a problem without deciding what to do about it. The exception enables each client to act as needed in its own particular situation. For simplicity right now, we will adopt the philosophy that methods should throw exceptions only in truly unusual circumstances, when no other reasonable solution exists. You can learn about exceptions in C++ Interlude 3.

**Security Note:** All methods should enforce their preconditions.

**Note:** A first draft of a module's specifications often overlooks or ignores situations that you really need to consider. You might intentionally make these omissions to simplify this first draft. Once you have written the major portions of the specifications, you can concentrate on the details that make the specifications complete.

### 1.3.3 Abstraction

**Abstraction** separates the purpose of a module from its implementation. Modularity breaks a solution into parts, or modules; abstraction specifies each module clearly before you implement it in a programming language. For example, what does the module assume, and what action does it take? What task is this module responsible for when called on? Such specifications clarify the design of your solution, because you can focus on its high-level functionality without the distraction of implementation details. In addition, they help you modify one part of a solution without significantly affecting the other parts. For example, you should be able to change the sorting algorithm in the previous example without affecting the rest of the solution.

As the problem-solving process proceeds, you gradually refine the modules until eventually you implement their actions by writing code—typically, classes and their methods. Separating the purpose of a module from its implementation is known as **functional (or procedural) abstraction.** Once a module is written, you can use it without knowing the particulars of its algorithm as long as you have a statement of its purpose and a description of its arguments. Assuming that the module is documented properly, you can use it knowing only its specifications. You do not need to look at its implementation.

Functional abstraction is essential to team projects. After all, in a team situation, you have to use modules written by others, frequently without knowledge of their algorithms. Can you actually use such a module without studying its code? In fact, you do this each time you use a C++ Standard Library function, such as `sqrt` in the C++ math library `cmath`. Because `sqrt` is precompiled, you do not have access to its source statements. Furthermore, it may be that `sqrt` was written in a language other than C++! There is so much about `sqrt` that you do not know; yet you can use it in your program without concern, as long as you know its specifications. If you pass `sqrt` a floating-point expression, it will return the floating-point square root of the value of that expression. You can use `sqrt` even though you do not know its implementation.

Consider now a collection of data and a set of operations on the data. The operations might include ones that add new data to the collection, remove data from the collection, or search for some data. **Data abstraction** focuses on what the operations do with the collection of data, instead of on how you implement them. The other modules of the solution "know" what operations they can perform, but they do not know how the data is stored or how the operations are performed.
For example, you have used an array, but have you ever stopped to think about what an array actually is? There are many pictures of arrays throughout this book. They might resemble the way a C++ array is implemented on a computer, and then again they might not. In either case, you can use an array without knowing what it “looks like”—that is, how it is implemented. Although different systems may implement arrays in different ways, the differences are transparent to the programmer.

For instance, regardless of how the array `years` is implemented, you can always store the value 1492 in location `index` of the array by using the statement

```cpp
years[index] = 1492;
```

and later display the value by using the statement

```cpp
cout << years[index] << endl;
```

Thus, you can use an array without knowing the details of its implementation, just as you can use the function `sort` without knowing the details of its implementation. Let’s explore this idea in more detail.

### 1.3.4 Information Hiding

As you have seen, abstraction tells you to write functional specifications for each module that describe its outside, or public, view. However, abstraction also helps you identify details that you should hide from public view—details that should not be in the specifications but rather should be private. The principle of information hiding tells you not only to hide such details within a module, but also to ensure that no other module can tamper with these hidden details.

While writing a module’s specifications, you must identify details that you can hide within the module. The principle of information hiding involves not only hiding these details, but also making them inaccessible from outside a module. One way to understand information hiding is to imagine walls around the various tasks a program performs. These walls prevent the tasks from becoming entangled. The wall around each task prevents the other tasks from “seeing” how that task is performed.

The isolation of the modules cannot be total, however. Although `MyProgram` does not know how the task `sort` is performed, it must know what the task `sort` is and how to initiate it. For example, suppose your program needs to operate on a sorted array of names. The program may, for instance, need to search the array for a given name or display the names in alphabetical order. The program thus needs a function `sort` that sorts an array of names. Although the rest of the program knows that `sort` can sort an array, it should not care how `sort` accomplishes its task.

Thus, imagine a tiny slit in the wall, as Figure 1-2 illustrates. The slit is not large enough to allow the outside world to see the function’s inner workings, but items can pass through the slit into and out of the function. This slit is the prototype, declaration, or header of the function. The slit comprises the function or method’s name, parameter list, and return type. For example, you can pass the array into the function `sort`, and the function can pass the sorted array out to you. What goes in and comes out is governed by the terms of the function’s specifications, or contract: If you use the function in this way, this is exactly what it will do for you.

Suppose that a faster sort algorithm is developed. Since the function `sort` is isolated from the other modules in the program, the new algorithm can be implemented in the `sort` function without affecting those other modules. Thus, if `MyProgram` uses the task `sort`, and if the algorithm and implementation for performing the sort changes, `MyProgram` will not be affected. As Figure 1-3 illustrates, the wall prevents `MyProgram`’s algorithm from depending on `sort`’s algorithm.