Object-Oriented Computation in C++ and Java

DH *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: 0-932633-58-7   Copyright © 2004   312 pages, softcover

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

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

More Secrets of Consulting: The Consultant's Tool Kit
by Gerald M. Weinberg
ISBN: 0-932633-52-8   Copyright © 2002   216 pages, softcover

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

The Psychology of Computer Programming: Silver Anniversary Edition
by Gerald M. Weinberg
ISBN: 0-932633-42-0   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: 0-932633-45-5   Copyright © 2005   400 pages, softcover

Waltzing with Bears: Managing Risk on Software Projects
by Tom DeMarco and Timothy Lister
ISBN: 0-932633-60-9   Copyright © 2003   208 pages, softcover

For More Information

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

✔ Sign up for *DHQ: The Dorset House Quarterly* in print or PDF.

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

✔ Visit Dorsethouse.com for excerpts, 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

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.

Trademark credits: All trade and product names are either trademarks, registered trademarks, or service marks of their respective companies, and are the property of their respective holders and should be treated as such.

Cover Design: Nuno Andrade

Copyright © 2006 by Conrad Weisert. Published by Dorset House Publishing, 353 West 12th Street, New York, NY 10014.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

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:

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

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, in participating in large-scale application development, and in 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 the new and still shaky C++ language. Since 1990, various clients have provided platforms for me 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.

It used to be customary to thank colleagues for their help in proof-reading and correcting errors, but people are busy these days. 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, but the responsibility for mistakes is entirely mine. My former IIT student, **Vijayram Gopu**, was helpful in reviewing early drafts of several chapters.

The skillful and patient editors at **Dorset House Publishing** have caught a number of typos and assured consistency of style.

Readers who discover errors of any kind should let me know at **cweisert@acm.org**. If the volume or seriousness demand, I'll post corrections and discussion on my company Website, **www.idinews.com**.

CONTENTS

PREFACE

Object-Oriented Computation in C++ and Java fills a gap in the literature of object-oriented programming. 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]

This book is about an important subset of application domain data: *numeric* data items. Numeric data are central both to most

[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, if not more.

business applications and to every engineering or scientific application.

For over a dozen years, I've been teaching courses in advanced object-oriented programming. My students have backgrounds in both commercial/business applications and scientific/engineering applications. In searching for a suitable textbook, I found none that adequately treated application-domain objects.

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 developed a large collection of course handout material, part of which has evolved into this book.

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 it covers an area neglected by most OOP textbooks, it also serves well as a supplementary text in a survey course in object-oriented programming for computer science majors.

Object-Oriented Computation in C++ and Java

Conrad Weisert

INTRODUCTION

I.1 Your background

This book is for experienced programmers. You 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 the cost of its future maintenance as well as by robustness, efficiency, ease of use, and potential reuse
- to be *highly productive*, solving problems in far less time than the average programmer

- to exercise *creativity and originality*, developing non-obvious 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 relevant 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 this chapter.

Chapters 3 through 6 examine particular categories of numeric data that appear in real-world applications. We cite common patterns, starting with the simplest *pure numeric* classes, and build up to families of interacting related classes. We 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 we develop in this chapter make heavy use of inheritance and polymorphism.

Problems and exercises

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 appendices contain the source code listings for most of the examples and suggested answers to selected exercises.

I.3 Methodology independence

Every program that performs nontrivial computation requires 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 C.A.S.E. 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 C++ programmers 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, we convert the most important examples to Java, noting the main differences between the two languages. The appendices contain source code in both languages.

The principles also apply 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-assisted software engineering.

1 Numeric Objects in Context

1.1 Data and objects

This chapter is about data. A solid understanding of data is not only vital to applying the object-oriented paradigm; it is a valuable aid to *all* kinds of data analysis and programming. A programmer who tries to develop application software without mastering these concepts is at a serious disadvantage.

We first examine the top levels of the natural hierarchy or *taxonomy* of data types. This natural taxonomy is not directly supported by any programming language, nor have the names of the types been legitimized by any standards body or other influential organization. Don't try to figure out which built-in data type in C or any other programming language corresponds to each of the natural types described here; just try to understand what they are and how they differ from one another.

We also introduce a lot of terminology, sometimes multiple terms for the same thing. Occasionally, one term describes two different things. Terminology conflicts are unfortunate—and annoying—but they arose as various branches of information technology evolved independently. The inconsistencies are now such well-established conventions that we have to put up with them.

Finally, we look at data representation, drawing a firm distinction between what a data item *is* and what it *looks like.*

1.2 Application-domain data

Application-domain data represent objects in the real world. They fall into two categories:

1. *Elementary* data items are not meaningfully decomposed into independent components. Alternative names for elementary data items include

    - *element*
    - *field* (within a composite data item)
    - *attribute* (of a composite data item)

2. *Composite* data items are fixed arrangements of other independent data items (components), which can themselves be either elementary or composite. Alternative terms for composite data items include

    - *record,* the common term in data processing and in Pascal programming
    - *dataflow,* the usual term in structured systems analysis
    - *struct,* in C programming
    - *group item,* in Cobol programming
    - *control block,* in operating system internals

Both kinds of application-domain data are common in real-world applications and are also well suited to object-oriented concepts and techniques.

We sometimes include a third fundamental category in the application domain:

3. *Container* data items are data structures that act as receptacles for other data items, which may be elementary, composite, or (rarely) other containers. Alternative terms for container data items include

    - *collection,* the usual term in the Java programming community, which reserves "container" for graphical user interface (GUI) objects, such as frames and windows

- *data structures,* the usual term in theoretical computer science, especially for dynamic containers that can change their size and shape during execution.

The data items stored in a container are called "elements" of the container, whether they're elementary or composite. A homogeneous container can hold elements of one type; a heterogeneous container can hold elements of multiple types.

Problems and exercises

1.2-1 Are the three fundamental categories of data sufficient to accommodate *pictures* or *audio* information? If so, into which of the three categories do pictures and audio fit? If not, how should we extend the top-level taxonomy?

1.2-2 Some writers on object-oriented technology prefer the term "business objects" over "application domain data." Is that term more or less descriptive? What does it imply about the writers' views of the role and the importance of such data?

1.3 Non-application-domain data

As your programming experience no doubt has shown, computer programs also manipulate many data items that correspond to nothing in the real world of the application. Programmers used to call such items "housekeeping data." Today, these items play a far greater role than that term implies. Here are some common examples:

What about pointers?

Pointers aren't really data at all, and they rarely represent anything in the application domain. They serve mainly as a mechanism for representing relationships among data items.

- graphical user interface (GUI) objects, such as screen windows and forms
- tables used to describe properties of other data
- program execution artifacts, such as user sessions
- initialization switches, record counters, check sums, end flags
- flags and semaphores used to synchronize concurrent processes

Non-application-domain data include the same three fundamental types that we encounter in application-domain data. Although this book is strictly about application-domain data, you'll find some of the concepts and techniques we'll be examining applicable to some non-application-domain data.

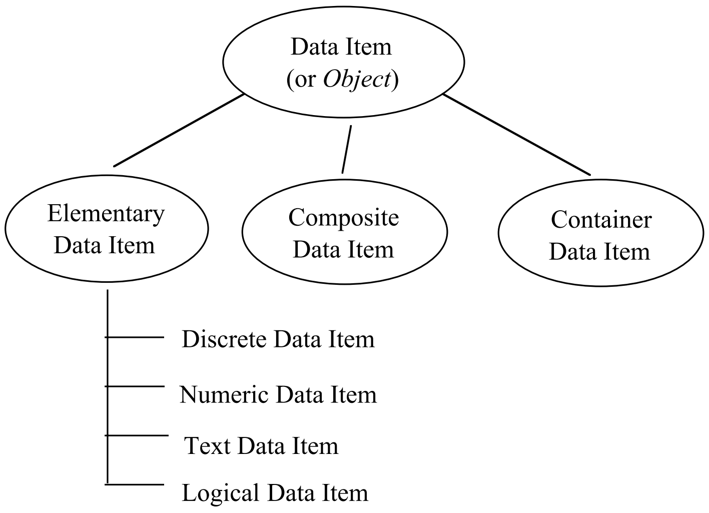
1.4 Four basic types of elementary data

The three fundamental categories are just the top level of a complete taxonomy of data types. We can further divide each of those three categories into useful families of data types. In particular, every *elementary* data item belongs to one and only one of these four basic elementary types:

1. A *discrete* (or *enumerated* or *coded)* data item takes on one from a set of possible values. Discrete data items often serve as identifiers (`productCode`) or state data (`maritalStatus`).
2. A *numeric* data item is one upon which some arithmetic operation is meaningful. Numeric data are the main focus of this book.
3. A *logical* (or *Boolean*[1] or *option)* data item takes on one of two possible truth-values (true/false, yes/no, on/off, present/absent, and so on).
4. A *text* (or character-string) data item is a sequence of characters. Text data items often serve as names of entities (people, companies, cities) or are used for communication in a natural language (messages, letters, dialogues).

Here's the taxonomy so far:

[1]After the British mathematician George Boole (1815-1864), who codified and popularized Boolean Algebra.



Problems and exercises

1.4-1 Many programmers whose first language was C or Java think of *text* data as *containers* of characters rather than as elementary items. Programmers whose first language was PL/I, Cobol, or Basic do not share this view.

a. Explain what you think accounts for these conflicting views.
b. Discuss the pros and cons of the two views, and their likely impact on database design and program structure.

1.4-2 Some programmers point out that we can view *logical* data as a special case of *discrete* data having only two possible values. Based on what you know now, would that complicate or simplify software design?

1.4-3 A `street address` composite item contains a `city` field and a `state` field. A designer has determined that `city` is a text data item while `state` is a discrete data item. Explain why the designer's determination is reasonable.

1.5 Avoiding false composites

Beginners sometimes confuse components of a *mixed-base* numeric representation with fields of a composite item. For example,

```
struct Weight {
    long  pounds;
    short ounces;};
```

Is a `Weight` object a composite data item or an elementary numeric data item? The answer is clear from the definitions: `pounds` and `ounces` aren't independent components of a weight object; together they express the internal representation of a single data item. A weight object, then, is an *elementary numeric* data item, no matter how we choose to represent it in a computer.

A more subtle case is a `Date` object. We may choose to represent a date as three components of the traditional Gregorian calendar representation, `year`, `month`, and `day`. Here again we normally consider a date to be a single elementary numeric data item, even though some programs that perform calendar manipulations may extract and apply special significance to *one* of those components. Chapter 5 explores date representation and date manipulation in depth.

Avoiding false numerics

When we determine that an elementary data item belongs to one of the four basic types, we're specifying what that item is, not what it looks like. *A U.S. postal ZIP code, for example, is represented by a sequence of numeric digits, but it is a discrete data item. Many false numerics have the word "number" in their data name, such as* `accountNumber`. *Since it would be non-sensical to perform arithmetic on ZIP codes or account numbers, they are not numeric data items.*

Some old-fashioned programming languages and tools, such as Cobol *and* Oracle, *use representation-based data declaration, rather than type-based. They encourage, but don't require, the designer to choose among such predefined pseudo types. As a result, many programs and data bases developed with those tools specify discrete data items as "numeric."*

Note that the old-fashioned term alphanumeric *never denotes a data type.*

1.6 Numeric data representation

1.6.1 Choosing the unit of measure

For each of the following kinds of numeric data item, which of the alternative units is better, and what other representations are worth considering?

Data item type	**Representation A**	**Representation B**
Weight	pounds and ounces	grams
Time of Day	hour, minutes, and seconds*	seconds since midnight
Temperature	degrees Fahrenheit	degrees Kelvin
Length / distance	miles, feet, and inches*	meters

* normalized

Experienced software designers understand that neither choice is "better." In a data-entry form or a report, many American end-users prefer the familiar representations in column **A**. Inside programs and databases, on the other hand, most *programmers* opt for the simpler representations in column **B**.

Thus, for most real-world data items we need *two* representations, not just one:

- An ***external data representation*** appears in anything seen by end-users, such as reports, input forms, inquiry displays, and shipping labels.
- The ***internal data representation*** appears in internal computer entities that are never seen by end users, such as programs, databases, master files, and work files.

1.6.2 Other properties of numeric data representation

In addition to the unit of measure, we have to specify

- the range of values
- the precision

The range is defined by a pair giving the minimum and maximum values. In a payroll system we might define the range of `hourlyWage` as `<$6.50, $90.00>` and `noOfDependents` as `<0, 18>`. The precision is the smallest significant change in value, for example, `1/2 cent` for `hourlyWage`, **1** for `noOfDependents`.

Problems and exercises

1.6-1 Some old-fashioned data-dictionary tools call for the *size* of a numeric item (the number of digits needed to contain it, in other words). Is that property equivalent to the range? If not, how do you suppose that tradition got started?

1.6.3 Criteria for external and internal data representations

The criteria for choosing internal representations are entirely different from the criteria for choosing external representations. External data representations must be

- *familiar* to the users of the application
- *not error-prone* (for input)

while internal data representations should be

- *simple*
- *efficient* (especially in terms of space)
- *standardized* for interchange among programs and organizations

It would be wrong to force end-users to cater to developers by adopting representations they don't encounter in their everyday work. It would be equally wrong to ask the programmers to deal repeatedly with messy "traditional" representations.

1.6.4 Object-oriented implementation of the representations

You should *always* draw a clear distinction between internal and external data representation, and object-oriented programming helps you to do so in a natural and systematic way. The internal representation of an object corresponds to the object's *member data*. The access rules of C++ and Java let us make sure that knowledge of the private internal representation is known only to a few closely related parts of the program.

Thus you can design a `Weight` class like either of these:

```
class Weight {
  long  pounds;
  short ounces;
    .
    .
  };
class Weight {
  double grams;
    .
    .
  };
```

and the only programs that will know which you chose will be some of the member functions of `Weight`.

An object-oriented programmer can also control *external* representations in a variety of ways. We will explore these later.

Problems and exercises

1.6-2. In the 1960's, systems analysts often confused *simple* with *familiar*. Cite two or three traditional and very familiar numeric data representations for which it's complicated to perform arithmetic on or to compare two data items.

1.6-3. In the late 1990's, vast efforts were expended on the so-called "Y2K crisis."

a. Discuss how that crisis arose and who, if anyone, was to blame for it.

b. Explain why some organizations had no Y2K troubles at all with their internally developed applications.

1.6-4 The 1990's saw the emergence of a new protocol for data interchange, *Extensible Markup Language* (XML). If you're not acquainted with XML, consult an introductory tutorial on it. Then discuss (a) how XML either supports or undermines the distinction between internal and external data representations, and (b) its likely impact upon program and database design.

1.6.5 Converting between internal and external representation

Although it's the programmer's responsibility to convert between external and internal representations of a data item, we don't think of that as an extra burden but rather as a simplification. In any large program or suite of programs only one or two functions need to have knowledge of an external representation. If the program is object oriented, then the internal representation is hidden from the rest of the program, and manipulation of the data items takes place through the public *client interface*.

1.6.6 Internal-to-external conversion (output)

Whenever a program needs to display a data item for the end user, either on a screen or on a printed report, the program must convert the internal representation to a suitable external representation. C++ and Java provide simple and elegant facilities for generating a standard or *default* external representation for a given type of data item.

C++ extends the meaning of the `<<` operator (originally C's *left shift* operator) as the *output-stream insertion* operator. Whenever we define a new class of data item, we can extend the meaning of that operator to convert data items of that class to any desired external format.

Consider a simple `Date` structure that a C programmer might define like this:

```
struct  Date {
      int  year;
      int  month;
      int  day;};
```

Now suppose the programmer codes this:

```
Date  dateHired = {1985,12,5};
      .
      .
rptFile << dateHired;
```

Since left-shifting makes no sense for a `struct`, the compiler would normally complain about an illegal structure operation. But if we first define an overloaded `<<` operator that takes an output stream as its left-side operand and a `Date` object as its right-side operand, the compiler will invoke it:

```
ostream& operator<< (ostream& ls, const Date rs)
  {return ls << rs.year <<   -   << rs.month
                          <<   -   << rs.day;}
```

Then the program would be compiled correctly and would display

```
1985-12-5
```

on the `rptFile`.

Java's equivalent facility is not tied to stream output but to conversion from internal form to a character string. When the Java compiler sees an object in a context where it wants a character string, it looks for a class member function with the name `toString` and generates code to invoke it. For example, Java interprets a call to the library output-stream function

```
System.out.print(dateHired);
```

as if the programmer had coded

```
System.out.print(dateHired.toString());
```

That would produce the desired result if the programmer had included the following member function in the `Date` class:

```
public  String  toString(final Date x)
  {String result = x.year +   -   + x.month
                          +   -   + x.day;
   return result;
  }
```

In Chapter 2 we shall review in more detail the rules for such functions in both C++ and Java. Meanwhile, the above will serve as a model. The point here was just to show that

- a default version of the conversion from internal to external representation is easy to code,
- it can be localized to a single place, and
- you have full control over what it does. Of course, you're always free to write more specialized versions when you need them.

Problems and exercises

1.6-5 The naïve C++ example was shown for a `struct` in which all members are publicly accessible rather than for a `class` that restricts access to the member data. What should we do differently to make the output-stream insertion operator work correctly without revealing the internal `Date` representation to the whole world?

1.6-6 The date conversion functions shown earlier produced the format "`1985-12-5`". Suppose users complain that the lack of a leading zero on the day or the month portion makes a columnar display look ragged and messy. They want to see 10 characters for every date (for example, "`1985-12-05`"). Modify either the C++ overloaded output-stream operator or the Java `toString()` function to satisfy those users. (This is a rather trivial exercise to illustrate the flexibility of localizing such conversions.)

1.6-7 Suppose it's decided that the default external output `Date` should be in the American English style, for example "`December 5, 1985`." Rewrite either the C++ overloaded output-stream operator or the Java `toString` function to satisfy that requirement.

For this version, it's obvious that we'll need a table of month names. The interesting design question is how to package that table and where to put it. Is the output conversion function the *only* function that's likely to need that table? We shall return to this example in Chapter 5. (Don't even think about the crude beginner's technique of implementing a table as a `switch .. case` flow-control construct.)

1.6.7 External-to-internal conversion (input)

While converting to an external representation is easy, converting *from* an external representation is usually much more difficult. It's easy to see why: We know exactly what the internal representation is and we can be confident that it's a valid value, but an external representation coming from, say, keyboard input may take a variety of forms and may exhibit many kinds of errors.

C++ experts are divided between two schools of thought:

- Some experts insist upon symmetry between input and output. Anything you can write to an output stream, you should be able to read back later from an input stream.
- Others concede that it's often impractical and inefficient to support such generality. They rely instead on the application to provide suitable input editing functions.

This book leans toward the second view, both because it's less work and also because many applications have no real need for a general input function for each type of data. We may want to *display* amounts of money with dollar signs and group separators (such as "`$1,202,499.20`"), but few if any data-input programs would need to read that format.

Definition

An **input-editing program** (or function) does two things:

1. It **converts** external data representations into internal representations.
2. It **validates** that the data item has a legitimate value.

Experienced programmers know that thorough input editing is essential in every application that gets data from an outside source, such as from a keyboard. In a large application, the input-editing functions greatly simplify the rest of the programs. Once the input data have been edited, the rest of the programs not only deal with the simpler and more efficient internal representation, but can also assume that values are valid. A computational function that gets a `Date` object parameter, for example, needn't check to verify that the month number is between 1 and 12.

Not an editor

An input-editing program is specific to an application or to a type of data. It is not the same as an editor (or "text editor"), the kind of program you've no doubt used to compose program source code.

Problems and exercises

1.6-8 Discuss how the distinction between internal and external data representation affects international application software that's designed to be used in many countries.

2

REVIEW OF C++ AND JAVA FACILITIES AND TECHNIQUES FOR DEFINING CLASSES

2.1 To the reader

This chapter is not a language tutorial. I assume you already have experience in defining object-oriented classes in C++ or Java or both. The emphasis here is on

- the *choices* we face among language facilities that have duplicate or overlapping functionality,
- the *background* of various traditions in C++ and Java programming, and
- established principles of *good programming practice* as they apply to building and using object-oriented classes.

Unlike later chapters, the following sections integrate corresponding topics in the two languages. Even if you have absolutely no immediate interest in one of the languages, you should resist the temptation to skip over those explanations. By understanding the fundamental approaches in C++ and Java and the differences between them, you'll develop a stronger command of object-oriented class design and an informed appreciation of the strengths and weaknesses of each language.

2.2 The basic goal—a major difference between C++ and Java

C++ encourages us to minimize the differences between built-in or *primitive* data and instances of user-defined classes. Bjarne Stroustrup, the principal designer of C++, advises language designers: "Provide as good support for user-defined types as for built in types."[1] Numeric data type classes are especially suited to such consistency because of the natural way in which programs manipulate them using C's rich set of operators.

Consider this program fragment, valid in C, C++, and Java:

```
double  creditLimit;
double  unitPrice   = 49.95;
double  totalPrice = 0;
int     quantityOrdered;
        .
        .
totalPrice += quantityOrdered * unitPrice;
if (totalPrice > creditLimit)
        .
        .
```

Now suppose we later discover or develop a `Money` class that supports everything a program might do to amounts of money and also alleviates auditors' anxiety about floating-point rounding error. What would we have to change in the above example to exploit the `Money` class?

In C++ we'd change only the type name in the three declarations:

```
Money  creditLimit;
Money  unitPrice   = 49.95;
Money  totalPrice = 0;
```

If the `Money` class supports the basic goal, then the executable statements will require no change at all. We wouldn't even have needed to change the three declarations if we'd had the foresight, as experienced C programmers do routinely, to localize the original choice of primitive type:

```
typedef double Money;
```

A basic goal in C++ for both the language itself and for anyone designing a class is the following:

[1]Bjarne Stroustrup, *The Design and Evolution of* C++, 3rd ed. (Reading, Mass.: Adison-Wesley Professional, 1994), p. 117.

Objects, especially elementary data types, should behave as much as possible like built-in primitive data.

In Java, it's just the opposite!

Java is actually two distinct expression languages in one package: one for manipulating *primitive* built-in data items, and a separate language for manipulating objects or *reference* data items. They are different in almost every way.

To change the program fragment to exploit a Java `Money` class, we'll need to change every statement that refers to a `Money` data item. The result might look like this:

```
Money  creditLimit;
Money  unitPrice  = new Money(49.95);
Money  totalPrice = new Money(0);
int      quantityOrdered;
        .
        .
totalPrice.addSet(unitPrice.mpy(quantityOrdered));
if (totalPrice.greaterThan(creditLimit))
    .
```

In Java, then, elementary objects behave *differently* from built-in primitive types in almost every context.

Although it's tempting to complain about this or even to argue against using Java for computation, we shall not do so in this book. Java's designers believed they had valid reasons for rejecting the C++ basic goal, and organizations often have valid reasons for choosing to develop applications in Java. We shall focus on making the best use of the facilities that Java does support, and we'll leave the language arguments to other forums.

In either C++ or Java you have to go to a lot of trouble to design and develop a robust and complete class for `Money` or any other numeric data type. It's marginally worth doing so for a single program or a single project. What justifies the effort is the huge multiplier that results from using those class definitions in every program developed in your organization or even in multiple organizations. Once such a class is developed, packaged, and distributed, that problem is solved forever.

Problems and exercises

2.2-1 Many Java programmers and some C++ programmers forgo defining classes for numeric data. Instead, they just use `double`, `int`, or another built-in primitive type. The executable statements are then similar to those in Fortran, C, or another procedural language. Discuss the pros and cons of that approach. Consider ease of coding, ease of debugging, ease of change, readability, reliability, and efficiency.

2.2-2 Other programmers go to the opposite extreme, defining "wrapper" classes, so that the numeric objects are bona fide *objects*. Then, instead of supporting operators and other functions to operate on the object, they provide *accessor* and *modifier* functions to retrieve and store the internal representation, and perform their operations on the built-in primitive value. The executable part of the `Money` example might look like this in either C++ or Java:

```
totalPrice.setValue(totalPrice.getValue()
  + quantityOrdered * unitPrice.getValue());
if (totalPrice.getValue() > creditLimit.get.Value()
     .
     .
```

Discuss the pros and cons of that approach.

2.3 Constructors and destructor

2.3.1 Purpose

A constructor is a function that is called, either explicitly or by compiler-generated behind-the-scenes code, for the purpose of initializing the state (or member data items) of an object. It is given raw uninitialized memory of the object's size. It can, of course, explicitly allocate memory for non-contiguous fields, but the compiler considers only the resulting pointer to be part of the actual object.

In both C++ and Java, a constructor is written as a function that has the same name as the class. It returns no value, not even `void`.

2.3.2 C++ constructors

C++ constructors are invoked in any of five ways:

- Declaring objects of the class, with or without initialization parameters:

  ```
          Complex x(2.5,-1.0), y, z;
  ```

- Declaring and initializing, using C syntax:

  ```
          Money  price = 49.95;
  ```

- Explicitly creating an unnamed temporary object:

  ```
          z = x + Complex(1.0, 1.0);
  ```

- Implicitly creating an unnamed copy of the object:

  ```
  Complex  fctn(Complex x) //  for the parameter
          {return  expr;}  //  and for the result
  ```

- Implicitly converting:

  ```
  price += 1.50;  //   calls single parameter constructor
  ```

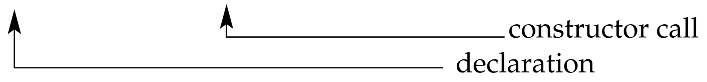
C++ constructors, like other functions, can specify default values for optional trailing parameters. This sometimes lets us avoid coding multiple constructors for a class:

```
class Complex {
   double rp, ip;
public:
   Complex(const double x=0.0, const double y=0.0)
           : rp(x), ip(y) {}
                .
                .
```

2.3.3 Java constructors

Java constructors, on the other hand, are always invoked explicitly, as the operand of a `new` operator, which allocates the memory for the object.

```
Money  price = new Money (49.95);
```



Java doesn't support default parameter values, but one constructor can invoke another constructor for the same class. To clarify that this is happening, we use the reserved word `this` instead of the class name:

```
public class Complex {
      double rp, ip;
public Complex(double x, double y) {rp = x; ip = y;}
public Complex(double x)           {this(x  , 0.0);}
public Complex()                   {this(0.0, 0.0);}
```

Java's `this` keyword does further duty by relieving us from having to think up names for constructor parameters that correspond to member data items. The first constructor above can then be written:

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

2.3.4 Special constructors

In both languages, a constructor with no parameters is called the *default* constructor and a function that takes a single parameter of the same class is called the *copy* constructor. In C++, of course, the copy constructor's parameter must be a *reference:*

```
Complex (Complex& x) : rp(x.rp), ip(x.ip) {}
```

Otherwise, the copy constructor would be invoked recursively to try to pass the parameter by value.

In C++, we sometimes have to specify a default constructor even when we don't want to give clients the ability to create an object without specifying an initial value. That's because when we create an array, C++ must create objects to fill it:

```
Money  priceTable[100];
```

Here the default constructor for `Money` will be invoked 100 times to initialize the array. For a numeric class, that's usually acceptable, since there's some value we can consider a default, usually zero.[2] Note that this doesn't occur in Java; see Chapter 8.

A constructor with parameters that all have a default value is an acceptable default constructor. For example,

```
Complex(const double x=0.0, const double y=0.0)
```

[2]`Date` is the only exception; see Chapter 5.

2.3.5 C++ Compiler-generated functions

Three functions are automatically generated by the compiler whenever the class definition omits them:

- the copy constructor
- the destructor (see Section 2.3.6)
- the assignment operator

These generated versions work just fine for *contiguous* objects, that is, wherever all the component data items belonging to an object lie *inside* the object itself. Since, except for the arrays in Chapter 8, and a simple character string class (see 2.7), nearly every object we shall use in this book is contiguous, we shall routinely let the compiler generate those default versions. As a courtesy to the future maintenance programmer, however, we customarily affirm in commentary that the omission was not an oversight:

```
//  The compiler will generate an
//  acceptable copy constructor
//  destructor, and assignment operator
```

2.3.6 C++ destructor

We do need to code an explicit destructor whenever the constructors allocate a resource such as memory. If a constructor allocates memory, the destructor for that class must free it. Furthermore, if *any* constructor for a class allocates a resource, then *all* constructors for that class should do so, unless some complicated scheme allows the destructor to figure out when to free the resource.

The destructor has the same name as the class with a prefix tilde character. It takes no parameters, since programs never invoke it explicitly:

`~Complex()` { . . *code to free resources*. . }

The destructor is invoked whenever the object is to be destroyed. That will occur

- when a local object passes out of scope (`return` from a function, for example), or
- when the user program explicitly deletes the object.

If we expect our class to be used as a base class in an inheritance hierarchy, then it's good practice to make the destructor a virtual function, that is, subject to polymorphic invocation:

```
virtual ~Complex(){   }
```

That insures that the right destructor will be called if the user program executes

```
         delete   objPtr;
   or    delete[] objPtr;
```

where `objPtr` is *declared* to be a pointer to the base class but actually contains a pointer to an object of a derived class.

2.3.7 Java destructor and garbage collection

Java has no destructor. Instead, the *garbage collector* examines the active references to an object and frees the storage when no such reference exists. That eliminates memory management bugs, but it's still possible to run out of memory if the program leaves long-lived references to data it no longer needs. Suppose there are active references to `obj`, which in turn contains a reference to an object `item` that the program no longer needs.

A good-practice solution recommended by Joshua Bloch is to destroy a reference whenever (a) the object it points to is no longer needed and (b) the reference itself (`obj`) is not about to become free.[3]

```
obj = null;
```

2.3.8 Java assignment operator

The assignment operator exists in Java, but for reference data it does something entirely different not only from C++ but also from almost every procedural programming language. Java's assignment operator assigns a *reference* to the same object. After the program executes

```
obj1 = obj2;      // reference assignment
```

any changes to the object referred to (and thought of) as either `obj1` or `obj2` will be reflected in both. If you want conventional assignment semantics, you have several choices. Suppose `obj1` and `obj2` are declared as references to instances of class `X`. Then you can

- implement a `clone()` method in class `X` to create a `new` copy of the object and return a reference to it. `X` must also implement the `Cloneable` [sic] interface.

  This is a popular Java convention, but unfortunately the returned reference is not to an instance of class `X` but rather to an instance of the root `Object` class. The user program has to *cast* it back to the intended class before using it:

  ```
  obj1 = (X) obj2.clone();
  ```

- implement a method that mimics ordinary assignment semantics:

  ```
  obj1.set(obj2).
  ```

  This assumes that `obj1` already exists (is not `null`).

- just have the client program invoke the copy constructor explicitly:

  ```
  obj1 = new X(obj2);
  ```

  This technique works whether `obj1` already exists or not.

2.3.9 Implicit conversion in C++

C++ provides two helpful facilities for converting a data item from one type to another without explicit casting, where at least one of the types is not a built-in primitive type. Suppose we've declared two objects:

```
TypeA  objA;
TypeB  objB;
```

Suppose the program then uses `objB` in a context that's invalid for a `TypeB` object but would make sense for a `TypeA` object. That will

[3]Joshua Bloch, *Effective Java: Programming Language Guide* (Reading, Mass.: Addison-Wesley Professional, 2001).

work, provided that one but not both of the following have been defined:

- a *single-parameter constructor* in class `TypeA` that takes a parameter of `TypeB`:

  ```
  TypeA::TypeA (const TypeB x);
  ```

  Of course, the constructor can have more parameters if they have default values.

  ```
  TypeA::TypeA (const TypeB x, long size=20);
  ```

- an *inverse conversion operator* in class `TypeB` that creates a `TypeA` object:

  ```
  TypeB:: operator TypeA() {  . . . . }
  ```

The second approach also works when `TypeA` is a built-in primitive type, such as `double`.

Implicit conversion is not transitive. C++ won't implicitly convert a `TypeB` object to a `TypeC` object if you've defined a rule for converting a `TypeB` to a `TypeA` and another rule for converting a `TypeA` to a `TypeC`.

2.3.10 Implicit conversion in Java

Java has no comparable facility, but provides implicit conversion in one special case. If an object reference `obj` appears in a context where the compiler expects a character string, the compiler will generate a call to method `obj.toString()`. That's handy for simple console output:

```
System.out.print(obj);
```

or for concatenating a string with an object:

```
msg =  Amount due is   + totalPrice;
```

Of course, the class designer can always provide other conversion methods that client programs will invoke explicitly.

2.4 Operator overloading in C++

2.4.1 Member function versus independent function

C++ provides two ways of defining the meaning of an operator applied to one or more objects. We can define an operator function either as a member function or as an independent function. Consider the following code:

```
Angle theta, phi;
      .
      .
phi = theta * 2.0;
```

Unless we've defined a meaning for the `*` operator, the compiler will complain that the operator is not defined for an `Angle` left operand. To legitimize the above code, we might define an *independent* function:

```
Angle  operator* (const Angle ls, const double rs)
{Angle  result = ls;
 result.value *= rs;   // (needs friend access)
 result.normalize();
 return result; }
```

We conventionally use the names `ls` and `rs` for the left and right operands of binary operator functions.

Alternatively, we could define `*` as a *member* function:

```
Angle  Angle::operator*  (const double rs) const
{Angle  result = *this;
 result.value *= rs;
 result.normalize();
 return result; }
```

Here, the left side operand is implied: the *object* for which the function was invoked.

In either case the compiler simply transforms the normal expression syntax into a function call, so that `. . theta * 2.0 . .` becomes either

```
. . operator*(theta, 2.0)..  //  operator* defined
                             //  as independent
```

or

```
. . theta.operator*(2.0)..   //  operator* defined
                             //  as member
```

As a general rule, we prefer

- the member function whenever
    - the left operand must be an object of the class, or
    - the function needs access to `private` members.
- the independent function whenever
    - the left side parameter is not a member of the class, or
    - we want to allow either operand to be converted implicitly, by invoking a single-parameter constructor.

C++ requires a member function for the assignment operator.

Neither version above takes care of all legitimate multiplications of an `Angle` by a pure number. The client program might have coded:

```
. . 2.0 * theta . .
```

That won't match the parameter signature of either the member or the non-member version. We need two multiplication functions: one of the above and

```
Angle operator* (const double ls, const Angle rs);
```

Because multiplication is commutative, implementing the second function is trivial regardless of whether the other one is a member or an independent function. We simply define it in terms of the other function:

```
Angle operator* (const double ls, const Angle rs)
    {return rs * ls;}
```

Problems and exercises

2.4-1 The last multiplication operator above would be valid for any class, as long as multiplication is commutative. Some designers might suggest, therefore, a global function template:

```
template<class T>
 T  operator* (const T ls, const T rs)
    {return rs * ls;}
```

What's wrong with that suggestion?

2.4.2 Sequence and localization

The last example illustrates defining some overloaded operator functions in terms of others. In order to simplify future maintenance, we should do this whenever it doesn't compromise efficiency.

Another obvious candidate is the combination of a binary arithmetic operator such as + and the corresponding compound assignment operator +=. Some programmers are irritated when they learn that they have to define both. If we've defined +, they argue, shouldn't the compiler know what += means?

Well, it doesn't, and tedious as it is, you still have to define both operators. An *obvious* but somewhat inefficient approach is to define the compound assignment operator as a member function in terms of the simple arithmetic operator:

```
Money  operator+ (const Money rs) const
  {Money result;
  result.value = value + rs.value; // (or whatever)
  return result;
  }
Money&  operator+= (const Money rs)
  {Money result = *this + rs; return *this;}
```

That works, but as Scott Meyers and others point out, it's unnecessarily expensive.[4] The *efficient* approach is to define the compound assignment operator first as primitive, and then define the simple binary operator in terms of it:

```
Money&  operator+= (const Money rs)
      {value += rs.value;
      return *this;}     //  Note:  no new object
Money  operator+(const Money rs) const
      (Money result = *this;
       return result += rs;
      }
```

[4]Scott Meyers, *More Effective C++* (Reading, Mass.: Addison-Wesley Professional, 1996).

The latter approach avoids creating a new object in the compound assignment operator function. With that in mind, we advise client programs to prefer compound assignments, especially where the expression contains only one binary operator.

Note that the second version of the simple binary operator function above knows nothing about the object's internal representation. It could therefore be implemented as a non-member, non-friend inline function. Some smart compilers may be able to optimize away the new result object if we rewrite the simple + operator to use an unnamed temporary object by explicitly calling the copy constructor:

```
Money  operator+(const Money rs) const
      (return Money(*this) += rs;
      }
```

Most examples in this book follow Meyers's recommendation.

Problems and exercises

2.4-2 Both versions of the compound assignment operator += return a reference to the object, while the simple + operator returns an actual object. Are both of those conventions necessary? Why?

2.4-3 Suppose we learn that the project for which we're developing a class needs only the simple operators and not the compound assignment ones. How would knowing that alter our strategy in defining binary arithmetic operators for the class?

2.4.3 Increment and decrement operators

Later chapters will examine when it's appropriate to overload the increment (++) and decrement ( ) operators. Here, we'll just look at some of the mechanics.

First, we have to distinguish between the prefix version (`++k`) and the postfix version (`k++`). C++ recognizes the following artifice:

```
const ClassName& operator++();    //  Prefix version
      ClassName operator++(int); //  Postfix version
```

The dummy `int` parameter to the postfix version is never used.

Second, note that the prefix version doesn't create a new object, but just changes the state of the object for which it's invoked. The result is a reference, so as to avoid creating a temporary object. We make it `const` for consistency with C, where the result is not a *Lvalue* into which the program can store a new value.

Finally, we can always define the postfix version in terms of the prefix version:

```
ClassName  operator++(int)
 {ClassName result = *this;
 ++(*this);
 return result;
 }
```

2.4.4 Inline versus separately compiled function

In object-oriented programming, many of the methods are much smaller than typical functions in purely procedural programs. An *accessor* function, for example, often consists only of a `return` statement. Since conventional subroutine linkage would then account for an unacceptably large percentage of the function's execution time, C++ needed a construct that provided the modularity of functions without the overhead of subroutine linkage. That construct is the `inline` function.

You tell the compiler that a function should be generated inline in either of two ways:

- For any function, member or independent, code the `inline` specifier.
- For a member or friend function, define the function body inside the class definition.

In Java, of course, we fully define *all* methods within the class definition. We trust the compiler to decide which functions should be generated inline.

2.4.5 What about exponentiation?

Programmers often complain about C's lack of the *exponentiation* operator supported by almost every other procedural programming language, even COBOL. The ability to define operators in C++ may tempt us to try to fill that need, but we'll be unsuccessful.

Syntactic ambiguities, precedence confusion, or semantic conflicts would result if we were allowed to define, say, `x**n` or `x^n` to mean exponentiation.

When would `x**p` mean `x*(*p)`? Should `a/b**c` mean `a/(b**c)` or should it mean `(a/b)**c`? When would `a^b` have its original Boolean exclusive *or* meaning? (If you're skeptical, you can read Stroustrup's discussion and explanation of this issue.[5]) So we're stuck with using a named function for exponentiation. The C library's function

Ambiguous syntax

C already exhibited syntactic ambiguity before anyone thought of operator overloading. A careless programmer might code

```
a/*p
```

intending to divide `a` *by the number pointed to by* `p`. *But* `/*` *starts a comment. The programmer must either provide explicit parentheses or insert a space between the two operators.*

Haste made much waste when they designed the syntax of that operator-rich language.

```
double pow(const double x,const double y);
```

takes care of the most general case of $\mathbf{x}^y$, but if you want to go to extra trouble for the common situation where the exponent is an integer, you can provide an efficient specialized version, such as this recursive function template:[6]

```
template<class T>  T power(const T x, const int n)
  {T  t;
   return  n == 0   ? 1              //  Base cases
    :  n == 1   ? x                  //     (optional)
    :  n <  0   ? 1 / power(x, -n)  //  Negative
                                     //     power
    :  n%2 == 1 ? x * power(x,n-1)  //  Odd power
    :  t = power(x,n/2), t * t;     //  Even power
   }
```

Note that the nested selection (`?:`) operators don't require parentheses, since they associate left to right. That lets us list the conditions in a column, with the corresponding actions to the right, a rather readable construct once you're familiar with it. If you're

[5]Bjarne Stroustrup, *The Design and Evolution of* C++, 3rd ed.

[6]If you're not comfortable with recursion, long supported by C, you can safely ignore this example, the only recursive function in this book.

acquainted with Lisp, you'll recognize this construct as equivalent to the `COND` function.[7]

Problems and exercises

2.4-4 The last line of the `power` function above is illegal in Java. Why? How can the Java programmer change it so that it's legal and yields the correct result?

2.4-5 Why is the second base case (n=1) optional? What would happen if we removed that line? Why should we leave it in?

2.4-6 How many multiplications will be performed for n=35? Is that optimal?

2.5 Operator overloading in Java

Not only does Java not support operator syntax for objects, but Java insiders vigorously disapprove of *all* operator overloading. Here's a typical explanation:

> *". . . the language designers decided (after much debate) that overloaded operators were a neat idea, but that code that relied on them became hard to read and understand."*[8]

Those language designers must have felt that

```
totalPrice += quantityOrdered * unitPrice;
if (totalPrice > creditLimit) . . .
```

is harder to read and understand than

```
totalPrice.addSet(unitPrice.mpy(quantityOrdered));
if  (totalPrice.greaterThan(creditLimit)) . . .
```

Many experienced application programmers express amazement, mild regret, or stern condemnation when they discover this, but whether or not we agree with the language designers' choice, we still have to accept it and work around it.

Of course, programs still have to do arithmetic on numeric objects and compare numeric objects. We must therefore define

[7]A 40-year-old list processing language widely used in artificial intelligence applications.

[8]David Flanagan, *Java in a Nutshell,* (Sebastopol, Calif.: O'Reilly an Associates, Inc., 2005).

named functions to take the place of operator syntax. This book uses the following mnemonics for those functions:

C++ operator	Java equivalent
`- a`	`a.minus()`
`a = b;`	`a.set(b)`
`a + b`	`a.add(b)`
`a - b`	`a.sub(b)`
`a * b`	`a.mpy(b)`
`a / b`	`a.div(b)`
`a % b`	`a.mod(b)`
`a += b`	`a.addSet(b)`
`a -= b`	`a.subSet(b)`
`a *= b`	`a.mpySet(b)`
`a /= b`	`a.divSet(b)`
`a %= b`	`a.modSet(b)`
`a == b`	`a.equals(b)`
`a < b`	`a.lessThan(b)`
`a > b`	`a.greaterThan(b)`

Problems and exercises

2.5-1 We provided equivalents for only three of the six possible Boolean operators. Is that acceptable? Advisable? Why?

2.5-2 Note that the `equals` function described above is not the one inherited from `object` and conventionally overridden by other Java classes. In what ways is it different? Why do we need both? What about Java's conventional `CompareTo` function and `Comparable` interface?

2.6 Flow control constructs

You undoubtedly already know how all the flow-control constructs work. We're just going to recommend two good practices.

2.6.1 *Prefer prefix increment and decrement*

When the increment or decrement operators are used in a separate statement or clause, we don't care whether the operation is performed before or after the value is returned because we don't use the value. For example, many C programmers habitually use and many textbooks recommend this loop-control idiom:

```
for (ctr=0; ctr < limit; ctr++);
```

If you have that habit, you should change it to:

```
for (ctr=0; ctr < limit; ++ctr);
```

That makes no difference, of course, when `ctr` is a built-in primitive data item, such as an `int`. It makes a big difference, however, when `ctr` is an object of a defined class, such as `Date`. We know that the suffix version of the overloaded `++` operator creates a new object, while the prefix version doesn't (see Section 2.3.13). The suffix version's extra overhead in execution time may or may not be significant, but if the program executes it billions of times it may have a noticeable impact. Anyway, there's no tradeoff, since it costs nothing to use the prefix version.

2.6.2 *Avoid switch case for implementing a table*

Some introductory textbooks illustrate the switch-case construct with a misguided example like this one:

```
switch(monthNumber) {
    case 1:    ndays = 31; break;
    case 2:    ndays = 28; break;
    case 3:    ndays = 31; break;
    case 4:    ndays = 30; break;
        .
        .
    case 12:   ndays = 31; break;
    default:   ndays = 0;       //  error
}
```

or as a function (`break` not needed):

```
int daysInMonth(const short monthNo)
{switch(monthNo) {
    case 1:  return 31;
    case 2:  return 28;
    case 3:  return 31;
    case 4:  return 30;
        .
        .
    case 12: return 31;
}
 return(0);          //  error
}
```

What's wrong with that? Well, all the code is doing is implementing a simple table. The following function does the same thing more clearly and probably more efficiently:

```
int daysInMonth(const short monthNo)
 {static const short ndays[] = {31, 28, 31, 30, 31, 30,
                                31, 31, 30, 31, 30, 31};
  if (monthNo >=1 && monthNo <=12)
           return ndays[monthNumber-1];
  return 0;
 }
```

The intent of `switch case` is to support doing something *different* depending on the value of some number, usually an *enumerated* data item. For example, a generalized transaction processor might contain something like this:

```
switch(transactionType)  {
    case addNewCustomer:
         .
    case changeAddress:
         .
         .
```

However, whenever you find yourself doing the *same* thing, such as `return` or assigning different values to the same variable, you should consider simplifying your logic with an explicit table.

Problems and exercises

2.6-1 Explain in what practical respects the second version of `daysInMonth` above is better than the first (`switch-case`) one.

2.6-2 Find an example in a C, C++, or Java textbook where the author has used switch case to do the same thing with different values. Rewrite it to use a straightforward table.

2.7 Manipulating character-strings in C++

This book is about numeric computing, not string manipulation. Therefore, we won't discuss in detail techniques of string handling. However, our numeric classes will occasionally need to generate a character string, for example, as the result of a function that generates an external data representation.

2.7.1 C-style strings

Among popular procedural programming languages, C provided the weakest support for character strings. The usual way of representing a character string in C was as an array of `char` terminated by the non-printing null character ACSII code 0, coded `\0`). Among the shortcomings that irritated C programmers were the following:

- A function couldn't return a string (`char*`) result without introducing potentially catastrophic memory management anomalies. That made it impossible, for example, to write a usable substring function.
- Strings couldn't be assigned (= operator) or compared (==, < operators) using ordinary expression syntax.
- A program could easily overrun the allocated space. A function had no way of determining how much space had been allocated to a `char*` parameter. This shortcoming alone led to innumerable subtle bugs and provided entrée to some notorious virus programs.
- Except when the string length was a compile-time constant, the program had to allocate the memory explicitly. Then the actual data could not be contiguous with a record (`struct`), further complicating record copying and input-output.

2.7.2 User-defined string classes

C's crude string capability inspired early C++ users to develop character string classes or packages of related classes. Most of those string packages, including some from major compiler vendors, had serious flaws, ranging from catastrophic bugs to clumsy user interfaces to violations of object-oriented concepts. Some of them even preserved the shortcomings of C-style strings!

The minority that didn't eventually brought first-rate string capabilities to C++ programmers in a few organizations, but none of them achieved widespread outside acceptance.

2.7.3 The standard string class

Eventually C++ got a string class that was blessed as a standard and supported by all modern compilers. Like C, it treats strings as containers rather than as elementary data items. Its main virtue is extremely *efficient* implementations, usually based on reference counting.[9] Nevertheless, there are some serious shortcomings:

- For simple and straightforward string handling, it's harder to use than it ought to be, with an excessive number of methods having overlapping functionality.
- It provides no support for fixed-length strings, common in business forms and databases.
- It provides no support for embedded (contiguous) strings within a record, a traditional need in business applications.

2.7.4 String handling in this book

First, we're going to avoid string handling wherever possible. For example, we'll avoid implementing external input functions (`istream& operator>>(. . .)`) if we don't absolutely need them.

However, our examples still need strings in several areas, especially external output or conversion *to* external representation. We have to use *some* string class, but none of the above, because

[9]A technique that avoids copying the string data in some cases, e.g. for a function return value. See Scott Meyers, *More Effective* C++.

- Although we know of some excellent easy-to-use, user-defined string classes, it would be too burdensome for you to have to copy them and set them up.
- The standard string class is too complicated and awkward to use for our simple needs.

What we're going to do, therefore, is to present the simplest possible string class, `SimpleString`, that permits

- string-valued functions
- string assignment
- concatenation
- size that is adjustable upon allocation

`SimpleString` doesn't support scanning and parsing. Our `SimpleString` class is meant only to support examples in this book. You can run the examples and experiment with `SimpleString`, but you should choose a more complete and robust string class for your serious software development.

2.8 Canonical class structure

A typical complete class definition, even one for a simple elementary data type, consists of several pages of dense code. The maintenance programmer needs to find items of interest quickly and reliably. Therefore, most organizations' standards for C++ or Java classes specify a preferred sequence.

Fashions change. In the early days of C++, class definitions typically began with the private data members. Today, however, many programmers place the data members last, beginning the class with the public interface (the constructors, accessors, overloaded operators, and so on). This recent practice caters less to the maintenance programmer than to the client programmer (user) who has to consult source code in the absence of usage documentation.

Actually, it makes very little practical difference. Each organization or project team should choose a preferred sequence and stick to it for the sake of consistency. Regardless of your own preference on this minor issue, you should be willing to follow the standards of the group you're working with at the time. Where there's a compelling reason to depart from the sequence your group expects, explain it in brief commentary.

The complete class examples in this book adhere to the following sequence:

1. private data members
2. constructors
3. destructor
4. accessor functions
5. inverse conversion operator
6. arithmetic operator functions
7. relational operator functions
8. I-O stream operator and external conversion methods

We put static data, static functions, and private methods near any other members to which they're closely related. Again, if your organization prefers a different sequence, that's fine. The sequence of class members is not an issue worth discussion.

2.9 Overcoming macrophobia

2.9.1 Bad macros

C++ insiders disparage the use of the preprocessor, inherited from C. Java goes further by not supporting a preprocessor in the standard language. Experts, including the designers of both languages, advise against using macros. Stroustrup warns:

> *"The first rule about macros is: Don't use them if you don't have to. Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer."*[10]

What really offends Stroustrup and other experts are these two common uses of macros in traditional C:

- defining constants:

  ```
  #define  NITEMS 32
  ```

- implementing generic pseudo-functions:

  ```
  #define  ABS(x) ((x) >= 0 ? x : -(x))
  ```

[10]Bjarne Stroustrup, *C++ Programming Language,* 3rd ed.

Both of those macro definitions are handled more flexibly and more reliably by features of C++:

```
const int NITEMS = 32;
template<class T>
 inline T abs(const T x) {return x >= 0 ? x : -x;}
```

or Java:

```
public static final int NITEMS = 32;
public T abs(final T x)      //  member of class T
   {return x.greaterThan(0) ? x : x.minus();}
```

2.9.2 Good macros

The main benefit of macros in any programming language is that they provide a way of capturing and packaging *patterns* of code. Specifically, macros are able to

- localize such patterns for ease of future change
- facilitate complying with standards for such patterns
- minimize or eliminate opportunities for error
- enhance source-code readability

There is no other way in C++ (and no way at all in Java) to capture code patterns that are not functions. Here's a small, simple example

```
#define ACCESSOR(name, rtnType, expr)  \
     rtntyp name() {return expr;}
```

A programmer defining the `Complex` class could use that macro like this in the class definition:

```
ACCESSOR(realPart, double, rp)
ACCESSOR(imagPart, double, ip)
ACCESSOR(rho,      double, sqrt(rp*rp + ip*ip))
ACCESSOR(theta,    Angle,  atan2(ip/rp))
```

Since maintenance programmers scanning the source-code would immediately spot those functions and understand their purpose, you wouldn't need any explanatory commentary. Don't worry; we're not going to use that sort of low-level macro coding in this book.

2.9.3 Packaging common patterns in elementary numeric classes

Defining a robust, production-quality class can be awfully tedious and error-prone. We need to implement dozens of methods, follow dozens of rules, and avoid dozens of pitfalls.

Preprocessor macros are the *only* C++ facility that can help. In Chapter 4 we shall package the *additive* pattern, and in Chapter 5 we shall package the *point-extent* pattern using mainly the simplest of all preprocessor facilities, the `#include` macro statement. In Chapter 6 we shall use some *ad hoc* macros to reduce repetition of smaller patterns of code, and thereby simplify future program maintenance.

Those simple examples may stir your imagination to consider this approach whenever you find yourself repeating the same pattern of code over and over.

Problems and exercises

2.9-1 Among standard higher-level languages, PL/I provides the most powerful macro (or preprocessor) language. Consult a PL/I manual to learn how macros work, and then try to find some examples that exploit those facilities. Discuss whether a similarly powerful preprocessor would help or hurt program development and maintenance in C++ or Java.

2.9-2 (*C) The `switch case` construct (2.3.17) works only for integer types. Using macro coding, devise an equivalent flow control construct `Switch Case` that works for *any* data type. Document any reasonable restrictions due to limitations of the macro language (forbidding nested occurrences of this construct, for example).

2.9.4 `#include` *dependencies*

Many `#include` files, especially class definitions, depend on definitions in other `#include` files. There are two ways of handling such situations:

1. The usage documentation for file A can tell the user that file B has to be included first, like this:

```
#include B
#include A
```

2. File A can itself contain the preprocessor statement `#include B`. If file B contains definitions that the compiler should see only once, then it's customary to surround the code with a *multiple-include guard*, usually `#ifndef . . . . #endif`.

The second technique has become common practice among C and C++ programmers, because it localizes knowledge and relieves the programmer of complicated bookkeeping. Nevertheless, there are a few situations where the first technique is preferable.

One such situation is the packaging of truly *global* definitions, such as constants, functions, and macros that nearly every program needs and that the programmer should hardly need to think about. Such definitions are like extensions to the C++ language itself.

In the code appendix is a minimal set of such definitions, file `global.hpp`. Most organizations will want to expand it to include more conventional names and functions that they wish to be standard for their programming staff or at least for an individual project team, for example an `#include` for a string class.

Some experts who disparage macros claim that use of macros impairs program readability by introducing unfamiliar syntax. Experience shows, however, that macros like these greatly facilitate readability among groups of programmers who are acquainted with them and use them every day.

This technique also reduces compile times, since `global.hpp` is opened only once per compile. When programs grow very large, compile times can become surprisingly long, and a lot of that time is wasted in opening `#include` files only to skip to the end and close the file because of the multiple-include guard.

Problems and exercises

2.9-3 C programmers often call `#include` files "header files." Discuss how that tradition may have originated and whether it's reasonable today to call every includable source-code file a "header."

2.9-4 Java's designers consider a general `#include` facility unnecessary and potentially dangerous. Consider their views and decide whether you concur. State persuasive reasons to support your position.

2.9-5 A few of the generic functions in `global.hpp`, for example `min(x,y)`, duplicate those available in the standard template library (STL). What considerations justify also having them in `global.hpp`?

2.9.5 Macros in this book

In later chapters, we shall occasionally exploit macros in a couple of situations where C++ provides no alternative facility

- to capture patterns of code for reuse in multiple components.
- locally, to "factor out" tedious repetition within a component to enhance readability and avoid error. The `global.hpp` example above uses a couple of such local macros.

We refrain, however, from using macros as extensively here as we do when we write production code. Thus, readers can skip this chapter and still be able to understand every example.

2.10 Program readability

2.10.1 Commentary and data names

A program is not only something to be run on a computer but also a *document* for people to read. We assume that the reader is an experienced programmer, often the original programmer at a later time.

Good programmers use commentary in four places:

- A *title* comment introduces a class definition, an important function, a package of macro definitions, some other nontrivial module, or an entire source-code file. For proprietary programs, it often includes a copyright notice.
- *Introductory* comments describe the purpose and usage of a class, function, or other module.
- *Block* comments describe the purpose and strategy of a group of related statements.
- *Line-by-line* comments explain an individual statement or even a part of a statement.

Program documentation is an integral part of programming, not a separate activity. Title and introductory comments are best written *before* the code. That helps you to clarify your thoughts and usually saves time. Line-by-line and block comments can be written before, during, or after the code. In complicated logic block comments are often useful to explain the state of data items at that point.

Line-by-line comments should avoid stating what's obvious from the code. Describe *what* is being done, not *how*. For example,

```
Not: ++posn;            //  Advance the position
But: ++posn;            //  Skip over the comma
Not: weight*=2.2;       //  Multiply by conversion factor
But: weight*=2.2;       //  Convert to pounds
Not: while(count>0)     //  Loop until count exhausted
But: while(count>0)     //  Examine all work orders
```

By choosing meaningful data names, we often avoid the need for any line-by-line comment:

```
Not: while(count>0)            // Examine all work orders
But: while(workOrderCtr>0)
Not: weight*=2.2;              // Convert to pounds
But: weightInPounds = weight * kgToPound;
```

Data names should be mnemonic, suggesting the purpose or usage of the data item from the point of view of the module. Names should be long enough to be mnemonic (or self-documenting) but not so long as to force typical statements to span multiple 80-character lines. Single character variable names, such as `k`, are sometimes appropriate for abstract mathematical quantities or bound variables having a short scope (such as a loop index).

2.10.2 Format criteria and editor support

Criteria for writing easy-to-read code haven't changed since the structured revolution of the 1970s. Today, however, it has become harder to satisfy those criteria, because many of the editors and development platforms that we use to create and maintain C++ and Java source code fail to support basic page layout facilities.

One excuse we hear is that programmers, especially younger ones, view their code only on a computer screen where the text is continuous. Instead of turning a page, the programmer scrolls the text, using either a vertical scroll bar or the up and down keyboard arrows. If we never print the source code on hard copy, then why, some ask, should we care about page layout?

Actually, experience shows that it's often much easier to comprehend a module on a 55-line page (or a pair of facing pages) than on a screen. No matter how comfortable you are with your online program editor, I strongly recommend that you print and save a paper copy every now and then, study it carefully, and make notes on it in red pencil.

Why monospaced font?

With the huge choice of fonts now available on most platforms, some experts have recently abandoned the uniform (e.g. Courier) font traditionally favored by programmers. If we display and print our programs in, say, Times Roman font, we can squeeze more characters onto a line.

The disadvantage of a variable width font is that the programmer no longer controls alignment of corresponding elements of a series of lines (see 1.62.3.33). That can make errors harder to spot. Even indentation may be less clear. I recommend sticking with monospaced font for displaying and printing your source-code.

2.10.3 Uncontrolled page breaks

One particularly irritating omission is the lack of any way to insert a hard page break. When the top line of a module appears at the bottom of a page, or when a four-line loop is split between pages, readability suffers.

Some programming organizations have developed their own source-code listing programs that interpret embedded commands, such as

```
//%EJECT
```

Others paste an entire module into a word processor and then edit the text to produce a readable *presentation* (or publication) copy. That's too much labor, however, for everyday use, and it runs the risk of last-minute changes to the presentation copy that aren't reflected in the compiled copy.

2.10.4 Page and line width

Even though we no longer prepare source code on 80-column punched cards, programmers still work with 80-position lines. Character display monitors and the "command windows" that mimic them are typically limited to 80-position lines. With 11-point monospaced font and normal margins, standard paper accommodates lines up to 80 characters.

Trouble arises because some of our source-code editors recognize no such limitation. They'll let you keep typing 200 or more characters on a line. When you try to view such a line on the screen you have to scroll the text horizontally, and when you print them on paper, they either get chopped off or continued (ruining your indentation) at the left margin of the next line.

Even without horizontal scrolling, the most popular screen resolution lets you enter 100 characters without realizing that you've gone off the page. You can still see the whole line on the page, but you can't print it on normal paper.

If your editor doesn't warn you when you've exceeded a standard line size, it probably displays the current character position in a corner somewhere. Keep an eye on it and don't go beyond position 80. Since both Java and C++ syntax accept line breaks between tokens, you can always split a statement between lines in a highly readable manner.

2.10.5 A macro convention

The `global.hpp` definitions we introduced in Section 2.3.25 contain definitions for a set of macro names that enhance source code readability:

```
#define INT    const int
#define DOUBLE const double
. (and so on)
```

If you use these macro names, you'll not only conserve horizontal space on the listing, but also reduce the chance of forgetting `const` in the parameter list of a function. We strongly recommend extending this convention to classes you define. For example,

```
#ifndef COMPLEX
#define COMPLEX const Complex
  class Complex  {
    .
    .
};
#endif
```

2.10.6 Indentation and white space

Compare these two source listings:

```
while(lbound <= hbound)
 {midp = (lbound+hbound)/2;
 if (arg ==tbl[midp])
     return midp;
 if (arg < tbl[midp])
    hbound = midp - 1;
 else lbound = midp + 1;
 }
return -midp;
```

```
while(lbound <= hbound) {
midp = (lbound+hbound)/2;
if (arg ==tbl[midp])
return midp;
if (arg < tbl[midp])
hbound = midp - 1;
else lbound = midp + 1;}
return -midp;
```

The compiler considers them identical, but a human reader does not. You probably recognize the one on the left as the familiar *binary search* algorithm.

Although indenting lines to show the scope of flow control constructs contributes greatly to source-code clarity, it's silly to worry about the exact number of spaces to indent, whether to put brackets alone on separate lines, or similar minutiae. The programmer should be concerned with clear and attractive presentation of source code, not with complying with arbitrary and restrictive *rules*.

Blank lines also help to set off blocks of related code. Since they get in the way when you view a program on a small screen, they provide another argument in favor of printing program listings on paper.

2.10.7 Alignment

A block of consecutive similar lines or of similar groups of lines is easier to read if corresponding elements are aligned. Note that in the binary search indentation example we also took care to align similar clauses.

Finally, compare these two:

```
INT Black   = 0;
INT Blue    = 1;
INT Green   = 2;
INT Red     = 4;
INT Cyan    = Blue + Green      ;
INT Magenta = Blue +         Red;
INT Amber   =        Green + Red;
INT White   = Blue + Green + Red;
```

```
INT Black = 0;
INT Blue = 1;
INT Green = 2;
INT Red = 4;
INT Cyan = Blue+Green;
INT Magenta = Blue+Red;
INT Amber = Green+Red;
INT White = Blue+Green+Red;
```

The version on the right would be even less clear if rendered in a variable pitch font.

2.11 Error detection and exceptions

Both C++ and Java support *exception*-handling facilities. In robust production-quality programs, of course, you'll want to take full advantage of them. For the examples in this book, however, we chose not to `throw` or `catch` exceptions, because the subject matter of this book is entirely independent of the error-handling mechanisms. The subtle complexities of `try` blocks, `catch` blocks, exception hierarchies, and `throws` declarations would have detracted from our focus on numeric objects and would have made the examples harder to grasp.

In the few places where our examples detect errors, we have relied on an old C facility, the `assert(p)` macro. If the Boolean expression parameter `p` is true, no action is taken, but if it's false, execution is terminated. No matter where this occurs in the program, termination is immediate and abrupt, with no draining of output buffers or graceful freeing of resources. For example:

```
Temperature(DOUBLE degreesKelvin)   // Constructor
 {assert (degreesKelvin > 0.0);
 value = degreesKelvin;
}
```

The `assert(p)` macro is admittedly crude, but it is far preferable to ignoring an error. If a constructor leaves an object in an illegal state, or if a function yields nonsense results, even more serious consequences will surely occur later in the program, and they may be very hard to diagnose. Never ignore an error.

On the other hand, object orientation eliminates the need for redundant "paranoid" error checking. A function that takes a `Date` parameter, for example, may assume that the `Date` object is legal and should not validate that the month is between 1 and 12.

Java (since version 1.4) has an `assert` *statement:*

```
   assert  p;
or assert  p : c;
```

where `p` is the Boolean expression to be validated and `c` is an expression whose `toString()` value is to be displayed in the error message. If `p` is false, an `AssertionError` exception is thrown.

Problems and exercises

2.11-1 The C library designers might have chosen to implement this facility as an ordinary function:

```
void assert(bool p)
      {if (!p) abort();}
```

Why did they elect instead to make it a parameterized macro?

2.11-2 In both the C and Java facilities, the Boolean expression `p` should not generate side effects. Why is this a sensible rule?