In some cases, any of the several options might work. The point of design is choosing one of the several good options for your case. Don't worry too much about choosing the best one. As Butler Lampson, a senior engineer at Digital Equipment Corporation, says, it's better to strive for a good solution and avoid disaster rather than trying to find the best solution (Lampson 1984).

Put the stair-step table lookup into its own routine. When you create a transformation function that changes a value like *StudentGrade* into a table key, put it into its own routine.

Other Examples of Table Lookups

A few other examples of table lookups appear in other sections of the book. They're used in the course of making other points, and the contexts don't emphasize the table lookups per se. Here's where you'll find them:

- Looking up rates in an insurance table: Section 15.3, "Creating Loops Easily—from the Inside Out"
- Cost of paging during a table lookup: Section 28.3, "Kinds of Fat and Molasses"
- Combinations of boolean values (A or B or C): "Substitute Table Lookups for Complicated Expressions" in Section 29.2
- Precomputing values in a loan repayment table: Section 29.4, "Expressions"

12.3 Abstract Data Types (ADTs)

CROSS-REFERENCE Abstract data types are usually implemented as modules. Both ADTs and modules are combinations of data and operations on the data. For details on modules, see Chapter 6, "Three out of Four Programmers Surveyed Prefer Modules."

An abstract data type is a collection of data and operations that work on that data. The operations both describe the data to the rest of the program and allow the rest of the program to change the data. The word "data" in "abstract data type" is used loosely. An ADT might be a graphics window with all the operations that affect it; a file and file operations; or even an insurance-rates table and the operations on it.

Traditionally, programming books wax mathematical when they arrive at the topic of abstract data types. They tend to make statements like "One can think of an abstract data type as a mathematical model with a collection of operations defined on it." Then they provide some boring examples of how to write access routines for a stack, a queue, or a list. Such books make it seem as if you'd never actually use an abstract data type except as a sleep aid.

Such dry explanations of abstract data types completely miss the point. Abstract data types are exciting because you can use them to manipulate real-world entities rather than computer-science entities. Instead of inserting a

node into a linked list, you can add a cell to a spreadsheet, a new type of window to a list of window types, or another passenger car to a bullet-train simulation. Don't underestimate the power of being able to work in the problem domain rather than the computer-science domain. This isn't to say that you shouldn't use abstract data types to implement stacks, queues, and lists; you should. It is to say that ADTs have much more power than is usually acknowledged.

Example of the Need for an Abstract Data Type

To get things started, here's an example of a case in which an ADT would be useful. We'll get to the theoretical details after we have an example to talk about.

Suppose you're writing a program to control text output to the screen using a variety of typefaces, point sizes, and font attributes (such as bold and italic). Part of the program manipulates the text's fonts. If you use an ADT, you'll have a group of font routines bundled with the data—the typeface names, point sizes, and font attributes—they operate on. The collection of font routines and data is an ADT.

If you're not using ADTs, you'll take an ad hoc approach to manipulating fonts. For example, if you need to change to a 12-point font size, you'll have code like this:

```
CurrentFont.Size = 12
```

If you change sizes in several places in the program, you'll have similar lines spread throughout your program.

If you need to set a font to bold, you might have code like this:

```
CurrentFont.Attribute = CurrentFont.Attribute or 02h
```

If you're lucky, you'll have something cleaner than that, but this is about the best you'll get with an ad hoc approach:

```
CurrentFont.Bold = True
```

If you program this way, you're likely to have similar lines in several places in your program.

Finally, if you set the face name, you'll have lines like this:

```
CurrentFont.Typeface = TIMES_ROMAN
```

Benefits of Using Abstract Data Types (ADTs)

The problem isn't that the ad hoc approach is bad programming practice. It's that you can replace the approach with a better programming practice that produces these benefits:

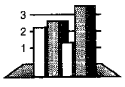
You can hide implementation details. Hiding information about the font data structure means that if the data structure changes, you can change it in one place without affecting the whole program. For example, unless you hid the implementation details in an ADT, changing the data structure from the first representation of bold to the second would entail changing your program in every place in which bold was set rather than in just one place. Hiding the information also protects the rest of the program if you decide to store data in external storage rather than in memory or to rewrite all the font-manipulation routines in another language.

Changes don't affect the whole program. If fonts need to become richer and support more operations (such as switching to small caps, superscripts, strikethrough, and so on), you can change the program in one place. The change won't affect the rest of the program.

It's easier to improve performance. If you need to improve font performance, you can recode a few well-defined routines rather than wading through an entire program.

The program is more obviously correct. You can replace the more tedious task of verifying that statements like *CurrentFont.Attribute = CurrentFont.Attribute or 02h* are correct with the easier task of verifying that calls to *SetCurrentFontToBold()* are correct. With the first statement, you can have the wrong structure name, the wrong field name, the wrong logical operation (a logical *and* instead of *or*), or the wrong value for the attribute (*20h* instead of *02h*). In the second case, the only thing that could possibly be wrong with the call to *SetCurrentFontToBold()* is that it's a call to the wrong routine name, so it's easier to see that it's correct.

The program becomes more self-documenting. You can improve statements like *CurrentFont.Attribute or 02h* by replacing *02h* with *BOLD* or whatever *02h* represents, but that doesn't compare to the readability of a routine call such as *SetCurrentFontToBold()*.



HARD DATA

Woodfield, Dunsmore, and Shen conducted a study in which graduate and senior undergraduate computer-science students answered questions about two programs—one that was divided into eight routines along functional lines and one that was divided into eight abstract-data-type routines (1981). Students using the abstract-data-type program scored over 30 percent higher than students using the functional version.

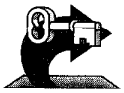
You don't have to pass data all over your program. In the examples just presented, you have to change *CurrentFont* directly or pass it to every routine that works with fonts. If you use an abstract data type, you don't have to pass *CurrentFont* all over the program and you don't have to turn it into global data either. The ADT has a structure that contains *CurrentFont*'s data. The

data is directly accessed only by routines that are part of the ADT. Routines that aren't part of the ADT don't have to worry about the data.

You're able to work with real-world entities rather than with computer-science structures. You can define operations dealing with fonts so that most of the program operates solely in terms of fonts rather than in terms of array accesses, record definitions, and *BOLD* as a true or false boolean.

In this case, to define an abstract data type, you'd define a few routines to control fonts—perhaps these:

```
SetCurrentFontSize( Size )
SetCurrentFontToBold()
SetCurrentFontToItalic()
SetCurrentFontToRegular()
SetCurrentFontTypeFace( FaceName )
```



KEY POINT

The code inside these routines would probably be short—it would probably be similar to the code you saw in the ad hoc approach to the font problem earlier. The difference would be that you would have isolated font operations in a set of routines. That would provide a better level of abstraction for the rest of your program to work with fonts, and it would give you a layer of protection against changes in font operations. Putting direct data manipulations behind access routines is like having an air lock in a submarine. You can go out, and you can go in, but the air stays inside and the water stays outside.

More Examples of ADTs

Here are a few more examples of ADTs:

Suppose you're writing software that controls the cooling system for a nuclear reactor. You can treat the cooling system as an abstract data type by defining the following operations for it:

```
GetReactorTemperature()
SetCirculationRate( Rate )
OpenValve( ValveNumber )
CloseValve( ValveNumber )
```

The specific environment would determine the code written to implement each of these operations. The rest of the program could deal with the cooling system through these functions and wouldn't have to worry about internal details of data-structure implementation, data-structure limitations, changes, and so on.

Here are more examples of abstract data types and likely operations on them:

Cruise Control
Set speed
Get current settings
Resume former speed
Deactivate

Set of Help Screens
Add help topic
Remove help topic
Set current help topic
Display help screen
Remove help display
Display help index
Back up to previous screen

List
Initialize list
Insert item in list
Remove item from list
Read next item from list

Light
Turn on
Turn off

Blender
Turn on
Turn off
Set speed
Start "Insta-Pulverize"
Stop "Insta-Pulverize"

Menu
Start new menu
Delete menu
Add menu item
Remove menu item
Activate menu item
Deactivate menu item
Display menu
Hide menu
Get menu choice

Pointer
Get pointer to new memory
Dispose of memory from existing pointer
Change amount of memory allocated

Fuel Tank
Fill tank
Drain tank
Get tank capacity
Get tank status

Stack
Initialize stack
Push item onto stack
Pop item from stack
Read top of stack

File
Open file
Read file
Write file
Set current location
Close file

Elevator
Move up one floor
Move down one floor
Move to specific floor
Report current floor
Return to home floor

You can derive several guidelines from a study of these examples:

Build typical computer-science data structures as ADTs. Most discussions of ADTs focus on representing typical computer-science data structures as ADTs. As you can see from the examples, you can represent stacks, lists, and pointers, as well as virtually any other typical data structures, as ADTs.

The question you need to ask is What does this stack, list, or pointer represent? If a stack represents a set of employees, treat the ADT as employees rather than a stack. If a list represents a set of billing records, treat it as billing records rather than a list. If a pointer represents a cell in a spreadsheet, treat it as a cell rather than a pointer. Treat yourself to the highest possible level of abstraction.

Treat common objects such as files as ADTs. Most languages include a few abstract data types that you're probably familiar with but might not think of as ADTs. File operations are a good example. While writing to disk, the operating system spares you the grief of positioning the read/write head at a specific physical address, allocating a new disk sector when you exhaust an old one, and checking for binary error codes. The operating system provides

a first level of abstraction and the ADTs for that level. High-level languages provide a second level of abstraction and ADTs for that higher level. A high-level language protects you from the messy details of generating operating-system calls and manipulating data buffers. It allows you to treat a chunk of disk space as a "file."

You can layer ADTs similarly. If you want to use an ADT at one level that offers data-structure level operations (like pushing and popping a stack), that's fine. You can create another level on top of that one that works at the level of the real-world problem.

Treat even simple items as ADTs. You don't have to have a formidable data structure to justify using an abstract data type. One of the ADTs in the example lists is a light that supports only two operations—turning it on and turning it off. You might think that it would be a waste to isolate simple "on" and "off" operations in routines of their own, but even simple operations can benefit from the use of ADTs. Putting the light and its operations into an ADT makes the code more self-documenting and easier to change, confines the potential consequences of changes to the *TurnLightOn()* and *TurnLightOff()* routines, and reduces the amount of data you have to pass around.

Provide services in pairs with their opposites. Most operations have corresponding, equal, and opposite operations. If you have an operation that turns a light on, you'll probably need one to turn it off. If you have an operation to add an item to a list, you'll probably need one to delete an item from the list. If you have an operation to activate a menu item, you'll probably need one to deactivate an item. When you design an ADT, check each service to determine whether you need its complement. Don't create an opposite gratuitously, but do check to see whether you need one.

Refer to an ADT independently of the medium it's stored on. Suppose you have an insurance-rates table that's so big that it's always stored on disk. You might be tempted to refer to it as a "rate *file*" and create access routines such as *ReadRateFile()*. When you refer to it as a file, however, you're exposing more information about the data than you need to. If you ever change the program so that the table is in memory instead of on disk, the code that refers to it as a file will be incorrect, misleading, and confusing. Try to make the names of access routines independent of how the data is stored, and refer to the abstract data type, like the insurance-rates table, instead. That would give your access routine a name like *ReadRateTable()* instead of *ReadRateField()*.

Handling Multiple Instances of Data with ADTs

Once you've defined the base operations for an ADT, you might find that you need to work with more than one piece of data of that type. For example, if you're working with fonts, you might want to keep track of multiple fonts. In

ADT terms, each font you want to keep track of would be an "instance." One solution is to write a separate set of ADT operations for each font instance you want to keep track of. A better solution is to design the single ADT so that it works with multiple font instances. That usually means including services to create and delete instances and designing the other services so that they can work with multiple instances.

The font ADT originally offered these services:

```
SetCurrentFontSize( Size )
SetCurrentFontToBold()
SetCurrentFontToItalic()
SetCurrentFontToRegular()
SetCurrentFontTypeFace( FaceName )
```

If you want to work with more than one font at a time, you'll need to add services to create and delete font instances—maybe these:

```
CreateFont( FontID )
DeleteFont( FontID )
SetCurrentFont( FontID )
```

The notion of a *FontID* has been added as a way to keep track of multiple fonts as they're created and used. For other operations, you can choose from among three ways to handle the ADT interface:

Use font instances implicitly. Design a new service to call to make a specific font instance the current one—something like *SetCurrentFont(FontID)*. Setting the current font makes all other services use the current font when they're called. If you use this approach, you don't need *FontID* as a parameter to the other services.

Explicitly identify instances each time you use ADT services. In this case, you don't have the notion of a "current font." Routines that use the ADT's services don't have to keep track of font data, but they do have to use a font ID. This requires adding *FontID* as a parameter to each routine.

Explicitly provide the data used by the ADT services. In this approach, you declare the data that the ADT uses within each routine that uses an ADT service. In other words, you create a *Font* data structure that you pass to each of the ADT service routines. You must design the ADT service routines so that they use the *Font* data that's passed to them each time they're called. You don't need a font ID if you use this approach because you keep track of the font data itself. (Even though the data is available directly from the *Font* data structure, you should access it only with the ADT service routines. This is called keeping the record "closed" and is discussed more later in this section.)

The advantage of this approach is that the ADT service routines don't have to look up font information based on a font ID. The disadvantage is that it's

dangerous for the rest of the program because the data is directly available to be fooled with.

Inside the abstract data type, you'll have a wealth of options for handling multiple instances, but outside, this sums up the choices.

Mixing Levels of Abstraction (Don't!)

If you directly access the data structure only within the service routines for the ADT and use the ADT service routines everywhere else, you'll have a consistent level of abstraction. If you don't, you'll have an inconsistent level of abstraction, and you'll defeat one of the principal benefits of using ADTs. The practical consequence of inconsistent abstraction is that modifications are more error prone. You create the false impression that changing an access routine catches all the accesses to the data.

To extend an earlier analogy: If the ADT's service routines work as an air lock that keeps water from getting into a submarine, inconsistent use of service routines are leaky panels in the routine. The leaky panels might not let water in as quickly as an open air lock, but if you give them enough time, they'll still sink the ship. In practice, this is what happens when you mix levels of abstraction. As the program is modified, the mixed levels of abstraction make the program harder and harder to understand, and it gradually degrades until it becomes unmaintainable.

Open and Closed Records

Along with the idea of mixing levels of abstraction comes the idea of "open" and "closed" records. A record, or structure, is considered to be open in a routine if the routine uses any of its fields directly. It's closed in the routine if none of its fields are used directly—that is, if it's used only as a complete structure. As mentioned earlier, you can use closed records as one way of handling multiple instances of an ADT. The fact that a record's available doesn't mean that you have to open it. Keep the record closed except within the confines of the ADT's service routines.

As you consider the three ways to handle multiple fonts, note that the temptation to open records is a good reason to prefer the *FontID* approach to the *Font*-record approach.

ADTs and Information Hiding, Modules, and Objects

Abstract data types and information hiding are related concepts. If you want to make an abstract data type, you hide its implementation details. That's the road that leads from abstract data types to information hiding. If you want to practice information hiding, you look for secrets you can hide. One of the

most obvious secrets is the implementation details of an abstract data type. That's the road leading from information hiding to abstract data types.

The abstract-data-type concept is also related to the module idea. In languages that support modules directly, you can implement each abstract data type in its own module. A module is a collection of data and operations on that data just as an ADT is a collection of data and operations on that data.

CROSS-REFERENCE
For more on object-oriented design, see Section 7.3, "Object-Oriented Design."

The ADT concept is closely related to the concept of objects too. "Object" is a loosely defined term, but it generally refers to a collection of data and operations on the data. In that limited sense, all objects are abstract data types. The idea of an "object," however, also makes use of the ideas of inheritance and polymorphism. One way of thinking of an object is as an abstract data type plus inheritance and polymorphism. Abstract data types are part of the notion of an object, not the whole idea.

Language Support for ADTs

Some languages provide better support for ADTs than others. Ada provides excellent support. For the font example, Ada would allow you to package all the font service routines into a single package. You could declare a few service routines to be public and have other routines that you used only within the package. You could restrict the definition of *Font* so that routines using ADT services wouldn't be allowed to operate on *Font*'s innards. They could declare *Font* data, but they wouldn't be able to operate on specific fields and would have no knowledge of how *Font* was structured.

In other languages—Pascal, C, Basic, and Fortran—you can package an ADT within a single source file, export some routines, and keep other routines private to varying degrees. C works well for ADTs because it always supports multiple source files (modules) and private routines. The ADT capabilities of the other languages depend on the capabilities of specific implementations. In other words, in other languages, your work won't necessarily be portable.

Regardless of how effectively you can hide routines in a given language, hiding data is problematic in all these languages. To extend the font example: If you declare a *Font* variable outside a service routine, you can operate on its innards anywhere that you can reference the variable. It is always an open record—which, unfortunately, often means it will be a broken record. The languages don't enforce closed records. Only discipline, programming standards, and painful memories of the sins of past indulgences will keep your records closed.

Key Points

- Properly structured data helps to make programs less complicated, easier to understand, and easier to maintain.
- Tables provide an alternative to complicated logic structures. If you find that you're confused by a program's logic, ask yourself whether you could simplify by using a lookup table.
- Abstract data types are a valuable tool for reducing complexity. They allow you to write your program in layers and to write the top layer in terms of the problem domain rather than in terms of computer-science, programming-language details.