Chapter 2: Issues and Overview

2.1 Why did C need a ++?

For application modeling. Shakespeare once explained it for houses, but it should be true of programs as well:

> **When we mean to build, we first survey the plot then draw the model.**
> . . . Shakespeare, King Henry IV.

2.1.1 Design Goals for C

C was designed to write Unix. C is sometimes called a "low level" language. It was created by Dennis Ritchie so that he and Kenneth Thompson could write a new operating system that they named Unix. The new language was designed to control the machine hardware (clock, registers, memory, devices) and implement input and output conversion. Thus, it was essential for C to be able to work efficiently and easily at a low level.

Ritchie and Thompson worked with small, slow machines, so they put great emphasis on creating an simple language that could be easily compiled into efficient object code. There is a direct and transparent relationship between C source code and the machine code produced by a C compiler.

Because C is a simple language, a C compiler can be much simpler than a compiler for C++ or Java. As a result, a good C compiler produces simple error comments tied to specific lines of code. Compilers for full-featured modern languages such as C++ and Java are the opposite: error comments can be hopelessly wordy and also vague. Often, they do not correctly pinpoint the erroneous line.

Ritchie never imagined that his language would leave their lab and become a dominant force in the world and the ancestor of three powerful modern languages, C++, C#, and Java. Thus, he did not worry about readability, portability, and reusability. Because of that, readability is only achieved in C by using self-discipline and adhering to strict rules of style. However, because of the clean design, C became the most portable and reusable language of its time.

In 1978, Brian Kernighan and Dennis Ritchie published "The C Programming Language", which served as the only language specification for eleven years. During that time, C and Unix became popular and widespread, and different implementations had subtle and troublesome differences. The ANSI C standard (1989) addressed this by providing a clear definition of the syntax and the meaning of C. The result was a low-level language that provides unlimited opportunity for expressing algorithms and excellent support for modular program construction. However, it provides little or no support for expressing higher-level abstractions. We can write many different efficient programs for implementing a queue in C, but we cannot express the abstraction "queue" in a clear, simple, coherent manner.

2.1.2 C++ Extends C.

C++ is an extension and adaptation of C. The designer, Bjarne. Stroustrup, originally implemented C++ as a set of macros that were translated by a preprocessor into ordinary C code. His intent was to retain efficiency and transparency and simultaneously improve the ability of the language to model abstractions. The full C language remains as a subset of C++ in the sense that anything that was legal in C is still legal in C++ (although some things have a slightly different meaning). In addition, many things that were considered "errors" in C are now legal and meaningful in C++.

Readability. C++ was no more readable than C, because C was retained as the basic vehicle for coding in C++ and is a proper subset of C++ . However, an application program, as a whole, may be much more readable in C++ than in C because of the new support for application modeling.

Portability. A portable program can be "brought up" on different kinds of computers and produce uniform results across platforms. By definition, if a language is fully portable, it does not exploit the special features of any hardware platform or operating system. It cannot rely on any particular bit-level representation of any object, operation, or device; therefore, it cannot manipulate such things. A compromise between portability and flexibility is important for real systems.

A program in C or C++ can be very portable if the programmer designs it with portability in mind and follows strict guidelines about segregating sections of code that are not portable. Skillful use of the preprocessor and conditional compilation can compensate for differences in hardware and in the system environment. However, programs written by naive programmers are usually not portable because C's most basic type, `int`, is partially undefined. Programs written for the 4-byte integer model often malfunction when compiled under 2-byte compilers and vice versa. C++ does nothing to improve this situation.

Reusability. Code that is filled with details about a particular application is not very reusable. In C, the typedef and #define commands do provide a little bit of support for creating generic code that can be tailored to a particular situation as a last step before compilation. The C libraries even include two generic functions (`quicksort()` and `bsearch()` that can be used to process an array of any base type. C++ provides much broader support for this technique and provides new type definition and type conversion facilities that make generic code easier to write.

Teamwork potential. C++ supports highly modular design and implementation and reusable components. This is ideal for team projects. The most skilled members of the group can design the project and implement any non-routine portions. Lesser-skilled programmers can implement the routine modules using the expert's classes, classes from the standard template library, and proprietary class libraries. All these people can work simultaneously, guided by defined class interfaces, to produce a complete application.

2.1.3 Modeling.

The problems of modeling are the same in C and C++. In both cases the questions are, what data objects do you need to define, how should each object work, and how do they relate to each other? A good C programmer would put the code for each type of object or activity in a different file and could use type modifiers `extern` and `static` to control visibility. A poor C programmer, however, would throw it all into one file, producing an unreadable and incomprehensible mess. Skill and style made a huge difference. In contrast, C++ provides classes for modeling objects and several ways to declare or define the relationship of one class to others.

What is a model? A *model of an object* is a list of the relevant facts about that object in some language. A *low level* model is a description of a particular implementation of the object, that specifies the number of parts in the object, the type of each part, and the position of each part in relation to other parts. C supports only low level models.

C++ also supports *high-level* or *abstract* models, which specify the functional properties of an object without specifying a particular representation. This high-level model must be backed up by specific low-level definitions for each abstraction before a program can be translated. However, depending on the translator used and the low-level definitions supplied, the actual number and arrangement of bytes of storage that will be used to represent the object may vary from translator to translator.

A high level model of a *process* or *function* specifies the pre- and post-conditions without specifying exactly how to get from one to the other. A low level model of a *function* is a sequence of program definitions, declarations, and statements that can be performed on objects from specific data types. This sequence must start with objects that meet the specified pre-conditions and end by achieving the post-conditions.

High level process models are not supported by the C language but do form an important element of project documentation. In contrast, C++ provides class hierarchies and virtual functions which allow the programmer to build high-level models of functionality, and later implement them.

Explicit vs. Implicit Representation. Information expressed explicitly in a program may be used by the language translator. For example, the type qualifier `const` is used liberally in well-written C++ applications.

This permits the compiler to verify that the variable is never modified directly and is never passed to a function that might modify it.

A language that permits explicit communication of information must have a translator that can identify, store, organize, and utilize that information. For example, if a language permits programmers to define types and relationships among types, the translator needs to implement type tables (where type descriptions are stored), new allocation methods that use these programmer-defined descriptions, and more elaborate rules for type checking, type conversions, and type errors. This is one of the reasons why C++ translators are bigger and slower than C translators. The greater the variety and sophistication of the information that is declared, the more effort it is to translate it into low-level code that carries out the intent of the declarations.

Semantic Intent A data object (variable, record, array, etc.) in a program has some intended meaning that can be known only if the programmer communicates or declares it. A programmer can try to choose a meaningful name and can supplement the code by adding comments that explain the intent, but those mechanisms communicate only to humans, not to compilers. The primary mechanism for expressing intent in most languages is the data type of an object. Some languages support more explicit declaration of intent than others. For example, C uses type `int` to represent many kinds of objects with very different semantics (numbers, truth values, characters, and bit masks). C++ is more discriminating; truth values are represented by a semantically distinct type, `bool`, and programmer-defined enumerations also create distinct types.

A program has *semantic validity* if it faithfully carries out the programmer's semantic intent. A language is badly designed to the extent that it lets the programmer write code that has no reasonable valid interpretation. A well-designed language will identify such code as an error. A strong type checking mechanism can help a programmer write a semantically valid (meaningful) program. Before applying a function to a data object, both C and C++ translators test whether the call is meaningful, that is, the function is defined for objects of the given type. An attempt to apply a function to a data object of the wrong type is identified as a semantic error at compile time.

However, the type system and the rules for translating function calls are much more complex in C++ than in C for the reasons discussed in Section 2.3. For these reasons and others, achieving the first error-free compile is more difficult in C++, but the compiled code is more likely to run correctly.

2.2 Object Oriented Principles.

Classes. The term "object-oriented" has become popular, and "object-oriented" analysis, design, and implementation has been put forward as a solution to several problems that plague the software industry. OO analysis is a set of formal methods for analyzing and structuring an application; the result of an OO analysis is an OO design. OO programs are built out of a collection of modules, often called *classes* that contain both functions and data. The most fundamental design goal in these classes is that a class should take care of itself.

The way a language is used is more important in OO design than which language is used. C++ was designed to support OO programming; it is a convenient and powerful vehicle for implementing an OO design. However, with somewhat more effort, that same OO design could also be implemented in C.[1] Similarly, a non-OO program can be written in C++.

Three principles central to object-oriented programming are locality, coherent representation, and generic functions.

Locality and Encapsulation. The effects of an action or a declaration can be global (affecting all parts of a program) or local (affecting only nearby parts). The further the effects of an action reach in time (elapsed during execution) or in space (measured in pages of code), the more complex and harder it is to debug a program. The further an action has influence, the harder it is to remember relevant details, and the more subtle errors seem to creep into the code.

A well-designed language supports and encourages locality. All modern languages permit functions to have local variables and minimize the need for global variables. C goes farther than many language by supporting static local variables that have the lifetime of a global object but only local visibility. OO languages go further

[1] The insertion sort code example is an OO design implemented in C that illustrates locality, coherent representation, and reusable generic code.

still by introducing "private" fields in structures (class objects) that cannot be seen or changed by functions outside the class. We say that the private members are *encapsulated* within the class.

Coherent vs. Diffuse Representation. A representation is *coherent* if an external entity (object, idea, or process) is represented by a single symbol in the program (a name or a pointer) so that it may be referenced and manipulated as a unit. A representation is *diffuse* if various parts of the representation are known by different names, and no one name or symbol applies to the whole.

Coherence is the most important way in which object-oriented languages differ from older languages. In Pascal, for example, a programmer can declare a new data type and write a set of functions to operate on objects of that type. Taken together, the data objects and functions implement the programmer's model. However, Pascal does not provide a way to group the data and functions into a coherent package, or declare that they form a meaningful module. C is a little better in this respect because separately-compiled code modules allow the programmer to group related things together and keep both functions and data objects private within the module. In an OO language, however, the class mechanisms do this and more, and provide a convenient syntax for declaring classes and class relationships.

Generic code. A big leap forward in representational power was the introduction of generic code. A generic function is one like "+" whose meaning depends on the type of the operands. Floating-point "+" and integer "+" carry out the same conceptual operation on two different representations of numbers. If we wish to define the same operation (such as "print") on five data types, C forces us to introduce five different function names. C++ lets us use one name to refer to several methods which, taken together, comprise a function. (In C++ terminology, a single name is "overloaded" by giving it several meanings.) The translator decides which definition to use for each function call based on the types of the arguments in that call. This makes it much easier to build libraries of functions that can be used and reused in a variety of situations.

OO Drawbacks. Unfortunately, a language with OO capabilities is complex. The OO extensions in C++ make it considerably more complicated than C. It is a massive and complex language. To become an "expert" requires a much higher level of understanding in C++ than in C, and C is difficult compared to Pascal or FORTRAN. The innate complexity of the language is reflected in its translators; C++ compilers are slow and give very confusing error comments.

The ease with which one can write legal but meaningless code is a hallmark characteristic of C. The C programmer can write all sorts of senseless but legal things (such as a<b<c). C++ has a better-developed system of types and type checking, which improves the situation somewhat. However C++ also provides powerful tools, such as the ability to add new definitions to old operators, that can easily be overused or misused. A good C++ programmer designs and writes code in a strictly disciplined style, following design guidelines that have been evolved from experience over the years. Learning these guidelines and how to apply them is more important than learning the syntax of C++, if the goal is to produce high-quality, debugged, programs.

2.3 Important Differences

- **Comments.** Comments can begin with // and end with newline. Please use only this kind of comment to write C++ code. Then the old kind can be used to /* comment out */ sections of text during debugging.
- **Executable declarations.** Declarations can be mixed in with the code. This makes it possible to print greeting messages before processing the declarations. Why is this useful? Because C++ declarations can trigger file processing and dynamic memory allocation and it is VERY helpful to precede each major declaration with a message that will let the programmer track the progress of the program.

  When you put a declaration in a loop, the object will be allocated, initialized, and deallocated every time around the loop. This is unnecessary and inefficient for most variables but it can be useful when you need a strictly temporary object of a class type.

  Declarations must not be written inside one `case` of a `switch` statement unless you open a new block (using curly braces) surrounding the code for the case.

- **Type identity.** In C, the type system is based on the way values are *represented*, not on what they *mean*. For example, pointers, integers, truth values, and characters are all represented by bitstrings of various lengths. Because they are represented identically, an `int` variable can be used where a `char` value is wanted, and vice versa. Truth values and integers are even more closely associated: there is no distinction at all. Because of this, one of the most powerful tools for ensuring correctness is compromised, and expressions that should cause type errors are accepted. (Example: `k < m < n`.)

  In C++, as in all modern languages, type identity is based on `meaning`, not representation. Thus, truth values and integers form distinct types. To allow backwards compatibility, automatic type coercion rules have been added. However, new programs should be written in ways that do not depend on the old features of C or the presence of automatic type conversions.

- **Type bool.** In standard C++ , type `bool`, whose values are named `false` and `true`, is not the same type as type `int`. It is defined as a separate type along with type conversions between `bool` and `int`. The conversions will be used automatically by the compiler when a programmer uses type `int` in a context that calls for type `bool`. However, good style demands that a C++ programmer use type `bool`, not `int` and the constants true (not 1) and false (not 0), for true/false situations.

- **Enumerated types.** Enumerated type declarations were one of the last additions to the C language prior to standardization. In older versions of the language, `#define` statements were used to introduce symbolic codes, and a program might start with dozens of `#define` statements. They were tedious to read and write and gave no clue about which symbols might be related to each other. Enumerations were added to the language to provide a concise way to define sets of related symbols. For example, a program might contain error codes and category codes, all used to classify the input. Using `#define`, there is no good way to distinguish one kind of code from the other. By using two enumerations, you can give a name to each set and easily show which codes belong to it.

  In C, enumeration symbols are implemented as integers, not as a distinct, identifiable type. Because of this, the compiler does not generate type errors when they are used interchangeably, and many C programmers make no distinction. In contrast, in C++, an enumeration forms s distinct type that is not the same as type `int` or any other enumeration. Complilers *will* generate error and warning comments when they are used inappropriately.

- **Type conversions.** C provides type cast operators that work from any numeric type (`double, float, int, unsigned, char`) to any other numeric type, and from a pointer of any base type to a pointer of any other base type. These casts are used automatically whenever necessary:

  - When an argument type fails to match a parameter type.
  - When the expression in a return statement does not match the declared return type.
  - When the types of the left and right sides of an assignment do not match.
  - When two operands of a binary operator are different numeric types.

  These rules are the same in C++, but, in addition, the programmer can define new type casts and conversions for new classes. These operations can be applied explicitly (like a type cast) and will also be used by the compiler, as described above, to coerce types of arguments, return values, assignments, and operands.

- **Assignment.** First, the operator = is predefined for all types except arrays. Second, the behavior of the assignment operator has changed: the C++ version returns the address of the location that received the stored value. (The C version returns the value that was stored.) This makes no difference in normal usage. However, the result of some complex, nested assignments might be different.

- **Operators can be extended to work with new types.** For example, the numeric operators `+, *, -, and /` are predefined for pairs of integers, pairs of floating point numbers, and mixed pairs. Now suppose you have defined a new numeric type, Complex, and wish to define arithmetic on Complex numbers. You can define an additional method for `+` to add pairs of Complex numbers, and you can define conversion functions that convert Complex numbers to other numeric types.

- **Reference parameters and return values.** C supports only call-by-value for simple objects and structures, and only call-by-reference for arrays. When a pointer is passed as an argument in C, we can refer to it as "call-by-pointer", which is an abbreviation for "call by passing a pointer by value". This permits the function to change a value in the caller's territory. However, it is not the same as call-by-reference, which is supported by C++ in addition to all the parameter passing mechanisms in C.

  In call-by-reference, an address (not a pointer) is passed to the function and the parameter name becomes an alias for the caller's variable. Like a pointer parameter, a reference parameter permits a function to change the value stored in the caller's variable. Unlike a pointer parameter, the reference parameter cannot be made to refer to a different location. Also, unlike a pointer, a * does not need to be used when the variable is used within the function. General restrictions on the use of references are:

  - A reference parameter can be passed by value as an argument to another function, but it cannot be passed by reference.
  - Sometimes reference parameters are used to avoid the time and space necessary to `copy` an entire argument value. In that case, if it is undesirable for the function to be able to change the caller's variable, the parameter should be declared to be a `const &` type.
  - Arrays are automatically passed by reference in both C and C++. You must not (and do not need to) write the ampersand in the parameter declaration.
  - You can't set a pointer variable to point at a reference.
  - You can't create a reference to a part of a bitfield structure.

- **Protection.** Structures, as in C are still available in C++, but a C++ `struct` may have function members as well as data members. In addition, C++ provides *classes*, which are structures with added encapsulation capabilities. Learning to use encapsulation wisely will be a major part of this course.

- **Function methods.** Any function can have more than one definition, as long as the list of parameter types, (the *signature*) of every definition is different. The individual definitions are called *methods* of the function. In the common terminology, such a function is called *overloaded*. I prefer not to use this term so broadly. In this course, I will distinguish among *extending*, *overriding*, and *overloading* a function.

- **I/O.** C and C++ I/O are completely different, but a program can use both kinds of I/O statements in almost any mixture. The C and C++ I/O systems both have advantages and disadvantages. For example, simple output is easier in C++ but output in columns is easier in C. Since one purpose of this course is to learn C++, please use only C++ output in the C++ programs you write for this course.

- **Using the C++ libraries.** To use one of the standard libraries in C, we write an `#include` statement of the form `#include <stdio.h>` or `#include <math.h>`. A statement of the same form can be used in C++ . For example, the standard input/output library can be used by writing `#include <iostream.h>`. However, this form of the include statement is "old fashioned", and should not be used in new programs. Instead, you should write two lines that do the same thing:

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

  The new kind of include statement still tells the preprocessor to include the headers for the iostream library, but it does not give the precise name of the file that contains those headers. It is left to the compiler to map the abstract name "iostream" onto whatever local file actually contains those headers. The second line brings the function names from the `iostream` library into your own context. Without this declaration, you would have to write `iostream::` in front of every function or constant name defined by the library.

2.4 Generic Insertion Sort

This is an object-oriented program written in a non-OO language, C. It embodies a variety of OO design, coding, and style principles, and also a few old C techniques. Notes follow the code. Use this example as a general guide for doing your own work. The most important themes are:

- Highly modular code with a streamlined main function.
- Reusable generic code instantiated using typedef and #define.
- The use of pointers, cursors, and sentinels to process an array.
- OO concept: The data pack, along with its setup, input, and print functions forms a class (or would, if written in C++). A data pack is an array packaged together with the information needed to use and manage it: the number of slots in the array and the number of slots that are currently filled with valid data. This implementation of a data pack includes a sort function, as well as the basic functions for construction, input and output from the pack. An addition that could be useful is a status code that indicates whether the data is sorted, and if so, in what order.
- OO design principle: A class should take care of itself. The functions that belong to a class operate on the class's data objects. No other functions should manipulate these objects.

A type declaration and the main program are given first, followed by detailed notes, keyed to the line numbers in the code. A call chart for the program is given in Figure 2.1. In the chart, white boxes surround functions defined in this application, light gray boxes denote functions in the standard libraries and dark gray denotes the tools library.

2.4.1 Main Program

1. A program that is built in modules has a pair of files for each module: a header file and a code file. The code file starts with a command to include the corresponding header file. The main program starts with commands to include one or more module headers. In this case, `main()` uses the Datapack module, so we include the header for that module.
2. The main program consists entirely of calls on the DataPack functions. All the work on the values in the `dataPack` is done by those modules.
3. The `setup()` function constructs a fully formed, legal, empty, DataPack. Then `readData()` fills it with data, `sortData()` sorts the data, and `printData()` shows us the data before and after sorting.

```
1  /*  ------------------------------------------------------------------------
2  //  Main program for Sorting DataPacks.                              2_main.c
3  //  Created by Alice Fischer on Mon Dec 22 2003.
4  //  ------------------------------------------------------------------------*/
5  #include "pack.h"
6
7  /*  ------------------------------------------------------------------------*/
8  int main( void )
9  {
10     DataPack theData;
11     banner();
12     say( "Construct pack, read data" );     setup( &theData );
13     say( "\n%d data items read:",           theData.n );
14     say( "\nUnsorted data:" );              printData( &theData, stdout );
15     say( "\nBeginning to sort." );          sortData( &theData );
16     say( "\nSorted results:" );             printData( &theData, stdout );
17     bye();
18     return 0;
19 }
```

2.4.2 The DataPack Header File

1. By changing the statements on lines 26. . . 28, we can change the program to work with any basic C data type instead of with float. We need only to change the formats and define `BT` (which stands for base type) to be the type of data that will be stored in the array and sorted. For example, these lines would be used for type int:

```
#define IN_FMT "%i"
#define OUT_FMT "%i\n"
typedef int BT;
```

2. Line 24 includes definitions from the local library, `tools`, that we will be using throughout the term. Note the use of quotes, not angle brackets. Note that we include the header file, not the code file, because we are doing a multi-module compile and separate link operation.

```
#include "tools.h"
```

3. The `dataPack` type makes a coherent grouping of *all* the parts that are needed to manage an array of data. In C++, it would be a class, not a structure.
4. The code of `main()` (lines 11...17) is simply an outline of the processing steps with some user feedback added. All work is delegated to functions. A call chart (Figure 2.1) shows the overall dynamic structure of a program and is an important form of documentation. The arrows indicate which functions are called by each other function. Frequently, the standard I/O functions are omitted from such charts.

```
20  /*  ------------------------------------------------------------------------
21  //  Header file for all DataPack programs.                            pack.h
22  //  Created by Alice Fischer on Mon Dec 22 2003.
23  */
24  #include "tools.h"
25  /* ---------------------------------- Instantiations for generic parts  */
26  #define IN_FMT "%g"
27  #define OUT_FMT "%.7g\n"
28  typedef float BT ;
29  #define LENGTH 20
30
31  /* ------------------------------------------- Generic type definition  */
32  typedef struct {
33      int n;           /* Allocation length. */
34      int max;         /* Data length. */
35      BT* store;       /* Dynamic data array. */
36  } DataPack;
37
38  /* ------------------------------------------------------- Prototypes  */
39  void setup( DataPack* pData );
40  void printData( DataPack* pData, stream outstream );
41  void sortData( DataPack* pData );
```

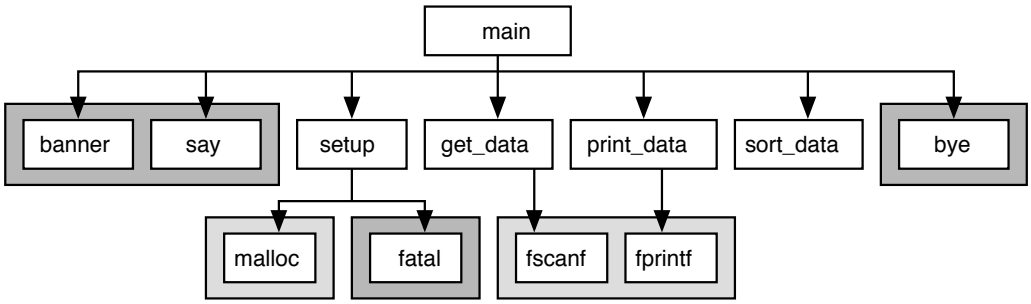

Figure 2.1: Call hierarchy chart for insertion sort.

2.4.3 The Data Pack Functions

Constructing the data structure. The `setup()` function (lines 52...61) is called from line 12 of `main()`.

1. Every data structure should have a function to build and initialize its parts. In C++, these are called constructors. At the end of construction, the object should be completely formed and internally consistent.
2. In C, this function must be called explicitly, while in a C++ program, the appropriate constructor function is called automatically whenever an object is allocated.
3. We allocate necessary storage on line 57 and check for successful allocation on the next two lines. In C++, this kind of error checking is not necessary. Note the call on fatal() (line 59) to handle an unrecoverable error.
4. Lines 55...57 initialize all the fields of the object so that they are consistent with each other and with the amount of storage allocated.

```
42  // ------------------------------------------------------------------------
43  //  Code file for all DataPack programs.                          pack.c
44  //  Created by Alice Fischer on Mon Dec 22 2003.
45
46  #include "pack.h"
47  static void readData( DataPack* pData );   // A private function
48
49  //-----------------------------------------------------------------
50  // Allocate memory for data.  Errors are fatal.
51
52  void
53  setup( DataPack* pData )
54  {
55      pData->n = 0;                              // Currently empty.
56      pData->max = LENGTH;                       // Space for LENGTH items.
57      pData->store = malloc(LENGTH*sizeof(BT));
58      if (pData->store == NULL)
59          fatal( " Error: not enough memory for %i BTs\n", LENGTH );
60      readData( pData );
61  }
62
63  //-------------------------------------------------------------------------
64  // Use sequential access pattern to store data from infile into data pack.
65
66  static void
67  readData( DataPack* pData )
68  {
69      char filename[80];              // For name of input file
70      stream infile;
71      BT* cursor, * end;
72      int status;
73
74      printf( "\nEnter name of data file to be searched: " );
75      scanf( "%79s", filename );
76      infile = fopen ( filename, "r" );
77      if (! infile)  fatal( "    Error: couldn't open input %s\n", filename );
78
79      cursor = pData->store;          // Scanning pointer, set to start of array.
80      end = pData->store + pData->max;    // An off-board sentinel
81
82      for( ; cursor<end; ++ cursor ) {
83          status = fscanf( infile, IN_FMT, cursor);
84          if( status != 1 ) break;    // Quit for bad data or for end of file.
85      }
86      pData->n = cursor-pData->store; // Actual # of items read.
87  }
88
```

Filling the DataPack. Every data structure needs one or more functions for entering data into it.

1. Ideally, these functions should not only provide access to the data structure; they should also be the *only* way to enter data into it. In C++, private data members allow us to restrict access in this way. However, in C, good programming style and lack of knowledge of the implementation are the only ways that unwanted access can be prevented.
2. Upon entry to `readData()` (lines 66...87), the information in the data pack is used to establish the pointers that will be used for sequential array processing. At no time is reference made to global variables or constants; the data structure is self sufficient.
3. On line 79, the cursor is initialized to the beginning of the array and will always point at the first unfilled slot. Line 80 establishes the end pointer, an off-board sentinel. It points to the first location that is not in the array. This is legal and sound practice that conforms to the C standard. When `cursor==end`, the array is full.

4. On line 83, there is a call on fscanf() that uses the format string `IN_FMT` defined at the top of the program. By changing the define statements and the typedef, we can change the type of data that we are sorting to any type for which `<=` is defined.

   Note that the return value from fscanf() (line 83) is stored and checked. When the format for `fscanf()` calls for reading one item, the return value of 1 indicates success. A smaller value indicates that a read error or an end-of-file caused the program to stop reading the data file before a data element was read. In a production program, more elaborate error handling would be needed.

5. It is important to maintain internal consistency within an object. Before returning, the data pack is updated (line 86) to reflect the number of items actually stored in it. We use pointer subtraction to calculate this number; the address of the head of the array is subtracted from the cursor. This tells us the number of array slots (not the number of bytes) between them.

```
 89  // Continuation of code file for  DataPack programs.                    pack.c
 90  // ------------------------------------------------------------------------
 91  // Print array values, one per line, to the selected stream.
 92
 93  void
 94  printData( DataPack* pData, stream outstream )
 95  {
 96      for( int k=0; k < pData->n; ++k)
 97          fprintf( outstream, OUT_FMT, pData->store[k] );
 98  }
 99
100  // ------------------------------------------------------------------------
101  // Generic insertion sort using a DataPack.
102  // Sort n values starting at pData->store by an insertion sort algorithm.
103
104  void
105  sortData( DataPack* pData )
106  {
107      BT* const end = pData->store + pData->n;    // Off-board sentinel.
108      BT* pass;                          // First unsorted item; begin pass here.
109      BT  newcomer;                      // Data value being inserted.
110      BT* hole;                          // Array slot containing no data.
111
112      for ( pass=pData->store+1; pass<end; ++pass ) {
113          // Pick up next item and insert into sorted portion of array.
114          newcomer = *pass;
115          for ( hole=pass; hole>pData->store; --hole ) {
116              if ( *(hole-1) <= newcomer ) break;   // Insertion slot is found.
117              *hole = *(hole-1);                    // Move item back one slot.
118          }
119          *hole = newcomer;
120      }
121  }
```

Printing the data. Some sort of output function is usually necessary in an application. It is also essential during debugging, and therefore is one of the first functions written for a new data structure.

1. The `printData()` function (lines 93...98) is simpler than `readData()` because we know at the outset how much data is in the array and that does not change, and because there is no need to check for errors or end-of-file. This makes a simple `for` loop (line 96) and subscripts (line 97) an attractive alternative to a cursor for scanning through the array values.
2. Lines 89...91 are a loop that calls `fprintf()` using the format string defined at the top of the program.

Sorting by insertion. The `sortData()` function (lines 104...121) implements insertion sort, the most efficient of the simple sorts (and reasonable to use in practice for small arrays, perhaps up to size 8 or so). This

is a typical double-loop implementation of the insertion sort algorithm. The implementation uses pointers, not subscripts, which slightly increases efficiency.

1. Each declaration has a comment that explains the usage of the variable. Note the use of brief, imaginative, meaningful variable names. This makes a huge difference in the readability of the code. In contrast, code is much harder to understand when it is written using identifiers like `i` and `j` for loop indices.
2. As in the previous functions, the pointer named `end` points at the first memory location that is not in the array being sorted.
3. At all times, the array is divided into two portions: a sorted portion (smaller subscripts) and an unsorted portion (larger subscripts). Initially, the sorted part of the array contains one element, so the sorting loop (lines 112...120) starts with the element at data[1].
4. The algorithm makes $N - 1$ passes over the data to sort N items. On each pass through the outer loop, it selects and copies one item (the `newcomer`, line 114) from the beginning of the unsorted portion of the array, leaving a `hole` that does not contain significant data.
5. The inner loop (lines 115...118) searches the sorted portion of the array for the correct slot for the newcomer. The `for` loop is used to guard the boundaries of the array. Within the loop body, an `if...break`, line 116, is used to leave the loop when the correct insertion slot is located.
6. To find that position, the loop scans backwards through the array, comparing the `newcomer` to each element in turn (line 116). If the `newcomer` is smaller, the other element is moved into the `hole` (line 117) and the `hole` pointer is decremented by the loop (line 118).
7. If the `newcomer` is not smaller, we break out of the inner loop and copy the `newcomer` into the `hole` (line 119).