Object-Oriented Computation in C++ and Java

a practical guide to design patterns for object-oriented computing

CONRAD WEISERT

Object-Oriented Computation in C++ and Java

Also Available from Dorset House Publishing

Best Practices for the Formal Software Testing Process: A Menu of Testing Tasks
by Rodger D. Drabick foreword by William E. Perry
ISBN-13: 978-0-932633-58-3 Copyright © 2004 312 pages, softcover

The Deadline: A Novel About Project Management
by Tom DeMarco
ISBN-13: 978-0-932633-39-2 Copyright ©1997 320 pages, softcover

Five Core Metrics: The Intelligence Behind Successful Software Management
by Lawrence H. Putnam and Ware Myers
ISBN-13: 978-0-932633-55-2 Copyright © 2003 328 pages, softcover

More Secrets of Consulting: The Consultant's Tool Kit
by Gerald M. Weinberg
ISBN-13: 978-0-932633-52-1 Copyright © 2002 216 pages, softcover

Peopleware: Productive Projects and Teams, 2nd ed.
by Tom DeMarco and Timothy Lister
ISBN-13: 978-0-932633-43-9 Copyright © 1999 264 pages, softcover

The Psychology of Computer Programming: Silver Anniversary Edition
by Gerald M. Weinberg
ISBN-13: 978-0-932633-42-2 Copyright ©1998 360 pages, softcover

Systems Modeling & Requirements Specification Using ECSAM: An Analysis Method for Embedded and Computer-Based Systems
by Jonah Z. Lavi and Joseph Kudish
ISBN-13: 978-0-932633-45-3 Copyright © 2005 400 pages, softcover

Waltzing with Bears: Managing Risk on Software Projects
by Tom DeMarco and Timothy Lister
ISBN-13: 978-0-932633-60-6 Copyright © 2003 208 pages, softcover

For More Information

✔ Contact us for prices, shipping options, availability, and more.

✔ Sign up to receive *INSIDE DORSET HOUSE (iDH)* by mail or fax.

✔ Send e-mail to subscribe to *iDH,* our e-mail newsletter.

✔ Visit Dorsethouse.com for savings, reviews, downloads, and more.

DORSET HOUSE PUBLISHING
An Independent Publisher of Books on Systems and Software Development and Management. Since 1984.
353 West 12th Street New York, NY 10014 USA
1-800-DH-BOOKS 1-800-342-6657
212-620-4053 fax: 212-727-1044
info@dorsethouse.com www.dorsethouse.com

Object-Oriented Computation in C++ and Java

a practical guide to design patterns
for object-oriented computing

CONRAD WEISERT

Dorset House Publishing
353 West 12th Street
New York, NY 10014

Library of Congress Cataloging-in-Publication Data

Weisert, Conrad.
Object-oriented computation in C++ and Java / Conrad Weisert.
p. cm.
Summary: "Introduces use of numeric data items in C++ and Java, object-oriented computer programming languages. Numeric data items are a subset of application-domain objects and are central to business and scientific software applications. Includes exercises and answers"--Provided by publisher.
ISBN-13: 978-0-932633-63-7 (trade paper : alk. paper)
ISBN-10: 0-932633-63-3 (trade paper : alk. paper)
1. Object-oriented programming (Computer science) 2. C++ (Computer program language) 3. Java (Computer program language) I. Title.
QA76.64.W4353 2006
005.1'17--dc22

2006002491

Quantity discounts are available from the publisher. Call (800) 342-6657 or (212) 620-4053 or e-mail info@dorsethouse.com. Contact same for examination copy requirements and permissions. To photocopy passages for academic use, obtain permission from the Copyright Clearance Center: (978) 750-8400 or www.copyright.com.

Cover Design: David McClintock

Distributed in the English language in Singapore, the Philippines, and Southeast Asia by Alkem Company (S) Pte. Ltd., Singapore; and in the English language in India, Bangladesh, Sri Lanka, Nepal, and Mauritius by Prism Books Pvt., Ltd., Bangalore, India.

Printed in the United States of America

Library of Congress Catalog Number: 2006002491

ISBN-10: 0-932633-63-3
ISBN-13: 978-0-932633-63-7

12 11 10 9 8 7 6 5 4 3 2 1

Digital release by Pearson Education, Inc., June, 2013

ACKNOWLEDGMENTS

I was first made aware of the value of well-organized, highly modular programs by Joe Myers (IBM), along with Tom Steel (System Development Corp.), George Mealy (Rand Corp.), and other designers of the SHARE Operating System (SOS) for the IBM 709, arguably still the most elegant large program I've seen. I'm grateful to my bosses at the Johns Hopkins Applied Physics Laboratory, Bob Rich and Lowell McClung, for encouraging and supporting me in maintaining and enhancing SOS, participating in large-scale application development, and contributing to our scientific library of reusable modules.

The structured revolution and the PL/I language helped me to reinforce and expand my appreciation of good programming practices in a higher-level language. I appreciated the support of my boss at Union Carbide Corp., Jim Rowe, in applying those practices to major business applications. We established a business-oriented library of reusable modules that eventually contained many of the kinds of building block that would later be formalized in object-oriented programming (OOP).

David Miller (DePaul University) introduced me to OOP in an advanced artificial intelligence programming course using CLOS (LISP OOP extensions). Another DePaul instructor, Glenn Lancaster, introduced me to then-new and still shaky C++ language. Since

1990, various clients have given me the opportunity to advise on and teach OOP concepts and techniques, and to develop useful OOP classes.

The Knowledge Systems Institute (KSI) and the Illinois Institute of Technology (IIT) gave me the chance to expand my short OOP courses into full-semester academic courses.

Colleagues, former mentors, and family members have encouraged my work on this book and expressed their wish to have a copy of the finished product. My former IIT student Vijayram Gopu was helpful in reviewing early drafts of several chapters, but the responsibility for mistakes is entirely mine.

Readers who discover errors of any kind should let me know at cweisert@acm.org. I'll post corrections and discussion at www.dorsethouse.com/books/ooc.html.

CONTENTS

PREFACE

Many C++ or Java textbooks, courses, and class libraries emphasize object-oriented classes for two kinds of data:

- one-dimensional *containers* (Java *collections),* such as vectors, lists, and sets
- graphical user interface (GUI) components, such as windows, forms, and menus

However, most of the data items our programs process belong to neither of those categories. Container structures and GUI components rarely belong to the *application domain.* They don't represent actual objects in the real world of a business or science application. True application-domain objects model real-world data items at the core of the very purpose behind developing a computer application.[1]

For more than a dozen years, I've taught courses in advanced object-oriented programming to students who have backgrounds in both commercial/business applications and scientific/engineering applications. In searching for a suitable textbook, I've found none that adequately treat application-domain objects.

[1]Application-domain objects are sometimes misleadingly called "business objects" although they're not limited to business or commercial applications. Scientific and engineering applications need and use application-domain objects just as much as do business applications, if not more.

Unfortunately, but hardly surprisingly, the omission of application-domain data from books and courses is mirrored by much application software. I frequently encounter allegedly object-oriented application systems in which nearly all numeric quantities are represented as floating-point numbers, as if the programmers have coded in FORTRAN.

In response, I've developed a large collection of course handout material, part of which has evolved into this book. *Object-Oriented Computation in C++ and Java* is designed to fill a gap in the literature of object-oriented programming. It treats an important subset of application-domain data: *numeric* data items. Numeric data are central both to most business applications and to every engineering or scientific application.

Object-Oriented Computation in C++ and Java is suited to an advanced programming course for senior undergraduates or masters-level students in engineering, business, or the sciences, as well as to self-study by practicing professionals. Since this book covers an area neglected by most OOP textbooks, it also serves well as a supplementary text for a survey course on object-oriented programming.

Chicago **—C.W.**
November 2006

INTRODUCTION

I.1 Your background

This book is intended for experienced programmers. Readers should either have completed a rigorous introductory course in object-oriented programming or have developed one or more nontrivial complete applications or object-oriented components.

I assume you already know

- the syntax and semantics of either C++ or Java
- facilities the language provides for defining classes and instantiating objects
- fundamental OOP notions of encapsulation, inheritance, and polymorphism

Whether you're an advanced student or a mature professional, you surely strive to be a *good programmer*. After mastering the concepts and techniques detailed in this book, you can expect to

- produce application software of *high quality*, especially as measured by cost of future maintenance as well as robustness, efficiency, ease of use, and potential reuse
- be *highly productive*, solving problems in far less time than the average programmer

- exercise *creativity and originality*, developing nonobvious solutions to problems that an average programmer either might not solve at all or would solve in a crude way

I.2 Reading guide

If you're a practicing application developer, you'll find it easy to read this book on your own. You should find each chapter's concepts and techniques directly and routinely applicable to the applications you work on.

Chapter 1 lays a foundation for numeric objects, showing their relationship to other kinds of data.

Chapter 2 reviews the language facilities you'll need in the later chapters. If you're already a world-class expert in C++ and Java, you may choose to skip that chapter.

Chapters 3 through 6 examine particular categories of numeric data that appear in real-world applications. I cite common patterns, starting with the simplest *pure numeric* classes, and build up to families of interacting related classes. I build representative and useful classes to support each category, working incrementally through the thought processes that a competent object-oriented designer would be likely to experience.

Chapter 7 examines the admittedly small potential for exploiting inheritance and polymorphism with numeric data.

Chapter 8 departs from numeric classes and objects to discuss *arrays* of numeric objects, emphasizing matrix manipulation and arrays of higher dimensionality. The container classes I develop in this chapter make heavy use of inheritance and polymorphism.

Problem-and-exercise guide

Most topics are followed by exercises. Some call for designing and writing code, while others call for analysis and discussion. Most can be easily solved in a few minutes. Those marked with a laurel wreath (see left) will take longer, and are suitable for small course projects. Those marked with a lightbulb (see left) call for creative insight that's reasonable to expect from a highly experienced professional, but may elude or startle students who are accustomed to being given low-level *how-to* specifications in an introductory programming course.

The Appendix contains suggested answers to selected exercises. You'll find source code for complete C++ and Java classes in the accompanying Website www.dorsethouse.com/books/ooc.html.

I.3 Methodology independence

Every program that performs nontrivial computation needs the kind of object-oriented class presented in this book, regardless of the tools and techniques used to specify and design it. Whether you love or hate UML,[1] favor or shun so-called *agile* methods, or design by hand or with CASE tools,[2] your object-oriented program will need exactly the same numeric classes. Two development teams may develop program components in a different sequence, or may document them in a different way, but the end-product software will contain essentially the same numeric classes.

Therefore, all software developers who work on computational applications will find this book compatible with the techniques they prefer.

I.4 Choice of language

This book is for both C++ and Java programmers. The exposition and examples use C++ mainly because C++ provides much stronger support for numeric data than Java does. If you're doubtful, bear with us in the early chapters and you'll soon see why.

But even if you're a committed Java programmer, you'll still find those presentations useful and relevant. The languages are similar enough that you should easily understand the C++ code examples. In addition, near the end of most chapters, I convert the most important examples to Java, noting the main differences between the two languages. The Website contains source code in both languages.

The principles apply also to most other programming languages that support objects. Even if your preferred programming language is C#, Python, Ruby, or Smalltalk, you'll find most of this book helpful.

---

[1] Unified Modeling Language, endorsed as a standard by the Object Management Group.

[2] Computer-aided software engineering.

This page intentionally left blank

3

DEFINING A PURE NUMERIC DATA TYPE

3.1 What does "pure numeric" mean?

A *pure numeric* data item doesn't represent an object in the real world. It's an abstract, mathematical entity.

Some pure numeric data types are built into our programming languages. Java calls them "primitive types." C supports integers (`short`, `int`, `long`) and real number approximations (`float`, `double`). C++ and Java inherit those built-in types from C with minor variations.

However, none of these languages supports such mathematical quantities as complex numbers, rational numbers, fixed-point numbers, transfinite numbers, or numbers of arbitrary precision.[1] If we need to use such numbers, we must construct them ourselves using OOP's class definition mechanism (or find a suitable existing class).

3.2 Example: Designing a `Complex` number class

We recall from elementary algebra that there is no real number `x` such that $x^2 = -9$. Nevertheless, square roots of negative numbers occur naturally in many problems, not only in scientific and engineering applications but also in some business applications. Many programs must, then, deal with numbers of the following form:

```
a + bi
```

[1] Complex numbers have recently been added to C as a built-in type.

where **a** and **b** are real numbers and **i** (often written **j** by engineers) is the *imaginary* square root of **-1**. Such a number is a complex number, with real part **a** and imaginary part **b**.

Is this example relevant?

If the applications you work on never require complex arithmetic and if you've forgotten what you learned about complex numbers in high school, bear with us anyway in exploring this interesting example, which lays a solid foundation for later examples that you'll find more directly applicable to your work.

3.2.1 Sketching a `Complex` class

An obvious way to represent a complex number is by a pair of floating-point numbers. A rough start on our C++ **Complex** class definition, then, is this:[2]

```
class  Complex {
// Member data
      double rp;          //  Real part
      double ip;          //  Imaginary part
// Constructors and accessors
 public:
  Complex(DOUBLE rp_=0, DOUBLE ip_=0)
        :       rp(rp_),       ip(ip_) {}
  double realPart() const {return rp;}
  double imagPart() const {return ip;}
// The compiler will generate an acceptable copy
// constructor, destructor, and
// assignment operator
};
```

With just that much, client programs can now declare, initialize, and assign complex numbers in various ways:

```
Complex  w;              //  0 + 0i
Complex  x = 2.5;        //  2.5+0i
Complex  y(2.5, -3.2);   //  2.5-3.2i
Complex  z = y;          //  2.5-3.2i
w = x;                   //  2.5 + 0i
y = 0.0;                 //  Why does this work?
```

Users can also retrieve the component parts in expressions:

```
. . . w.imagPart() . .      //  0
. . . x.realPart() . .      //  2.5
```

[2]We're still following the readability convention explained in Chapter 2 where **DOUBLE** means **const double**.

That's a promising start, but users still can't directly perform calculations involving complex numbers.

Problems and exercises

3.2-1 Two accessor functions, **`realPart()`** and **`imagPart()`**, allow the user program to retrieve the component parts of a complex number but not to modify them. Such functions are sometimes called "getters" or "read accessors." We could also provide functions specifically to change each of the component parts; such functions are sometimes called "modifiers," "setters," or "write accessors."

Should we provide modifier functions for users of our **`Complex`** class? If not, explain why we don't need them. If so, write the code.

3.2-2 Some programmers give their read accessor (getter) functions names that begin with **`get`**, such as **`getRealPart()`** and **`getImagPart()`**. Discuss the pros and cons of that convention. Consider traditions both in mathematics and in other programming languages.

3.2-3 Built-in complex numbers have recently been added to standard C and are now part of that non-object-oriented language. C++ and Java may or may not follow suit in the future. Discuss the pros and cons of adding complex numbers as another primitive built-in type and assess the impact of such a future enhancement on existing programs that use the **`Complex`** class we develop in this chapter.

3.2-4 We chose **`double`** as the type for the member data components, very likely the preferred choice of most client programs. In most implementations, then, a **`Complex`** object will consume 16 bytes. Some users, however, might have occasion to want a smaller object—for example, with single-precision **`float`** members, perhaps to save space in a huge array. Compare various ways of giving users such a choice.

3.2-5 We allowed the compiler to generate a default version of three essential methods: the copy constructor, the destructor, and the assignment operator. Was it acceptable and even desirable to do that instead of coding explicit versions? If so,

explain why we felt obliged to include the source-code comment to affirm that strategy? If not, code explicit versions and insert them into the class.

3.2.2 Complex arithmetic

In elementary algebra, we learned how to derive the following rules for complex arithmetic from basic definitions. If **x** and **y** are complex numbers such that

```
x ≡ a + bi
y ≡ c + di
```

then

```
x + y ≡ (a+c) + (b+d)i
x - y ≡ (a-c) + (b-d)i
```

We can first implement the compound assignment operators **+=** and **-=**. Because they're so simple, let's just insert them in-line into the class definition:

```
Complex& operator+= (COMPLEX rs)
      {rp += rs.rp; ip += rs.ip; return *this;}
Complex& operator-= (COMPLEX rs)
      {rp -= rs.rp; ip -= rs.ip; return *this;}
```

Then we can implement the corresponding simple arithmetic operators **+** and **-** as nonmember, non-friend, **inline** functions:

```
inline  Complex operator+ (COMPLEX ls, COMPLEX rs)
        {return Complex(ls)+=rs;}
inline  Complex operator- (COMPLEX ls, COMPLEX rs)
        {return Complex(ls)-=rs;}
```

Problems and exercises

3.2-6 Explain why we chose above to implement the simple arithmetic operators in terms of the corresponding compound assignment operators. What other two choices do we have? What disadvantages would each present?

3.2-7 Determine the formulas for multiplication and division of complex numbers. (If you don't remember them from high-school algebra, you can easily derive them from the basic definition of a complex number.)

a. Decide whether it's appropriate to implement those two operations as **`inline`** functions, the way we did for addition and subtraction, or whether it's better to code them in a separately compiled implementation file (like **`Complex.cpp`**). Justify your choice with persuasive reasons.

A convention for constants

The convention we introduced in Chapter 2 to conserve horizontal coding space and enhance readability has another use. If the class definition file contains

```
#define COMPLEX const Complex
```

we can then use that name for the conventional multiple-inclusion guard in the class definition source-code file.

```
#ifndef COMPLEX
    .
    .
#endif
```

b. Implement the compound assignment operators **`*=`** and **`/=`** for **`Complex`** objects.

c. Write the code to implement the simple arithmetic operators **`*`** and **`/`** in terms of the compound assignment operators. Explain why it's acceptable to code them **`inline`** even if you implemented **`*=`** and **`/=`** in a separate source-code file.

d. Does that complete the arithmetic operators for **`Complex`** objects? If not, then what additional arithmetic operators are needed?

3.2-8 Mathematicians don't consider complex numbers ordered. That is, given two complex numbers **`x`** and **`y`**, we can't say which is greater. We still need, however, to define the equality and inequality operators.

a. Write the code for the equality operator (**`==`**) as an **`inline`** nonmember function, using the accessors defined earlier. (Two complex numbers are said to be equal if and only if *both* corresponding parts are equal.)

b. Is it necessary to implement the inequality operator (**`!=`**)? If not, explain why. If so, implement it in terms of the equality operator defined in Exercise 3.2-8a and explain why it's a good idea to implement the function that way.

3.2.3 Supporting an external representation for complex numbers

Client programs can now use our **Complex** class for the full range of arithmetic and comparison operations, but how can they display the results? They might contain certain code such as this:

```
cout << "x = " << x;
```

But if **x** is a **Complex** object, the compiler will complain that no output-stream insertion (or left-shift) operator is defined for that type.

We need to choose an appropriate external representation and then define the overloaded operator to produce it. An easy representation is just a pair of numbers enclosed by parentheses, such as **(1.1, 2.3)** for **1.1+2.3i**. The following nonmember function will do:

```
inline ostream& operator<<(ostream& ls, COMPLEX rs)
   {return ls << '(' << rs.realPart()
              << ',' << rs.imagPart() << ')';}
```

Problems and exercises

3.2-9 Suppose users express a strong preference for the external form

```
1.1+2.3i
```

Rewrite the overloaded operator function to display the output in that form instead. This is trickier than the original version since you have to pay attention to the sign of the imaginary part and, in accord with common practice in mathematics, suppress either part entirely when it's zero. Should this version be **inline**?

3.2-10 We've taken care of stream output, but what about stream *input?* That's harder since the program doesn't have control of what it receives and since you'd have to parse punctuation. As we recall from Chapter 1, there are two schools of thought:

a. Anything that gets written to an output stream should also be readable from an input stream.

b. Complex input isn't worth the bother since the client program can easily read a pair of floating-point values and then invoke the **Complex** constructor.

Discuss the pros and cons of those alternatives with respect to complex numbers.

3.2-11 It's common for an overloaded output-stream insertion operator to be declared a **friend** of the class; so common, in fact, that a few textbooks advise the reader always to do so. Why is that practice unnecessary and undesirable for the **Complex** class?

3.2.4 Interactions between complex and real numbers

Mathematical formulas that involve complex numbers also typically involve real numbers. Given the class as we've developed it so far, we're pleased to discover that many such computations already work and yield the correct answer. That's because the compiler implicitly invokes the **Complex** constructor whenever it encounters a real number in a context where it wants a **Complex** object. Our constructor, taking a single parameter, defines a *conversion rule* for coercing **double** to **Complex**. Therefore, when a client program codes

```
double  x;
Complex w;
   .
   .
w *= x;
```

the compiler generates

```
w *= Complex(x);
```

which, because the constructor supplies a default parameter, is also

```
w *= Complex(x, 0.0);
```

That's welcome news after the drudgery of implementing all the methods; we don't have to do anything further to enable most mixed

real/complex operations. However, if we expect client programs to make heavy use of mixed-mode arithmetic, then we shouldn't stop there. When we know that the imaginary part is zero, the operations, especially multiplication and division, can be a lot simpler and faster. Perhaps reluctantly, then, for the sake of efficiency, we're obliged to implement another set of overloaded arithmetic operators taking one real and one complex operand. For the sake of efficient mixed-mode arithmetic, then, we'll end up with three simple multiplication operators as nonmember functions:

```
Complex  operator* (COMPLEX ls,  COMPLEX  rs);
Complex  operator* (COMPLEX ls,  DOUBLE   rs);
Complex  operator* (DOUBLE  ls,  COMPLEX  rs);
```

Even after we develop the mixed-mode arithmetic operators, one issue remains: Should the following code be legal and if so, what should it do?

```
x = w;   // Assign complex to real
```

C++ doesn't let us redefine the assignment operator (=) for a built-in type. Furthermore, an overloaded assignment must be defined as a member operator, so we need another approach. C++ supports an *inverse conversion* operator, which gives the compiler a conversion rule. You might write it as a member of **Complex**:

```
operator  double() const {return   rp;}
```

Note that this inverse conversion provides no additional functionality. It is, at best, a notational convenience compared with an explicit accessor call:

```
x = w.realPart()
```

You may prefer to make sure that the value is a real number:

```
operator  double() const
{assert (ip == 0.0); return   rp;}
```

Problems and exercises

3.2-12 Design and develop the full set of mixed-mode real/complex arithmetic operators that a client programmer might reasonably expect. Implement as many as possible in terms of others.

3.2-13 An advanced C++ textbook proposes a separate class for *pure imaginary* numbers like `-3.5i`, and then forms a **`Complex`** number class like this:

```
class  Complex {
   double    realpart;
   Imaginary imagpart;
     .
     .  };
```

In his **`Imaginary`** class, the author then declares these member operators:

```
Imaginary  operator* (const Imaginary)  const;
Imaginary  operator/ (const Imaginary)  const;
```

Are those sensible functions? If so, write the code for their implementation. If not, write declarations for what the author probably meant.

3.2-14 It is sometimes proposed that instead of using C's primitive floating-point numbers, we should use a real-number class, such as

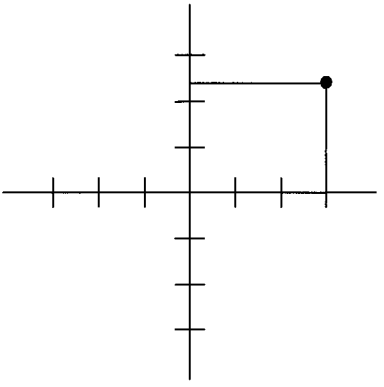
```
class  Real {
           double  value;
     public:
   .                              // and so on
   . };
```

a. What advantages would such a class yield? (Hint: Consider C++ templates.) What drawbacks would it impose?

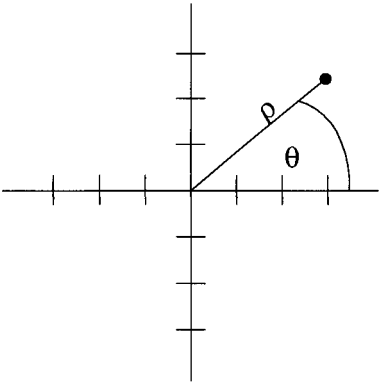
b. Write the code to implement the **`Real`** class, and modify the **`Complex`** class to interact properly with **`Real`**.

3.2.5 Polar coordinates

We can think of complex numbers as points in the complex *plane*, where the horizontal axis represents the value of the real part and the vertical axis the value of the imaginary part. Thus, the number `3.0+2.3i` is represented by the point shown below:



An alternative way to represent a complex number is in polar coordinates ρ and θ, where ρ is called the *magnitude* of the complex number (the distance from the origin to the point), and θ is the *angle* between the real axis and the line from the origin to the point.



Problems and exercises

3.2-15 Code accessor functions **`rho()`** and **`theta()`** to retrieve the values of the polar coordinates of a **`Complex`** object. Unlike the two accessor functions we saw earlier, these functions require you to compute a result since the only member data are still **`rp`** and **`ip`**. (If you don't remember the formulas, you

can derive them from basic plane geometry and elementary trigonometry.)

3.2-16 Some writers prefer to reserve the term *getter* (or *accessor*) for functions that retrieve a single member data item, performing no computation or transformation. Those writers would thus consider `realPart()` an accessor, but not `rho()`. Is that a sensible terminology convention? Why? (Hint: Consider Exercise 3.2-18.)

3.2-17 Suppose users of your `Complex` class want to be able to specify polar coordinates in a constructor invocation. Devise and describe a way to let them do so while retaining the two-parameter constructor (real and imaginary parts) that we already implemented. (Note: Requiring a third constructor parameter to indicate which coordinates are being given is too awkward and user-unfriendly. Think of an object-oriented solution.)

3.2-18 While most complex number calculations are faster in rectangular (`rp, ip`) coordinates, a few (which ones?) are faster in polar coordinates. Prepare an alternative version of the `Complex` class, storing polar coordinates `rho` and `theta` as the member data instead of `rp` and `ip`. The criterion for success is that any client program that uses `Complex` can switch from one version to the other, recompile, and still produce identical results.

3.2-19 Explain how the preceding exercise demonstrates the practical value of hiding the internal representation of objects from client programs. Discuss the impacts of our earlier decisions to implement some `Complex` methods in terms of others.

3.3 Packaging and using the `Complex` class

If you've followed the discussions so far, you can now put together a complete, usable, and robust `Complex` class. On the Website www.dorsethouse.com/books/ooc.html, there are three complete C++ implementations of the `Complex` class:

- Version 1 is a slightly naïve version that an intelligent but inexperienced C++ programmer might create.

The class definition file, **ComplexV1.hpp**, is self-contained, and all functions are **inline**. The compound assignment operators (**+=**, and so on) are defined in terms of the corresponding simple arithmetic operator—an obvious but inefficient choice.

- Version 2, **ComplexV2.hpp**, switches to internal polar representation. (This is a solution to Exercise 3.2-18.)
- The final version, **Complex.hpp**, returns to Cartesian representation and makes three improvements on Version 1:

    - The larger methods are moved to a separate implementation file, **Complex.cpp**.
    - The simple arithmetic operators are defined in terms of the corresponding compound assignment operators.
    - An additional constructor allows users to specify polar coordinates, but depends upon another class. (This is a solution to Exercise 3.2-17.)

This version is suitable for production use. If you can improve upon it, then by all means do so.

The effort we spent in preparing our **Complex** class was formidable. We probably couldn't justify that investment for any single project, but now that we've packaged our class, our colleagues on other project teams can begin to exploit its benefits. This high-multiplier effect makes object-oriented classes, even simple ones, attractive to a software development organization.

Problems and exercises

3.3-1 Many FORTRAN textbooks present problems that require complex arithmetic. Find one and recast some of the solutions in C++. Discuss the relative strengths and weaknesses of the two languages (or any other languages you know) for such problems.

3.3-2 Many C++ class libraries provide a complex number class. Find one and examine how it differs from the one we developed. Discuss where you prefer one approach to the other.

3.3-3 Advocates of Extreme Programming (XP) advise us to implement only the functionality we need for the immediate problem at hand. They call this the YAGNI (You Aren't Going to Need It) principle.

Suppose we're developing an application that needs to add and multiply complex numbers, but doesn't ever subtract or divide them. Should we omit the overloaded subtraction and division operators from the initial operational version of our **Complex** class and plan to add them later, when we need them?

3.3-4 In PL/I, you can do operations on structures on an element-by-element basis without defining any *overload* rules. Thus, if **a**, **b**, and **c** are compatible (for example, **Complex**) structures, the expression **c = b + c** will yield the expected result. Would that facility be useful in C++? What drawbacks would it present?

If you thoroughly understand the process of designing and developing the **Complex** class, you can safely skip the following section.

3.4 Some other pure numeric classes

3.4.1 Rational numbers (exact fractions)

Suppose we have to compute 5.5 percent of an account balance prorated for four months (one-third of the year), and your accountant auditors insist that the result be exact to the penny. Since neither .055 nor 1/3 has an exact, nonterminating floating-point binary representation, we're uncomfortable using floating point for either the interest rate or the fraction of the year.

Of course, you could customize the fractions into an expression such as this:

```
(balance * 55) / 3000
```

But that's both awkward to change and confusing to read. Consider this alternative:

```
Fraction interestRate(55,1000); //  5 1/2%
                .
balance * interestRate * Fraction(nDays, 365);
```

Suppose the **Fraction** class maintains a numerator and a denominator as **long** integer member data:

```
class Fraction {
      long numerator, denominator;
      .
      .
};
```

Then client programs will be able to represent and use in computation any rational number, as long as neither numerator nor denominator exceeds the range of a **long** integer (32 bits in many implementations, 64 bits in some).

Problems and exercises

3.4-1 Develop a complete, efficient, and robust **Fraction** class. The design of **Fraction** is similar to that of **Complex** in many ways, but with a couple of complications. You have to decide

- whether to reduce the fraction to lowest terms after every operation (for example, should the **interestRate** we specified earlier as 55/1000 be stored initially as 11/200?)
- how to represent a *negative* fraction (denominator always positive, for example)
- what to do if an integer overflow occurs in either part
- whether to support interactions with floating-point numbers, and if so, how

Hints

- You'll probably need functions to compute the greatest common divisor of two integers (**gcd(m,n)**) and perhaps also their least common multiple (**lcm(m,n)**). But those functions are not logically part of the client program interface; that is, users of your class don't need to compute those functions themselves as part of the behavior of **Fraction** objects. Therefore, you'll either have to find them in an accessible library or implement them as **private static** class methods.

- Upon overflow, you can **`throw`** an exception, abort the run (**`assert`** macro), or ignore the problem. Choose on the basis of the likelihood of such an occurrence and the likely preference of your client user community.
- Since rational numbers, unlike complex numbers, are ordered, you'll need to support all the relational operators.

3.4.2 Decimal numbers

Some programming languages (COBOL, PL/I) support fixed-point non-integer decimal arithmetic. Business applications programmers traditionally represent amounts of money as a fixed decimal with two places to the right of the point. This practice was so widespread that a number of computers (such as the IBM 360 and successors) supported a decimal number format in the hardware (or microcode).

Problems and exercises

3.4-2 If you're not already acquainted with IBM's "packed decimal" data format, consult a reference to learn exactly what it is. Then develop a complete **`PackedDecimal`** class using that internal representation.

3.4-3 Design and develop a **`FixedDecimal`** class that associates the appropriate scaling (the number of decimal places to be carried) with each object. You can use the packed decimal internal representation in the previous exercise or choose a simpler one.

3.4-4 Assess and discuss whether it's practical to use either **`PackedDecimal`** or **`FixedDecimal`** in a business application that processes a high volume of transactions.

3.4.3 Integers of unusual sizes

In simulating various digital devices, especially older computers, we need to manipulate binary integers of unusual sizes. A number of computers of the first and second generation had word sizes of 36 bits or 48 bits. Some of them used "sign-magnitude" representation for negative numbers, others used "1's complement" or "2's complement."

To simulate such machines, we must perform integer arithmetic, shifting, comparison, and so on, and get bit-for-bit the same results as the original hardware. We must also be able to detect overflow.

Problems and exercises

3.4-5 Find a programming manual or other specification for a historical computer that used binary integer data that was a different size than the native sizes of the C++ implementation you're using. Determine how that computer performed integer arithmetic operations. Develop a complete C++ class to emulate those operations and produce exactly the same results.

3.4-6 Using the results of the previous exercise, develop a complete interpretive simulator for that computer. Naturally, you'll have to decide how to represent the target computer's memory and registers as well as any console switches and external devices.

In addition, you'll confront a couple of troublesome issues involving data formats:

- If the simulated machine had built-in floating point, you may have to go to a lot of trouble to simulate it accurately.
- The simulated machine may have used a different set of character codes, perhaps 6-bit. You may have to map characters when they're involved, for example, in output to a simulated printer.

3.5 Java equivalents

3.5.1 Constructors and accessors

We can model a Java **`Complex`** class on our final C++ version, making just those changes needed to comply with Java syntax and to circumvent Java restrictions. We begin, as with the C++ version, with a rough outline:

```
public class  Complex {

// Member data
// -----------

 double rp;          //  Real part
 double ip;          //  Imaginary part

//  Constructors and accessors
//  --------------------------

public Complex(final double rp, final double ip)
  {this.rp = rp; this.ip = ip;}

public Complex(final double rp)
  {this(this.rp, 0.0);}

public Complex()
  {this(0.0, 0.0);}

public Complex(final Complex x)
  {this(x.rp, x.ip);}

public  double realPart() {return rp;}
public  double imagPart() {return ip;}
     .
     .
}
```

Since Java doesn't support default function parameter values, we have to provide three constructors instead of the original one. We'll also need an explicit copy constructor. Note that

- three of the constructors invoke the first, via **this**
- the parameters have the same name as the member data items, disambiguated by **this**, a common Java idiom

For converting to external representation instead of C++'s overloaded output-stream operator (<<), we code a **toString()** method in the obvious way:

```
public String toString() {return '(' + rp +
                                 ',' + ip + ')';}
```

Remember that Java interprets the + sign as concatenation when either operand is a **String** or a **char**.

Problems and exercises

3.5-1 Instead of a copy constructor, some Java enthusiasts favor a `clone()` method.

a. Compare the two approaches in terms of ease of client program coding, ease of future change, and efficiency.
b. Modify the class definition to implement Java's `Cloneable` interface

3.5.2 Arithmetic and comparison operators

Since we know that Java doesn't support operator overloading, it will be impossible to support ordinary expression syntax with `Complex` or any other Java numeric class. Yet a `Complex` class will be useless if users can't manipulate its objects in the usual mathematical ways. Therefore, we're obliged to give client programs the ability to do such manipulation in the most natural and convenient way that Java will permit.

Chapter 2 suggested a mapping of operations to conventionally named functions. If you know of a more natural and convenient convention, then follow it. Otherwise, use the one in Chapter 2.

Problems and exercises

3.5-2 Complete the `Complex` class definition in Java. Develop and run a thorough test driver (`main`) program, either in the same class or in a separately compiled public class file.

3.5-3 Find and examine an existing complex number class in a Java library. Assess how it is better or worse than the one we developed here, from the points of view of ease of use, functionality, and flexibility.

3.5-4 Choose one of the *other* C++ pure numeric class examples from Section 3.4, and implement a close equivalent in Java.

3.5-5 Many FORTRAN textbooks present problems that require complex arithmetic. Find one and recast some of the solutions in Java. Discuss the relative strengths and weaknesses of using the two languages (or any other languages you know) to express complex arithmetic for such problems.

This page intentionally left blank

INDEX