Library of Congress Cataloging-in-Publication Data

Carrano, Frank M.
Data abstraction and problem solving with C++: walls and mirrors / Frank M. Carrano, Janet J. Prichard.—3rd ed.
p. cm.
ISBN 0-201-74119-9 (alk. paper)
1. C++ (Computer program language) 2. Abstract data types. 3. Problem solving—Data Processing. I. Prichard, Janet J. II. Title.

QA76.73.C153 C38 2001
005.7'3—dc21 2001027940
CIP

Readability. For a program to be easy to follow, it should have a good structure and design, a good choice of identifiers, good indentation and use of blank lines, and good documentation. You should avoid clever programming tricks that save a little computer time at the expense of much human time. You will see examples of these points in programs throughout the book.

Choose identifiers that describe their purpose, that is, are self-documenting. Distinguish between keywords, such as `int`, and user-defined identifiers. This book uses the following conventions:

Identifier style

- Keywords are lowercase and appear in boldface.
- Names of standard functions are lowercase.
- User-defined identifiers use both upper- and lowercase letters, as follows:
  - Class names are nouns, with each word in the identifier capitalized.
  - Function names within a class are verbs, with the first letter lowercase and subsequent internal words capitalized.
  - Variables begin with a lowercase letter, with subsequent words in the identifier capitalized.
  - Data types declared in a `typedef` statement and names of structures and enumerations each begin with an uppercase letter.
  - Named constants and enumerators are entirely uppercase and use underscores to separate words.

Two learning aids

- Two other naming conventions are followed as a learning aid:
  - Data types declared in a `typedef` statement end in `Type`.
  - Exception names end in `Exception`.

Use a good indentation style to enhance the readability of a program. The layout of a program should make it easy for a reader to identify the program's modules. Use blank lines to offset each function. Also, within both functions and the main program, you should offset with blank lines and indent individual blocks of code visibly. These blocks are generally—but are not limited to—the actions performed within a control structure, such as a `while` loop or an `if` statement.

You can choose from among several good indentation styles. The four most important general requirements of an indentation style are that

Guidelines for indentation style

- Blocks should be indented sufficiently so that they stand out clearly.
- Indentation should be consistent: Always indent the same type of construct in the same manner.
- The indentation style should provide a reasonable way to handle the problem of **rightward drift**, the problem of nested blocks bumping against the right-hand margin of the page.
- In a compound statement, the open and closed braces should line up:

```
{   <statement1>
    <statement2>
        ⋮
    <statementn>
}
```

  Although it is preferable to place the open brace on its own line, space restrictions in this book prevent doing so except at the beginning of a function's body.

Within these guidelines there is room for personal taste. Here is a summary of the style you will see in this book:

Indentation style in this book

- A `for` or `while` statement is written for a simple action as

```
while (expression)
    statement
```

  and for a compound action as

```
while (expression)
{   statements
}   // end while
```

- A `do` statement is written for a simple action as

```
do
    statement
while (expression);
```

  and for a compound action as

```
do
{   statements
}   while (expression);
```

- An *if* statement is written for simple actions as

```
if (expression)
    statement1
else
    statement2
```

and for compound actions as

```
if (expression)
{   statements
}

else
{   statements
}   // end if
```

One special use of the *if* statement warrants another style. Nested *if* statements that choose among three or more different courses of action, such as

```
if (condition1)
    action1
else if (condition2)
        action2
      else if (condition3)
          action3
```

are written as

```
if (condition1)
    action1
else if (condition2)
    action2
else if (condition3)
    action3
```

This indentation style better reflects the nature of the construct, which is like a generalized *switch* statement:

```
case condition1 : action1; break;

case condition2 : action2; break;

case condition3 : action3; break;
```

- Braces are used to increase readability, even when they are not a syntactic necessity. For example, in the construct

```
while (expression)
{   if (condition1)
        statement1
    else
        statement2
}   // end while
```

the braces are syntactically unnecessary because an `if` is a single statement. However, the braces highlight the scope of the `while` loop.

Documentation. A program should be well documented so that others can read, use, and modify it easily. Many acceptable styles for documentation are in use today, and exactly what you should include often depends on the particular program or your individual circumstances. The following are the essential features of any program's documentation:

KEY CONCEPTS ***Essential Features of Program Documentation***

1. An initial comment for the program that includes
   a. Statement of purpose
   b. Author and date
   c. Description of the program's input and output
   d. Description of how to use the program
   e. Assumptions such as the type of data expected
   f. Statement of exceptions; that is, what could go wrong
   g. Brief description of the major classes
2. Initial comments in each class that state its purpose and describe the data contained in the class (constants and variables)
3. Initial comments in each function that state its purpose, preconditions, postconditions, and functions called
4. Comments in the body of each function to explain important features or subtle logic

Consider who will read your comments when you write them

Beginning programmers tend to downplay the importance of documentation because the computer does not read comments. By now, you should realize that people also read programs. Your comments must be clear enough for someone else to either use your function in a program or modify it. Thus, some of your comments are for people who want to use your function, while others are for people who will revise its implementation. You should distinguish between different kinds of comments.

You benefit from your own documentation by writing it now instead of later

Beginners have a tendency to document programs as a last step. You should, however, write documentation as you develop the program. Since the task of writing a large program might extend over a period of several weeks, you may find that the function that seemed so obvious when you wrote it last week will seem confusing when you try to revise it next week. Why not benefit from your own documentation by writing it now rather than later?