file for the Flex class. The function definitions are in Figure 16.12. This class is used in the selection sort program in Figure 16.19.

First box: dealing with the environment.

- This is the first example program that is composed of three classes and a main program, and the first where a module needs to `#`include two user-defined header files. At this level of complexity, we begin to have a problem with the `#`include statements: the code will not compile with too few of them, or if a header is included twice in the same module or if there is a *circular include.*
- The `#pragma` statement on the first line tells the compiler to include this file once, but if it has already been included in a module, do not include it again. It is good style to include the `#pragma once` whether or not it is needed. In this particular program, it is needed in the file `trans.hpp` but not in the files `charges.hpp` and `flex.hpp`.

```
#pragma once
#include <iostream>
using namespace std;

#include "trans.hpp"
typedef Transaction BT;

#define START 4                 // Default length for initial array.

class Flex {
private:  // ---------------------------------------------------------------
    int max = START;            // Current allocation size.
    int n = 0;                  // Number of array slots that contain data.
    BT* data = new BT[max];     // Allocate dynamic array of base type BT.

    void grow();                // The Flex is full, so double the allocation.

public:  // ----------------------------------------------------------------
    Flex() {}                   // An array of Transactions.
    ~Flex() { delete[] data; }  // Free dynamically allocated data.

    void push_back( BT data );  // Store data and return the subscript.
    int size() { return n; }    // Provide read-only access.
    BT& operator[]( int k );
    void print(ostream& out) { for (int k=0; k<n; ++k) out <<data[k] <<endl; }
};
```

Figure 16.11. Flex: a growing array.

Second box: dealing with the environment.

- This is a generic class, defined in terms of an abstract base type, BT. This class can be used to hold data of any base type by defining BT to mean the target type. We use Flex in the selection sort program to hold a series of transactions. To make that possible, we need to include the transaction header here.
- We use a typedef to say that the BT, is really the Transaction class.

Third and fourth boxes: Private data members.

- In Flex, as in most classes that define data structures, the data members of the class must not be modified by any other class. If they *were* modified, the data structure would malfunction. Thus, the data members are private.
- In this class, it is possible to give meaningful initializations for the data members at compile time. We choose an arbitrary small number, `START` as an initial allocation size. If it is too large, little memory is wasted. If it is too small, it will grow. We don't need to guess, and this number does not need to depend on the application.
- Since the initial array length is a constant, and a new Flex is always empty, it is possible to initialize all of the data members in the class declaration. That permits us to have a constructor that does nothing.

Fifth box: a private function.

- Just as it would be disastrous to let another class modify n or max, it would make a mess if another class called `grow()` at the wrong time. Therefore the `grow()` function is private.

```
#include "flex.hpp"
// ----------------------------- Store object in array.  Grow first, if needed.
void Flex ::
push_back( BT obj ) {
    if ( n == max ) grow();        // Create more space if necessary.
    data[n] = obj;
    ++n;
}
// ---------------------------------------------- Double the allocation length.
void Flex ::
grow() {
    BT* temp = data;                            // Hang onto old data array.
    max *= 2;
    data = new BT[max];                         // Allocate a bigger one.
    for (int k=0; k<n; ++k) data[k] = temp[k]; // Copy info into new array.
    delete temp;                                // Recycle (free) old array.
}
//---------------------------------------- Access the kth char in the array.
BT& Flex ::
operator[]( int k ) {
    if ( k >= n || k < 0 ) fatal("Flex bounds error.");
    return data[k];              // Return reference to desired array slot.
};
```

Figure 16.12. Flex functions.

- In this class, it is possible to give meaningful initializations for the data members at compile time. We choose an arbitrary small number, `START` as an initial allocation size. If it is too large, little memory is wasted. If it is too small, it will grow. We don't need to guess, and this number does not need to depend on the application.
- Since the initial array length is a constant, and a new Flex is always empty, it is possible to initialize all of the data members in the class declaration. That permits us to have a constructor that does nothing.

Sixth box: the constructor and destructor.

- The Flex constructor is empty; there is no work to be done. All the initializations were done in the fourth box.
- The Flex class does dynamic allocation, therefore it must take the responsibility of freeing the dynamic memory when it is no longer needed. This is what a destructor is for.
- The name of a destructor is a tilde (∼) followed by the name of the class. All dynamic memory blocks created anywhere in the class must be deleted by the destructor.
- There is only one dynamic memory block, an array, and its pointer is stored in `data`. Therefore, we write `delete[] data;`

Seventh box and Figure 16.12: the class functions.

- Two of the class functions are inline, and one-line definitions are given here. They are `size()` and `print()`. The other three functions, `grow()`, `push_back()`, and `operator[]` are defined in `flex.cpp`, the implementation file, which is shown in Figure 16.12 and included in the comments here.

- The names `push_back()` and `size` are the same as the names of the corresponding function in the standard `vector` class.
- `push_back()`.
  This function is the only correct way to store new data in the Flex array. It tests for a full array and, if found, extends the array. The class member n is always an accurate count of the number of objects in the array.
- `grow()`.
  The `grow()` function was explained in Section 16.4. It is a private function because only the Flex class should ever use it. It is called when there is a request to store another thing in the Flex array but the array is full.
- `subscript()`.
  To define the subscript operator for this class, we write a definition of `operator []( int );` Writing operator definitions is beyond the scope of this text. However, an operator definition is necessary here to allow subscript to be used in the ordinary way. The student does not need to understand this code at this time.
- In this definition, the Flex class delegates the subscript operation to the dynamic array and returns the value that the array subscript returns. The return value is the address of one slot in the array. This allows us to use subscript for either reading or writing an array element.
- Note that the code performs a bounds check, and will not allow storing into a subscript outside of the filled portion of the Flex array.

16.5 Application: Insertion Sort

The first application of the Flex array is an insertion sort algorithm. Insertion sort is one of two easy sorting algorithms that are often useful and should be learned[7].

On the surface, the insertion sort algorithm looks a lot like the selection sort studied earlier; both have an outer loop that executes $n-1$ times to sort n items. Both have an inner loop, but this is where the similarity ends. When doing selection, we repeatedly examine the remaining *unsorted items* to find the largest one. We must look at all of them to find the largest. When found, we swap it with the last unsorted value in the array.

In the **insertion sort** algorithm, the data array is divided into two segments: the sorted portion first, followed by the unsorted portion. To sort, we pick the first unsorted item and then scan backwards through the list of *sorted items* from the high end to the beginning of the array, looking for the correct position for the new item. As soon as we locate an item smaller than the one being inserted (assuming we sort in ascending order), we stop looking. On the average, this search for the correct insertion slot takes fewer comparisons than looking through the remaining unsorted items to find the next smallest item during a selection sort.

Since we can't just create new space for a value between two others in an array; we have to move some of them to make space for an insertion. (See Figure 16.13.) We do this by shifting each sorted data item one slot toward the end of the array as we pass it during our search, in essence moving a hole backwards along the array, so that it is always adjacent to the item we are testing. When the proper insertion slot is found, we simply store the current value in our hole. This sequence of movements requires more work than the single data swap in a selection sort. However, because the number of comparisons needed is smaller, insertion sort usually is faster.

This insertion process is repeated `n-1` times. After each of the `n-1` passes, the sorted part is one item longer and the unsorted part is one item shorter. An example is illustrated by the diagrams in Figure 16.16, which show the intermediate states of an array of 10 items after each pass.

[7] The other useful easy sort is selection sort, which is presented later in this chapter. A third sort in the same general category is bubble sort, which should never be used for any purpose because it can take twice as long to execute as insertion sort.

This represents the fifth pass through the array.

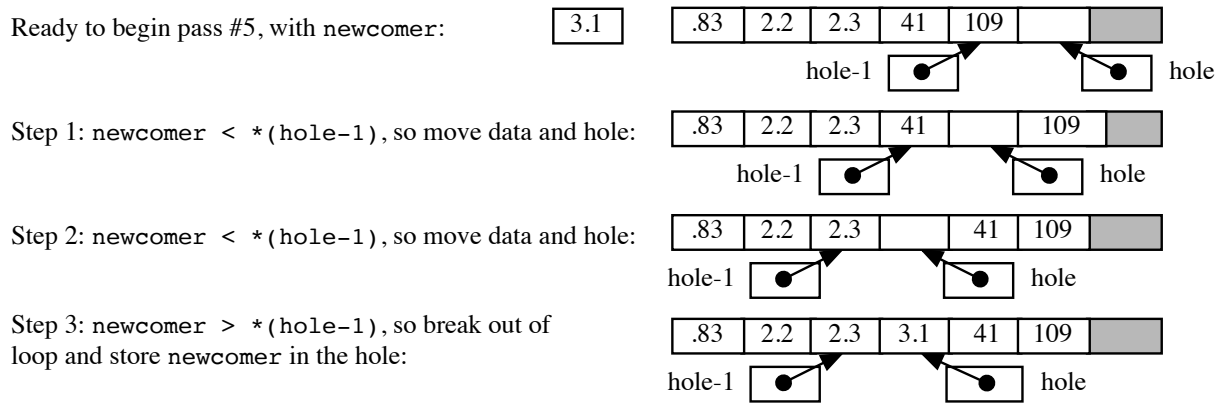


Figure 16.13. Inner loop of the Insertion sort.

The insertion algorithm is implemented by the function `insertionSort()` listed in Figure 16.18. Pointers are used, instead of subscripts, to access the data array. The main program for insertion sort is in Figure 16.15. Following that are the controller class, `Sorter` and the data class, `Transaction`. The structure of the application is diagrammed in Figure 16.16.

The controller class uses a Flex array to store all the data from an input file. Flex arrays are dynamically allocated data structures that create, manage, and free the dynamic memory they use. The client class, in this case Charges, can rely on this dynamic flexibility but does not need to take any responsibility for it, since all management is handled by the Flex class.

Notes on Figure 16.15: Insertion sort. This is the main program for the insertion sort. It uses the Sorter class defined in Figures 16.17 and 16.18 and the Flex array class in Figures 16.11 and 16.12.

First box: Dealing with the environment.

- This main program uses the class `Sorter`, and includes the header for that class.
- The `#define` statement supplies the name of the input file as a quoted string, a form that is appropriate for opening the file.

Second box: Opening the file.

- The input stream is declared and opened in the same line. This is better style than using two lines to do the two actions separately.
- As usual, we test for a properly opened file. The error comment has three parts: it begins and ends with a quoted string and has the `#defined` symbol between the quoted strings. This uses two advanced features of the C language: macros and string merging.
- Macros: `#define` commands are interpreted at compile time, not at run time. Wherever you write the `#defined` symbol, the compiler's preprocessor removes what you wrote and replaces it by the definition of the symbol. So these two lines:
  ```
  ifstream fIn( FILE );
  if (!fIn.good()) fatal( "Cannot open " FILE " for reading."); are replaced by this code
  ```
  prior to compilation:
  ```
  ifstream fIn( "insr.in" );
  if (!fIn.good()) fatal( "Cannot open " "insr.in" " for reading.");
  ```
- String merging: Whenever two strings are written in a program and the quote marks are separated by nothing except possible whitespace, the compiler will merge the two strings into a single string. In this

Sorting 10 items with a pointer-based insertion sort.

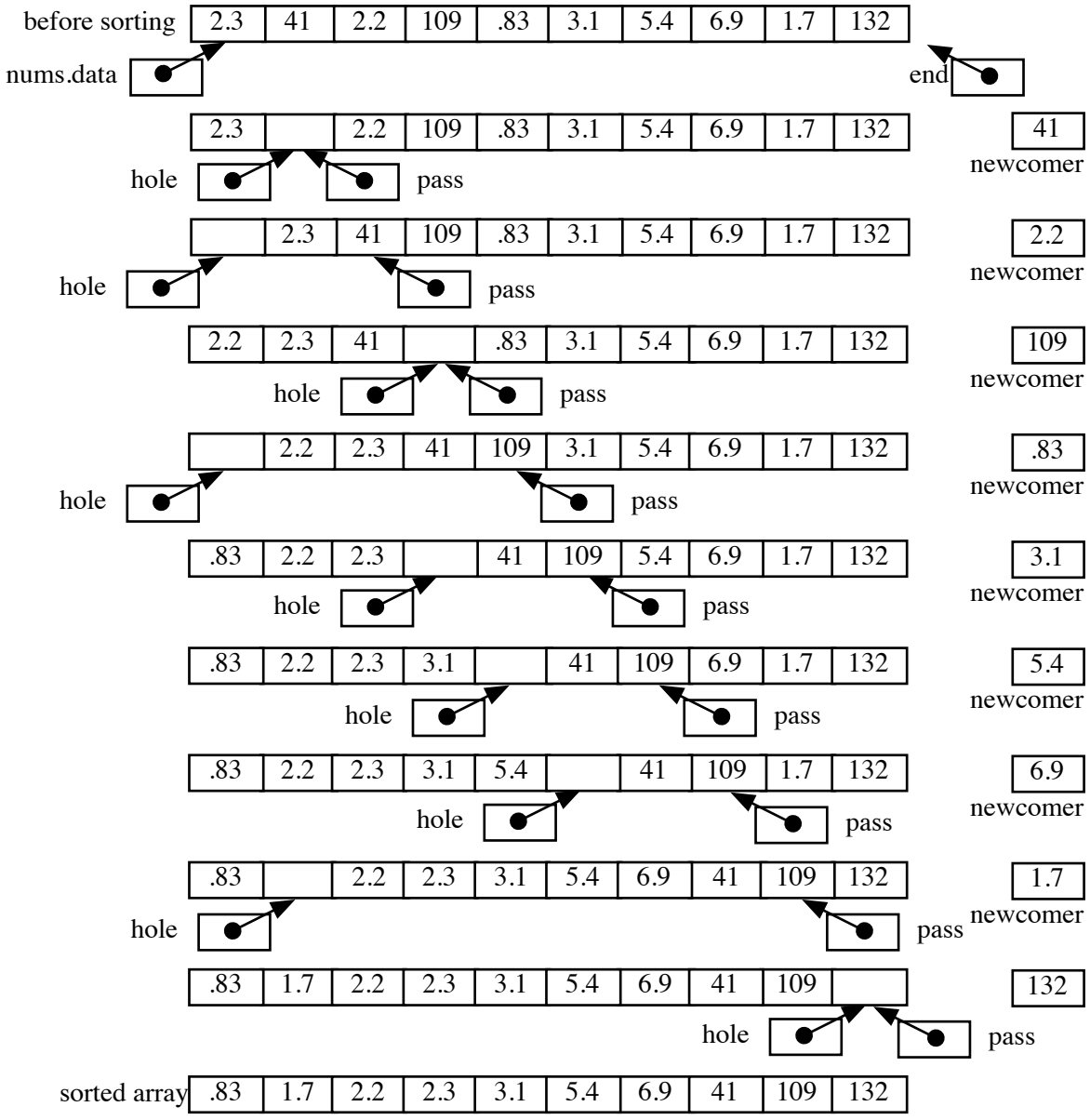


Figure 16.14. Insertion sort.

```
#include "sorter.hpp"                // File:  main.cpp
#define FILE "insr.in"

int main( void ) {
    cout <<"Insertion Sort Program for type float\n";
    ifstream fIn( FILE );
    if (!fIn.good()) fatal( "Cannot open " FILE " for reading.");

    Sorter nums( fIn );
    cout << nums.getN() <<" numbers were read; beginning to sort.\n";
    nums.insertionSort();
    cout <<"\Data sorted, ready for output\n";
    nums.print( cout );
    return 0;
}
```

Figure 16.15. Insertion sort using a dynamic array.

case, the result from the preprocessor is three adjacent strings, and they are merged to form one string, thus:

```
if (!fIn.good()) fatal( "Cannot open insr.in for reading.");
```

This is what the compiler sees and compiles: a single string.

Third box: Doing the work.

- This main function follows the ordinary form for an OO main program: (1) Call the constructor to create an object. (2) Use it to get the work done. (3) Print the results.
- We don't try to sort the data or print it here in the main function. The Sorter class is the expert on this data set, so we delegate the sorting and printing operations to the expert.

A call-chart for the insertion sort.

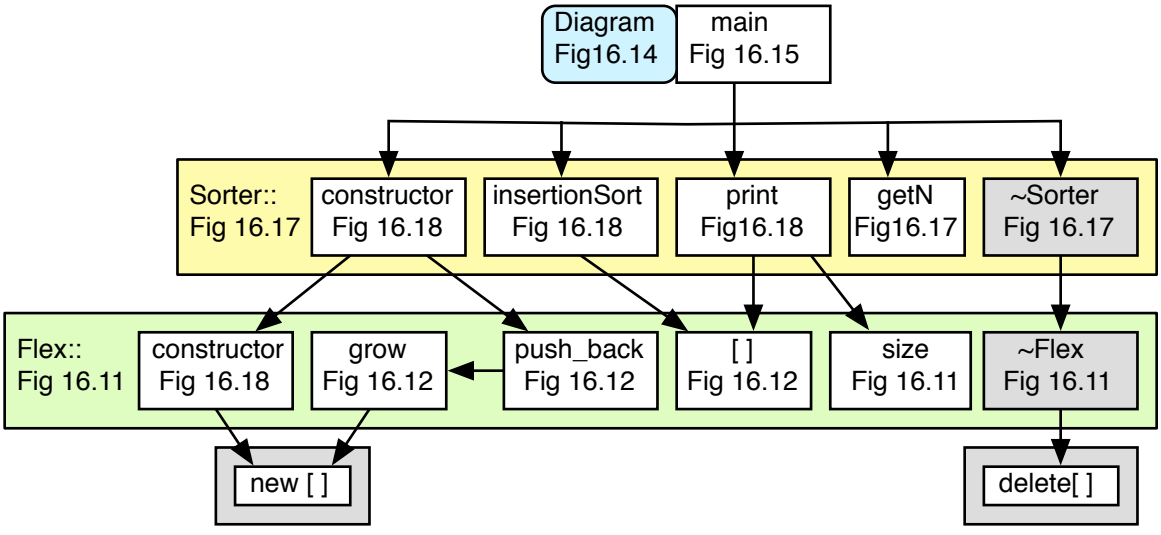


Figure 16.16. Insertion sort.

```
#pragma once                              // File:  sorter.hpp
#include "tools.hpp"
#include "flex.hpp"

class Sorter {
private:
    Flex data;                            // Dynamic data array.
    istream& inFile;

public:
    Sorter( istream& inF );               // Constructor for the controller.

    void print( ostream& out );           // Output the data array.
    void insertionSort();
    int getN() { return data.size(); }
};
```

Figure 16.17. The Sorter class.

- After each phase is a line of output. This kind of output is important during program development and debugging. It also lets the user know what is going on inside the program.

Notes on Figures 16.17 and 16.18: The Sorter class. This is the header file for the Sorter class. The function definitions are in Figure 16.18. This class is used in the insertion sort program in Figure 16.15.

First box: Dealing with the environment.
This controller class instantiates a Flex array and uses it to store floats. Therefore, we include the header file for Flex arrays. By including the tools header file, we bring in all the standard headers we are likely to need, and the required namespace declaration.

Second box: Data members. There are only two data members: an open istream to supply the data and a Flex array to store it. They are not initialized because we do not have the information, at compile time, to write meaningful initializations. Both will be initialized by the constructor.

Third box Figure 16.17 and first box, Figure 16.18: the constructor

- Main will open the input file and pass an open `istream&` as a parameter. The only way to use an `&` parameter is a ctor (constructor initializer), shown in the small box on the first line of the constructor. The ctor takes the parameter (named inF) and stores it in the data member named `inFile`.
- Bringing a database from a file into the program is something that a constructor often does. It is part of making the class ready to use.
- The second box in the constructor is a loop that reads one line of the input file at a time and does a minimal check for read errors and end-of-file. If either happens, the loop ends and the program processes the data it already has.
- Any illegal character in the file will end the input and will not warn the user about the problem. This is not a good way to design an application! It is done here as a bad example of file handling. The selection sort program later in this chapter does a better job of error handling.
- If there *was* good data, then we push it onto the end of the Flex array. If the array is full, it will grow.

Fourth box in Figure 16.17: an accessor. The `getN()` function is a typical accessor. The main function calls it in order to provide high-quality user feedback. It is not used by the sorting algorithm. It is inline to increase efficiency.

Fourth box in Figure 16.17 and second box in Figure 16.18: sorting.

- This sort algorithm is written to use pointers, not subscripts. Note that pointers can be incremented and decremented, just like subscripts, and that we use an asterisk here instead of a subscript to access a data value.
- Note that this code uses helpful names, `newcomer` and `hole`, instead of using variables named `i` and `j`. Using meaningful names makes code far, far easier to understand.

```
Sorter::Sorter( istream& inF ) : inFile( inF ) {
    float temp;
    int k;
    for (k=0; ; k++) {
        inFile >> temp;
        if (! inFile.good()) break;
        data.push_back( temp );
    }
}
```

```
void Sorter::
insertionSort() {
    float* head = &data[0];   // The beginning of the array.
    float* end = head + data.size();            // Off-board sentinel.
    float* pass;              // Starting point for pass through array.
    float* hole;              // Array slot temporarily containing no data.
    float newcomer;           // Data value being inserted on this pass.

    for (pass = &data[1]; pass < end; ++pass) { // Outer loop; n-1 passes.
        newcomer = *pass;                       // Pick up next item,
        // ...and insert into the sorted portion at the head of the array.
        for (hole = pass; hole > head; --hole) {
            if (*(hole-1) <= newcomer) break;   // Insertion slot is found.
            *hole = *(hole-1);                  // Move item back one slot.
    }
    hole = newcomer;
}
```

```
void Sorter::
print( ostream& outFile ) {
    float* cursor = &data[0];
    float* end = cursor + data.size();  // an off-board sentinel

    for( ; cursor<end; ++cursor) {      // Use a pointer to walk the array.
        outFile << *cursor <<'\n';
    }
}
```

Figure 16.18. Insertion sort using a Flex array.

- The logic of the `insertionSort()` algorithm is as follows:
  1. To sort n items, write an outer loop that will execute $n - 1$ times.
  2. In the body of the loop, pick up a copy of the first unsorted data value and call it the *newcomer*. A list of one item is always sorted, so start the first pass with the value in data[1]. This leaves a hole in the array at subscript 1.
  3. Each time around the outer loop, walk backwards from the hole to the head of the array. At each step compare the *newcomer* to the value in the current array slot. If the *newcomer* is smaller, move the other value into the hole, then decrement the hole-pointer and repeat the inner loop.
  4. Eventually, we come to either the end of the array or a slot containing a value smaller than `newcomer`. At this point, the current `hole` position is where `newcomer` must be inserted, so we terminate the inner loop and put the *newcomer* in the hole.
  5. We might shift as few as no items or as many as currently are in the sorted portion of the array. On the average, we move about half of the sorted items.
  6. The data is sorted after the last value in the array has been inserted into its place.

Fourth box in Figure 16.17 and third box in Figure 16.18: printing.

- The print function uses a pointer-loop to access the data values. To set it up, pointers are defined for the beginning and end of the data in the Flex array.
- As is customary in C and C++, the end pointer is `off-board`, that is, it points at the first unoccupied array slot, not the last occupied slot. Using an off-board pointer makes it easier to write a loop. The language standard guarantees that this is OK.
- At each step, one number is printed with whitespace to separate it from all the other output. It is easy to forget that the whitespace is essential. When forgotten, the output is unreadable.

16.6 Selection Sort

In this section, we present a second version of the selection sort algorithm. This algorithm was first presented in C in Figure 10.29. The version presented here has been transformed to an object-oriented design implemented in C++. Although the logic is the same, the organization of the code is totally changed. The main program is now brief and all the work is done in a controller class (Charges) and a data class (Transaction).

The form and content of this program are very much like the insertion sort program, above. Details are different, and the complexity is greater because the data being sorted are objects, not simple numbers. The controller class uses a Flex array to store the data from an input file. The client class, in this case Charges, can rely on this dynamic flexibility but does not need to take any responsibility for it, since all management is handled by the Flex class.

Notes on Figure 16.19: Selection sort.

First box: Dealing with the environment.

- This main program uses the class `Charges`, and includes the header for that class.
- The `#define` statement supplies the name of the input file as a quoted string, a form that is appropriate for opening the file.

Second box: the body of main(). The code here is exactly parallel to the insertion sort code in Figure 16.15. The comments will not be repeated here.

Notes on Figure 16.20: The Charges class. This is the header file for the Charges class. The function definitions are in Figure 16.21. This class is used in the selection sort program in Figure 16.19. The unboxed parts of this class are exactly parallel to the Sorter class in Figures 16.17 and 16.18, so the comments will not be repeated here.

This is the main program for the selection sort. It uses the controller class, Charges, in Figures 16.20 and 16.21. The data class, Transaction, is in Figures 16.22 and 16.23.

```
#include "charges.hpp"
#define FILE "charges.txt"

int main( void ) {
    cout <<"Club Snack Bar Accounting Program\n";
    ifstream fIn( FILE );
    if (!fIn.good()) fatal( "Cannot open " FILE " for reading.");
    Charges snacks( fIn );
    cout << snacks.getN() <<" transactions were read; beginning to sort.\n";
    snacks.sort();
    cout <<"\nData sorted, ready for output\n";
    snacks.print( cout );
    return 0;
}
```

Figure 16.19. Selection sort using a dynamic array.

This is a controller class. It is instantiated by the main function in Figure 16.19 and carries out the logic of the application. To do this, it creates and operates on an array of data objects (Transactions).

```
#pragma once                                                    // File:  charges.hpp
#include "trans.hpp"
#include "flex.hpp"

class Charges {
private:
    const char* name = "ECECS Snack Club";
    istream& inFile;
    Flex mems;
    int findMaxID( int last);
public:
    Charges( istream& in );
    int getN() { return mems.size(); }
    void sort();
    void print( ostream& out );
};
```

Figure 16.20. The controller class: Charges.

First box: Dealing with the environment.
This controller class instantiates a Flex array and uses it to store Transactions. Therefore, we include the header files for both classes.

Second box in Figure 16.20 and second box in Figure 16.21: a private function.
This class has one private function and three public functions. The `findMaxID()` function is private because it intended for use by the public sort function, and not for use by a client class. The code could be written as an inner loop in the sort function, but the selection sort algorithm is clearer this way.

Third box Figure 16.20 and first box, Figure 16.21: the constructor

- The first inner box in the constructor reads one line of the input file and does a complete error check. The first if statement looks for a normal end of file and leaves the input loop if it is found.
- The second if statement tests for all other input errors. If the stream state is not good, no further input can happen. It must be corrected. We reset the stream's error flags to the good state by calling `clear()`.
- The most minimal action that could possibly correct the problem is to eliminate the character that caused the input error. `ignore(1)` does that job.
- If there *was* good data, then we use it to instantiate a new transaction and immediately put the transaction into the Flex array. If the array is full, it will grow.

Fourth box in Figure 16.20 and third box in Figure 16.21: sorting.

- The `sort()` function implements a selection sort:

  1. To sort n items, write an outer loop that will execute $n - 1$ times.
  2. Each time around the loop, find the largest value in the array between slot 0 and slot $n - 1$, and swap it to the end of the array.
  3. Subtract one from n and repeat the loop.

- Note that this code uses helpful names, `last` and `where`, instead of using variables named `i` and `j`. Using meaningful names makes code far easier to understand.

Fourth box in Figure 16.20 and fourth box in Figure 16.21: printing.

- The last class function, `print()`, relies on two major OO techniques: expertise and delegation. The Flex array is the expert on storing Transactions and it knows how many have been stored. Similarly, the Transaction class is the expert on formatting and printing a transaction. The Charges class is not expert on either of these things.

  So `Charges::print()` calls on `mems`, the Flex array to find out how many items to execute its print loop, then calls `Transaction::print()` to do the printing. In both cases, the action is *delegated* to the expert.

Notes on Figures 16.22 and 16.23: Transactions. This is the data class. It is used in the selection sort program in Figure 16.19.

First box, Figure 16.22: Dealing with the environment.

- This header file must be included twice: in flex.hpp and in charges.hpp because both classes refer to Transactions. Therefore the `#pragma once` declaration is essential.
- By including the tools header file here, we make it unnecessary to include it in flex.hpp and charges.hpp

Second box, Figure 16.22: Data members.
This very simple class records one transaction made by one person. It has only a member's ID and the price of the item he took from the snack bar.

```
Charges::Charges( istream& in ) : inFile( in ) {
    int ID;
    float owes;
    for (;;) {                      // Growing array will hold all data in file.
        inFile >> ID >> owes;
        if (inFile.eof()) return;
        if (!inFile.good()) {       // Remove faulty line after input error.
            inFile.clear();
            cerr <<"Bad data on line " <<mems.size() <<endl;
        }
        mems.push_back( Transaction( ID, owes ) );
    }
}
```

```
int Charges::                           // Find the maximum ID in the array
findMaxID( int last ) {
    int finger = 0;                     // Put your finger on the first ID.
    // Now compare the fingered value to the values that follow it.
    for (int cursor = 1; cursor < last; ++cursor) {
        // If next is bigger, move your finger to it.
        if (mems[cursor].bigger( mems[finger] )) finger = cursor;
    }
    return finger;                      // Your finger is on the biggest value.
}
```

```
void Charges::
sort() {
    int last = mems.size();   // Number of actual data items in the array.
    int where;                // Position of largest value in the index array.
    while (last > 0 ) {
        where = findMaxID( last-- );
        // Swap the two transactions.
        Transaction temp = mems[where];
        mems[where] = mems[last];
        mems[last] = temp;
    }
}
```

```
void Charges::
print( ostream& out ){
    for( int k=0; k<mems.size(); ++k ) mems[k].print(out);
}
```

Figure 16.21. Functions for the Charges class.

This is a data class. It is used by the Charges class, and indirectly, by the main program in Figure 16.19.

```
#pragma once                                                    // File:  trans.hpp
#include "tools.hpp"

// ----------------------------------------------------------------------------
class Transaction {
private:
    int ID;                // Transaction number
    float owes;            // Amount owed by Transaction

public:
    Transaction() = default;
    Transaction( int ID, float owes );

    ostream& print( ostream& out );
    bool bigger( Transaction& b ) const { return ID > b.ID; }
};

inline ostream& operator<<(ostream& out, Transaction& t){ return t.print(out); }
```

Figure 16.22. The data class: Transaction.

Third box, Figure 16.22 and second box in Figure 16.23: Constructors.

- There are two constructors in this class. The default constructor is called by the system when a Charges object is created because Charges contains a Flex array of Transactions. When an array is created for any type, a default constructor is required for that type.
- The second constructor is the most ordinary form there is: it has one parameter per data member, used to initialize the new object.
- The definition of this method in Figure 16.23 is also very ordinary: one assignment statement per pair of a parameter and a data member. The parameters have the same names as the data members of the class, so `this->` must be used to refer to the data members. If the names are different, `this->` is not needed.
- No destructor is needed because there is no call on `new` anywhere in this class.

Fourth box, Figure 16.22: Comparing two transactions.

- The function `bigger()` is public and is called from `Charges::findMax()` to compare two transactions. The Transaction class is the expert on which transaction is bigger, so the Charges class delegates the comparison to it. This is a better design than having `Charges::findMax()` do the comparison itself, after using a getter function to get each of the money amounts.
- We made the function `bigger()` inline because it fits on one line. Being inline improves the time and space efficiency of short functions.

Fifth box, Figure 16.22 and third box, Figure 16.23: Printing.

- The Transaction class declaration is followed by an inline function definition for `operator<<`, the output operator. Operator extensions are beyond the scope of this book, and the student does not need to understand this code at this time.
- Defining this method for `operator<<` enables us to use `<<` to print Transactions. The definition, itself, simply reaches inside the class to call the public print function in the Transaction class.

These are the function definitions for teh Transactions class in Figure 16.22

```
#include "trans.hpp"                                          // File:  trans.cpp
// ------------------------------------------------------------------------------
Transaction:: Transaction( int ID, float owes ) {
    this->ID = ID;
    this->owes = owes;
}

ostream& Transaction:: print( ostream& out ) {
    return out <<"[" <<setw(2) <<ID <<"] " <<fixed <<setw(7) <<setprecision(2)
               <<owes <<endl;
}
```

Figure 16.23. Functions for the data class, Transaction.

- `Transaction::print()` returns an `ostream&` result which is then returned again by `operator<<`. This allows us to use one line of code with a chain of calls on `<<` to print several things, including a Transaction and a newline.

16.7 What You Should Remember

16.7.1 Major Concepts

Pointer operations. To use pointers skillfully, several pointer operations must be understood:

- Direct assignment. To set a pointer, `p`, to point at a selected referent, `r`, write `p = r` for arrays or functions but write `p = &r` for structures or simple variables.
- Indirect assignment. To assign a new value, `v`, to the referent of pointer `p`, use `*p = v` (`v` cannot be an array or function).
- Direct reference. To copy pointer `p` into another pointer `q` write `q = p`.
- Indirect reference. To use the referent of pointer `p` in an expression, write `*p` for simple variables and array elements, `p->member_name` if `p` points at a structure, and simply use `p(...)` if it points at a function.
- Pointer increment. To make pointer `p` point to the next (or previous) slot of an array, write `++p` (or `--p`).
- Pointer arithmetic. To calculate the number of array slots between two pointers, `p` and `q` (where `q` points to a later slot), write `q - p`. Accessing an array element has two equivalent forms; `*(p+5)` and `p[5]` reference the same value. The latter is preferred.
- Pointer comparison. To test whether two pointers, `p` and `q`, refer to the same object, simply use `p == q`. To compare the values of the referents, compare `*p` to `*q` using an appropriate comparison operator or function (`==` or your own function for comparing two objects).

Array and pointer semantics. In a very strong sense, `C` doesn't really have array *types*. An array is simply a homogeneous, sequential collection of variables of the underlying type. We can do nothing with an array as a whole except initialize it and apply `sizeof` to it. When a pointer refers to an array, whether subscripted or not, it refers to only one slot at a time.

Array allocation time. If an array is declared with square brackets and a `#define`d length, its size is fixed at compile time and can be changed only by editing and recompiling the source code. At very little additional cost in terms of time and space, many programs can be made more flexible by using dynamic memory. The maximum expected amount of data is determined at run time, and storage is then allocated for an array of the appropriate size. Such an array is declared in the program as an uninitialized pointer variable, `p`. Then at run time, either `malloc()` or `calloc()` is called to allocate a block of memory, and the resulting starting address is stored in `p`. (Of course, this must be done before attempting to use `p`.)

Resizeable arrays. It is possible to resize a dynamic array, making it either longer or shorter. If the array is lengthened, it may be reallocated starting at a new memory address. Therefore, when implementing your own growing data structures, you must copy all the data from the old block to the new block. Resizing an array is an appropriate technique for applications in which the amount of data to be processed cannot be predicted until after processing has begun.

Recycling storage. A program that uses dynamic memory is responsible for freeing that memory when no longer needed. Blocks used during only one phase of processing should be recycled by calling `delete` or `delete[]` as soon as that phase is complete. Some memory blocks remain in use until the end of the program. If all is working properly, such blocks will be freed by the system automatically when the program ends, and so the program should not need to free them explicitly. However, relying on some other program to clean up after yours is risky. It is better if every program frees the dynamic storage it allocates. Recycling memory is especially important if a program requests either several very large memory blocks or many smaller ones. Failure to free salvageable blocks can cause program performance to deteriorate. If the virtual address space becomes exhausted by many requests to allocate memory, and none to free it, there will be a fatal run-time error.

Sorting. Insertion sort is a simple sorting algorithm, implemented here using a double loop that moves data within the array. The sorting strategy is to pick up items from the unsorted part of the data set and insert them, in order, into the sorted portion. Insertion sort should not be used for large data sets, because it is very slow compared to other sorts such as quicksort, which is covered later in this book. However, it is considered the best sorting algorithm for small data sets (up to about 25 or 30 items on a modern computer).

16.7.2 Programming Style

Data encapsulation. Object-oriented languages such as C++ permit the programmer to define objects and data structures in a way that encapsulates everything about them. The `Flex array` implements this philosophy – it gathers together the information about an object into a structure and treats it as a single object.

Pointers vs. subscripts. A pointer can be used (instead of a subscript) to process an array. The scanning technique of using cursor and sentinel pointers is easy and very efficient when the array elements must be processed sequentially. When random access to elements is made, using a subscript is more practical with either an array name or a pointer to an array.

Sorting preferences. Always use the appropriate sorting algorithm for a particular situation. Speed usually is the deciding factor. For small data sets, the insertion sort performs well. For larger data sets, a fundamentally different sort is needed, like the quicksort algorithms discussed in Chapters 19 and 20.

Coding idioms. Errors with pointers are hard to track down because the symptoms may be so varied. Any computation or output that happens after the original error could be invalid because the program may be storing information in the wrong addresses and, thereby, destroying the data used by the rest of the program. This kind of error generally requires that the whole job be rerun and is doubly frustrating because the cause of the error can be hard to localize and, therefore, hard to fix. We address this problem by using

coding idioms, presented throughout this chapter, that ensure all references to an array (either through subscripts or through pointers) refer to slots that are actually part of the array. A few of these are

- Initialize pointers to `nullptr`. Using a null pointer should cause the program to terminate quickly and make the problem easier to find.
- When referencing an element, `i`, of an array using a pointer, use the syntax `p[i]` rather than `*(p+i)`.
- Use an off-board sentinel pointer to mark the end of an array for a scanning loop.

16.7.3 Sticky Points and Common Errors

Pointers out of bounds. One danger of pointers in C++ is pointing at something unintended. Common errors are to use a subscript that is negative or too large or to increment a pointer beyond either end of an array and then attempt to use the pointer's referent. You cannot use a declaration to restrict a pointer to point at a legitimate array element. Also, C++ does not "enforce" the boundaries of an array at run time. In general, it does not trap pointer errors. An attempt to use a pointer that is out of bounds (or `NULL`) may cause the program to crash but, on some systems, may not be detected at all; the program will simply walk on adjacent memory values.

Uninitialized sentinel. Another common error is to use a pointer to process an array but forget to initialize the end pointer. The loop almost certainly will not terminate at the end of the array. After processing the actual array elements, the cursor will fall off the end of the array and keep going until, eventually, the program malfunctions or crashes. When you use a sentinel value, be careful to set your loop test correctly, depending on whether you are using an on-board or off-board pointer.

Pointers and arrays. Since an array name, without subscripts, is translated as a pointer to the first slot in the array, some books say that "an array is a pointer." However, this clearly is not true. Since we can use an unsubscripted array name in almost any context where a pointer is expected,[8] we, accurately, could say that an unsubscripted array name becomes a pointer to the beginning of the array. But an array is not a pointer. A pointer requires only a small amount of storage, often 4 bytes.[9] In contrast, an array is a series of storage objects of some given type and can occupy hundreds of bytes. Conversely, a pointer certainly is not an array, it is not limited to use with arrays and cannot be used where an array is needed unless it refers to an array.

A common error of this sort is to attempt to use a string variable for input, but forget that an array must be declared to hold the characters that are read in. When the pointer is dereferenced, the program will usually crash.

Wrong reference level. The most common reason for pointer programs to fail is that the wrong number of ampersands or asterisks is used in an expression. Although most compilers give warning comments about mismatched types, they still finish the translation and generate executable code. Do not ignore warnings about pointer type errors.

Incorrect referencing. It also is common to write syntactically correct code with pointers that do not do what you intend. For example, you may wish to change the value of a pointer's referent and, instead, change the value of the pointer itself. Also, you may forget to insert parentheses in the proper places, such as using `*p+5` rather than `*(p+5)` to access an array element. Finally, you may attempt to use a pointer of one data type to access the memory area in which data of another type are stored. Dereferencing such a pointer will result in a garbage value.

[8] For instance, we cannot do an assignment such as `arrayname = pointer`.

[9] This is true of many modern computers. However, some computers may have pointers of different lengths. Microprocessors in the Intel 80x86 family have two kinds of pointers, local (near pointers) and general (far pointers). The near pointer occupies only 2 bytes of storage. In the near future, computers may have such large memories that they will need more than 4 bytes for a pointer.

Precedence. When dereferencing a pointer using `*`, don't be afraid to use parentheses around the dereferenced value when other operators are involved, thereby making sure that the proper precedence is both understood by you and used by the computer. Watch out for the precedence of `*` in an expression involving pointers, such as `*p++`. Depending on the situation, this expression might have been intended to increment the contents of the address in `p`, but at other times it might have been necessary to increment the contents of `p` and then use the new address. Use parentheses where needed for clarity.

Pointer diagrams. Errors sometimes stem from having an unclear idea about what the ampersand and asterisk operators really mean. The best way to avoid such trouble is to learn to draw diagrams of your pointers, objects, and intended operations, following the models at the beginning of this chapter. Having a clear set of diagrams can help reduce confusion when you begin to write code.

Pointer arithmetic and logic. It is not meaningful to use address arithmetic on a pointer unless the pointer refers to an array. Similarly, it is not meaningful to compare or subtract two pointers unless they point at parts of the same array.

16.7.4 Vocabulary

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

base type	pointer with subscript	scanning loop
referent	pointer assignment	dynamic allocation
`&` (address of)	indirect assignment	constructor
`*` (indirect reference)	pointer arithmetic	destructor
`->` (dereference or select)	pointer comparisons	Flex array
new	off-board sentinel	insertion sort
delete	tail pointer (sentinel)	selection sort
delete[]	head pointer	reference level error
	scanning pointer (cursor)	

16.7.5 Self-Test Exercises

1. Given the array of values that follows, show the positions of the values after each pass of a selection sort that arranges the data in descending order:

21	4	13	17	24	8	15

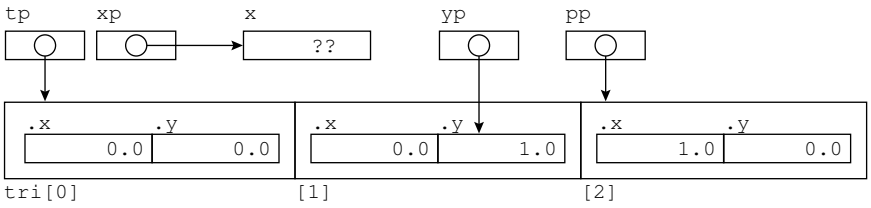
2. Repeat the previous problem on the data above using the insertion sort algorithm for ascending order.
3. Add the correct number of asterisks to the following declarations. Also add the correct number of ampersands in the initializers so that the declaration actually creates the object described by the phrase on the right. In the last item, replace the `???` by the correct argument.

   (a) `char a[12] = "Ohio";` An array of chars
   (b) `char b ;` An array of char pointers
   (c) `char p = b[0];` A pointer to the first slot in an array of char pointers
   (d) `char q = new char[12];` A pointer to a dynamically allocated array of 12 chars
   (e) `char s = new ??? ;` A pointer to a dynamically allocated array of 4 char pointers

4. You are given the declarations and diagram that follow:

```
typedef struct { double x, y; } PointT;
double x;
PointT tri[3];
```

```
double* xp;
double* yp;
PointT* pp;
```



(a) Change the declaration of `tri` to include an initializer for the values shown in the diagram.
(b) Declare another pointer named `tp` and initialize it to point at the beginning of the array that represents the triangle, as shown.
(c) Declare and initialize `triEnd`, an off-board sentinel for the triangle.

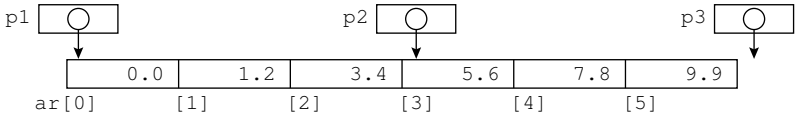
5. Use the diagram in the preceding exercise to write the following statements:

   (a) Write three assignment statements that make the pointers `xp`, `yp`, and `pp` refer to the objects indicated.
   (b) Using only these four pointers (and not the variable names `tri` and `x`), write an assignment statement that copies the `x` coordinate of the last point of the triangle `tri` into the variable `x`.
   (c) Using the array name `tri` and a subscript, store the value 3.3 in the referent of `yp`.
   (d) Using the pointer `tp` and no subscripts, print the `x` coordinate of the last point in the array.

6. Using the representation for triangles diagrammed and described in problem 5, write a function, `triIn()`, with no parameters that will read the coordinates of a triangle from the `cin` stream. Dynamically allocate storage for the triangle within this function and return the completed triangle by returning a pointer to the first point in the array. Use pointers, not subscripts, throughout this function.

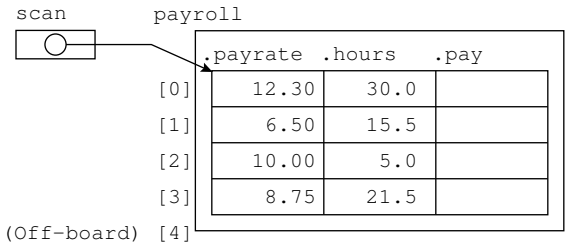
16.7.6 Using Pencil and Paper

1. Show the changes that must be made to adjust the insertion sort program to sort numbers in descending order. Then use the data array first presented in Figure 16.5 to trace the steps of execution of the new insertion sort algorithm, as was done in Figure **??**.
2. Given the data structure in the following diagram, look at each of the statements. If the operation is valid, say what value is stored in `x` or in `p2` by each of these operations. If the operation is not valid, explain why. Assume that all numbers are of type `double` and all pointers are of type `double*`.



(a) `x = *p1;`
(b) `x = p1 + 1;`
(c) `x = *(p2 + 1);`
(d) `x = *p2 + 1;`
(e) `p2 = p1[4];`
(f) `p2++;`
(g) `p2 = p1 + 1;`
(h) `p2 = p3 - p1;`

3. Given the data structure diagrammed in the next problem, write a type declaration and the set of variable declarations to construct (but not initialize) the payroll structure, the pointer `scan`, and pointers (not illustrated) `end` and `p10`, which will be used to process the structure.
4. Given the data structure in the following diagram, write a statement for each item according to the instructions given. Assume that all numbers are type `float`. Declare the pointers appropriately.



   (a) Set `scan` to point to the first structure in the array, as illustrated.
   (b) Set `end` to be an off-board pointer for the array.
   (c) Move `scan` to the next array slot.
   (d) Set pointer `p10` to point at the slot with pay rate $10.00.
   (e) Give the last person a $.50 pay raise.
   (f) Calculate and store the pay for the person who is the referent of `p10`.

16.7.7 Using the Computer

1. Pointer selection.
   Rewrite the selection sort program presented in Chapter 10 to use pointers for array searching and data accessing as in the insertion sort program in this chapter.
2. Sorting a poker hand.
   This program asks you to begin implementing a program that runs a poker game. To start, you will need to define two enumerated types:

   (a) Represent the suits of a deck of cards as an enumeration (clubs, diamonds, hearts, spades). Define two arrays parallel to this enumeration, one for input and one for output. The input array should contain one-letter codes: `{'c', 'd', 'h', 's'}`. The output array should contain the suit names as strings.
   (b) Represent the card values as integers. The numbers on the cards, called *spot values*, are entered and printed using the following one-letter codes: `{'A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K'}`. These should be translated to integers in the range 1 ... 13. Any other card value is an error.
   (c) Represent a card as class with two data members: a suit and a spot value. In the Card class, implement these functions:
      i. `Card::Card()`. Read and validate five cards from the keyboard, one card per line. Each card should consist of a two-letter code such as `3H` or `TS`. Permit the user to enter either lower-case or upper-case letters.
      ii. `void print()`. Display the cards in its full English form, that is, `9 of Hearts` or `King of Spades`, not `9h` or `KS`.
   (d) Represent a poker Hand as array of five cards. In the public part of the class, implement the following functions:
      i. `Hand::Hand()`. Read and validate five cards from the keyboard, one card per line. Each card should consist of a two-letter code such as `3H` or `TS`. Permit the user to enter either lower-case or upper-case letters.
      ii. `void sort()`. Sort the five cards in increasing order by spot value (ignore the suits when sorting). For example, if the hand was originally `TH`, `3S`, `4D`, `3C`, `KS`, then the sorted hand would be `3S`, `3C`, `4D`, `TH`, `KS`. Use insertion sort and pointers.

Value	Description
Royal flush	The suits all match and the spot values are `T`, `J`, `Q`, `K`, `A`
Straight flush	The suits all match and the spot values are consecutive
Four of a kind	The hand has four cards with the same spot value
Full house	The hand has one pair and three of a kind
Flush	All five cards have the same suit
Straight	The spot values of all five cards are consecutive
Three of a kind	The hand has three cards with the same spot value
Two pairs	The hand has two pairs of cards
One pair	The hand has two cards with the same spot value
Bust	Five cards that include none of the above combinations

Figure 16.24. Poker hands, high to low.

iii. `void print()`. Display the five cards in a hand, one card per line.

(e) Write a main program to instantiate a Hand and test these functions.

3. Beginner's poker.
The object of poker is to get a hand with a better (less likely) combination of cards than your opponents. The scoring combinations are listed in Figure 16.24, from the best possible hand to the worst. Define an enumerated type to represent these values and a parallel array for output. Start with a program that can read, sort, and print a poker hand, as described in the previous problem. Add these functions to begin implementing the game itself:

   (a) Write nine private functions, one for each scoring combination. Each function will analyze the five cards in this hand and return `true` if the hand has the particular scoring combination that thefunction is looking for. (Return `false` otherwise.) Some of these functions might call others.

   (b) `int handValue()`. Given a sorted hand, evaluate it using the scoring functions in Figure 16.24. Return the highest value that applies. For example, the hand `TS`, `JS`, `QS`, `KS`, `AS` is a royal flush, a straight flush, a flush, and a straight. Of these possibilities, royal flush has the highest value and should be returned.

   (c) Write a main program that reads two hands, calls the evaluation function twice, prints each hand and its value, then says which hand wins, that is, has a higher value. If two hands have the same value, then no one wins. (This is a slight simplification of the rules.)

4. Average students.
At Unknown University, the Admissions department has discovered that the best applicants usually end up going to the big-name schools and the worst often do poorly in courses, so the school wants to concentrate recruitment efforts and financial aid resources on the middle half of the applicants. You must write a program that takes a file containing a list of applicants and prints an alphabetical list of the middle half of these applicants. (Eliminate the best quarter and the worst quarter, then alphabetize the rest.)

   Define a class Applicant with data members of a name and a total SAT score. This can be done using a simple iterative technique. Following the example in Figures 16.22 and 16.23, define a constructor, and the functions print and bigger and smaller.

   Read the data from a file named `apply.dat` into a Flex array of Applicants. Let N be the number of objects in the Flex array.

   Then begin a loop that eliminates applicants two at a time, until only half are left. Within this loop, let `first` be a pointer to the first array element that is still included, `last` be a pointer to the last one, and `r` (the number remaining) be initialized to $N/2$. Each time around the loop, `first` is incremented, `last` is decremented, and `r` decreases by 2. At each step,

(a) Using `smaller()` find the remaining applicant with the lowest SAT score and swap it with the applicant at position `first`. Then increment `first` . Then find the remaining applicant with the highest SAT score and swap it with the applicant at `last`. Then decrement `last`, subtract 2 from `r`, and repeat.

(b) Quit when `r` $\leq N/2$. Return the values of `first` and `r`.

When only half the applicants are left, use any sort technique to sort them alphabetically using an insertion sort. Write the sorted list to a user-specified output file.

Chapter 17

Vectors and Iterators

This chapter gives an overview of the C++ standard template library, introduces the standard `vector` template class and explains how it works by analogy to the Flex array. Two applications are presented that rely on the dynamic nature of vectors.

17.1 The Standard Template Library–STL

A student of computer science must learn about data structures– these are classes that organize a collection of data for efficient storage and retrieval. With any particular type of data structure, the operations performed on it depend wholly on the structure and not on the data stored in it. STL is a library of pre-programmed data structures written by the best C++ developers, in the form of templates.

A template is an abstract class definition. When a real type is supplied as a parameter, the template code is *instantiated* with that type to create a real class definition that can then be compiled. The result is code that is customized for the given type parameter. For example, STL provides a template for a stack class. Suppose your program defines an Item class. Then you would create a stack of Items, named `s`, like this: `stack<Item> s;`

In the implementation of the Flex class, a typedef for an abstract type `BT` was used to instantiate the Flex for the desired base class. This use of typedef is an old C trick that is still useful, but it is not as powerful as a template implementation.

The STL library. STL was designed with extreme care so that it is complete and portable and as safe as possible within the context of standard C++. Among the design goals were:

- To provide standardized and efficient template implementations of common data structures, and of algorithms that operate on these structures.
- To produce correct and efficient code. The level of efficiency is greater than Java and Python are able to produce.
- To unify array and linked list concepts, terminology, and interface syntax. This supports plug-and-play programming and permits a programmer to design and build much of an application before committing to a particular data structure.

There are three major kinds of components in the STL:

- Containers manage a set of storage objects (vector, list, stack, map, etc). Twelve basic kinds are defined, and each kind has a corresponding allocator that manages storage for it.
- Iterators are pointer-like objects that provide a way to traverse a container.
- Algorithms (sort, swap, make_heap, etc.) use iterators to act on the data in the containers. They are useful in a broad range of applications.

In addition to these components, STL has several kinds of objects that support containers and algorithms including key–value pairs, allocators (to support dynamic allocation and deallocation) and function-objects.

Code	Meaning
vector< *BT* > vc;	Construct an empty vector of default size to hold *BT* objects.
vc.push_back(bto) vc.pop_back() vc.clear()	Insert an object at the end of the vector. Grow if necessary. Remove and discard the most recently inserted object. Remove all of the elements from the vector.
k = (int) vc.size() c = (int) vc.capacity() vc[k] = bto bto = vc[k] bto = vc.front() bto = vc.back()	Return the number of objects stored in vc (type size_type) Return the number of slots currently allocated (type size_type) Store a value in the contents of the kth slot of the vector. Return the contents of the kth slot of the vector. Return the first element in the vector. Return the last element in the vector.

Figure 17.1. Vector operations.

17.1.1 Containers

The C++ standard gives a complete definition of the functional properties and time/space requirements that characterize each container class. The intention is that a programmer will select a class based on the functions it supports and its performance characteristics. Although natural implementations of each container are suggested, the actual implementations are not standardized: any semantics that is operationally equivalent to the model code is permitted[1].

Member operations. Some member functions are defined for all containers. These include: Constructors a destructor, traversal initialization: (begin(), end()), size() (current fill level), capacity() (current allocation size), and empty() (true or false).

Other functions are defined only for a subset of the containers, or for a particular container. For more information, go to `cplusplus.com` and click on Reference, then Containers, then <vector> or the name of another container.

17.2 The vector Class

The growing array is probably the most important data structure in use today. There are simple implementations, such as Flex array, and more powerful, general solutions such as the ArrayList class in Java and the C++ template class *vector*.

Both FlexArray and vector copy the data values into the data structure. This means that the base type must be copyable. Once copied, the vector "owns" the data object and will properly delete it at the end of the program. If the base type of the vector is a pointer type, pointing to a dynamic object, you must write a loop to delete those objects yourself.

It is essential to remember that any iterators in use are invalidated if the data structure grows or if elements are removed from the data structure.

A `vector` works the same way as a Flex array: it tracks its current capacity and the number of items stored in it. When those numbers are equal and more data is inserted, the vector "grows" in the same way that a Flex array grows: by doubling its capacity. The FlexArray is simpler and easier to use for those things it does implement. However, vector is a standard type that presents the same interface as the other STL container classes. Figure 17.1 lists the most important `vector` functions. Many functions are omitted here; for more information, consult a standard reference. In this chart, *BT* stands for the base type of the array, bto stands for an object of that type, and vc stands for the vector object.

The vector class is not as restricted as a Flex array. It implements more functions, including `swap()` and `sort()`. It can be used with any base type, whereas the implementation shown for Flex arrays cannot be used to store objects that contain dynamic parts and have a destructor that manages them.

[1]Big-Oh notation is used to describe performance characteristics. In the following descriptions, an algorithm that is defined as time O(n), is no worse than O(n) but may be better.

Code	Meaning
vector< *BT* >:: iterator p1, p2, pos;	Create three iterators to point at vector elements.

p1 = vc.begin();
p2 = vc.end();
p1++ , ++p2
−−p2 , p2−−
if (pos == vc.end())
	A iterator that points to the first element in the vector.
	An offboard iterator pointing to the first unused slot in vc.
	Move the iterator forward to the next array slot.
	Move the iterator backwards to the previous array slot.
	Test whether an iterator has reached the end of the vector.

pos = find(p1, p2, bto); vc.sort (p1, p2)	Return pointer to first copy of bto in the vector; *p*2 for failure. Sort the elements of the vector from p1 to but not including p2

Figure 17.2. Vector iterators.

Iterators. An iterator is a generalization of a pointer, and is used the way a pointer would be used. You can think of an iterator as a "smart pointer". All STL containers have associated iterator types that "know about" the internal structure of the container and are able to move from the first element to the last element, hitting each element on the way. It is important to remember that any iterators in use are invalidated if the data structure grows or if elements are removed from the data structure. Figure 17.2 lists the essential `vector::iterator` functions.

17.2.1 Using the STL vector Class

In this example program (Figure 17.3) we create a vector of ints and an iterator for it, populate the vector, sort it, search it, remove some items and print it. The STL vector functions used are the default constructor, `push_back(), pop_back(), [ ], size(), sort(), find(), erase(), front(), begin(), and end().`

Notes on Figure 17.3: Vector demo program.

First box: The environment.

- The vector header file is needed to use the vector class.
- The algorithm header file is needed to use the sort and find functions.

Second box: A global print function.

- This is just a demo program, not a proper OO design. There is no class declaration. `main()` instantiates a vector and uses it. So there is nowhere to put a print function except in the global namespace. Because it is not in a class, this function must take the vector as a parameter. It is passed by reference (`&`) to avoid copying the data structure.
- The type qualifier `const` means that the print function will not modify the vector.
- Subscripts are used in this function to access the elements of the vector. Iterators are used for the same purpose in box 4.
- We use `vec.size()` to find our how many ints are stored in the vector, then we use the information to control the printing loop.
- An ordinary `for` loop was used in this example to illustrate calls on `size()` and and to give better quality output. However, it is not the easiest or most modern way to write the print loop. Here is the modern alternative, using a for-each loop:

```
void print(const vector<int>& vec) // print the elements of the vector
{
    for (int k : vec) cout << k << endl;
    cout << "--- done ---\n";
}
```

  Read this code as "for each int (call it k) in vec (the vector of ints), print k".

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void print(const vector<int>& vec) { // print the elements of the vector
    int count = vec.size();
    for (int idx = 0; idx < count; idx++)
        cout << "Element " << idx << " = " << v[idx] << endl;
    cout << "--- done ---\n";
}

int main( void ) {
    vector<int> ages;             // create a vector of int's
                                  // insert some numbers in random order
    ages.push_back(11);    ages.push_back(82);    ages.push_back(24);
    ages.push_back(56);    ages.push_back(6);

    cout << "Before sorting:  " <<endl;
    print(ages);                  // print vector elements
    sort(ages.begin(), ages.end());
    cout << "\nAfter sorting:  " <<endl;
    print(ages);                  // print elements again

    cout << "First element in vector is " << ages.front() << endl;
    int val = 3;                  // search the vector for the number 3
    vector<int>::iterator pos;
    pos = find(ages.begin(), ages.end(), val);
    if (pos == ages.end())
        cout << "\nThe value " << val << " was not found" << endl;

    cout << "\nNow remove last element and element=24 "<< endl;
    ages.pop_back();

    // remove an element from the middle; the hole will be closed up.
    pos = find(ages.begin(), ages.end(), 24);
    if (pos != ages.end()) ages.erase(pos);
    print(ages);                  // print 3 remaining vector elements

    return 0;
}
```

Figure 17.3. Vector demo program.

Third box: Creating and filling the vector.

- To instantiate a vector, we must supply a base type enclosed in angle brackets. This type can be any defined primitive type or class. Initially, this vector is empty.
- We put five elements into the vector (each goes at the end).
- Using `push_back()` is the only proper way to insert more data into a vector. Only `push_back()` will cause the structure to grow.

Fourth box: Before and after sorting.

- `ages.begin()` is an iterator pointing at the first item in the vector. `ages.end()` is an offboard iterator pointing at the vector slot not occupied by data. The functions `begin()` and `end()` are defined on all containers.
- Calling `sort(ages.begin(), ages.end())` sorts the data from the beginning to the end. A smaller portion of the data can be sorted by supplying iterators to slots in the middle.
- The contents of the vector are printed before and after sorting. Here is the output, condensed into two columns. Read the left column first.
- The contents of the vector are printed before and after sorting. Here is the output, condensed into two columns. Read the left column first.

```
Before sorting:          After sorting:
Element 0 = 11           Element 0 = 6
Element 1 = 82           Element 1 = 11
Element 2 = 24           Element 2 = 24
Element 3 = 56           Element 3 = 56
Element 4 = 6            Element 4 = 82
--- done ---             --- done ---
```

Fifth box: Accessing and searching a vector.

- The call on `ages.front()` gets the first element in the vector but does not erase it from the vector.
- We prepare for searching by declaring an iterator named `pos` of the right kind for vector¡int¿.
- Like `sort()`, a call on `find()` takes two iterator parameters to mark the beginning and the end of the part of the vector to be searched. The third parameter is the value to search for.
- The result of `find()` is an iterator pointing at the desired value, if it exists, or a copy of the second parameter if the value is not in the array.
- In this box, the sought-for value is not on the vector, so the call `pos = find(ages.begin(), ages.end(), val)`, sets the iterator `pos` to point at `ages.end()`, the first vector slot not occupied by data. We test for this condition and the error comment is printed.
- The output:

```
First element in vector is 6
The value 3 was not found
```

Sixth box: Removing data from a vector

- We can remove the last vector element by calling `pop_back()`. This function is very efficient: it simply decrements the vector's counter for data items.
- We can also remove any element that an iterator is pointing at. Here, we use `find()` to set an iterator to the slot containing 24, then use `erase()` to remove the 24. This operation is slow because the `remove()` function must shift each element that follows the 24 one slot to the left to fill up the hole.
- A final call on `print()` verifies the actions described here. The output is:

```
Now remove last element and element=24
Element 0 = 6
Element 1 = 11
Element 2 = 56
--- done ---
```

The Pleasant Lakes Club

A group of neighboring families got together, bought two cottages on a lake, and incorporated as the Pleasant Lakes Club. Their bylaws say that there must be no more than 20 families in the club, so that each family can use a cottage one week per summer. Although the club currently has fewer than 20 families, they are willing to buy another cottage if the membership grows.

The software.
The club secretary maintains a file of member information and decided to write a program to help with that task. He has found that the sizes of families and the length of their names vary dramatically, so he decided that his program will use dynamic allocation (vectors and strings) to organize this information. When a new family joins, a Family object is created and pushed onto the membership vector. The family object contains some family data and a vector that stores the name of each person in the family.

Figure 17.4. Problem Specification.

17.3 A 3-D dynamic object.

As a first example of the application of vectors, we implement the software described in Figure 17.4. Our solution creates a vector of objects, where each object contains a vector of strings. We use this data structure to model the membership of a small club. This is a 3-dimensional dynamic data structure: (1) the vector of families can grow as large as needed, (2) the vector of names in each family can grow during input, and (3) each name (a string) can be as long as needed.

This program is a preliminary version of a membership-management tool that intended to maintain a club's membership database file. The use of vectors and strings makes it easy to handle all the variability in the data.

The overall strategy is to read in the current file of members and display them so that the secretary can see who is there and who is not. Then, if he has a new member family, the family information can be typed in, stored in the vector, and ultimately written back out to the file.

Input and output for variable-length objects is significantly trickier that input for a flat database. The main program and the functions in the FamilyT class must work closely together to handle the nature of the data file and the possible kinds of input errors. They share responsibility for detecting end-of-file and for interacting with the user.

The input file and the output file have the same name to make it easy to use the system over and over. After reading the input, the file is closed and renamed as a backup file. Then a new file of the same name is opened for output. By doing this, we avoid the danger of losing the data entirely if the program crashes while the file is open for output.

Notes on Figure 17.5: The Pleasant Lakes Club.

First box: The environment.

- By including `family.hpp` we are including `tools.hpp` and, indirectly, including `<vector>`, `<string>`, `<iostream>`, `<fstream>`, `<iomanip>`, several C header files, and the namespace command. Therefore we do not need to list all of these things at the top of this program.
- File names are defined at the top of the program and used throughout. This is good practice. Names for both the backup file and the output file are provided.

Second and fourth boxes of main program: The input and output files.

- The second box opens an input file in the ordinary way. The third box uses the file, and the fourth box closes it.
- It is always good to close a file as soon as the program is done with it. In this case, however, it is not just good, it is necessary. We want to rename the input file as a backup file, and we want to reuse the name for a different file. It needs to be closed first.
- The `rename()` function is from the C standard I/O library (`<cstdio>`). If a backup file already exists, it will be overwritten.
- After renaming the input tile, we open a new output file with the original file name. If a file already exists by this name in the directory, it will be overwritten.

```
#include "family.hpp"
#define FILE "famFile.txt"
#define BAKFILE "famFile.bak"

int main( void )
{
    char reply;                    // For a work loop.

    ifstream iFams( FILE );
    if( !iFams.is_open()) fatal( "Cannot open " FILE " for input." );

    cout <<"\n Pleasant Lakes Club Membership List \n";

    vector<FamilyT> club;
    for(;;) {
        FamilyT f;                 // Create an empty family.
        f.realize( iFams );        // Read 1 family's data from the file.
        if (iFams.eof()) break;
        f.print( cout );           // If data, display and add to club vector.
        club.push_back( f );
    }

    cout <<"\n------- Done reading club members -------\n";

    iFams.close();
    rename( FILE, BAKFILE );
    ofstream oFams( FILE );
    if( !oFams.is_open()) fatal( "Cannot open " FILE " for output." );

    for(;;) {
        cout <<" Do you want to enter a new family?  (y/n): ";
        cin >>reply;               // Read.
        cin.ignore(1);             // Remove newline from stream.
        if (tolower( reply )!='y') break;

        FamilyT f;
        f.input();                 // Interactive input.
        club.push_back( f );

    }

    // Finally, write it all out again.  --------------------------------

    for (FamilyT f : club) f.serialize( oFams );
    oFams.close();
    cout <<" Data is in file " <<FILE <<'\n' <<" Normal termination." <<endl;
}
```

Figure 17.5. The Pleasant Lakes Club.

```
#include "tools.hpp"

class FamilyT {
private:
    int n = 0;                      // Number of people in this family.
    string fName;                   // The family's last name (father or mother).
    vector<string> fam;             // Dynamic array of strings.

public:
    FamilyT() = default;

    void input();                       // Input a family interactively.
    void print( ostream& out );         // Display a family, formatted nicely.

    void realize( istream& in );        // Read a family from a file.
    void serialize( ostream& out );     // Write a family to a file.
}
```

Figure 17.6. The FamilyT class.

Third box of main program: File input.

- To begin, we create an empty vector of families and, inside the loop a single empty family, `f`, with a default initialization.
- Having `f` allows us to call functions from the FamilyT class. First, we call `realize()`. We use this name for a function when it initalizes a data structure by bringing in data from backup storage (in this case a disk file).
- All the details of reading the data are inside the FamilyT class, which is the expert on how to read the data for a family. Each time it is called, it reads one family from the stream.
- The end of file flag will be set during `serialize()` when we try to read another family that is not there. `Serialize()` does test for eof. However, we also need to know about eof in this function that we can end the loop.
- When eof happens in a stream, the stream eof flag is set and stays set until the stream is closed. We can test for eof anywhere in the program., even in a different function in a different module.
- If there was no `eof()`, then `f` has real data in it, so we call `FamilyT::print()` to display the data on the screen, then push the new family object into the vector.
- During the `push_back()`, the object is copied from the local variable `f` into the vector. The next time around the loop, `f` is recreated and reinitialized. We are never writing new family data on top of a former family.

Fifth box: Interactive input.

- This is written as a query loop. The user is asked each time around the loop whether he wishes to enter another family.
- After reading the single-character answer to the query, a newline character is left unread in the stream buffer. It must be removed; `ignore(1)` does the job.
- Inner box: Interactive input. Again, the FamilyT class hides all the details of reading the input for a new family. When the function returns, `f` has been initialized and can be pushed into the vector.

```
#include "family.hpp"
void FamilyT:: input() {
    string name;                          // For input of family's name.
    cout <<"\n Please enter the surname of the head of the family: ";
    getline( cin, fName );
    cout <<"\n Now enter the names of family members.\n The surname"
         <<" may be omitted if same as family's name.\n"
         <<" Enter RETURN to quit.\n";
    for (;;) {                        // Read & install new names.
        cout <<" > ";
        getline( cin, name );
        if (name.length() == 0) break;
        fam.push_back(name);
    }
    n = fam.size();
}
// ---------------------------------------------------------------------------
void FamilyT:: print( ostream& out ) {
    int n = fam.size();
    cout <<"\n The " <<fName <<" family has " <<n <<" members.\n";
    for (string s : fam) cout <<'\t' <<s <<'\n';
}
// ---------------------------------------------------------------------------
void FamilyT:: realize( istream& in ) {| string name;
    in >> n >> ws;
    if (!in.good()) {
        if (in.eof()) return;
        else fatal( "Family size is corrupted." );
    }
    getline( in, fName);
    for (int k=0; k<n; ++k) {
        getline( in, name );
        if (!in.good()) fatal( "Family name is missing." );
        fam.push_back( name );
    }
}
// ---------------------------------------------------------------------------
void FamilyT:: serialize( ostream& out ) {
    out << fam.size() <<" " <<fName <<endl;
    for (string s : fam) out <<s <<'\n';
}
```

Figure 17.7. The FamilyT functions.

Notes on Figures 17.6 and 17.7: The FamilyT class.

First box: Data members.

- The number of people in the family is needed to be able to read the database back in after writing it to a file.
- This is a prototype implementation – in a more developed implementation there would be data members to store a family's contact information.
- The vector of strings in this prototype would become a vector of PeopleT in a more developed version, so that a variety of personal information could be stored.

Second box: the constructor.

- A constructor without parameters is needed because `main()` FamilyT variables. This constructor performs a default initialization, setting the name to an empty string and the vector to an empty vector.

Third box of Figure 17.6 and first box of Figure 17.7: Interactive input.

- This function reads the data for one family from the keyboard and stores it in the implied parameter (this).
- The calls on `cout <<` show that effort and thought have gone into communicating with the user.
- An input prompt, `>`, tells the user that input is expected. A short prompt like this is all you need. Long, wordy prompts repeated every time around a loop cause visual clutter and are not helpful.
- Inner boxes: We call `getline()` twice to read strings that might have embedded spaces. The output file was planned so that reading these strings would be easy: they are the last thing on each line.
- `getline()` reads and stores characters up to the end of the line. It removes the newline character from the stream and discards it. A newline is not stored in the string.
- If the user types RETURN instead of a name, the length of the string that is read will be 0. This is an easy way to end an input loop.

Third box of Figure 17.6 and second box of Figure 17.7: Interactive output.

- This function formats the data for humans to read. The `serialize()` function formats it for the computer to read on the next run. These formats are significantly different. The screen output for a family has a heading consisting of the family name and number of members.
- A for-each loop is used to print out the contents of the `fam` vector. Read this line as "for each family (call it s) in fam, display s on cout".
- Use the for-each loop in preference to a for loop or a loop with iterators whenever you need to process all of the data stored in a vector.

Fourth box of Figure 17.6 and third box of Figure 17.7: File input.

- This function reads the data for one family from the input stream and stores it in the implied parameter (this).
- You simply cannot use a "while not end of file" loop to do this job because the data structure is variable length in three dimensions.
- The first item to be read for each family is the number of family members. The end-of-file flag will not go on in the stream until an attempt has been made to read this number for a family that does not exist. This code distinguishes between the normal end of file and an error caused by damage to the contents of the input file. Corrupted data will terminate the run. EOF will cause a return to the caller.

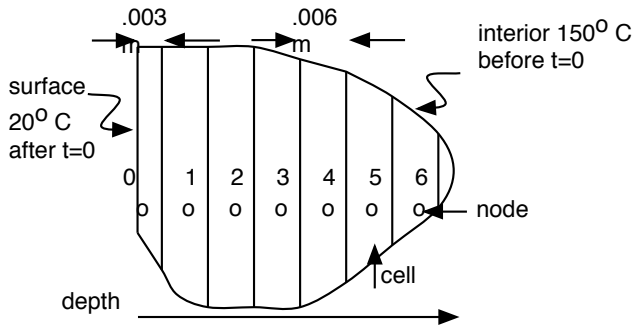


Figure 17.8. Heat conduction in a semi-infinite slab.

- An input prompt, `>`, tells the user that input is expected. A short prompt like this is all you need. Long, wordy prompts repeated every time around a loop cause visual clutter and are not helpful.
- Inner box : The first `getline()` reads the family name. The second one (in the loop) reads an individual name. The stream could be not good after either read, but it is adequate to test just once because the two calls are so close together and nothing will happen to the stream state after the first error.
- An effort was made to provide a meaningful error comment, but a variety of errors could cause a failure here and no error comment will be right for all of them.
- As always, if control gets past the input and error checks, we have good data and it is stored in the vector.

Fourth box of Figure 17.6 and fourth box of Figure 17.7: File output.

- To "serialize" a data structure means to write it to backup storage. This `serialize()` function is very much like the `print()` function, but simpler because the file output format is simpler than the interactive output format.

17.4 Using Vectors: A Simulation

The technique we use in the next example, a simulation, involves two pointers, `old` and `new`, that switch back and forth between pointing at two vectors, first referring to one, then to the other. This lets us represent an indefinite series of arrays, where the array values in `new` at each time step are derived from the ones in `old`. There is no need to allocate a long series of separate arrays or constantly move the data values from one array to another; we just **swap pointers** (or the addresses in them).

17.4.1 Transient Heat Conduction in a Semi-Infinite Slab

Problems that involve changes over time in a property of a solid or fluid often can be solved by analytical techniques when the geometry, boundary conditions, and material properties are simple. This is the preferred method, because a valid result can be determined at any continuous point inside the material at any time. However, when an analytical solution is not possible, numerical techniques can give an approximate solution at discrete points inside the material at specific times.

Transient heat conduction in a solid slab forms a class of problems suitable for **numerical approximation.** A slab, as shown in Figure 17.8, is divided by imaginary boundaries into equal-sized regions called *cells*. For each cell, the temperature is determined at a discrete point called its *node*. An energy equation is used to derive a formula for the temperature at the cell's node, for each cell at each time step, in terms of the temperature at each of the surrounding nodes on the previous time step. This **finite-difference equation** (a form of the heat conduction equation) for each cell can be modeled by a computer program.

For the specific example in Figure 17.8, the slab's initial uniform temperature is 150°C. It has a thermal diffusivity of $6 \cdot 10^{-7}$ m^2/s. Suddenly, at time $t = 0$, it is exposed to a cooling liquid so that the surface is instantly cooled to 20°C, where it remains throughout the process.

We want to determine the temperature at various depths below the surface of the slab as time passes. As mentioned, this process can be represented by a partial differential equation. The numerical approach to solving it assumes that the slab can be split into cells and that each cell has the same uniform temperature throughout as at its node, which is at the midpoint of the cell. The nodes are labeled $0, 1, 2, 3, \ldots$ beginning at the surface and are spaced 0.006 meter apart. The set of finite-difference equations used to compute the temperature T at nodes $1, 2, 3 \ldots$ is

$$T_m^{t+1} = \frac{1}{2}\left(T_{m-1}^t + T_{m+1}^t\right) \qquad \text{for } m = 1, 2, 3, \ldots \quad \text{and} \quad t = 0, 1, 2, 3, \ldots$$

where m denotes the node and t is the number of elapsed time steps, each corresponding to a 30-second interval. That is, $t = 0$ at time $0, t = 1$ after 30 seconds, $t = 2$ after 60 seconds, and so on.

At time 0 in the example, the cooling source at 20°C is applied at the edge of the slab, which initially is at a uniform temperature of 150°C. Therefore, the temperatures of the first four nodes at time 0 become $T_0^0 = 20, T_1^0 = 150, T_2^0 = 150, T_3^0 = 150$. At the next time step (30 seconds later), $t = 1$ and we can compute the nodal temperatures as

$$\begin{aligned}
T_0^1 &= 20 \\
T_1^1 &= \frac{1}{2}\left(T_0^0 + T_2^0\right) = \frac{1}{2}(20 + 150) = 85 \qquad \text{at a depth of 0.006 m} \\
T_2^1 &= \frac{1}{2}\left(T_1^0 + T_3^0\right) = \frac{1}{2}(150 + 150) = 150 \qquad \text{at a depth of 0.012 m} \\
T_3^1 &= \frac{1}{2}\left(T_2^0 + T_4^0\right) = \frac{1}{2}(150 + 150) = 150 \qquad \text{at a depth of 0.018 m}
\end{aligned}$$

As time passes, the cooling effect penetrates deeper into the slab. The next time step corresponds to 60 seconds; at that time, the nodal temperatures are

$$\begin{aligned}
T_0^2 &= 20 \\
T_1^2 &= \frac{1}{2}\left(T_0^1 + T_2^1\right) = \frac{1}{2}(20 + 150) = 85 \\
T_2^2 &= \frac{1}{2}\left(T_1^1 + T_3^1\right) = \frac{1}{2}(85 + 150) = 117.5 \\
T_3^2 &= \frac{1}{2}\left(T_2^1 + T_4^1\right) = \frac{1}{2}(150 + 150) = 150
\end{aligned}$$

After 90 seconds, node 3 will begin to cool. Eventually, if the cooling source remains constant, the cooling effect will reach the end of the slab, and the entire slab will approach a steady-state temperature of 20°C.

17.4.2 Simulating the Cooling Process

Vectors and iterators are used to implement this process. A call chart for this application is shown in Figure 17.9. The program is given in Figures **??** through 17.14.

Notes on Figures ??: A heat flow simulation.

Main is simply the boss. This is what a main program should look like if there are no files involved – all the details of the object model and the process are hidden in the class.

- Include the header file for the primary class.
- Instantiate that class.
- Call its functions to get the work done and the results printed.

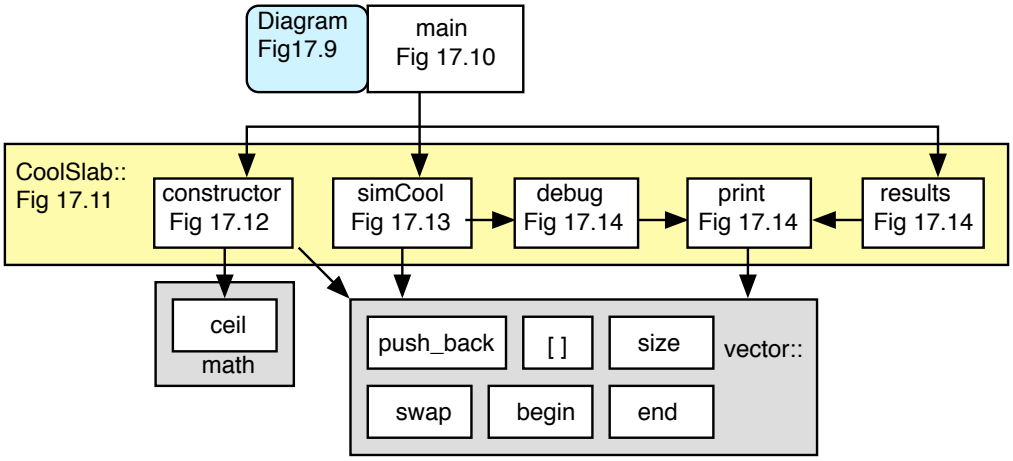


Figure 17.9. A call chart for the heat flow simulation.

Notes on Figures 17.11 and 17.14: The simulation class.

First box of Figure 17.11: simulation parameters.

- The first three variables are used to hold the simulation parameters entered by the user.
- All parts of the simulation depend on the depth at which the temperature is to be monitored. We use this depth to calculate `slot`, the number of the node at the desired depth, which then controls vector growth and looping.
- `p` counts the simulation steps. It is needed for output and to limit the length of the simulation in the event that the user enters unrealistic parameters.

Second box of Figure 17.11: vectors and iterators.

- We need two vectors to model heat in the cells of a slab. The vector named `old` contains the information for time step t, and `next` represents time step $t + 1$. Each value in `next` is calculated from two values in `old`. This process cannot be done in one array – it requires two.
- An iterator is defined for `next`. It will be used for printing.
- `p` counts the simulation steps. It is needed for output and to limit the length of the simulation in the event that the user enters unrealistic parameters.

Third box of Figure 17.11 and Figure 17.12: Constructor and destructor.

- Sometimes a main program handles all user interaction. In this program, however, the input will technical data about the simulation, so it belongs in the Slab constructor, which is the expert on simulations.

This code calls the functions from Figures 17.11, through 17.14.

```
#include <iostream>
#include "slab.hpp"

int main( void ) {
    cout <<"\nSimulation of Heat Conduction in a Slab\n";
    CoolSlab s;
    s.simCool();
    s.results( cout );
}
```

Figure 17.10. A heat flow simulation.

- The Slab constructor interacts with the user to input the parameters for the simulation. It uses those parameters to initialize the data members and the two vectors.
- Because all parts of the simulation depend on the depth at which the temperature is to be monitored, limits are defined for that depth and enforced by an input validation loop. If the depth value is large, the number of iterations needed to reach the goal temperature also will be very high and may exceed the predefined iteration limit, `MAX_STEPS`. If this happens, the simulation will be halted prior to reaching the goal. Therefore, as a practical matter, we limit the observation depth to $0.25m$ and `#define` this constant (`DEPTH_LIMIT`) at the top of the program. Obviously, negative values are rejected.
- Based on a valid depth, the slot number is calculated, according to the specification.
- The `ceil()` function from the C math library takes a double parameter and rounds it to the nearest greater integer.
- Second box: Reasonable bounds for the other inputs depend on the real process being simulated and cannot, in general, be established. So there are no validation loops for the three temperatures.
- Third box: After reading the temperatures, the two vectors can be initialized. To begin, we simply need the first two slots initialized. One new value will be added to the end of each vector on each simulation step. When these later elements are pushed into the vector, they will be initialized to the `initTemp`.
- Here is an example of the output produced by the constructor:

```
Simulation of Heat Conduction in a Slab
Enter depth to monitor (up to 0.25 meters): .055
```

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
using namespace std;

#define MAX_STEPS 1000
#define DEPTH_LIMIT .25

class CoolSlab {
private:
    double initTemp, goalTemp;          // Beginning and ending temperatures.
    double surf;                        // Temperature at surface of slab.
    double depth;                       // Depth at which to monitor temperature.
    int slot;                           // Array subscript at given probe depth.
    int p;                              // The actual number of simulation steps.

    vector<double> old;                 // The initial conditions.
    vector<double> next;                // The next step of the simulation.
    vector<double>::iterator nIt;

public:
    CoolSlab();
    ~CoolSlab() = default;

    void simCool();

    void results( ostream& out );
    void debug( ostream& out );
    void print( ostream& out );
};
```

Figure 17.11. The Slab class.

```
CoolSlab:: CoolSlab() {
    do {
        cout <<"Enter depth to monitor (up to " <<DEPTH_LIMIT <<" meters):  ";
        cin >> depth;
    } while (depth<0 || depth > DEPTH_LIMIT);
    slot = ceil( (depth-.003) / .006 );

    cout <<"Enter initial temperature of the slab:  ";
    cin >>initTemp;
    cout <<"Enter target temperature for depth " <<depth <<":  ";
    cin >>goalTemp;
    cout <<"Enter temperature of cold outer surface:  ";
    cin >>surf;

    // Initialize first two slots of both vectors to the initial conditions.
    old.push_back( surf );
    next.push_back( surf );
    old.push_back( initTemp );
    next.push_back( initTemp );
}
```

Figure 17.12. The Slab constructor and destructor.

```
Enter initial temperature of the slab: 100
Enter target temperature for depth 0.055: 85
Enter temperature of cold outer surface: 0
```

- The destructor: Although we are using dynamic allocation, it is encapsulated within the `vector` class, which takes all responsibility for allocating and freeing the memory. The client class (Slab) does not need to free anything. We emphasize this by explicitly defining a default do-nothing destructor.

Fourth box of Figure 17.11 and Figure 17.13: the simulation. The `simCool()` function will simulate the heat flow process at successive time steps as the slab cools. It will stop when the selected node reaches a specified temperature or when the number of steps reaches the limit (`MAX_STEPS`). On each iteration, the new temperature at each node is computed, based on the temperatures of its neighboring nodes during the preceding time step.

- First box: At each time step during cooling, the leading edge of coolness progresses on slot to the right. Therefore, to begin each iteration, we extend the data in the vectors by one slot, initialized to the initial temperature of the slab. Even though we add an element to the vector every time around the loop, the vector only lengthens itself occasionally, when its available storage is full. This growth process is invisible to the client program.
- Second box: Walk down the vectors using `k` to subscript both of them. Calculate the next value by averaging the two old values to its left and right. This simple formula is a good model of what actually happens during cooling!
- During development, it was necessary to see what was happening in the arrays. So we defined a debug function to make that easy. Using a function removes the details of debugging from the flow of the main logic.
- Third box: The end of the simulation. When the temperature at subscript `slot` is $\leq$ the goal temperature, the simulation is finished. The part of the test that follows the `&&` tests this and breaks out of the simulation loop when it happens.

  The first part of the test in this `if` statement acts as a *guard* for the second part, which is the termination condition. However, in the early stages of the simulation, `next[slot]` may not exist yet, or it may exist but

This function is called by the main program in Figure **??** to perform the steps of the simulation.

```
// ---------------------------------------------------------------------------
void CoolSlab:: simCool() {
    for (p=1; p<=MAX_STEPS; ++p) {
        old.push_back( initTemp );     // Lengthen initialized parts by 1 slot.
        next.push_back( initTemp );    // to prepare for next simulation step.

        //Calculate the next set of conditions based on the old ones.
        for (int k = 1; k <= p; ++k) {
            next[k] = (old[k-1] + old[k+1]) / 2.0;
        }

        debug(cout);
        if (slot < next.size() && next[slot] <= goalTemp )  break;

        next.swap( old );
    }
}
```

Figure 17.13. `simCool()`: **doing the simulation.**

not have any valid data pushed into it. We must avoid testing vector slots that do not exist or do not hold valid data! Therefore, we compare `slot` to the `next[size]` before trying to access the data at `next[slot]`. If `slot > next[size]`, the if statement fails and we stay in the simulation loop[2].

- Fourth box: The swap. At the end of each time step, we are done forever with the values stored in `old`, and `next` will become the basis for computing the next iteration. We also need a vector to hold the temperatures we calculate on the next iteration. The solution is called `swing buffering`: swap the old and the next so that the current next becomes the new old and the current old vector is reused for more calculations.

  The vector class provides a function `swap()` that swaps the values of two vectors and does it efficiently, by swapping pointers, not copying all the data.
- The iterations continue until the goal temperature is reached or the maximum number of steps is exceeded. If the goal is attained, the program breaks out of the loop and returns to `main()`, which prints the final temperature values.

Notes on Figures 17.14: Output functions for the simulation. We want two kinds of output: (1) a brief format, appropriate for debugging, that will print the contents of a vector at one time step and (2) a full version giving the results of the simulation including the final temperatures reached. The common part of the two formats is printing the vector at the end of one time step. Thus, we define a `print()` function to do the common part, and two other output functions to give the two views we need.

- The `debug()` function. For debugging, we want to see a sequence of iterations, each numbered consecutively. So the `debug()` function prints the iteration number and a visual divider to make the output more readable. Here is a sample output; you can see the size of the vector increasing.

```
1. ------------------------------
[ 0]=  0.000  [ 1]= 50.000  [ 2]=100.000

2. ------------------------------
[ 0]=  0.000  [ 1]= 50.000  [ 2]= 75.000  [ 3]=100.000

3. ------------------------------
[ 0]=  0.000  [ 1]= 37.500  [ 2]= 75.000  [ 3]= 87.500  [ 4]=100.000
```

[2]Remember that logical operators are evaluated left to right using lazy evaluation.

The `results()` function is called from `main()` in Figure **??**; `debug()` is called from `simCool()` in Figure 17.13. Both of these functions delegate the common parts of the job to `print()`.

```
    // ------------------------------------------------------------------------
    void CoolSlab:: debug( ostream& out ){
        cout <<\n' <<setw(3) <<p <<". ------------------------------\n";
        print( cout );
        cout <<endl;
    }
    // ------------------------------------------------------------------------
    void CoolSlab:: results( ostream& out ){
        cout << "\nTemperature of "<<next[slot] <<" reached at node "<<slot
             <<"\n\tin "<<p <<" seconds (= " <<fixed <<setprecision(2)
             <<p/60.0 <<" minutes or " <<p/3600.0 <<" hours).\n"
             <<"\nFinal nodal temperatures after " <<p <<" steps:\n";
    print( cout );
    cout <<"\n\n";
}
// ------------------------------------------------------------------------
void CoolSlab:: print( ostream& out ){
    int k = 0;
    out << fixed << setprecision(3);
    for (double d : next) {
        out <<" [" <<setw(2) <<k <<"]=" <<setw(7) << d;
        if (++k % 5 == 0) out <<"\n";         // Newline after every five items.  }
    }
}
```

Figure 17.14. Print(), results(), and debug().

- The `results()` function. At the end of execution we want to see summary information and the final results of the simulation. We print the summary information and call `print()` to print the rest. This sample output corresponds to the parameters given earlier:

```
Temperature of 84.614 reached at node 9
in 39 seconds (= 0.65 minutes or 0.01 hours).

Final nodal temperatures after 39 steps:
  [ 0]=  0.000  [ 1]= 12.537  [ 2]= 25.074  [ 3]= 36.417  [ 4]= 47.760
  [ 5]= 57.041  [ 6]= 66.322  [ 7]= 73.181  [ 8]= 80.041  [ 9]= 84.614
  [10]= 89.187  [11]= 91.931  [12]= 94.675  [13]= 96.152  [14]= 97.630
  [15]= 98.341  [16]= 99.052  [17]= 99.357  [18]= 99.662  [19]= 99.778
  [20]= 99.893  [21]= 99.932  [22]= 99.971  [23]= 99.982  [24]= 99.993
  [25]= 99.996  [26]= 99.999  [27]= 99.999  [28]=100.000  [29]=100.000
  [30]=100.000  [31]=100.000  [32]=100.000  [33]=100.000  [34]=100.000
  [35]=100.000  [36]=100.000  [37]=100.000  [38]=100.000  [39]=100.000
  [40]=100.000
```

  Note that slots 28 – 40 all print as 100.000 degrees. This is an artifact of the print formatting (three places of precision). The number in slot 40 is actually 100.000. The numbers in slots 28 – 39 are all between 99.999 and 100, but appear as 100.000 when they are rounded to three decimal places.

- The `print()` function, outer box. Formatted output, aligned in columns, is important for readability. We

are printing double values and we would like them rounded to three decimal places. Sso we write `<<fixed <<setprecision(3)`. These manipulators stay set until explicitly changed, so we write them once, outside the loop.

The loop is a for-each loop that defines `d` as the name of the current element. We write `<< d`. This code is equivalent to writing an iterator expression:
`for (nIt = next.begin(); nIt != next.end(); ++nIt)`

The width of an ouput field must be set separately for every field written out. Therefore, calls on `setw()` are written inside the loop, not before it.

- The `print()` function, inner box. If we printed the temperatures one per line, it would consume too many lines. So we use modular arithmetic to print five values per line in formatted fields. If the array subscript mod 5 equals 4, we know that we have printed five values on this line, so we print a newline character and enough spaces to indent the beginning of the next line.

17.5 What You Should Remember

17.5.1 Major Concepts

Arrays and vectors of strings. An earlier chapter introduced the ragged array of strings, which was initialized by string literals in that chapter. The data structure is even more versatile when implemented with C++ strings, which are dynamically allocated, because each string can contain data of any length.

17.5.2 Programming Style

Use the proper arguments. If you are writing a function to process a row of a matrix, pass as the argument a row of the matrix. Do not send the entire matrix as the parameter along with a row index to be used by the function in accessing the data. Write your functions to pass the appropriate amount of data, no more.

Use type cast. The `void*` pointer returned by `malloc()` and `calloc()` normally is stored in a pointer variable. While explicitly performing a type cast is not necessary, doing so can be a helpful form of documentation. Prior to the ANSI C standard, the explicit cast was necessary, and many programmers continue to use it out of habit.

Use `free()` ***properly.*** It is a good practice to recycle memory at the first possible time. Sometimes it is possible to reuse a memory block for different purposes before giving it back. This can improve program performance because the memory manager need not be involved as often.

Check for allocation failure. It is proper to check the pointer value returned by the allocation functions to make sure it is not `NULL`. If no memory is available, program termination is a logical course of action.

Use `realloc()` ***properly.*** Do not use `realloc()` too often, because it can reduce the performance of a program if data are copied frequently. Therefore, choose a size that is appropriately large but not too large. A common rule of thumb is to double the current allocation size. A final call to `realloc()` can be made to return excess capacity to the system when the dynamic data structure is complete and all data have been entered.

Use the proper data structure. It is important to choose which form of 2D structure you will use. Should it be defined at compile time or dynamically? Should it be a 2D matrix, an array of arrays, or an array of pointers to arrays? If the size is not known at compile time, dynamic memory is chosen. If the processing of rows is significant, one of the array structures is better suited. It is best to use a 2D matrix when the size can be defined at compile time and all the elements are treated equally.

Use a setup function. Use a setup function to organize data initialization and memory allocation statements. This improves code legibility and localizes much of the user interaction in one function.

Use one buffer for string input. It is a common practice to have one large buffer for string input, since the lengths of such input vary greatly. Each data item is read, then measured (using `strlen()`). An appropriate amount of memory is allocated dynamically for the input, then the input is copied into the new memory block and attached to some data structure. This frees the single long buffer for reuse.

17.5.3 Sticky Points and Common Errors

Dynamic memory allocation is a powerful technique but is prone to a variety of errors that cause programs to crash. The first four errors, below, all relate to overuse or underuse or misuse of the `free()` function. In avoiding one of these errors, it is important not to fall into another!

Misuse of `free()`**.** The two most common mistakes involving the use of `free()` are attempting to recycle a storage area that was not dynamically allocated and attempting to free a memory block that already has been recycled. E ventually, each of these is likely to result in an attempt to free storage that is part of some active object. The visible result may be garbage output or a sudden program termination. The cause may be very difficult to track down because it is not caused by the most recently executed part of the program. Further, the time and manner of crash will probably be different each time the program is run with different data.

Memory leaks. A memory leak occurs when the last pointer to a dynamic memory area is lost; the area was allocated but not freed, and remains a drain on the memory management system until the program terminates. If this happens repeatedly, and if the program runs for hours or days without terminating, system performance will be degraded. Thus, it is important to learn to free memory when it is no longer needed.

Using a dangling pointer. After a memory block has been freed, it never should be accessed again. When space is deallocated, it is logically "dead" but physically still there. If more than one variable is set to point at the area, a common error is to continue using the memory block. Once reassigned to a different portion of the program, the competing use eventually will corrupt the data.

Using `realloc()`**.** The dangling pointer problem also arises with regard to `realloc()`. If a new memory block is assigned, pointers into the old memory block become obsolete. Care must be taken to save the new memory address and use it to reset other pointers to elements within the data block.

Uninitialized pointers. It is easy to forget that every pointer needs a referent before it can be used. A pointer with no referent is like a pocket with a hole in the bottom; it looks normal from the outside but is not functional. A common error is to declare a pointer but forget to store in it the address of either a variable or a dynamically allocated memory block. This frequently results in a program crashing.

Remembering array sizes. Whenever an array is used, the actual number of data items in it must be remembered. Functions that process arrays must have two parameters, the array and the count. The count commonly is forgotten. Logically, these two items always should be grouped. In the next chapter, we use a structure that contains these two members so that we can pass the structure to functions as a single entity.

Null character. When allocating a memory block for an input string, remember to include space for the null character at the end.

17.5.4 New and Revisited Vocabulary

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

vector<Type>	front()	vector<Type>::iterator
push_back()	back()	++ on an iterator
size()	begin()	* on an iterator
capacity()	end()	rename() for a file
subscript	find()	cin.ignore(1)
swap	sort()	cin >>ws

17.6 Exercises

17.6.1 Self-Test Exercises

1. A basic step in many sorting programs is swapping two items in the same array. Write a function that has three parameters, an array of strings and two integer indexes. Within the function, compare the strings at the two specified positions and, if the first comes after the second in alphabetical order, swap the string pointers.

2. What standard header files must be included to use the following C++ features:

   (a) A vector
   (b) A string
   (c) sort() on a vector
   (d) ignore(1)

17.6.2 Using Pencil and Paper

1. What standard header files must be included to use the following C/C++ features:

   (a) new and delete
   (b) The `ceil()` function
   (c) A C++ string

2. In the heat flow program (Figure **??**), we begin the simulation with an empty vector with size()= 0. Assume the initial capacity()= 16. Then we added values to the vector and it doubled every time the capacity was used up. Assume that we were monitoring node 9 and the simulation ran for 100 steps.

   (a) How many times did we double the array?
   (b) What was the final capacity of the array?
   (c) Altogether, how many `double` values were copied during the growing process?

3. Given these declarations, explain what (if anything) is wrong with the following allocation and deallocation commands:

```
double* p, *q;
int arr[5];
```

   (a) `p = new(double);`
   (b) `q = double[10];`
   (c) `delete q;`
   (d) `delete[] arr;`
   (e) `q = new double[10];`
   (f) `p = &q[2]; delete[] p;`

17.6.3 Using the Computer

1. Generating test data. Write a program that uses `srand()` and `rand()` to calculate a series of 1000 random floating point numbers. Write these to a file named "randfloats.in".
2. Sorting a file of numbers. Modify the program from Figure 16.19 in three ways:
   - Change it from interactive input and output to file input and output.
   - Sort a file of numbers instead of a set of objects.
   - Delete the two lines that prompt for and read the number of items to be sorted. Use a vectpr so that all of the numbers in the file can be stored in it.

   Debug and test your program on a short file that contains ten numbers. Be sure that all ten inputs occur in the output file, in order. Then test your program on the long data file generated in the previous exercise.
3. Simulation of heat conduction.

   Given the semi-infinite slab in Figure 17.8, determine the temperatures at nodes 1 through 30 inside the slab after time periods of 5, 10, 15, 20, 25, and 30 minutes have passed. Start with the functions in Figures **??** through 17.14 and modify them to print the results every 5 minutes and terminate after half an hour. Note that 30 minutes corresponds to $p = 60$.

4. A dictionary.

   Make a dictionary data structure by reading in the contents of a user-specified text file one word at a time. Use C++ strings and store them in a vector. Sort these strings by using vector::sort. Print out the entries in your dictionary in alphabetical order. Do not display a word more than once. Instead, display a count of how many times each word appeared in the file.

5. A better dictionary.

   Modify the program from the previous exercise. Do not display a word more than once. Instead, display a count of how many times each word appeared in the file.

6. String math.

   Write a program to read any two integers as character strings (with no limit on the number of decimal digits). Hint: Read a number into a string that will expand when necessary. Add the two numbers digit by digit, using a third string to store the digits of the result. Print the input numbers and their sum. Hint: To convert an ASCII digit to a form that can be meaningfully added, subtract `'0'`. To convert a number $0 \ldots 9$ to an ASCII digit, add `'0'`.

7. Easier said than done.

   Read in the contents of a file of real numbers for which the file length is not known ahead of time and could be large. Write the numbers to a new file in reverse order. Abort the program if the file is so long that the data cannot be held in the computer's memory.

8. Sparse matrix.

   Many applications involve using a matrix of numbers in which most entries are 0. We say that such a matrix is *sparse*. Assume that we are writing a program that uses a sparse 100 by 100 matrix of `double`s that is only 1% full. Thus, instead of 10,000 entries, there are only approximately 100 nonzero entries. However, the actual fullness varies, and there might be more than 100 nonzero entries. One representation of a sparse matrix is to store each nonzero entry as a structure of three members: the row subscript, the column subscript, and the value itself; that is, the value `20.15` in `matrix[15][71]` would be represented as the triple `{ 15, 71, 20.15 }`. ,

   (a) Define a class named `triple` to implement the structure for one element of the sparse matrix.
   (b) Declare a class Matrix with a data member that is a vector. Use the vector to store all the triples in the matrix.
   (c) Write a function, `getMat()` in the Matrix clasa. Write a sentinel loop that will read data sets (a pair of subscripts and a matrix value) from the keyboard until a negative subscript is entered. Create Triple objects and push them into the vector.
   (d) Write a function, `showMat()`, that prints the matrix as a table containing three neat columns (two subscripts and a value).
   (e) Write a main program that tests these functions. Testing programs on huge amounts of data is impractical. During development, use only a few Triples. Call `getMat()` and `showMat()` to read and print the values of the matrix. You will need more than one data set too test this program.

9. Partially sorting the matrix.

   Start with the matrix program in the previous problem. Add to it a function, `sortMat()`, that sorts the array elements in increasing order by the first subscript. If two elements have the same first subscript, sort them in increasing order by their second subscripts. If both subscripts are equal, display an error comment. To perform the sort, adapt the sorting program in Chapter 16, printing the matrix before and after sorting.

Chapter 18

Array Data Structures

Two-dimensional arrays commonly are used in applied physics, engineering, and mathematics to hold numerical data and represent two-dimensional (2D) physical objects. In this chapter, we explore several different array data structures, including the matrix, arrays of arrays, arrays of pointers to arrays, and arrays of strings. We explore applications of these compound arrays, the type definitions used to facilitate their construction, and two-dimensional array processing. We consider multi-dimensional arrays briefly.

18.1 Concepts

18.1.1 Declarations and Memory Layout

Figure 18.1 shows a two-dimensional array, sometimes called a **matrix**, used to implement a 4-by-4 multiplication table. We declare such an array by writing an identifier followed by two integers in square brackets. The first (leftmost) number is the row dimension; the second is the column dimension. Visually, **rows** are horizontal cross sections of the matrix and **columns** are vertical cross sections.

We initialize a two-dimensional array with a set of values enclosed in nested curly brackets; each pair of inner brackets encloses the values for one *row*. The result can be viewed, conceptually, as a rectangular matrix, but physically, it is laid out in memory as a flat data structure, sequentially by row. Figure 18.2 shows two views of a 3-by-4 array of characters: The two-dimensional conceptual view and the linear physical view. Technically, we say it is stored in **row-major order**. All the slots in a given row will be adjacent in memory; the slots in a given column will be separated. This can have practical importance when dealing with large matrices: Row operations always will be efficient; column operations may not be as efficient because adjacent elements of the same column might be stored in different memory segments.

We refer to a single slot of a matrix using **double subscripts**: the row subscript first, followed by the column subscript. Each subscript must have its own set of square brackets. For example, to refer to the first and last slots of the matrix in Figure 18.1, we would write `mult_table[0][0]` and `mult_table[3][3]`.[1]

[1]Experienced programmers who are new to C must take care. They might be tempted to write `multTable[j,k]`, which is correct in FORTRAN. This means something obscure in C, and it will compile without errors. However, it does not mean the same thing as `multTable[j][k]`. (The comma will be interpreted as a comma operator, which is beyond the scope of this text.)

```
short multTable[4][4] = { {1, 2,  3,  4 },
                          {2, 4,  6,  8 },
                          {3, 6,  9, 12},
                          {4, 8, 12,16}
                        } ;
```

multTable

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	2	4	6	8
[2]	3	6	9	12
[3]	4	8	12	16

Figure 18.1. A two-dimensional array and initializer.

The conceptual view of a 3-by-4 array of characters is shown on the left; the subscripts of each cell are written in the corner of the cell. Note that subscripts are used to refer to individual cells but are not actually stored in the cell. The actual storage layout of this array is shown on the right with the subscripts under each cell.

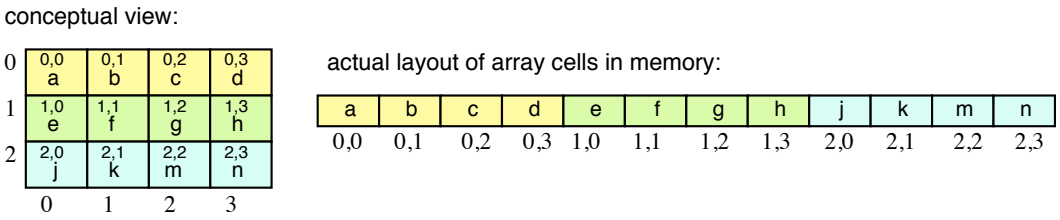


Figure 18.2. Layout of an array in memory.

18.1.2 Using `typedef` for Two-Dimensional Arrays

A programmer may create and use arrays, strings, and multidimensional arrays without defining any new type names. We never actually *need* to use a `typedef`, because it just creates an abbreviation or synonym for a type description. However, `typedef` often should be used with array types in place of the basic syntax, because `typedef` helps simplify the code and clarify thinking. More important, it often enables a programmer to work at a higher conceptual level and have less involvement with the actual implementation of a data structure. There are two conceptually different kinds of two-dimensional arrays, and the appropriate type definitions are quite different for the two varieties, although they are initialized and stored identically.

18.1.3 Ragged Arrays

Another 2-D array data structure is the ragged array: an array (dynamic or declared) that points to C-strings (dynamic or constant). A ragged array is a natural data structure for use with a menu display. It is also the data structure used by C and C++ systems to pass command-line arguments to a main function. (This application will be covered in Chapter 20.)

In this section we introduce `menu()`, a function that is part of the tools library. It uses a ragged array to represent the menu selections. This array is diagrammed as part of the next demo program, Figure 18.5.

Notes on Figure 18.4: The menu function from tools.

First box: Declarations in tools.hpp.

- C++ makes it easy to use C++ strings, but there are often incompatibilities with non-constant C strings. When working with type `char*`, we often find ourselves adding `const` to the type. So we introduced a type name, `ccstring`, for constant-c-string, to make it more convenient to use the simple C-strings for simple applications.
- The menu function shown below is one of two in the tools library. The other is menus where the selection code is a character, not a digit.
- Parameters to `menu()` are a title, the number of possible choices, and a ragged array of item descriptions to display.

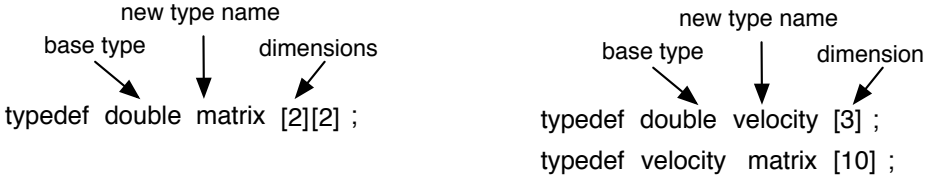


Figure 18.3. Using `typedef` for 2D arrays.

These declarations are in tools.hpp.

```
typedef const char* ccstring;
int menu( ccstring title, int n, ccstring menu[] );
```

This code is in tools.cpp.

```
// Display a menu then read and validate a numeric menu choice.  Handle errors.
int
menu( ccstring title, int n, ccstring menu[] ) {
    int choice;
    for(;;) {
        cout <<'\n' <<title <<'\n';
        for( int k=0; k<n; ++k ) cout <<"\t " <<k <<". " << menu[k] <<endl;
        cout <<" Enter number of desired item:  ";

        cin >> choice;
        if (!cin.good()) {
            cin.clear();    // Reset stream state to good.
            cin.ignore(1);  // Clean garbage out of input buffer.
            choice = -1;    // Set invalid choice to prevent loop exit.
        }

        if ( choice >= 0 && choice < n) break;
        cout << " Illegal choice or input error; try again.  \n";
    }
    return choice;
}
```

Figure 18.4. A menu function.

Second box: Displaying the menu.

- The menu is an array with n menu selections. An ordinary for loop is used to display the selections from slot 0 to $< n$, so the menu will appear with selections numbered from 0 to $n - 1$. The program using this function must expect to get a 0-based subscript as the return value.
- The prompt for the menu choice makes clear that a numeric answer is expected.

Third and fourth boxes: Reading a valid choice.

- Reading the choice is easy. Deciding whether it is valid is not.
- The user could input a number that is too small or too large, or he could input a non-digit. We need to test for all three kinds of errors and produce an error comment if there is any kind of problem.
- If a letter is entered, it will cause the `>>` operation to fail. The fail flag in the stream will be set. At that point, nothing more can be done with the stream until the flag is cleared. Thus, to handle errors properly, the first thing to do is check the stream state and fix it if necessary.
- To prevent an infinite loop of the same error again and again, it is necessary to clear the offending character out of the stream. `ignore(1)` does this.
- Finally, to be sure that the error comment is printed, we set the choice to an invalid number.
- When control reaches the fourth box, the variable `choice` has a number in it and we can test whether that number is in the right range. If so, we break out of the validation loop. If not, the error comment is printed.

18.1.4 A Matrix

In one kind of 2D array, the data are a homogeneous, two-dimensional collection. Columns and rows have equal importance, and each data element has as close a relationship to its column neighbors as to its row neighbors. Programs that process this data structure typically have no functions to process a single row or column. Instead, they might process single elements, groups of contiguous elements, or the entire matrix. For example, consider an image-processing program in which each element of a matrix represents one pixel in a digital image. A **pixel**, which is short for "picture element," is one dot in a rectangular grid of dots that, taken together, form a picture. That pixel has an equally strong relationship to its vertical and horizontal neighbors. Functions operate on entire images or on a rectangular subset of the elements, called a **processing window**. Rows and columns are equally important.

The general form of the `typedef` declaration for this kind of data structure is given on the left in Figure 18.3; note that the dimensions, not the type name, are written last. An example of its use for image processing is given in Section 18.3.

Using a matrix. The next program example illustrates the use of a 2D array for a practical purpose. Many road atlases have a table of travel times from each major city to other major cities, such as that in Figure 18.5. To use such a table, you find the row that represents the starting city and the column that represents the destination. The number in that row and that column is the time that it should take to drive from the first city to the second. We implement a miniature version of this matrix and use it to calculate the total driving time for a two-day trip, where you start in city 1, stay overnight in city 2, and end up in city 3. The program is in Figure 18.5.

Notes on Figure 18.5. Travel time for a two-day trip.

At the top: Data diagrams.

- The names of the cities covered by the table are defined as a ragged array in the second box.
- The table of travel times is a square matrix with one row and one column for each city. The value in each slot will be the number of minutes needed to drive from the row-index city to the column-index city.
- The integers `row` and `col` will be used in nested `for` loops to read the travel times from a file.

First box: the environment.

- This program uses the `menu()` function from the tools library.
- The number of towns is a constant used throughout the code. Such constants should always be defined at the top, not buried in the code.
- The file "minutes.in" contains the data shown in the diagram on the right.

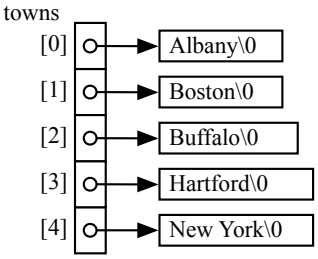
Second box: the matrix.

- The names of the cities covered by the table are defined as a ragged array (diagram at upper left) that will be used as a menu. The menu will be displayed three times to permit the user to select the source city (`city1`), layover city (`city2`), and destination city (`city3`). These cities will be used as subscripts for the travel-time matrix.
- Here we declare and allocate the matrix in the diagram at the upper right. We initialize it in the third box.

Third box: reading the input file.

- The nested `for` loops used here are a typical control structure for processing a matrix.
- Note that we use meaningful names (instead of i and j) for the row and column indices. This makes the code substantially easier to understand.
- We keep this example simple by omitting normal error checking. We assume that the data in the file are not damaged and that the file contains the correct number of data values. In a realistic application, error detection would be necessary.

These are the cities and the matrix of city-to-city travel times used in the travel-time program.



timetable

		0	1	2	3	4
Albany	0	0	194	330	145	193
Boston	1	194	0	524	115	265
Buffalo	2	330	524	0	475	506
Hartford	3	145	115	475	0	150
New York	4	193	265	506	150	0
		0 Alb	1 Bos	2 Buf	3 Hfd	4 NY

```
#include "tools.hpp"
#define NTOWNS 5
#define INFILE "minutes.in"

int main( void )
{
    ccstring towns[NTOWNS] = { "Albany","Boston","Buffalo","Hartford","New York" };
    int timetable[NTOWNS] [NTOWNS];

    int city1, city2, city3; // Cities along route of trip.
    int time;
    cout <<"\n Travel Time \n";

    ifstream minutes( INFILE ); // Data for travel-time matrix
    if (!minutes.is_open()) fatal( " Cannot open " INFILE " for reading.");

    for (int row = 0; row < NTOWNS; ++row)        // Read travel-time matrix.
        for (int col = 0; col < NTOWNS; ++col) {
            minutes >>timetable[row][col];
        }

    city1 = menu( " Where will your trip start?", NTOWNS, towns );
    city2 = menu( " Where will you stay overnight?", NTOWNS, towns );
    city3 = menu( " What is your destination?", NTOWNS, towns );

    time = timetable[city1][city2] + timetable[city2][city3];

    cout <<"\n Travel time from "<<towns[city1] <<" to "<<towns[city2]
         <<" to "<<towns[city3] <<"\n\t will be "<<time <<" minutes.\n\n";
    return 0;
}
```

Figure 18.5. Travel time for a two-day trip.

Fourthbox: choosing three cities.

- These three calls on `menu()` permit the user to select three cities from the `towns` list. The menu function lets the use select a prompt, the number of menu items, and an array of strings that describe the choices.
- The return value from `menu()` is the subscript for the chosen city in the `towns` array. The city numbers will be used to access both the list of cities and the matrix of travel times.

Fifth box: calculating and printing the travel time.

- To access a time from the table we give two subscripts; the number of the source city (the row) and the number of the destination city (the column).
- The total time is the sum of the times on each of the two days. Sample output follows. Normal operation is shown in the left column, error handling on the right:

```
Travel Time

Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 0

Where will you stay overnight?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 1

What is your destination?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 4

Travel time from Albany to Boston
to New York will be 459 minutes.
```

```
Travel Time

Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 6
Illegal choice or input error; try again.

Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: -1
Illegal choice or input error; try again.

Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: w
Illegal choice or input error; try again.
```

18.1.5 An Array of Arrays

In the other kind of 2D array, an **array of arrays**, the data are a collection of rows, where each row has an independent meaning. The data elements in each row relate to each other but not to the corresponding elements of nearby rows. Programs that process this data structure typically have functions that process a single row. The general form of the `typedef` declaration for this data structure is given on the right in Figure 18.3.

For example, consider a program that makes weather predictions. One of its data structures might be an array of winds measured by weather stations in a series of locations. Each wind is represented by an array of three `double` values, which give the magnitude and direction in Cartesian coordinates (x, y, z). In Figure 18.6, we use `typedef` to give the name `velocity` to this kind of array. The variable `wind` is an array of velocities, representing the winds in several locations. The first coordinate of each wind is related to the second and third; taken together, they specify one physical object. However, the first coordinate of one wind has little relationship to the first coordinate of the next wind. You would expect various functions in this program to have parameters of type `velocity`, as does the function `speed()` in Figure 18.7.

```
typedef double velocity[3];Type velocity is an array of 3 doubles.
double speed( velocity v)Given velocity, calculate wind speed.

velocity calm = {0, 0, 0};No wind.
velocity wind[5];      // The winds for 5 towns.
```

Figure 18.6. Declaring an array of arrays.

Using an array of arrays. The program in Figure 18.7 implements a sample wind array representing five locations. Altogether, it contains 15 `double` values, five locations with three coordinates each. In this example, `wind[0]` is the entire velocity array for Bradley Field and `wind[3][0]` is the x coordinate of the velocity at Sikorsky Airport.

Weather stations at five locations phone in their instrument readings daily to a central station running this program. When a weather station reports its data, the data are recorded in the wind table for that day. The program accepts a series of readings, then prints a table that summarizes the data and the wind speeds at the locations that have reported in so far.

Notes on Figure 18.7. Calculating wind speed.

First box: the type declaration. We declare a type to represent the velocity of one wind. We use this type to build a two-dimensional array (several winds with three components each) and to pass individual winds to the `speed()` function.

Second box: the calculation functions.

- We define a 1-line function for squaring a number because it will make our formulas more readable.
- One reason to use `typedef` is that a typedef name makes it easier and clearer to write correct function headers. The `speed()` function operates on velocity arrays: the use of the typedef name makes that clear.

Third box: the data structure.

- Three objects are declared here as parallel arrays. Together, they form a masked table of wind velocities for several weather locations, whose names are listed in a form that can be passed to `menu()`.
- The array of names has one extra item on the end to make it simple to end menu processing and finish the program.
- The mask array is initialized to `false` values (0) to indicate that, initially, no stations have called in their data. Recall that if an initializer is given that is too short for the array, all remaining array locations will be initialized to 0.

Fourth box: entering the data.

- The `name` and `mask` arrays are parallel to the `wind` array. Once a city is chosen, that city number is used to subscript all three. When the wind information for that city is entered, the corresponding mask is set to `true`. If the same city reported a second set of data, it simply would replace the first.
- Even though `wind` is declared as an array of arrays, not a matrix, a single velocity component is accessed using two subscripts.
- For simplicity, error checking is omitted here. In a realistic application, the numbers entered would be tested for being reasonable and an error recovery strategy would be implemented to recover from accidental input of nonnumeric data.

Fifth box: calculating one wind speed.

- The argument to the `speed()` function is a single wind velocity vector, not the whole array of winds, because that calculation involves only one velocity. By passing only the relevant row of the wind array, we simplify the code for `speed()`. Inside the function, we focus attention on that single wind and can access the individual velocity components using only one subscript.

Sixth box and Figure 18.8: printing the wind speed table.

- Sample output follows, with dashed lines replacing repetitions of the menu:

```
Wind Speed

0. Bradley
1. Bridgeport
2. Hamden MS
3. Sikorsky
4. Tweed
5. --finish--
Enter number of desired item: 3
```

This program creates a table of wind velocities at several weather stations, calculates the wind speeds, and prints a report. It calls the function in Figure 18.8.

```
#include "tools.hpp"
#include <cmath>
#define N 5
typedef double velocity[3];

void printTable( ccstring names[], bool mask[], velocity w[] );
double sqr( double x ) { return x * x; }
double speed( velocity v ){ return sqrt(sqr(v[0]) + sqr(v[1]) + sqr(v[2])); }

int main( void )
{
    int city;
    double windspeed;
    velocity wind[N];
    bool mask[N] = { false };         // Initialize all masks to false.
    ccstring names[N+1] = { "Bradley", "Bridgeport", "Hamden MS",
                            "Sikorsky", "Tweed", "--finish--" };
    cout <<"\n Wind Speed\n" );
    for (;;) {
        city = menu( " Station reporting data:", N + 1, names );
        if (city == N) br/*ak;User selected "quit"
        cout <<" Enter 3 wind components for " <<names[city] <<":  ";
        cin >>wind[city][0] >>wind[city][1] >>wind[city][2];
        mask[city] = true;
        windspeed = speed( wind[city] );
        cout <<"\t Wind speed is " <<windspeed <<".\n";
        printf( "\t Wind speed is %g.\n", windspeed );
    }
    printTable( names, mask, wind );
    return 0;
}
```

Figure 18.7. Calculating wind speed.

This function is called from Figure 18.7.

```
void
printTable( ccstring names[], bool mask[], velocity w[] )
{
    int k;
    cout <<"\n Wind Speeds at Reporting Weather Stations\n";

    for (k = 0; k < N; ++k) {
        if (!mask[k]) continue;
        cout <<" " <<left <<setw(15) <<names[k] <<right;
        cout <<fixed <<setprecision(2) <<"(" <<setw(7) <<w[k][0]
             <<setw(7) <<w[k][1] <<setw(7) <<w[k][2] <<" ) speed:  "
             <<setw(7) <<speed( w[k] ) <<"\n";
    }
}
```

Figure 18.8. Printing the wind speed table.

```
Enter 3 wind components for Sikorsky: 1.30    2.10   -1.10
Wind speed is 2.7037.

-------------------------------
Enter number of desired item: 4
Enter 3 wind components for Tweed: 1.50    2.00    0.00
Wind speed is 2.5.

-------------------------------
Enter number of desired item: 0
Enter 3 wind components for Bradley: 0.73    1.60   -2.10
Wind speed is 2.73914.

-------------------------------
Enter number of desired item: 5

Wind Speeds at Reporting Weather Stations
Bradley        (   0.73   1.60  -2.10 )  speed:    2.74
Sikorsky       (   1.30   2.10  -1.10 )  speed:    2.70
Tweed          (   1.50   2.00   0.00 )  speed:    2.50
```

- The parameters to the `print_table()` function are the three parallel arrays that make up the wind table. We use the `names` array to print the locations and the `mask` array to avoid printing "garbage" values for weather stations that have not reported in.

18.1.6 Dynamic Matrix: An Array of Pointers

Dynamic allocation of memory is not limited to one-dimensional arrays. Since 2D arrays are stored in contiguous memory, they also can be allocated as a single large area, a **dynamic 2D array**. However, accessing a particular element using the normal, two-subscript notation is not possible. (A function to perform this operation is considered in an exercise on image processing.) Alternatively, a 2D object can be represented as an array of pointers to arrays, using dynamic allocation repeatedly to create each part of the structure.
The result is a **dynamic matrix** data structure in which the rows can be efficiently manipulated and swapped, as illustrated in Section18.4. Elements of this matrix can be accessed using the matrix name and two subscripts, as if the entire matrix were allocated contiguously. This is ideal for applications where entire rows of the matrix are treated as units. For example, in Gaussian elimination, each row represents an equation, and entire equations are swapped as the solution progresses.

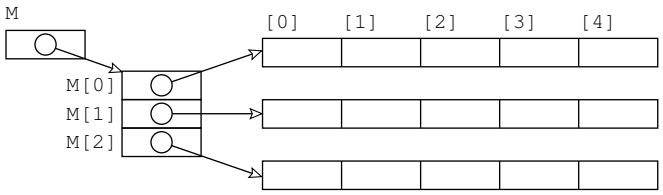


Figure 18.9. An array of dynamic arrays.

18.1.7 Multidimensional Arrays

Scientists and engineers use multidimensional arrays to model physical processes with multiple parameters. As in the two-dimensional case, a distinction should be made between multidimensional objects and arrays of matrices or matrices of arrays. C supports all these multidimensional data structures. Usage should be the guiding factor in deciding which to implement.

When using arrays of matrices or matrices of arrays, the rules for type compatibility of array parameters can become confusing. It is especially helpful to use `typedef` to define names for the subtypes, such as rows, and use those names to declare function parameters. The manipulation of uniform multidimensional structures can be more straightforward.

Three-dimensional arrays. The **dimensions** of a three-dimensional (3D) array usually are called *planes, rows,* and *columns*. The layout in memory is such that everything in plane 0 is stored first, followed by plane 1, and so forth. Within a plane, the slots are stored in the same order as for a two-dimensional matrix. Figure 18.10 shows a diagram of a 3D array with its subscripts.

When declaring a 3D array or a type name for a 3D type, each dimension must have its own square brackets, as shown in Figure 18.10. A 3D object may be referenced with zero, one, two, or three subscripts, depending on whether we need the entire object, one plane, one row of one plane, or a single element. For example, the middle plane in Figure 18.10 is `three_d[1]`. This plane is a valid two-dimensional array and could be an argument to a function that processes 2-by-4 matrices. Three-dimensional arrays are referenced analogously to matrices. For example, the last row of the last plane is `three_d[2][1]` and the last slot in that row is `three_d[2][1][3]`. A 3D function parameter is declared using a `typedef` name or with three sets of square brackets. Of these, the leftmost may be empty, but the other two must give fixed dimensions.

A `typedef` for a 3D array would extend the form for two dimensions, with the additional dimension, in square brackets, at the end. Multidimensional arrays may be initialized through nested loops or by properly structured initializers; a **3D initializer** would use sets of brackets nested three levels deep.

18.2 Application: Transformation of 2D Point Coordinates

An interesting application of two-dimensional arrays is the production of graphic images on the screen, often animated images. To move "objects" around on the screen, it may be necessary to rotate or translate them from their current position. This requires transforming the coordinates of the points constituting the object. In this section, we show a program that reads a set of point coordinates representing an object from a file and produces a new, transformed set of points. The code is written to transform a 2D image. However, by changing a `#define`, the same code will work with 3D images.

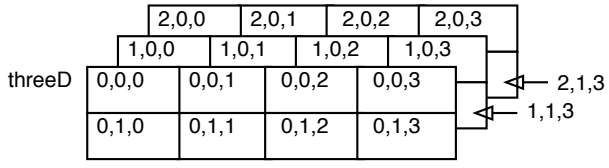


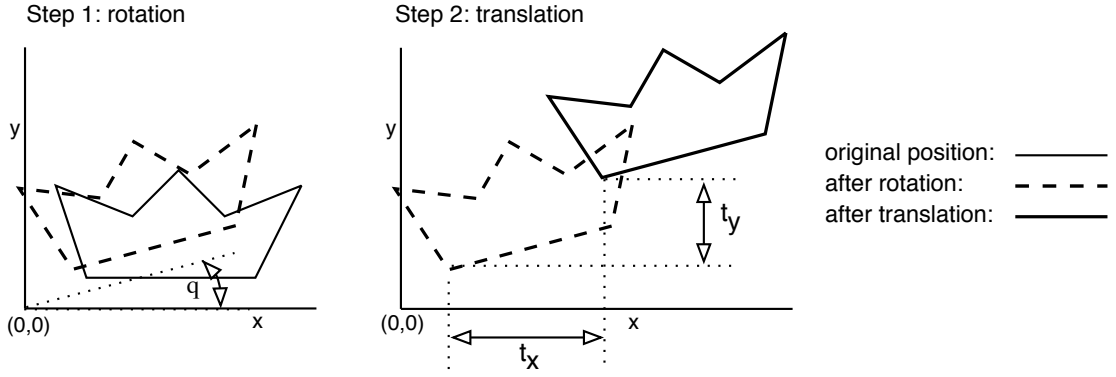
Figure 18.10. A three-dimensional array.

Problem: Write a program that reads in a set of points representing an object and produces another set of points based on a transformation (rotation and translation) of the original.

Input: (1) The name of a data file containing point coordinates, one point per line, each line containing the coordinates of one point, separated by a space. For a 2D drawing, there will be two coordinates, x and y, per point. For a 3D drawing, there will be also be a z coordinate.

(2) The transformation will be entered in the form of a counterclockwise (ccw) rotation angle, θ, and numbers that are the translational changes in each of the 2 or 3 dimensions. To avoid confusion, the rest of this discussion is in the context of 2D drawings.

The diagram shows a rotation angle of 15 degrees and translation of (10, 6).



Constant: π.

Formula: The transformation usually is represented in symbolic matrix form as

$$p_{new} = R\, p_{old} + T$$

where p_{new} and p_{old} are given by (x, y) coordinate pairs, R is a 2-by-2 rotation matrix for an object rotating counterclockwise (left) or clockwise (right) through angle θ:

$$R_{ccw} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad \text{or} \qquad R_{cw} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

and T is a translation given by another (x, y) pair. Expanded, the matrix formula for counterclockwise rotation becomes two equations:

$$\begin{aligned} p_{new_x} &= p_{old_x} \cdot \cos\theta - p_{old_y} \cdot \sin\theta + T_x \\ p_{new_y} &= p_{old_x} \cdot \sin\theta + p_{old_y} \cdot \cos\theta + T_y \end{aligned}$$

The `sin()` and `cos()` functions are in the standard `math` library, which is included by `tools.hpp`.

Output: The original point coordinates are echoed to the screen in a table, and next to them are the new point coordinates that have resulted from the transformation. Display two decimal places.

Limitations: Since we are using vectors to store the points, there is no limitation on the number of points in a drawing.

Figure 18.11. Problem specifications: 2D or 3D point transformation.

This program is specified in Figure 18.11. It calls functions can be found in Figures 18.13 through 18.15.

```
#include "tools.hpp"                        // File:  crown.cpp
#include "drawing.hpp"

int main( void )
{
    cout <<"Welcome to the " <<DIM <<"D drawing transformation program\n";

    Drawing dOld;                           // Input the points of the drawing.
    Transform trans;

    Drawing dNew( dOld, trans );
    dOld.displayPoints( dNew );             // print result table

    return 0;
}
```

Figure 18.12. 2D point transformation—main program.

A two-dimensional transformation is composed of two parts: a rotation and a translation. First, the object is **rotated** counterclockwise about the origin by an angle θ. Then it is **translated.** This is a straight-line motion described by offsets in position along each of two orthogonal axes (usually the x and y axes). Given theta and the two offsets, an appropriate equation can be written for calculating the transformed coordinates of a point based on its original location. This situation is depicted in Figure 18.11 along with the specifications for the 2D transformation. The main program is listed in Figure 18.12, and the supporting functions are shown in Figures 18.13 through **??**.

Classes are defined for a Drawing, a Point, and a Transformation. The first has only one data member: a vector. We define a class in order to have a place to put the relevant functions. The data members of the other two classes are arrays of doubles. We need to wrap these arrays inside classes to provide a place for functions and to make it possible to put the Points into a vector. The base type of a vector can be a primitive type or a class type, but it cannot be an array.

Notes on Figure 18.12: 2D point transformation—main program.

First box: the environment.

- We need the C math library, `<cmath>`, which is included by `tools.hpp`.
- The file `drawing.hpp` contains the class declarations for Drawing, Point, and Transform.

Second box: Constructing the classes.

- The Drawing constructor asks the user for a file that contains a drawing and reads it into memory. Errors during opening and reading are handled.
- The Transform constructor reads the rotation amount and translation amount from the keyboard and computes the transformation matrix.

Third box: Using the transformation.

- A second Drawing constructor initializes a drawing by transforming an existing drawing.
- Having computed the new drawing, both old and new are displayed, side by side, for comparison.
- This program does not write the new drawing to a file, but that might be useful.

Notes on Figure 18.13: Three class declarations.

First box: Data members of Point.

- Some real-lift objects can be represented either by a structure or an array. In a previous chapter, we used a structure to represent a point in 2-space. Here we are using an array of doubles instead. There are several reasons:

- The processing done during a transformation is the same for both x and y components of a point.
- The code for that processing is easier to write and briefer using nested loops than using individual statements for each component.
- This code is written in such a way that it can be used for a 3D drawing as well as a 2D drawing. You can't do that if you must write the component name (x or y or z) in the code instead of the component number ($0 \ldots 2$).

Second box: Function members of Point.

- All of the Point functions are defined here, in the class declaration because they are all very brief. All three fit on one line.

```
#include "tools.hpp"                        // File:  drawing.hpp
#define DIM 2                               // For 2-dimensional drawings.
#define PI 3.1415927
//---------------------------------------------------------------------
class Point {
private:
    double pt[2];
public:
    Point() = default;
    Point(double x, double y){ pt[0]= x; pt[1]= y; }
    double& operator[]( int k ){ return pt[k]; }
    void print(){ cout <<" ( " <<pt[0] <<", " <<pt[1] <<" )\n"; }
};
//---------------------------------------------------------------------
class Transform {
private:
    double rotation[DIM][DIM];
    Point translation;
public:
    Transform();
    void transformPoint( Point oldP, Point& newP );
};
//---------------------------------------------------------------------
class Drawing {
private:
    vector<Point> pts;                      // A drawing has many points
public:
    Drawing();
    Drawing( Drawing& oldD, Transform& trans );
    void displayPoints( Drawing& dNew );
    void getDrawing();
};
```

Figure 18.13. Three Class Declarations

- We need the default constructor to declare Point variables, such as the one in the Transform class.
- The constructor with parameters is needed to create a Point from the coordinates we calculate, prior to pushing the point into a vector that represents a new Drawing.
- The definition of `operator[]` is needed to enable the programmer to write code that looks like double subscripting on an array. The Point class delegates the subscript operation to its underlying array. The subscript operator returns a reference to a double so that the result of subscript can be used to store a double value.
- The print function is very ordinary. It was necessary during debugging, although it is not used anywhere in the finished application.

Third box: Data members of Transform
This program was originally written in C, with all the data members and functions thrown into one file. Translating it to a reasonable OO design was a challenge. The key came in realizing that Transformation needed to be a class on its own. It is the first example in this book of a `process` class. All other classes have been data or controller or container classes (like vector).

- A *process class* contains data members and functions needed to carry out a process. Instances of data classes are sent to it as parameters and returned by its functions.
- To perform a transformation, we need a 2-by-2 matrix of doubles for the rotation and an array of two doubles for the translation. These members define the transformation and really should not be part of any other class or of main().

Fourth box and Figure ??: the Transform functions.
We imagine that this class could be part of a large graphical drawing package such as the one used to produce the drawings in this book. The prototypes

- The Transform constructor must get the data to define the transform. In this case, the data comes from the keyboard but it could come from a file or (more likely) from a GUI interface. The actual input data is used to compute the transformation matrix.
- There is only one function, and it carries out the process for which the class was designed. It applies the transformation to a point and returns another point. The algorithms come from linear algebra, where a *vector* is an array of numbers and a *matrix* is a rectangular array of numbers. Vectors are often used to represent points in 2-space or 3-space. Do not confuse this terminology with the C++ meaning of vector, which is a container class capable of storing an array of numbers.
- The transformation process starts by computing an algorithm called *dot product*, once for each dimension. The dot product of vectors a and b is:

$$a \cdot b = \sum_{k=0}^{DIM-1} a_k \times b_k$$

  This is used twice to calculate the product of the rotation matrix and the old point, where row_0 and row_1 are the two rows of the translation matrix, considered as vectors.

$$newP = \begin{bmatrix} row_0 \cdot oldP \\ row_1 \cdot oldP \end{bmatrix}$$

- In Figure **??**, the fifth box carries out the matrix operation. The outer loop calculates the two components of the new point. The inner loop does the dot product to calculate one new coordinate. Double subscripting is used to access the elements of the rotation matrix.
- The sixth box does the simpler operation of translating the figure, that is moving it up or down and right or left by adding or subtracting a shift amount from each coordinate.

Fifth box of Figure 18.13: Data members of Drawing

- This class is simply a wrapper for a vector of Points. We need a class instead of simply using a vector because we need a place to put the related functions.

Sixth box of Figure 18.13 and Figure 18.15: Functions in the Drawing class.

- There are two constructors. One is used to initialize the old Drawing from a file, the other initializes a new Drawing by transforming the old one.
- The code for handling the input stream is like all the other examples except for one feature: the Drawing class (not main()) prompts the user for the name of a file. This is a somewhat arbitrary decision. In this case, by opening and closing the file within the constructor, we avoid having a stream member of the class.
- The file handling follows the usual pattern of checking for all kinds of errors and handling errors by calling fatal. In this case, there is no reasonable way to recover from a missing or corrupted file, and little effort is required from the user to restart the program.
- For each line in the input file, two double values are read and used to construct a Point, which is then pushed into the vector. The final line of output was useful during debugging to prove that the vector was

---

These functions are called from Figure 18.12. The constructor inputs a rotation angle and a translation vector from which the 2D transformation is constructed, assuming a ccw rotation of the object. The function `transformPoint()` applies this transformation to a single passed point and returns new coordinates.

```
Transform:: Transform(){
    double theta;                       // rotation angle
    for (;;) {
        cout <<"Please enter counterclockwise rotation angle, in degrees:  ";
        cin >>theta;
        if (theta >= 0 && theta <= 360) break;
        cout <<"Error:  Angle must be in the range 0 - 360 degrees\n";
    }
    theta = PI * theta / 180.0;         // convert angle to radians for math

    rotation[0][0] = cos( theta );      // compute rotation matrix elements
    rotation[0][1] = -sin( theta );
    rotation[1][0] = sin( theta );
    rotation[1][1] = cos( theta );

    cout <<"Please enter the translation amount ( X, Y ): ";
    cin >>translation[0] >>translation[1];
}
// ---------------------------------------------------------------------
Point Transform:: transformPoint( Point oldP ) {
    Point newP; int r, c;               // loop counters
    for (r = 0; r < DIM; ++r) {
        newP[r] = 0;                    // rotate by calculating the dot product
        for (c=0; c<DIM; ++c) newP[r] += rotation[r][c] * oldP[c];
    }

    newP[0] += translation[0];          // add translation vector to point
    newP[1] += translation[1];

    //oldP.print(); newP.print();       // debugging outputs
    return newP;
}
```

Figure 18.14. Functions for the Transform class.

being filled properly.

- The second constructor does all the work of the program, it transforms the points of the first drawing and pushes new points into the second drawing. As usual, the details of the transformation are hidden in the Transform class.
- The `displayPoints()` function is unusual because it deals simultaneously with two Drawings: the original is displayed on the left and the tranformation on the right.
- This function illustrates an important fact about privacy in C++[2]: the parts of the implied parameter (the old point) and the parts of the explicit parameter (the new point) are both visible to this function. There is no need to use getter functions to access the private vector of either one.
- This code is also interesting because it uses double subscripts. There is a lot of design and a lot of machinery behind this simple-looking clause: `dNew.pts[k][1]`. In this expression, the first subscript is interpreted by the vector class, the second one by the Point class.

Results of transforming a point set. The program was run on a data set defining the crownlike object originally shown in Figure 18.11. The object was rotated ccw by 15° and moved 6 units in the x direction and 10 units in the y direction. This would move the crown a little to the right and somewhat more upward. The output from the program follows. It is left to the reader to connect the dots.

```
Welcome to the 2D drawing transformation program
Please enter name of file containing object points: crown.in
    The drawing has 7 points
Please enter counterclockwise rotation angle, in degrees: 13
Please enter the translation amount ( X, Y ): 2 6

  Transformation of coordinates

Pt  Old X  Old Y     New X  New Y
 1   2.00   8.00      2.15  14.24
 2   6.00   7.00      6.27  14.17
 3  10.00   9.00      9.72  17.02
 4  14.00   7.00     14.07  15.97
 5  18.00   8.00     17.74  17.84
 6  16.00   2.00     17.14  11.55
 7   4.00   2.00      5.45   8.85
```

18.3 Application: Image Processing

This example introduces several new programming techniques that are useful in a variety of applications:

1. Binary files.
2. Closing and reopening a file for a different use.
3. Reading and writing a huge file in one operation, using low-level I/O.
4. Allocating a dynamic array for a 2D image, and processing it using a subscript function to convert logical two-dimensional subscripts to physical one-dimensional subscripts.
5. The concepts of deep copy and shallow copy.
6. Skipping whitespace and comments in a file, as part of parsing an image header.

18.3.1 Digital images.

One task for which we use computers is processing digital images, for example, cropping them or restoring corrupted pictures. Consider the "snow" you see on the screen of a TV set with a poor antenna. Removing the snow, thereby producing a clearer picture, makes other image processing tasks easier. This image restoration can be accomplished using many different techniques, both simple and complex, with varying levels of success. The program we develop here performs a simple technique known as *image smoothing*. Snow is removed by introducing an overall blurring effect.

[2]This is unlike Java. In Java, one object cannot access the private parts of another object, even if they belong to the same class

These functions are called from Figure 18.12.

```
// ---------------------------------------------------------------------
// Make a new drawing by reading points from an input file.
Drawing:: Drawing() {
    string fname;                          // name of data file
    double x, y;

    cout <<"Please enter name of file containing object points:  ";
    getline( cin, fname );
    ifstream fin( fname );
    if (!fin.is_open()) fatal( "Cannot open %s for input.", fname.data() );

    // Read data points until end of file or error occurs.
    for (int k = 0; ; k++ ) {
        fin >>x >>y;
        if (!fin.good()) {
            if (fin.eof()) break;          // End of file -- leave loop.
            else fatal( " Error reading file %s\n", fname.c_str() );
        }
        pts.push_back( Point(x, y) );
    }
    fin.close();
    cout <<" The drawing has " <<pts.size() <<" points\n";
}

// ---------------------------------------------------------------------
// Make a new drawing by rotating and translating the pts of an old one.
Drawing:: Drawing( Drawing& oldD, Transform& trans ) {
    Point newP;
    for (Point p : oldD.pts) {
        newP = trans.transformPoint( p );
        pts.push_back( newP );
    }
}

// ---------------------------------------------------------------------
void Drawing:: displayPoints( Drawing& dNew ) {
    cout <<"\n Transformation of coordinates\n" <<endl;
    cout <<"Pt Old X  Old Y     New X  New Y\n";

    for (int k = 0; k < pts.size(); k++) {   // put both points on one line
        cout <<fixed <<setprecision(2)
             <<setw(2) <<k+1 <<setw(7) <<pts[k][0] <<setw(7) <<pts[k][1]
             <<setw(10) <<dNew.pts[k][0] <<setw(7) <<dNew.pts[k][1] <<endl;
    }
}
```

Figure 18.15. Functions for the Drawing class.

Inside the computer, a picture must be stored in a discrete form. The format of a **digital image** typically is a header, containing information about the nature and size of the picture, followed by a grid of numbers, called pixels, where each number corresponds to the amount of light captured at that location by a camera. These numbers are typically scaled to a range of $0 \ldots 255$, where 0 corresponds to black, 255 is white, and the levels in between are shades of gray. This scaling is done so that each pixel can be stored in memory using only a single byte, a great memory savings when a typical image grid could be a square of size 1000-by-1000 numbers.

Smoothing eliminates extreme pixel values. Various situations (a dirty camera lens, dust inside the camera) can introduce speckles, called "snow" into a photograph. Usually we throw such damaged images away, but sometimes we prefer to rescue what we can of the photo.

The idea behind **smoothing** is that, in general, most pixel values are similar to those of their neighbors and the image intensity changes gradually across the picture. If one of the pixels in the image is corrupted, then its value probably has become quite a bit different than the values of the pixels surrounding it. Therefore, perhaps, a better value for the pixel would be based on its neighbor's values. The simplest kind of smoothing calculation is to replace every pixel value by the average pixel value in a small square window centered about the pixel's location. If a pixel is corrupted, this average should be much closer to the true value than the original value. If it is not corrupted, the true value will be changed slightly. The resulting image typically does not include the extremely erroneous pixel values, but some blurring of the picture does occur, especially for larger calculation windows.

Specifications for an image smoothing program are given in Figure 18.16. The various type declarations and functions that compose the program's implementation are shown in Figures **??**–**??**. The results of the smoothing operation are in Figure 18.25. The `fread()` and `fwrite()` functions are used in this application to read and write binary data files.

18.3.2 Smoothing an image.

Notes on Figure 18.17: Main program for image smoothing.

First box: definition two global functions.

- This main program is much like all earlier main programs: it sets up the environment, gets information from the user, opens a stream, instantiates the Pgm class, and calls the Pgm functions.
- It is different in two significant ways. First, the user interaction is extensive and has been moved out of main into a function. The prototype is here and the call is in the second box. The code is in Figure 18.18
- Second, a sophisticated technique is used by main to verify that the file name supplied by the user will not cause an unintentional file deletion. The function header is given here and the call is in the second box. The technique will be explained in the notes on Figure 18.18
- These two functions are written as global functions because they belong to `main()`, not to a class. They are declared to be `static` functions so that they *stay* inside the main module and are not visible to any other part of the program.

  All information passed between the functions and `main()` must be passed through parameters. Note the ampersands in the first function prototype: they signify that the function will use these parameters to return information to `main()`. In the second function, the information is going from `main()` to the function, so no `&` is needed.

Second box: User inputs. `main()` is responsible for getting two file names and a mask size from the user. Two of these inputs will be validated.

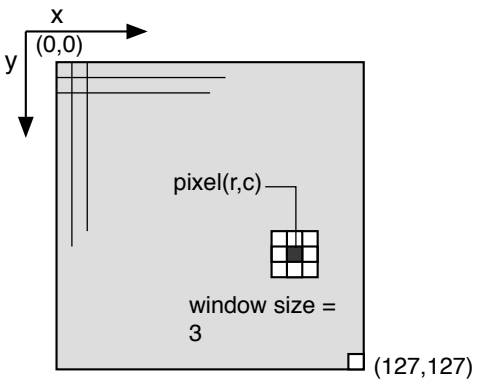
Third box: Opening the output file.

- A basic principle of program design is: *acquire all necessary resources before doing anything irreversible or heavily wasteful.* A corollary is: *open your output file before doing extensive processing or asking a human being to invest time in the application.*
- In this case, we want to be sure that an output file is available before beginning to input and process a potentially large input file. If we cannot open the output file, we abort now.

Problem scope: Write a program that will read in a digital image stored in P5 format and produce a new image in the same format by smoothing the pixel values of the original image.

Input: (1) The name of a data file that contains a digital image. (2) The name of a data file into which the new digital image will be stored. (3) The size of a processing window centered about each pixel, defining the neighborhood of values to be used in the averaging calculation.

The processing window follow:



Constants: The file format code is "P5".

Formula: The average value of a processing window is the sum of the pixel values in the window divided by the number of pixels in the region. This formula does not hold for pixels around the border of the image, for which the window does not fit completely in the image. In such cases, the dimensions of the window must be cropped to confine the window within the bounds of the image.

Output: A new image is to be generated, with the same file header as the original image, but modified pixel values. Each new pixel value is the average value of the pixels in the window of the old image that is centered at the corresponding location.

Limitations: The input file must have the proper amount of data in it. The processing window size should be an odd number in the range 3 . . . 11.

Figure 18.16. Problem specifications: Image smoothing.

Fourth box: Creating the Pgm objects.

- The object named `p` is the original photo that we wish to smooth, and `q` is the smoothed photo that we will calculate.
- The first function called here is the primary constructor for the Pgm class. It will initialize `p` by reading a photograph from the file whose name is stored in `inname` The constructor reads in a brief header telling the dimensions and grayscale of the picture, then it allocates a dynamic array of the correct size and reads all the pixels into it.
- Before we can calculate the smoothed image, we must create a .pgm data structure and allocate memory for the pixels. The second function called here is a copy constructor. It initializes `q` by copying the non-dynamic parts of `p`. By doing so we make `q` the same size and grayscale as the original photo, but we do not copy the pixels. Then this copy constructor allocates a new array to hold the pixels that we will compute.

Fifth box: Doing the work. At this point, we have read in the original and prepared memory for the smoothed version. We can go ahead with the smoothing, then write out the result.

Sixth box: cleanup. Good programming style dictates that we free everything we allocate, as soon as we are done using it, so the file should be closed. We do this even when termination will be immediate.

Notes on Figure 18.18: Getting parameters for the smoothing.

This program is specified in Figure 18.16. It uses the parts in Figures 18.18 through 18.24.

```
#include "tools.hpp"
#include "pgm.hpp"                              // File:  smooth.cpp
static void getParameters( string& inname, string& outname, int& maskSize );
static void checkOkToOpen( string outname );

int main( void )
{
    cout <<"This program smoothes a greyscale image\n";
    string inname;
    string outname;
    int maskSize;
    getParameters( inname, outname, maskSize );
    checkOkToOpen( outname );                   // abort if not ok.

    ofstream outFile( outname );
    if (!outFile.is_open()) fatal( "Cannot open %s for output", outname.data());

    Pgm p( inname );
    Pgm q( p );

    q.smooth( p, maskSize );
    q.write( outname, outFile );

    outFile.close();
    return 0;
}
```

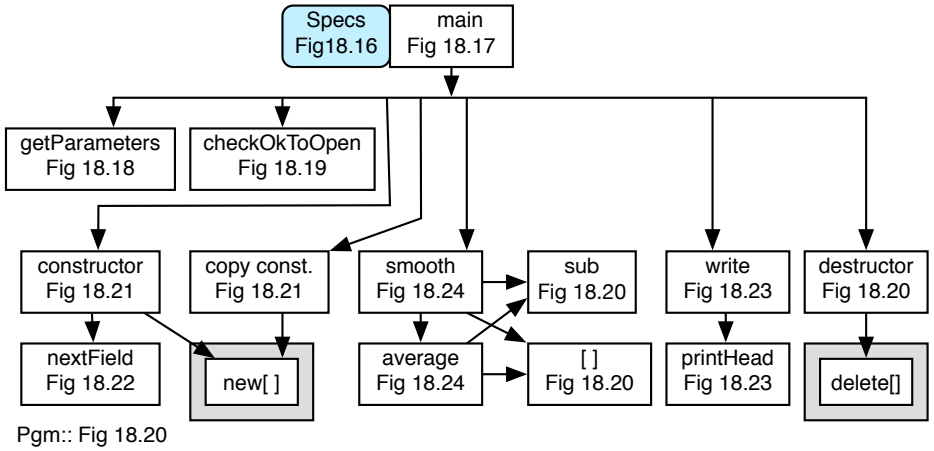


Figure 18.17. Image smoothing—main program.

```
void getParameters( string& inname, string& outname, int& maskSize )
{
    cout <<"Please enter name of image file to open:  ";
    getline( cin, inname );
    if (!cin.good()) fatal( "Error reading input file name" );
    cout <<"Please enter name of file for result image:  ";
    getline( cin, outname );
    if (!cin.good()) fatal( "Error reading output file name" );

    cout <<"Enter size of square smoothing mask:  ";
    cin >>maskSize;               // should be 3, 5, 7, 9 or 11
    if (!cin.good()) fatal( "Error reading size of mask." );
    if (maskSize % 2 == 0)
        fatal( "Error: mask size must be an odd number.\n" );
    if (maskSize < 3 || maskSize > 11)
        fatal( "Error: mask size must be in range 3 - 11.\n" );
}
```

Figure 18.18. Getting parameters for the smoothing.

First box: the parameters. All three parameters are passed by reference (`&`), so the parameter names will be the addresses of main's variables. (The call is in Box two of the main program.) In the boxes below, input is read into the parameter variables, which actually stores the input in main's variables. When the `getParameters()` function returns, that input will be known to main().

Second box: the file names. To run this program, we need an input file and an output file. This code is the minimal amount of work to make an appropriate interface. The program would work if the first test were skipped, but the user-interface would be worse. If there is an error reading the first file name, a program should not ask the user to enter the second hame.

Third box: the smoothing mask. Here, we want only one simple number, but need to deal with three ways that the input could fail.

- The read operation will fail if the user enters a non-numeric character.
- The mask value could be out of the range permitted by the program.
- The value could be even, which is not allowed.

```
void checkOkToOpen( string outname ) {
    ifstream testFile( outname );
    if (! testFile.is_open()) return;

    char ch;
    testFile.close();
    cout <<"File " <<outname <<" exists.  Do you want to overwrite it?  [y/N] ";
    cin >> ch;
    if (tolower(ch) != 'y') fatal("Aborted");
}
```

Figure 18.19. Prevent accidental overwriting of existing file.

In any of these cases, we call fatal and expect the user to run the program again. Very little effort has been wasted.

Notes on Figure 18.19: Prevent accidental overwriting of existing file.

First box: testing whether the file exists.

- Suppose you wish to open a file named "x.pgm" for output. If a file by the same name already exists in the current active directory, writing `ofstream myOut("x.pgm")` will delete the existing file. Sometimes that is OK. Often it would be an unintended problem.
- Some applications actually *ask* the user whether he wishes to overwrite an existing file, and this is considered good practice. So we want to find out whether the output file already exists, and we want to do it in a way that works for any operating system.
- We do this by trying to open that output file name as an input file. If that effort fails, we are happy: the file does not exist. So we return from this function.

Second box: the file exists – now what?

- Close that input file! We do not really want it.
- But we have discovered a potential problem and need to ask the user what to do next. If he does want to overwrite the file, we are happy and just return.
- But if the user says that he does *not* want to overwrite the file, we call `fatal()` and abort.

Notes on Figure 18.21: the Pgm class.

First box: the typedef for pixels. We use a typedef to provide a short, meaningful name for a long, nonspecific type. We are using chars because the pixels are each one byte, and unsigned because a pixel value is in the range 0...255.

Second box: declaration of a window type.

- The smoothing process will compute the new pixel value by averaging the pixel values in a square area surrounding each pixel of the image. We need this type to define the position of that averaging window as we move it across the image.
- We do not need to define a type for this purpose; we could use an array of four integers. The reason for declaring a type name and part names is to make the code easier to read and understand.
- This type is a struct, not a class, for simplicity and because there are no functions associated with it. It is a helper-type for the Pgm class. All of its members are public.

Third box: data members. The list of data members includes all required fields of the Pgm format plus one member for internal use.

- A Pgm file in P5 format has these fields, separated by whitespace:
  - The characters "P5" must come first. We reserve a 3-char array for this code to allow space for a null terminator. All three slots are initialized to null, and the first two will be changed by the constructor. The null in the third slot allows us to print the 2-character code as a string.
  - The width and height of the image, in pixels, come next.
  - The gray scale value tells us how many different pixel values are allowed in the photo. Normally, this is 255, but it could be a more restricted number.
  - At least one whitespace (probably a newline) must follow the maxGray.
  - After that are the pixels of the image.
- There should be $length \times width$ pixels. If there are too many or too few, it is an error. This number is computed here and stored in a data member because it is used by several of the functions in this class.
- Comments are allowed anywhere in the header and are likely to come after the "P5". A comment starts with a `#` character and ends at the next newline.

Fourth box: private functions.

- These two functions are private because they are not intended for use outside the Pgm class.
- These functions are both *const functions: in a function prototype, the keyword* `const` *just before the semicolon declares that the function does not modify the data members of the class.*
- *The* `nextField()` *function is called several times from the Pgm constructor to help parse the header of the Pgm file.*
- *The* `average()` *function is called from smooth to perform the calculation on which smoothing is based.*

Fifth box of Figure 18.20: constructors and a destructor. Here we give an overview of the three methods. See Figure 18.21 for more detail.

- The primary constructor for this class opens the image file, reads in the header, allocates dynamic memory, and then reads in the pixels. Errors due to incomplete files and inconsistent files are handled.
- The second constructor is a copy constructor[3] that defines how to initialize a new class instance by copying an existing one. Every class automatically has a copy constructor that copies all the bits of the original

[3] A constructor whose parameter is type `const classname&` is always a copy constructor.

```
#pragma once                                    // File:  pgm.hpp
#include "tools.hpp"
typedef unsigned char pixel;

// A rectangular window, given by its upper left and lower right corners
struct Window { int left, top, right, bot; };

class Pgm {
private:
    char filetype[3] = {'\0'};       // 2 characters designate the file format.
    int width;
    int height;
    int maxGray;
    int len;                         // length of pixel array = width * height
    pixel* image ;

    void nextField( istream& s ) const;
    pixel average( Window w ) const;
public:
    Pgm( string inname );
    Pgm( const Pgm& oldP);
    ~Pgm() { delete image; }

    pixel& operator[]( int k ){ return image[k]; }
    int sub( int row, int col ) const { return row * width + col; }

    void smooth( const Pgm& p, int maskSize );
    void printHead( ostream& out ) const;
    void write( string outname, ofstream& outFile );
};
```

Figure 18.20. The Pgm class.

object into the new one. However, this default definition can be changed, and that is what we are doing here.

- The class destructor is the third line. We need an explicit destructor in this class because the constructor uses dynamic memory. It allocated a new pixel array named `image`, and we delete that array here.

Sixth box: subscript functions. These two functions are both inline. When an inline is called, the entire body of the function is written in the caller's code in place of the function call. For one-line functions, this increases efficiency.

- The first line defines how to do a subscript operation on a Pgm object. The function body delegates that subscript operation to the image inside the Pgm object. It returns the address (`&`) of the selected array slot, which can then be used on either side of an assignment statement.
- The syntax that you see in this function declaration is used only to define new methods for subscript. Calls on subscript use the familiar syntax for subscripting. For example, the last line of the smooth function calls `operator[]`.
- The second function, `sub` translates a logical 2D subscript to a physical 1D subscript. This is necessary because dynamically allocated arrays are all 1D arrays. The programming student should learn this computation and recognize it when it appears.

Seventh box: smoothing and output.

- The smooth function performs the main task of this application. It is in Figure 18.24.
- The `write()` function outputs the completed, smoothed image to the previously-opened output file. It is in Figure 18.23.
- The `printHead()` function is called while writing the modified image to a file. It was written as a separate function because it was also very useful to monitor the program's actions during debugging.

Notes on Figure 18.21: The primary Pgm constructor. This is an unusually complex function for reading the contents of a file because a Pgm file is written partly as a text file and primarily as a binary file. On an OS platform that distinguishes between text and binary modes, the file must be opened, closed, opened again, and closed again. Both the open operation and the read operation could generate errors that must be checked. So a seemingly simple job takes a whole page of code.

First box: reading the file header.

- The Pgm header is written in text mode. It must start with the 2-letter code "P5". Reading these chars one at a time, as chars, seems to be the easiest and safest way to do the input. The 3rd slot in the char array has previously been initialized to a '\0' so that we can print out the code as a string.
- Next, we expect to find three integers. They might be separated by a single whitespace character, but there could also be several spaces or an entire comment to skip. This is enough complexity to write a function to skip over this irrelevant material. See Figure 18.22. We alternate reading a number and skipping whitespace.
- Finally, we read the single required whitespace char at the end.

Second box: checking for and handling errors. Note: if any kind of error happens while reading a file, nothing in the file itself or in the stream will change until the stream error flags are cleared.

- It is not necessary to check for errors after every read operation because any error, anywhere in the header, makes the file invalid. We can attempt to read the whole header, and check at the end to be sure the stream is still "good".
- The two `if` statements in this box check for all error that might have happened while reading this file header. Anything bad causes immediate termination.

Third box: calculation and output. When we get here, we have finished reading the part of the file that is written in text mode. Only the binary portion follows.

- We need to change mode from a text file to a binary file. To do that, we must close the file and reopen it using `ios::in | ios::binary`.

```
Pgm:: Pgm( string inname ) {             // Initialize a pgm object from a file.
    int n, ch;
    // Open file the first time to read in text mode.
    ifstream pgFile( inname );
    if (!pgFile.is_open()) fatal( "Cannot open input file %s\n", inname.data() );

    // Scan and parse file header ----------------------------------------------
    pgFile >> filetype[0] >> filetype[1];    // Get file type
    nextField( pgFile );                     // skip to start of width field
    pgFile >> width;
    nextField( pgFile );                     // skip to start of height field
    pgFile >> height;
    nextField( pgFile );                     // skip to start of maxgray field
    pgFile >> maxGray;
    ch = pgFile.get();                       / Required whitespace character preceding pixels.

    if (!pgFile.good() || !isspace(ch) || strcmp( "P5", filetype) != 0)
        fatal( " %s is not a PGM file.", inname.data() );
    if (maxGray<1 || maxGray > 255)      // Check range of maxgray
        fatal( "MaxGray must be between 1 and 255" );

    // Remember current position in the file for seeking back to it later.
    streampos filePos = pgFile.tellg();
    if (filePos<0) fatal("Can't get file pointer after reading header");
    pgFile.close();
    pgFile.open(inname, ios::in | ios::binary); // Reopen file in binary
    pgFile.seekg(filePos);  // Restore file pointer.
    if (!pgFile.good()) fatal("Can't restore file pointer");

    image = new pixel[len];              // Allocate storage for pixel data.
    pgFile.read( (char*)image, len );    // Get pixel data.
    if (!pgFile.good()) fatal("Incomplete pgm file '%s'", inname.data());

    // Check for eof -----------------------------------------------------------
    pgFile.get();
    if (!pgFile.eof()) fatal("File '%s' contains unread data", inname.data());
    pgFile.close();
}

// ---------------------------------------------------------------------------

// Copies an existing Pgm image into this Pgm object.
Pgm:: Pgm( const Pgm& old ) {
    this = old;                          // Shallow copy.
    image = new pixel[ len ];            // Allocate new pixel array
}
```

Figure 18.21. The Pgm constructors.

```
// Skip past inter-field whitespace and comments in pgm header. ---------------
void Pgm:: nextField( istream& s ) {
    char c;
    string junk;
    for(;;) {
        s >> c;
        if (s.eof()) return;
        if (c != '#') break;
        getline( s, junk );
    }
    s.unget();
}
```

Figure 18.22. Parsing and printing the header.

- But after reopening, we need to get back to exactly the current position in the file. So before closing,we need to find out and remember where we are. That is what `tellg()` does: it tells us the current position of the file pointer, returned as a value of type `streampos`. Of course, we check for errors. That is tedious but important.
- After executing `tellg()`, `close()`, then `open()` and seekg(), we are back at the end of the file header and ready to read in binary.

Fourth box: allocation.

- To prepare for reading the pixels, we allocate an array long enough to hold `length * width` pixels.
- The function `read()` is a low-level input operation that will read a specified number of bytes into a given array. Unfortunately, it has methods for reading and writing type char, but no methods for type unsigned chars. We "work around" this limitation by casting the image array pointer to type `char*` for the read operation, and again for the writ operation later. This works without problems.
- If an eof occurs during the read operation, there are not enough pixels in the file to satisfy the read request. This could mean that the Pgm header is corrupted or that the end of the file has been removed. In either case, there is no point in going on, so we abort.

Notes on Figure 18.21, last box: The Pgm copy constructor. Every class has a copy constructor, defined by default. It copies the bits from one object to another. However, if the first object has dynamic allocation attached, only the pointer is copied, not the entire information. The result is a recipe for trouble: two objects end up pointing at the same dynamic allocation.

The Pgm copy constructor.

- In main, we allocate two Pgm objects, p and q. p is initialized by the primary Pgm constructor reading an image from a file. q is created to hold the modified image. The modified image should be the same size and shape as the original, so the header information for q is the same as for p. However, the two objects must end up pointing at different pixel arrays.
- The first line of this method copies the entire old object into the new one, including the image pointer. The second line computes the total size of the dynamic area.
- The second line allocates new memory and stores it in the new image. q is now a complete Pgm object, independent of p, and ready to use to store the modified pixels.

Notes on Figure 18.22. Parsing the header. The function `nextField()` is a private helper function for the Pgm constructor. Its purpose is to skip comments and whitespace embedded in the Pgm header. The use of `unget()` is new to this text.

```
// Print the pgm header. ------------------------------------------------------
void Pgm:: printHead( ostream& out ) const {
    out << filetype << "\n"
        << width << " " << height << "\n" << maxGray << endl;
}

// Print the entire Pgm image. ------------------------------------------------
void Pgm:: write( string fileName, ofstream& pgmFile ) {
    printHead( pgmFile );                              // Write .pgm header data
    pgmFile.close();

    pgmFile.open( fileName, ios::out | ios::app | ios::binary );
    pgmFile.write((char*)image, len);                  // Write pixel data.
    if (!pgmFile.good()) fatal("Error writing pgm file '%s'", fileName.data());
    pgmFile.close();
}
```

Figure 18.23. Writing the Pgm file.

First box: the skipping loop.

- There is no simpler way to eliminate whitespace and comments than to read the file one byte at a time and test each byte. Remember that the operator `>>` skips leading whitespace, which is part of the goal in this application.
- In a Pgm file, comments extend from the `#` to the next newline character. We read the header, one character at a time, checking for the `#`. When we find it, we skip the rest of the comment line.
- A non-whitespace character that is not a `#` causes us to leave the loop. At that point, we have read the first digit of one of the three numbers that describe the image. We have gone too far!

Second box: Put it back! This is a common situation, and C++ provides a function to handle it. The call on `s.unget()` moves the stream cursor back one character, effectively putting the most recent character read back into the stream. Now the stream cursor is positioned properly to read the next number. Note: you can only "unget" one character.

Notes on Figure 18.23: Writing the Pgm file. This is considerably simpler than reading a Pgm file.

First box: Writing the header. This code is straightforward. It is simpler than reading the header because no parsing is necessary. We supply spaces and newlines where they are needed.

Second box: Writing the entire image file. We need to reverse the process of reading the file. However, again, there are no unknowns and the code is much much simpler.

- The output file was opened in text mode in the Pgm constructor. After printing the header, we need to close the stream and reopen it in binary mode.
- When we reopen the output file in binary mode, we also specify append mode: the pixels will be written at the end of the header information that is already in the file.
- `write()` is a low-level output command that writes a specified number of bytes to a file. As with `read()`, we need to cast the pixel array from type `unsigned char*` to type `char*` before write can use it. This works without problems.

Notes on Figure 18.24. The smoothing calculation. The smoothing calculation computes each pixel of the new image by averaging the values of its neighbors in the old image.

First box (outer): Processing one pixel.

- The smoothing process involves nested loops, where the inner loop processes all the columns in one row of the image and the outer loop processes all the rows of the image. The nested loop structure in this function follows the typical form for processing a 2D array.
- Each pixel must be processed independently of the others. It is necessary to save the results of processing each pixel into a new image; otherwise, the processing of successive pixels would incorporate both old and new pixel values, which would give a distorted result.

First inner box: determining window bounds.

- The processing window is computed separately for each pixel. It is centered about the pixel in question and extends `maskSize/2` pixels in each direction from that center. Integer division by 2 makes sure that integer limits are generated.
- Around the borders of the image, portions of the window may extend past the image boundary. Since there are no pixels in these areas to use, the window bounds must be restricted to stop at the actual edges of the image. For example, if the proposed processing window would have a negative subscript on the left side, it is set to 0 instead.

Second inner box: calculating the new pixel value.

```
void Pgm:: smooth( const Pgm& oldP, int maskSize )
{
    const int offset = maskSize/2;         // offset from center to edge of window

    for (int r = 0; r < width; r++)
        for (int c = 0; c < height; c++) {
            Window mask;                   // boundary limits of processing mask
            mask.left = max( 0, c-offset );
            mask.top = max( 0, r-offset );
            mask.right = min( c+offset, width-1 );
            mask.bot = min( r+offset, height-1 );

            // compute and store new pixel value ---------------------------------
            image[ sub(r, c) ] = oldP.average( mask );
        }
}

// ---------------------------------------------------------------------------
// Compute average value of pixels in bounded window of image.
//
pixel Pgm:: average( Window mask ) const
{
    int sum = 0, count = 0;                // sum and pixel-count for average

    for (int r = mask.top; r <= mask.bot; r++)
        for (int c = mask.left; c <= mask.right; c++) {
            sum += image[ sub(r, c) ];
            count++;
        }
        return sum / count;
}
```

Figure 18.24. Smoothing.

Figure 18.25. Results of image smoothing program.

- The original image and the bounds of the processing window are passed to the `average()` function to do the actual calculation. By putting the calculation in a separate function, we keep the process of cycling through all the pixels separate from the process of calculating the value of one pixel. We also avoid writing a nest of `for` loops that is four levels deep. Both of these goals are worthy design goals.
- The single line of code in this box calls three of the functions we have defined. First, the `average()` function is called to compute the new pixel value. Then the row and column numbers of the current pixel sent to `sub()` and converted to a 1-D subscript. Finally, that subscript is used by `operator[]` to access `image`, the dynamic pixel array.

Last box: the double loop in the `average()` ***function.***

- This is a straightforward summing loop to add up the pixel values in a square area. The Window object contains the beginning and ending subscripts for the rows (outer loop) and for the columns (inner loop).
- When the loop ends, all n^2 values have been summed. The last line computes and returns the average.

Results of smoothing an image. The program was run on a sample image. The results are shown in Figure 18.25. The original image on the left was corrupted by a small amount of "snow." A window size of 3 was chosen for processing, which produced the image on the right. As can be seen, the extreme white values have been removed, but the image has been blurred. Other image restoration techniques, which are more complex, can remove the corrupted values without the blurring.

18.4 Application: Gaussian Elimination

Gaussian elimination is an algorithm for solving a system of m linear equations in m unknowns, as shown next. The goal is to find values for the variables $a, b, \ldots, m$ that simultaneously satisfy all the equations

$$\begin{aligned} c_{1,1}a + c_{1,2}b + \cdots + c_{1,m}m &= X_1 \\ c_{2,1}a + c_{2,2}b + \cdots + c_{2,m}m &= X_2 \\ &\vdots \\ c_{m,1}a + c_{m,2}b + \cdots + c_{m,m}m &= X_m \end{aligned}$$

We can use an M by $M+1$ matrix to contain the coefficients of the variables in the M equations, one row for each equation, one column for each unknown, and a final column for the constant term. To find the set of variable values that satisfy the system, we apply to the coefficient matrix a series of arithmetic operations that are valid for systems of equations. These operations include

- *Swapping.* Since the order of the equations in the matrix is arbitrary, we can exchange the positions of any two equations without changing the system. (But we cannot swap two columns because each column position is associated with a particular variable.)
- *Scaling.* Both sides of an equation may be divided or multiplied by the same number. That is, if we divide or multiply all of the coefficients and the constant term of an equation by a single number, the meaning of the equation remains unchanged.
- *Subtraction.* We can subtract one equation E_1 in the system from another, E_2, and replace E_2 by the result, without changing the constraints on the variable values that satisfy the system.

An algorithm that uses these operations, Gaussian elimination, can be used to compute the variable values that solve the **system of equations**.

We implement this algorithm by writing a function to perform each of the preceding operations, and a function, `solve()`, that uses them appropriately. The algorithm solves the system of equations in stages. At stage k, we select one equation, place it on line k of the matrix, then scale it by dividing all of the row's entries by its own kth coefficient. This process leaves a value of 1 in the kth column of the kth row of the matrix. The new kth equation then is used by the scaling and subtraction functions to reset the kth coefficient of every other equation to 0 while simultaneously adjusting the other coefficients. After M such elimination steps, the matrix (except for the last column) has a single element with the value 1 in each row and in each column. This corresponds to a set of equations of the following form, in which the last column of the matrix contains the solution to the system of equations:

$$\begin{aligned} 1 \cdot a &= X'_1 \\ 1 \cdot b &= X'_2 \\ &\cdots \\ 1 \cdot m &= X'_m \end{aligned}$$

18.4.1 An Implementation of Gauss's Algorithm

To solve a particular system of equations using this code, the user must first create a data file, `gauss.in`, that contains the data. The number of equations must be on the first line. Following that must be the coefficients of the equations, one equation per line. The constant term of each equation is the last entry on a line. The file we use in the example looks like this:

```
4
1    2    2   1   10
3   -1    0   5   -4
2    2    3   0   15
0   .5 0  -1   7   -9
```

The corresponding output, before solving the equations i:

```
Linear Equations to Solve:
------------------------------------------------------------
  1.000 * a +    2.000 * b +    2.000 * c +    1.000 * d =   10.000
  3.000 * a +   -1.000 * b +    0.000 * c +    5.000 * d =   -4.000
  2.000 * a +    2.000 * b +    3.000 * c +    0.000 * d =   15.000
  0.000 * a +    0.500 * b +   -1.000 * c +    7.000 * d =   -9.000
------------------------------------------------------------
```

Use Gaussian elimination to solve m equations in m unknowns.

```
#include "gauss.hpp"

int main ( void )
{
    Gauss matrix;                // Array of equation pointers.
    matrix.print( cout, "\nLinear Equations to Solve:");

    bool solvable = matrix.solve();
    matrix.print( cout, "\n Equations after Elimination" );
    if (!solvable) fatal(" Equations inconsistent or not independent.\n");

    matrix.answers( cout );
    return 0;
}
```

Figure 18.26. Solving a system of linear equations.

Notes on Figure 18.26. Solving a system of linear equations. This is a typical OO main program. It instantiates `matrix`, an object of class Gauss, and displays the system of equations that will be solved. Then `main()` uses `matrix` to call functions that solve the system of equations and to print out the final solution. This is the right way to build a program: keep `main()` simple. The other functions, in the classes, do all the work.

The call chart in Figure 18.27 shows the structure of the Gaussian elimination program; the main program is in Figure 18.26, the classes in Figures 18.28 and 18.29, and the functions in Figures 18.30 through 18.36. A call chart is given in Figure 18.27; most standard system functions have been omitted from the chart.

Notes on Figure 18.28: The data structure.

- The Gaussian Elimination algorithm is based on operations on entire equations, where each equation is represented by one row of a matrix. Although we *could* use a two-dimensional array to hold the equation's coefficients, this data structure does not reflect the nature of the problem well: the program really is working with an array of equations, not a two-dimensional array of numbers.
- We wish to use dynamic allocation so we can solve a system of equations with any reasonable number of unknowns. A matrix that represents a system of n equations will have n rows and $n + 1$ columns. Each column except the last represents one unknown. The last column represents the constant term in the

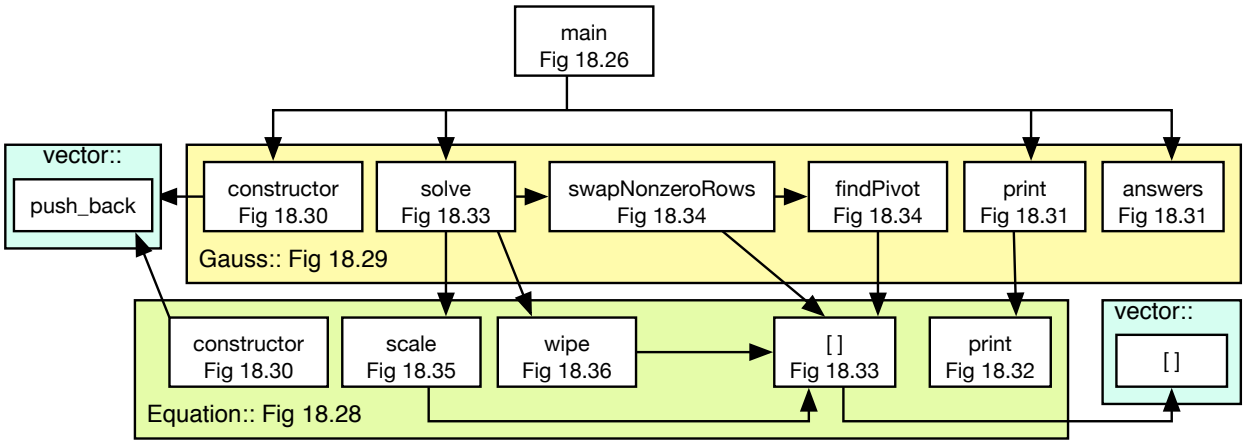


Figure 18.27. A call chart for Gaussian elimination.

This class is used by the Gauss class in Figure 18.29 to model one equation.

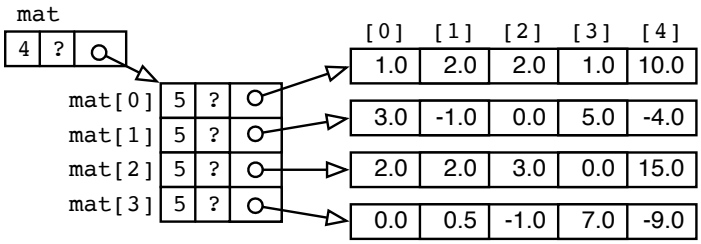
```
#pragma once
#include "tools.hpp"            // File:  equation.hpp
class Equation {
private:
    vector<double> coeff;
public:
    Equation( istream& eqIn, int nEq );
    void print( ostream& out );
    double& operator[]( int k ){ return coeff[k]; }

    void scale( int col );
    void wipe( Equation& piv, int pivRow );
};
```

Figure 18.28. The Equation class declarations.

equation.

- When the number of variables increases, both the number of rows and the number of columns increase. To allow any reasonable number of variables, we define the class `Equation` as an vector of `double` values and the class `Gauss` as a vector of equations. For the data file given earlier, the data structure will eventually have dynamically allocated parts that are related as shown in the diagram below. The question marks represent the data member in `vector` that tracks the actual length of the dynamic allocation. We simply do not know that length, and do not need to know.



Notes on Figure 18.28. The Equation class.

First box: the data member. he only data member of Equation is a vector. The class was created as a place to define functions that implement the row-operations of the elimination method.

Second box: the usual suspects.
There is nothing unusual here. These first two methods are what we expect to see in every class. The constructor prototype tells us that the coefficients for one equation will be read from an open stream. The definition of the subscript operator is expected: the outside world needs a way to access data stored in the coefficient vector. The operator definition is inline, the other two definitions are in Figure **??**.

Third box: the mathematics.
The last two functions implement the row-operations that are part of the Gaussian Elimination algorithm. The Equation parameter is passed by reference (`&`) because it will be modified within the `wipe()` function. The definitions are in Figure 18.34.

Notes on Figure 18.29. The Gauss class.

This class is used by `main()` to model a system of equations.

```
#pragma once                              // File:  guass.hpp
#include "equation.hpp"
#define EPS  .0001                        // Comparison tolerance for zero test.

class Gauss {
private:
    vector< Equation > mat;
    int nEq;

public:
    Gauss();
    ~Gauss() = default;

    void print( ostream& out, ccstring message );
    void answers( ostream& out );

    bool solve();                         // k is the column subscript
    int findPivot( int k );
    bool swapNonzeroRow( int k );
};
```

Figure 18.29. The Gauss class declaration.

First box: the constant. We need a comparison tolerance in certain portions of the algorithm to test whether coefficients equal (or nearly equal) zero. The `find_pivot()` function uses this constant to identify which equations have a nonzero coefficient in the current column. The tolerance is needed to avoid floating-point overflow caused by division by 0 or a nearly 0 value. The constant `EPS` is defined here, rather than in the function that uses it, to make it easy to find and change if the user requires more or less precision.

Second box: the data members.

- This program is able to handle a system of equations with any reasonable number of equations, *neq*. Since we do not know at compile time how many equations there will be, we use a vector to store them.
- The variable `neq` is the first thing read from the input file and determines how many lines of data (one equation per line) will be read. This number is used throughout the Gauss class. It is not necessary to store it as a class member because the same number will be stored inside the vector. However, the code is easier to write, read, and debug with this number stored in a convenient place.

Third box: the constructor and destructor.

- The class constructor is defined in Figure 18.30 and will be discussed in the notes for that Figure.
- It is not necessary to define a destructor for this class because all the dynamic allocation is hidden inside the vectors, which manage it. If a class does not have a definition for a destructor, C++ will provide a do-nothing destructor for the class. However, it is considered good style to define a destructor even if it does nothing. The phrase `= default` declares that the destructor should do nothing.

Fourth box: output. These function are defined in Figure 18.31.

- The `answers()` function is used once at the very end of the program to print the solution.
- The `print()` function is used just after the input phase is finished and just before the answers are printed. During debugging, it was used after every stage of the solution and was essential to track and correct the progress of the algorithm.

The Gauss constructor is called from `main()` in Figure 18.26; it opens the input file, reads equation data from it, and calls the Equation constructor.

```
Gauss:: Gauss() {
    // Open input file and read number of equations.  ---------------------
    ifstream eqInput( "gauss.in" );
    if (! eqInput.is_open()) fatal ( " Cannot open gauss.in" );
    eqInput >>nEq;
    // Read in all the equations.  ----------------------------------------
    for (int k = 0; k < nEq; ++k)   // Read and install an equation.
        mat.push_back( Equation( eqInput, nEq ));
    eqInput.close();
}
// ---------------------------------------------------------------------------
Equation:: Equation( istream& eqIn, int nEq ) {
    double temp;
    // Read coefficients for one equation.  -------------------------------
    for (int col=0; col<=nEq; ++col) {
        eqIn >> temp;
        coeff.push_back( temp );
    }
    if (!eqIn.good()) fatal( " Read error or unexpected eof." );
}
```

Figure 18.30. Constructing the model.

Fifth box: the mathematics.
These function are defined in Figure 18.33. They will be discussed in the notes that follow that Figure.

Notes on Figure 18.29. Constructing the model.

First box: the input stream. The Gauss constructor opens and closes the input stream.

- The first line of the file determines how many other lines will be read.
- For each input line (one equation), Gauss calls the Equation constructor and passes it a reference to the open stream. The number of equations must also be a parameter because that determines how many coefficients will be read for this equation.
- When the Equation constructor returns, the result is immediately pushed into the vector of Equations.

Second box: the input loop. This loop reads all the coefficients and the constant term for one equation. As each number is read, it is pushed into the vector of coefficients.

Third box: answers.

- Whether we are printing the equations or the answers, formatting the output is important. Here we specify fixed point with three decimal places, right justified in 8 columns. (Right justification is the default.)
- The second line of the output statement uses a character-code "trick". We want to print variable names a, b, c, d, etc. But we do not know how many letters we will need. So we calculate the ASCII code to print by adding the row number to the ASCII character 'a'. This produces as much of the alphabet as needed. It is easy, fast, and portable to any system where the alphabet is given consecutive character codes.

These functions are called from `main()` in Figure 18.26 and used in various other places to provide debugging output.

```
// Print the matrix, formatted so that each row is an equation.
void Gauss:: print( ostream& out, ccstring message ) {
    out << message <<endl;

    out << " ------------------------------------------------------------\n";
    for (int row=0; row<nEq; ++row) {       // Print all equations.
        mat[row].print( out );
    }
    out << " ------------------------------------------------------------\n";
}

// ---------------------------------------------------------------------------
// Print each variable with its value (from last column of matrix).
void Gauss:: answers( ostream& out ) {
    for (int row=0; row < nEq; ++row) {
        out <<fixed <<setprecision(3) <<setw(8)
            <<" " <<(char)('a'+row) <<" = " <<mat[row][nEq] <<"\n";
    }
    out << endl;
}
```

Figure 18.31. Output functions for the Gauss class.

Third box: error handling. Throughout this application, we are careful to do responsible error detection and print appropriate error comments. If any one of the equations in the system is unreadable, there is no way to solve the system and termination is appropriate.

To be OK, the *neq* expected coefficients and the constant term must all be present. If any one is missing or causes a read error, we can detect the problem with a single test at the end of the read loop.

Notes on Figure 18.31: Output functions for the Gauss class.

First box: the print function's message. The `Gauss::print()` function was used during debugging to display the model after every row or column operation. To make sense of the display, it was important to know *which* operation had just been done. This problem was easily solved by including a message as a parameter to print; the messages (shown below) give the name of the function that just finished its work.

Second box: the outer print loop.

- A page full of columns of number is hard to read. We improve the situation a lot by printing a line of dashes above and below the system of equations, so that the many steps of the solution are clearly separated. The output below shows the first quarter of the solution process and the final answers. (The rest has been omitted for brevity.)
- The loop in Gauss::print() is repeated once for each equation in the system. It delegates the actual printing of the equation to the expert: Equation::print(). That method contains a loop to print all the coefficients.

```
Linear Equations to Solve
 ------------------------------------------------------------
   1.000 * a +    2.000 * b +    2.000 * c +    1.000 * d +   10.000
   3.000 * a +   -1.000 * b +    0.000 * c +    5.000 * d +   -4.000
   2.000 * a +    2.000 * b +    3.000 * c +    0.000 * d +   15.000
   0.000 * a +    0.500 * b +   -1.000 * c +    7.000 * d +   -9.000
```

This function is called by Gauss::print() in Figure 18.31. It prints one row of the matrix formatted so that it looks like an equation.

```
// ---------------------------------------------------------------------------
void Equation:: print( ostream& out ) {
    int nEq = coeff.size()-1;   // The vector has n coefficients and 1 constant.
    char op= '+';               // To print this between terms in output equation.
    for (int col=0; col < nEq; ++col ) {
        if (col == nEq) op = '=';
        out << fixed << setprecision(3) << setw(8)
        << coeff[col] <<" * " <<(char)('a'+col) <<" " << op <<" ";
    }
    out << setw(8) << coeff[nEq] <<"\n";
}
```

Figure 18.32. Output for the Equation class.

```
 ------------------------------------------------------------
Starting solve loop at pivRow 0
After swap.
 ------------------------------------------------------------
   3.000 * a +   -1.000 * b +    0.000 * c +    5.000 * d +   -4.000
   1.000 * a +    2.000 * b +    2.000 * c +    1.000 * d +   10.000
   2.000 * a +    2.000 * b +    3.000 * c +    0.000 * d +   15.000
   0.000 * a +    0.500 * b +   -1.000 * c +    7.000 * d +   -9.000
 ------------------------------------------------------------
After scale.
 ------------------------------------------------------------
   1.000 * a +   -0.333 * b +    0.000 * c +    1.667 * d +   -1.333
   1.000 * a +    2.000 * b +    2.000 * c +    1.000 * d +   10.000
   2.000 * a +    2.000 * b +    3.000 * c +    0.000 * d +   15.000
   0.000 * a +    0.500 * b +   -1.000 * c +    7.000 * d +   -9.000
 ------------------------------------------------------------
After wiping all rows.
 ------------------------------------------------------------
   1.000 * a +   -0.333 * b +    0.000 * c +    1.667 * d +   -1.333
   0.000 * a +    2.333 * b +    2.000 * c +   -0.667 * d +   11.333
   0.000 * a +    2.667 * b +    3.000 * c +   -3.333 * d +   17.667
   0.000 * a +    0.500 * b +   -1.000 * c +    7.000 * d +   -9.000
 ------------------------------------------------------------
Starting solve loop at pivRow 1
. . .
 ------------------------------------------------------------
        a = 1.000
        b = 2.000
        c = 3.000
        d = -1.000
```

Notes on Figure 18.32: Output functions for the Equation class. The `Equation::print()` function prints one equation, with variable names and $*$, + and = signs.

Outer box: the loop.

- The formatting is the same here as in Gauss::print().

This function performs the Gaussian elimination algorithm. It uses row operations to reduce the first M columns of the matrix to an identity matrix. At that point, the last column contains the solution to the problem. This function is called from Figure 18.26. It uses the functions in Figures 18.34 through 18.36.

```
bool Gauss:: solve() {
    for (int pivRow = 0; pivRow< nEq; ++pivRow) {
        cout <<"Starting solve loop at pivRow " <<pivRow <<"\n";
        if (! swapNonzeroRow( pivRow )) // Do all rows have 0 in this column?
            return false;

        mat[pivRow].scale( pivRow );    // Make a 1 in mat[k][k].
        print( cout, "After scale." );  // For debugging.

        // Use the 1 in mat[k][k] to zero out the rest of column k.
        for (int row=0; row<nEq; ++row)
            if (pivRow != row) mat[row].wipe( mat[pivRow], pivRow);

        print( cout, "After wiping all rows." );
    }
    return true;
}
```

Figure 18.33. The solve() function.

- In a system of n equations, there are n unknowns. Each equation in the system starts with n coefficients, one for each unknown, and ends with a constant term. We want to print a + sign between every pair of coefficients and a = before the constant term. The char variable `op` is initially set to `'+'` and changed to `'+'` at the end of the loop.
- Currently, when a term is negative, the program prints both a `+` and a `-` sign in front of it. For even fancier output, we could test the sign of each coefficient and print either a `+` sign or a `-` sign, but not both.

Inner box: the variable names.
The actual variable names are computed here, as they were in Gauss::print().

Notes on Figure 18.33: The elimination algorithm. The main loop of this function is executed once for each equation in the system. Its purpose is to manipulate the matrix so that, eventually, it has ones on the main diagonal (elements with the same row and column subscripts) and zeros elsewhere. To do this it calls three functions (in the three boxes).

Outer box: To solve or not to solve, that is the question. Each time around this loop one equation is selected from the set of remaining equations. It is called the *pivot equation* and is swapped into the first row of the matrix that has not already been processed.

First inner box and Figure 18.34: Finding the next pivot row. Remember that swapping the order of two equations in a system does not change the solution to that system. Here we swap rows and do row operations.

- On pass k, the `swapNonzeroRow()` function has two purposes:
  (a) To find the equation with the largest non-zero coefficient in column k and row $>= k$.
  (b) To swap that equation with whatever is in row k.
  (c) To return `false` if all remaining equations have 0's in column k. The false is then returned to `solve()` to indicate that the system of equations has no solution.
- The first task is to select an equation, called the *pivot equation*, for the next phase of the process. On pass k, we look at all of the equations in rows k and greater. We select the equation with the largest coefficient in column k because this maximizes computational accuracy.

The `swapNonzeroRow()` function is called from `solve()` in Figure 18.33.

```
bool Gauss:: swapNonzeroRow( int k ) {
    int row = findPivot( k );

    // Coefficient k of pivot equation must be non-zero.  ------------------------
    if ( fabs( mat[row][k] ) < EPS ) return false;
    swap( mat[row], mat[k] );

    print( cout, "After swap." );                 // For debugging.
    return true;
}

// ---------------------------------------------------------------------------------
// Find the equation with the largest coefficient in column k.
int Gauss:: findPivot( int k ) {
    double coefficient = fabs( mat[k][k] );       // Largest value so far.
    int big = k;                                  // Index of current coefficient.

    for (int row = k+1; row<nEq; ++row) {
        if (fabs( mat[row][k] ) > coefficient) {
            coefficient = fabs( mat[row][k] );    // A bigger coefficient.
            big = row;                            // Remember where it was found.
        }
    }
    return big;     // Line number of equation with biggest coefficient.
}
```

Figure 18.34. Finding the next pivot row.

- `swapNonzeroRow()` delegates to the `findPivot()` function the job of finding the coefficient in column k that is farthest from zero.
- All comparisons must be made using the absolute value of the coefficient. `fabs()` is the C-standard function for computing the absolute value of a floating point number (type double or float).
- `findPivot()` calculates the maximum value in column k and returns the subscript of the row it is in. That row is swapped with the current row and becomes the next pivot equation.
- If the largest coefficient in column k in the remaining equations is zero or very close to zero, dividing by it will cause floating point overflow. If this happens, the problem cannot be solved by this method, either because it contains inconsistent formulas, or because one of the equations is a linear combination of some of the others. So `swapNonzeroRow()` checks for this and returns an error code.
- If one of the equations is a linear combination of some of the others, there are not enough constraints to determine a unique solution. If the equations are inconsistent, there are too many constraints and they cannot be simultaneously satisfied.
- The constant `EPS` defines what we mean by "too close to zero". The output below shows the beginning and end of the output from running the program on an unsolvable system of equations.

```
Linear Equations to Solve
 ------------------------------------------------------------
   1.000 * a +    2.000 * b +    2.000 * c +    1.000 * d +   10.000
   3.000 * a +   -1.000 * b +    0.000 * c +    5.000 * d +   -4.000
   2.000 * a +    2.000 * b +    3.000 * c +    0.000 * d +   15.000
   2.000 * a +    2.000 * b +    3.000 * c +    0.000 * d +   14.000
 ------------------------------------------------------------
Starting solve loop at pivRow 0
. . .
```

This function is called from Figure 18.33.

```
// Scale kth equation in matrix to have 1 in kth place
// Assumes places 1..k-1 are already zero.
// ------------------------------------------------------------------------------
void Equation:: scale( int col ) {
    double factor = coeff[col]; // scale factor
    for (int k=col; k <= coeff.size(); ++k)
        coeff[k] /= factor;
}
```

Figure 18.35. `Equation::scale()`.

```
Equations after Elimination
 ------------------------------------------------------------
   1.000 * a +    0.000 * b +    0.000 * c +    2.600 * d +   -1.600
   0.000 * a +    1.000 * b +    0.000 * c +    2.800 * d +   -0.800
   0.000 * a +    0.000 * b +    1.000 * c +   -3.600 * d +    6.600
   0.000 * a +    0.000 * b +    0.000 * c +    0.000 * d +   -1.000
 ------------------------------------------------------------
 Equations inconsistent or not independent.
```

Second inner box of Figure 18.33 and Figure 18.35: Scale an equation. Remember: Dividing every term of an equation by the same number does not change the solution of the equation.

- We divide all terms of the equation in row k by the value in column k. This leaves a value of 1 in `matrix[k][k]`.
- The loop starts at column k because all prior columns in row k are zero .

Third inner box of Figure 18.33 and Figure 18.36: Wipe an equation. After scaling row k, there is a 1 in matrix[k][k].

- The next step is to use that 1 to wipe out all the coefficients below it in column k.
- Take c, the coefficient in column k of row m. Multiply the entire pivot equation by c, then subtract the result from the equation in row m. This row operation does not change the solution to the system.
- Repeat this process for every equation in the matrix except the pivot equation. Now there are 0's in column k in every row except the pivot row.

18.5 What You Should Remember

18.5.1 Major Concepts

- The nature of control structures follows the form of the data being processed. Using 1D arrays requires a single loop control structure. Using 2D arrays requires nesting one loop within another to process all data elements. This effect continues as the data complexity increases.

This function is called from Figure 18.33.

```
// Clear coefficient k1 of equation k2 by subtracting mat[k2][k1] * equation k1
// from equation k2.  Assumes that mat[k1][k1] == 1.
// ------------------------------------------------------------------------------
void Equation:: wipe( Equation& piv, int pCol ) {
    double factor = coeff[pCol];
    for(int col=pCol; col<=coeff.size(); ++col) {
        coeff[col] -= factor * piv[col];
    }
}
```

Figure 18.36. Equation::wipe().

- A `typedef` name stands for the entire type description, including all the asterisks and array dimensions. In the basic syntax, this information is scattered throughout a variable declaration, with variable names written in the middle. This makes C declarations hard to read and easy to misunderstand. When you use a `typedef` name, all the type information goes at the beginning of the line and all the variable names come at the end.
- A table of data can be represented conceptually in two different manners. An array of arrays implies a unity of the elements in each of the rows in the table, whereas a matrix implies that each element in the table is independent of the others.
- Whether declared as an array of arrays or a matrix, the data elements are still stored sequentially in memory in row-major order. This layout can be exploited when filling or removing data from the structure, as in reading and writing data from files using `read()` and `write()`.
- The C language allows for arrays of many dimensions, although in practice using more than three or four dimensions is unusual.
- Example applications of 2D arrays include matrix arithmetic and image processing. Matrix arithmetic uses both 1D vectors and 2D matrices in various calculations. Image processing programs use 2D arrays of various sizes and typically store the data in binary files.

  Two kinds of matrix. C++ supports two distinctly different dynamic implementations of a matrix. In one implementation, all of the memory is contiguous and allocated by a single call on `new`. In the other, the matrix is represented as an array of pointers or a vector of vectors, each pointing at an array of data elements. Here, `new` is called several times, once for the dynamic array of pointers and once for each of the dynamic arrays of elements.

  The dynamic 2D array. A dynamic 2D array is useful for applications such as image processing, in which data sets of varying sizes will be processed. The size is generally known at the beginning of run time and a properly sized block of storage can be allocated then. Unfortunately, the programmer has to write a function to do the address calculations for converting the conceptual 2D subscripts to a physical 1D subscript before accessing a particular element. This calculation is examined in the exercises.

  The dynamic matrix data structure. A dynamic matrix can be an array of pointers to arrays. Like static two-dimensional arrays, elements are accessed by using two subscripts. The first subscript selects a row from the array of pointers, the second selects a column from that row. This data structure is appropriate if rows of the matrix must be interchanged.

  A matrix made of vectors. A dynamic matrix can also be a vector of vectors because the vector class supports the subscript operator.

  Gaussian elimination. Gaussian elimination is a well-known method for solving systems of simultaneous linear equations in several unknowns. The algorithm is easily implemented using a dynamic matrix data structure and a set of functions that perform row operations on the equations.

18.5.2 Programming Style

- Avoid deeply nested control structures. If the nesting level is greater than three, the logic can be very difficult to follow. It is better to break up the code by defining a new function that performs the actions of the inner loops on the particular data items selected by the outer loops.
- The processing of data in a 2D array typically is done using two `for` loops. Other loop combinations can be used, but the double `for` loop has become almost standard.
- The programmer must take care when using nested loops to make sure that the outer and inner loops each process the appropriate dimension of a 2D array. It helps a great deal to use meaningful identifiers for the subscripts, such as `row` and `col`, rather than the simpler `j` and `k`.
- In a set of nested loops, the number of times the innermost loop body is executed is the product of the number of times each loop is repeated. It is important to make the innermost statements efficient, since they will occur many times.

- Use `#define` appropriately. As with 1D arrays, defining the array dimensions as constants at the top of a class makes later modification simple.
- Using `typedef` to name the parts of nested array types makes the code correspond to your concepts. This makes it easier to write, compile, and debug a program, because it allows you to declare parameters with the correct types and enables the compiler to help you write the appropriate number of subscripts on array references and arguments.
- Any legal C identifier may be used as the type name in a `typedef`. Some caution should be used, however, to avoid names that sound like variables or keywords. Types are classes of objects, so the type name should be appropriate for the general properties of the class.
- Usually a choice must be made as to whether to use an array of arrays or a matrix. If each of the data elements is independent of the others, then the matrix is the appropriate structure. If the data in a single row have a meaning as a group, then it is correct to use the array of arrays. When data in a column also have a meaning, either structure can be used.
- Continuing the preceding reasoning, always pass the proper parameter. When using a function to process a multidimensional array, pass only the part of the array that is needed, if possible. When a function works on all elements of the array or on an entire column, the parameter should be the entire array. However, if a function processes only one row, simply pass the single row, not the whole matrix. And if a function processes a single array element, pass just that element.

18.5.3 Sticky Points and Common Errors

- When using nested loops, it is fairly common to write a statement at the wrong nesting level. The action then will happen either too often or not often enough. Use curly brackets where necessary to make sure statements are in the correct loop.
- As always, all subscripts, even in a multidimensional array, start at 0, not 1.
- The programmer must be careful always to use subscripts within the bounds of the array. When using a 2D array, it is possible for a column subscript to be too large and still have the referenced element be within the memory area of the array. This is a serious error and usually harder to find than simply referencing outside the array's memory.
- Always beware of using subscripts in the wrong order. This may cause you to access outside the array's memory, or it might not. Using meaningful subscript names reduces the frequency of these errors.
- It also is important to use the proper number of subscripts. Because legally you can reference a single row in a matrix, most such references are not deemed incorrect by the compiler and may generate only a warning. However, rather than get the data item you desire, you will be using a memory address in your calculations.
- Use correct C++ syntax: `a[row][col]` rather than the syntax common in other languages: `a[row, col]`.

18.5.4 New and Revisited Vocabulary

These are the most important terms, concepts, keywords, and C library functions discussed in this chapter:

array `typedef`	vector of vectors	digital image
dimension	double subscripts	pixel
vector	column	smoothing
matrix	row	processing window
2D and 3D initializers	row-major order	2D transformation
multidimensional array	binary file mode	plane
array of arrays	`read()`	translate
array of strings	`write()`	rotate
dynamic 2D array	binary data	system of equations
dynamic array of arrays	nested loop	Gaussian elimination

18.6 Exercises

18.6.1 Self-Test Exercises

1. Assume that `mat` is an N by N matrix of `float`s. Declare two pointers, `begin` and `end`, and initialize them to the first and last slots of `mat`. Declare a third pointer, `off`, and initialize it as an off-board pointer.
2. Given the declarations and code below, show the output of the loops.

```
#define Z 3
char square[Z][Z];
char* p;
char* start = &square[0][0];
char* end = &square[Z-1][Z-1];

for (p = end; p >= start; p -= 2) *p = '1';
for (p = start; p < end; ++p) {
    ++p;
    *p = '0';
}

for (p = start; p <= end; ++p) cout <<*p <<" ";
cout <<endl;
```

3. Assume your program contains this code:

```
short int box[5][4];
int j, k;

for (k = 1; k <= 3; k++)
    for (j = 1; j <= 3; j++)
        box[j][k] = 2*j-k;
```

   (a) Draw a flowchart of this code.
   (b) Make a diagram of the array named `box` showing the array slots and subscripts. In the diagram, write the values that would be stored by the `for` loops. If no value is stored in a slot, write a question mark there.
   (c) What are the subscripts of the slot in `box` in which the value `5` is stored?
   (d) What is `sizeof box`?
   (e) Which array slot will have the lower memory address, `box[1][2]` or `box[2][1]`? Why?
   (f) What happens if you execute this line: `box[5][4] = 10;`?

4. Diagram the array created by the following declarations. In the drawing, identify the slot numbers and show the values stored in those slots by the loops that follow. Also, write a new declaration statement that has an initializer to set the contents of the array to that produced by these loops.

```
#define Y 4
#define Z 3
int a, b;
float lumber[Z][Y];

for (a = 0; a < Z; ++a)
    for (b = 0; b < Y; b++) {
        if (b > a)          lumber[a][b] = 0;
        else if (b == a)    lumber[a][b] = 2.5;
        else                lumber[a][b] = (a + b) / 2.0;
}
```

5. Write a function, `findMin()`, that has one parameter, `data`, that is a 10-by-10 matrix of `float`s. The function should scan the elements of the matrix to determine the element with the minimum value, keeping track of its position in the matrix. Return the value itself through the normal return mechanism and its row and column subscripts through pointer parameters.

6. Trace the execution of the following code using a table like this one:

m	
k	
Output	

Show the initial values of the loop variables. For each trip through a loop, show how these values change and what is displayed on the screen. In the table, draw a vertical line between items that correspond to each trip through the inner loop.

```
int k, m;
int data[4][3] = { {1,2,3}, {2,4,6}, {3,6,9}, {4,8,12} };

for (k = 0; k < 4; ++k) {
    for (m = k; m < 3; ++m)
        if (k != 1) cout <<" " <<data[m][k] ;
        else        cout <<" " <<data[k][m] ;
    cout << '\n' ;
}
```

7. Trace the execution of the following loop, showing the output produced. Use the array `data` and initial values declared in the previous exercise. Be careful to show the output formatting accurately.

```
for (k = 0; k < 3; ++k) {
    for (m = 0; m < 3; ++m) {
        if ((k+m) % 2 == 0) cout <<data[m][k];
        else                cout <<"   ";
    }
    cout <<'\n';
}
```

8. Happy Hacker wrote and printed the following code fragment in the middle of the night, when some of the keys on his keyboard did not work. The next day, he looked at it and saw that it was not quite right: All of the curly brackets were missing, there was no indentation, there were too many semicolons, and the loops were a little messed up. The code is supposed to print a neat matrix with three columns and seven lines. In addition, the column and row numbers are supposed to be printed along the top and left sides of the matrix. Please fix the code for Happy.

```
#define Z 3
#define W 7
int k, m;

cout <<"    " ;
for (k = 0; k < Z; ++k);
cout <<setw(2) << k;
cout <<"\n --------------\n" );
for (m = 0; m < Y; ++m;
cout <<setw(4) <<m;
for (k = 0; k < W; ++k;
cout <<setw(2) <<mat[m][k];
cout <<'\n';
```

18.6.2 Using Pencil and Paper

1. The pair of loops that follow initialize the contents of a 2D array, `data`. Draw a picture of this array, labeling the subscripts and showing the contents of the slots after the loops are done.

```
int j, k;
int data[4][3];

for (k = 0; k < 4; k++)
    for (j = 0; j < 3; j++ )
        data[k][j] = j * k + 1;
```

2. Trace the following loop and show the output exactly as it would appear on your screen:

```
#define Z 3
int a, b;
char square[Z][Z+1] = { "cat", "ode", "dog" };

for (a = 0; a < Z; ++a) {
    cout << a;
    for (b = 0; b < Z; b++) {
        if (a == 0) cout <<setw(2) <<square[a][b];
        else cout <<setw(2) <<square[b][a-1];
    }
    cout <<'\n';
}
```

3. Write a function, `countZeroRows()`, with one matrix parameter. It should count and return the number of rows in the matrix that have all zero values. Assume the matrix is a square array of size N-by-N integers, where $N \geq 1$ and is defined as a global constant.

4. Given the declarations and code that follow, show the output of the loops:

```
int k, m;
int data[4][3] = { {1,2,3}, {2,4,6}, {3,6,9}, {4,8,12} };

for (k = 0; k < 4; ++k) {
    for (m = 0; m < 3; ++m) {
        if ((k*m)%2==0) cout <<setw(2) <<data[k][m];
        else            cout <<"   ";
    }
    cout << '\n' ;
}
```

5. Given the declarations and code that follow, trace the code and show the output of the loops:

```
#define Z 3
char square[Z][Z+1] = { "---", "---", "---" };
int r, c;

for (r = 2; r >= 0; --r) square[r][r] = '0';
for (r = 0; r < 3; ++r) square[r][(r + 1)%Z] = '1';
for (r = 0; r < 3; ++r) {
    for (c = 0; c < 3; ++c) cout <<square[r][c];
    cout <<"\n";
}
```

6. For each item that follows, write two C++ statements that perform the task described. Write the first using subscripts, the second using pointers and dereference. Explain which is clearer and easier to use and why. Use these declarations:

```
const int rows = 5, cols = 3;
int k, m;
float A[cols];
float M[rows][cols];
float* N = new float[(rows * cols ];
float* p, * end;
```

(a) Make `p` refer to the first slot of `A`.

(b) Write a `for` loop to set all elements of `A` to one.

(c) Double the number in `M[4][2]` and store it back in `M[4][2]`.

(d) Set `end` as an off-board pointer for `M`.

(e) Make `p` refer to the first `float` in `N`.

18.6.3 Using the Computer

1. Changing a figure.

   Extend the crown program (point transformation) to permit the user to add a point to the figure between any two existing points or delete any point in the figure after it is originally constructed. Provide a menu so that the user can do several point transformations. When "quit" is selected, write out the new data set.

2. Sales Bonuses. A company with three stores, A, B, and C, has collected the total dollar amount of sales for each store for each month of the year. These numbers are stored in a file, `sales.in`, that has the information for January, then February, and so on. Within each month, the sales amount for store A is first, then store B, then store C.

   Write a function, `readFile()`, that will read in the data from the sales file and store the amounts in a matrix where each row represents a store and each column represents a month.

   The company pays each store a $1,000 bonus if its sales exceed $30,000 in any particular month. Next, write a function, `bonus()`, that has one parameter, a matrix of `float` values. It returns an integer to the caller. In this function, search through the `sales` matrix, starting at the first month for the first store. Test each value in the matrix to see if the sales amount qualifies for a bonus. If it does, print the store name and month corresponding to the row and column, as well as the sales amount. Count and return the number of bonuses given. If no amount qualifies for a bonus, return 0.

   Finally, write a main program that first reads in all the data, then prints the sales amounts and total sales for each store in a neat table, with all of store A's sales first, followed by those of store B, and finally those of store C. Last, call the `bonus()` function to determine the bonuses for the year. Print the total sales amount, the number of bonuses awarded, and the total amount of bonus money that the company will give out.

3. Array averages.

   An instructor teaches courseof all sizes, and needs to compute grade averages. He typically gives between 5 and 10 quizzes each term.

   (a) Define two classes: Course and Student. A Course will be a represented by a float (the overall quiz average), the number of quizzes for the term, and a vector of Students.

   A Student has a `name`, a vector of `scores`, and an`average`) , where each score is an integer between 0 and 200. The Student constructor should accept one parameter, the number of quizzes. It must read the name and quiz scores for one student, compute the average, and store use the data to initialize a new Student.

   The Course constructor should open an input file and, call the Student constructor to read students, and store all the Students in a vector.

   (b) Write a main program that will compute and print the grade averages. Call the Course constructor and the `computeAverage()` function described below. Finally, print each student's name, quiz grades, and quiz average in a neat table, followed by the overall average of all students on all quizzes.

(c) The function `Student::computeAverage()` should compute the grade average for one student and store it in the current Student.

(d) The function `Course::computeAverage()` should compute the grade average for the entire class and store it in the Course object.

4. Matrix transpose.

(a) Define a class `Matrix`. Its data members should be the number of rows, R, number of columns, C, and two-dimensional $R \times C$ array of `double`s.

(b) Define `SIZE` as a constant. The Matrix constructor should take R, C, and a file name as parameters and read in an $R \times C$ matrix of numbers from the specified file.

(c) In the Matrix class, write a function, `print()`, to print out a matrix in neat rows and columns. Assume that `C` is small enough that an entire row will fit on one printed line.

(d) In the Matrix class, write a function, `transpose()`, to perform a matrix transpose operation, as follows:

i. If $R \neq C$, abort and print an error comment. You can only transpose square matrices.

ii. To transpose a matrix, swap all pairs of diagonally opposite elements; that is, swap the value in $M[I][J]$ with the value in $M[J][I]$ for all pairs of $I < SIZE$ and $J < SIZE$ The transposed matrix will replace the original matrix in the same memory locations.

iii. Try to accomplish the operation with a minimal number of swaps. Warning: This cannot be done by the usual double loop that executes the body $I \times J$ times.

When the operation is over, the original matrix should have been overwritten by its transpose. Elements with the same row and column subscripts will not change. As an example, the 3-by-3 matrix on the left is transposed into the matrix on the right:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

(e) Write a main program to instantiate your Matrix class and test your functions. Print the matrix before and after you transpose it.

5. Matrix multiplication.

(a) Define a class `Matrix` as in parts (a) through (c) of the previous problem. Add two more methods to this class:

- A constructor with two integer parameters M and N (no file name). Allocate but do not initialize space for a 2D $M \times N$ array of doubles (the answer).
- A function, `multiply()`, to perform a matrix multiplication operation, as follows:

i. There will be two matrix parameters, `A` and `B` of the sizes just mentioned. The implied parameter will be `C`, the result matrix.

ii. The result elements of `C` are defined as:

$$C_{ij} = \sum_{k=0}^{P-1} A_{ik} B_{kj}$$

where i is in the range $0 \ldots M-1$ and j is in the range $0 \ldots N-1$. This essentially is the dot product operation shown earlier in the chapter. An example calculation would be

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \times \quad \begin{bmatrix} 4 & 7 \\ 5 & 8 \\ 6 & 9 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 32 & 50 \\ 77 & 122 \end{bmatrix}$$

(b) This problem will work with three matrices of sizes $M \times P$, $P \times N$, and $M \times N$. Write a main program that will instantiate matrices A and B from user-specified files using #defined constants M, P, and N. Create matrix C for the results. Perform matrix multiplication on them, write the results to a file. Display the input matrices on the screen, then display the result matrix.

6. Bingo.

   In the game Bingo, a playing card has five columns of numbers with the headings B, I, N, G, and O. There are five numbers in each column, arranged randomly, except that all numbers in the B column are between 1 and 15, all numbers in the I column are between 16 and 30, N has 31–45, G has 46–60, and O has 61–75. The word *FREE* is in the middle slot in the N column instead of a number. Your job is to create and print a Bingo card containing random entries in the proper ranges. Define a type, `card`, to be a 5-by-5 array of integers. Generate 25 random numbers and use them to initialize a card, `C`, as follows:

   (a) Declare an array of length 15 named `ar`. Repeat the next step five times, with $x = 0, \ldots, 4$.

   (b) Store the number $(1 + 15 \times x)$ in `ar[0]`, then fill the remaining 14 slots with the next 14 integers. Repeat the next step five times, once for each of the five rows on the Bingo card.

   (c) Randomly select a number, `r`, between 1 and 15. If the number in `ar[r]` is not 0, copy it into column x of the current row of the Bingo card and store 0 in `ar[r]` to indicate that the number already has been used. If `ar[r]` already is 0, generate another `r` and keep trying until you find a nonzero item. This trial-and-error method works adequately because the number of nonzero numbers left in the array always is much larger than the number of zero items. You would have to be very unlucky to hit several used items one after another.

   (d) When all 25 random numbers have been filled in, store a code that means FREE in the slot in the middle of the card. Print the resulting Bingo card as follows, with spaces between the numbers on each row. You may use lines of minus signs for the horizontal bars and omit the vertical bars

B	I	N	G	O
3	17	32	49	68
12	23	38	47	61
11	18	FREE	60	70
2	29	36	50	72
9	22	41	57	64

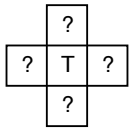
7. Crowded items.

   Define R and C as global constants of reasonable size. Define a class, `matrix`, with R rows and C columns of type `unsigned char` elements.

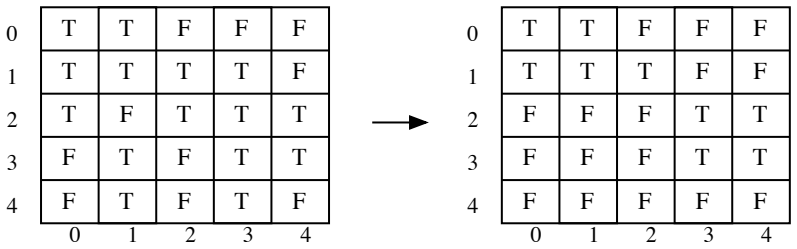
   - In the Matrix class, write a function, `findCrowdedElements()`, with a`matrix` parameter for output. The implied parameter will be the input to this process.

     The values in the input matrix will be either 1 or 0, representing `true` and `false`. The function will set each value of the output matrix to 1 or 0 according to whether the corresponding element in the input matrix is "crowded."

   - Crowded is defined thus: First, an element's value must be `true` for it to be crowded. Second, there are potentially four meaningful neighbors of the test element (up, down, left, right):



     – If three or more of the current element's four neighbors have a value of `true`, then the current element is crowded.
     – If the current element is on a border, but not in a corner, then only two neighbors have to be `true` to be crowded.
     – Corner elements need only one of their two neighbors to be `true` to be crowded.



- Write a main program. Input a matrix of values from a file, compute the "crowded" matrix, and write it out to another file. Assume the input file contains data for a matrix of the proper size. In addition to saving the result in a file, display output similar to that above, using 'F' and 'T'.

8. Bar graph.

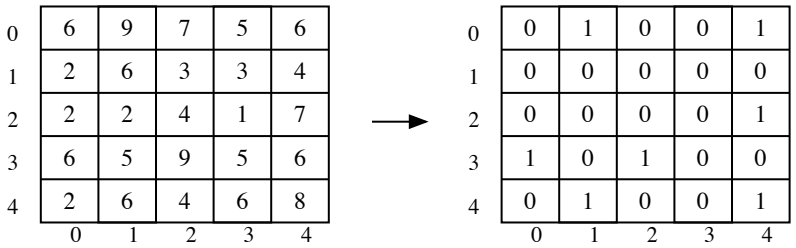
   A professor likes to get a visual summary of class averages before assigning grades. Her class averages are stored in a file, `avg.in`. You are to read the data in this file and from it make a bar graph, as follows:

   (a) Your graph should have 21 lines of information, which are printed with double spacing.
   (b) Down the left margin will be a list of 5-point score ranges: $0 \ldots 4$, $5 \ldots 9$, up to $100 \ldots 104$. Following this, on each line, should be a single digit (defined later) for each student whose average falls within that range. There may be scores over 100 because this professor sometimes curves the grades.
   (c) Any negative scores should be treated as if they were 0. Any scores over 104 should be handled as if they were 104.

   Represent this graph as a matrix of characters with 21 rows and 35 columns initialized to 0 (null characters). You may assume that the number of scores put into any single row will not exceed 35. Also, have an array of integer counters corresponding to the rows. As each average is read, determine which row, r, it belongs in by using integer division. Also, isolate the last digit, d, of each score; that is, for 98, $d = 8$ and for 103, $d = 3$. To record the score, convert digit d to the ASCII code for the digit by adding it to `'0'`. Store the resulting character in the next available column of row r and update the counter for row r. When the end of the file is reached, print the matrix on the screen using a two-column field width for each score, and a blank line between each row.

9. Local maxima in a 2D array. Assume that you have a data file, `max.in`, containing 1-byte integers in binary format. The file has 50 numbers, representing a matrix of values that has five rows containing 10 numbers each. The numbers are stored in a row-major order. Write a program that will do the following:

   (a) Read the data values into a 2D array on which further processing can be performed.
   (b) Call the `localMax()` function described next and print the values of both the input and output matrices.
   (c) Write a function, `localMax()`, that takes two 2D arrays of integers as parameters. The first parameter is the input matrix; the second is an output matrix. Set the value at a particular location in the output matrix to 1 if the corresponding input value is a "local maximum"; that is, the value is larger than the four neighboring values to its left, right, above, and below. Set the output value to 0 if the corresponding position cannot be labeled as a local maximum. Matrix positions on a corner or an edge will have only two or three neighboring points, which must have lesser values for the given location to be a maximum. As an example, the input matrix on the left would generate the matrix on the right:



10. Local maxima and minima.

    In solving this problem, refer to the smoothing program in Figure 18.17 and generate a new program to do the following:

    (a) Open and read a user-specified ASCII file that contains a 10×10 matrix of integer values in the range 0–9 and store them in a 2D array.

    (b) Then generate a corresponding 10×10 array of character values, each of which is determined by the following:

        i. Generate a `'+'` if the value in the original array is a local maximum, where *local maximum* is defined as the highest value in a 3×3 area centered on that position in the matrix. For values on the border, examine only the portion of this area containing actual values.

        ii. Generate a `'-'` if the value in the original array is a local minimum, where *local minimum* is defined as the lowest value in the same 3×3.

        iii. Generate a `'*'` if the value in the original array is a saddle point, where *saddle point* is defined as a value higher than the neighboring values in the same column but lower than the neighboring values in the same row. Saddle points cannot occur on a border.

        iv. Finally, if the value is not classified as any of the preceding three, simply convert the integer value of 0–9 into a character `'0'`–`'9'` by adding the number to the character value `'0'`.

        v. As an example, using a 5×5 array,

$$\begin{bmatrix} 0 & 2 & 3 & 2 & 4 \\ 6 & 5 & 6 & 0 & 3 \\ 7 & 4 & 1 & 2 & 3 \\ 5 & 8 & 7 & 4 & 8 \\ 1 & 2 & 9 & 3 & 2 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} - & 2 & 3 & 2 & + \\ 6 & * & + & - & 3 \\ 7 & 4 & 1 & 2 & 3 \\ 5 & 8 & 7 & * & + \\ - & 2 & + & 3 & - \end{bmatrix}$$

    (c) Display both the numeric matrix and the character matrix on the screen.

11. Computing the wind chill.

    The wind-chill index is a measure of the increase in heat loss by convection from a body at a specific temperature. Consider the wind-chill table below.

		Actual Air Temperature (F)									
		−10	**−5**	**0**	**5**	**10**	**15**	**20**	**25**	**30**	**35**
Wind Speed (mph)	**5**	5	5	5	5	4	4	4	3	3	2
	10	24	22	22	20	19	18	17	16	14	13
	15	35	35	33	30	28	26	26	24	19	19
	20	42	41	40	37	34	32	29	29	27	23
	25	48	47	45	42	39	37	35	32	30	28
	30	53	51	49	46	43	41	38	36	32	30
	35	57	55	52	48	45	42	40	38	34	32

In `main()`, instantiate Chill (below) then enter a query loop that will call a function, `getChill()`, to perform the main process as many times as the user wishes. Print instructions and prompt the user to enter real values for the actual temperature, t, and the wind speed, s.

Define a class Chill to contain this table and the related functions. The constructor must read the wind-chill table from a text file, `chill.in`, and store it in a two-dimensional array (the column and row headings are not in the data file).

The `getChill()` function should have the prototype `void chill( double t, double s )`, Within the `chill()` function do the following:

(a) Validate s and t. Print an error comment and return if s is not in the range $2.5 \leq s < 37.5$ mph or t is not in the range $-12.5^\circ \leq t < 37.5^\circ$ Fahrenheit. Use these values to look up the effective temperature decrease in the wind chill table as follows.

(b) Calculate the row subscript. For any input wind speed, s, we want to use the closest speed that is in the table. If s is a multiple of 5 mph, the table entry is exactly right. If not, we must calculate the closest speed that is in the table. For example, if the speed is between 7.5 and 12.5 mph, we want to use the row for 10 mph, which is row 1. To calculate the correct row number, subtract 2.5 from s, giving a result less than 35.0. Divide this by 5.0 and store the result in an integer variable. Since fractional parts are discarded when you store a `float` in an integer, the integer will be between 0 and 6 and usable as a valid row number.

(c) Calculate the column subscript. As for the wind speed, use the temperature in the table closest to the input temperature, t. For example, if the temperature is between -2.5° and $+2.5^\circ$, use the values for 0°, which are in column 2. To calculate the correct column number, add 12.5 to t, giving a positive number less than 50.0. Divide this by 5.0 and store the result in an integer variable. This will give an integer between 0 and 9 that can be used as a valid column number.

(d) Use the computed row and column subscripts to access the wind chill table and read the decrease in temperature. To compute the effective temperature, subtract this decreased amount from the actual air temperature. Echo the inputs and print the effective temperature.

12. Matrix averages.
Write a program that will input a matrix of numbers and output the row and column averages. More specifically,

(a) Read a file containing two integers on the first line: a number of rows and a number of columns. Then allocate memory for a matrix of `float`s with those dimensions. Read the rest of the data from the file, row by row, into the matrix.

(b) Allocate an array of `float`s whose length equals the number of rows and store the average of each row in the corresponding array slot.

(c) Allocate another array with one slot per column and store the average of each column in the corresponding array slot.

(d) Print the matrix in spreadsheet form, with row numbers on the left, row averages on the right, column numbers on top, and column averages at the bottom.

Use arrays, not vectors.