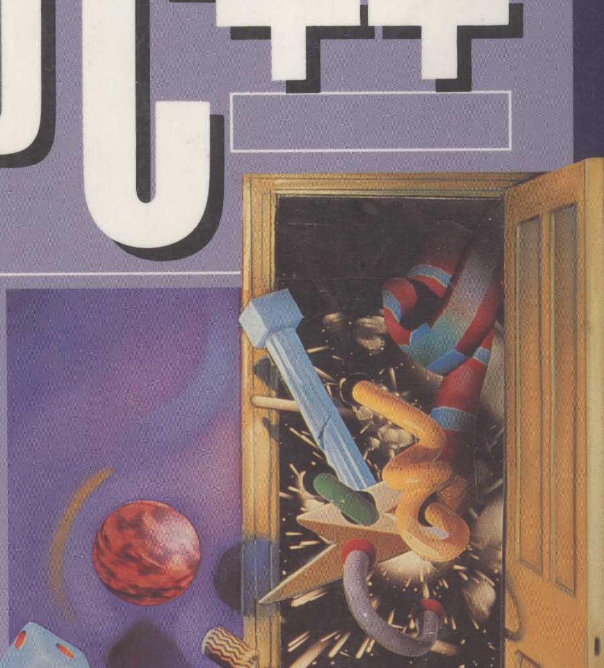
CLASS CONSTRUCTION IN C AND C++

OBJECT-ORIENTED PROGRAMMING FUNDAMENTALS



ROGER SESSIONS

Class Construction in C and C++
Object-Oriented Programming Fundamentals

Roger Sessions

International Business Machines Corporation
Austin, Texas

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

SESSIONS, ROGER.
Class construction in C and C++ : object-oriented programming fundamentals / Roger Sessions.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-630104-5
1. Object-oriented programming. 2. C (Computer program language) 3. C++ (Computer program language) I. Title.
QA76.64.S42 1992
005.1—dc20 91-48099
CIP

Cover design: *Lundgren Graphics, Ltd.*
Source: *Image Bank*
Illustrator: *Sandra Lilippucci*
Copy editor: *Maria Caruso*
Acquisitions editor: *Greg Doench*
Editorial assistant: *Rene Wilkins*
Prepress buyer: *Mary E. McCartney*
Manufacturing buyer: *Susan Brunke*

The publisher offers discounts on this book when ordered in bulk quantities. For more information, write: Special Sales/Professional Marketing, Prentice Hall, Professional & Technical Reference Division, Englewood Cliffs, NJ 07632.

Printed in the United States of America

10 9 8 7 6 5 4 3 2

ISBN 0-13-630104-5

PRENTICE-HALL INTERNATIONAL (UK) LIMITED, *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
SIMON & SCHUSTER ASIA PTE. LTD., *Singapore*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*

In order to find one's place in the infinity of being,
one must be able both to separate and to unite.

- I Ching
Chun Hexagram — Difficulty at the Beginning
Translated by Richard Wilhelm and Cary F. Baynes

Preface

This book is about object-oriented programming and how the concepts of object-oriented programming can be applied in C and in C++. The book's goal is to demystify object-oriented programming; to show that object-oriented programming is really just a common sense extension of structured programming; to show that many of the principles of object-oriented programming are applicable to any language; and to show that there are just a few new language features in C++ that must be learned to start using the language effectively.

There are two parts to learning object-oriented programming. The first is learning the object-oriented paradigm. The second is learning an object-oriented language, in this case, C++.

Learning both a new paradigm and a new language can be a daunting goal. The object-oriented paradigm is for most of us a new way of thinking about programming. The new language we are interested in, C++, is generally regarded as quite complex, with many new syntactic enhancements over and above C, a language most consider already complicated enough.

This book simplifies the material through two approaches. The first is by separating the paradigm from the language. The second is by focusing on only the important language features.

Separating the paradigm from the language means teaching as much of the object-oriented paradigm as possible using standard C code. This allows the reader to become familiar with the concepts of object-oriented programming without having to deal with the overhead of a new language.

Once the paradigm is firmly established, we start discussing C++. We look at where C has weaknesses in implementing object-oriented concepts, and how C++ supplements C to address these weaknesses. We purposely ignore the syntactic fluff of C++ which has little to do with object-oriented programming.

Many influential authors suggest using C++ as a better C, even if one never makes the shift to object-oriented programming. Their reasoning is that C++ is a superset of C, and therefore any C programmer can start using C++ immediately by just using the C subset of C++. Then, one can gradually make more and more use of the new C++ language features as one learns them. Since so few of the new language features are directly related to object-oriented programming, this argument goes, why wait to make the paradigm shift?

This argument has one major flaw. The most important advance offered by C++ is not its myriad collection of C enhancements, but its direct support for the object-oriented paradigm. The programmer who successfully makes the paradigm shift, but does not know every last C++ feature, will be far ahead of the programmer who memorizes every C++ ampersand and keyword, but never learns the new approach to thinking about programming.

This book focuses on the paradigm. We discuss those C++ language features which are essential to the paradigm and ignore those that are not. Those features which we do cover are covered in considerable depth, much greater depth than can be covered in books which cover every detail of the language.

The purpose of this book is to get you to use object-oriented programming. To teach you the important features of C++. To teach you those features well. You will then have plenty of time to learn the details, and there are plenty of books available from which you can learn it.

This book is targeted at two main groups of readers. The first is the C programmer who wants to learn object-oriented programming and C++. The second is the large group of C++ programmers who have never made the paradigm shift, who use C++ but only to write better procedural code than they could have written in C.

This book teaches object-oriented programming by looking at a lot of object-oriented code. This book includes over 7000 lines of code, almost all of which is shown as fully running programs complete with output. Almost every feature we discuss is demonstrated by actual running code.

Overview of Book

This book can be thought of as having three parts. The first part (Chaps. 1–5) teaches the C programmer the basic concepts of object-oriented programming, all in the C programming language. The second part (Chaps. 6–9) teaches the fundamentals of using C++. The final part (Chaps. 10–12) examines selected C++ issues in much greater depth. This last part will be of interest even to seasoned C++ programmers.

The next chapter, Chapter 1, provides a quick refresher course in the more advanced features of C. Although readers are expected to already be familiar with

C, many will not have used some of the more advanced features of the language such as pointers to functions and dynamic memory allocation. These features are used extensively in object-oriented programming, and all such features are reviewed in this chapter.

Chapter 2 reviews the concepts of structured programming. We consider a reasonably complex problem, counting excessively used words in a text file. This chapter gives a fully coded structured solution to this problem.

Chapter 3 introduces object-oriented programming. It defines most of the new object-oriented terminology in terms designed to be comfortable to the C programmer. It discusses the meaning of object-oriented programming. It recodes the problem of the previous chapter using an object-oriented solution, still in C, giving us a concrete example to contrast structured and object-oriented approaches to programming. The main purpose of this chapter is to give an intuitive understanding of what we mean by the term object-oriented programming.

Chapter 4 introduces more rigor to the concept of object-oriented programming in C. We discuss how programs must be organized to allow multiple instantiations of classes and maximum flexibility in the use of classes. As an example of a well organized object-oriented program, we look at software designed to manage a doctor's waiting room. This program uses many object-oriented data structures designed with minimal compile time limitations.

Chapter 5 discusses some of the problems one faces using C to develop object-oriented programming. Since C++ was developed primarily to address these limitations, this chapter essentially discusses the design goals of C++. Understanding the issues C++ was designed to address makes it easier to understand the new syntax of the language, and why the features work the way they do.

Chapter 6 gives an introduction to object-oriented programming in C++. This chapter covers the basics: defining classes, instantiating objects, and invoking methods. We look at C++ code designed to manage point of sale transactions as an example of how C++ can be used to solve real life problems.

Chapter 7 discusses inheritance, or class derivation. Inheritance is difficult to program in C, so this concept is introduced now for the first time. Class derivation is an important feature of C++, providing a fundamental technique for writing generic and reusable code.

Chapter 8 discusses method resolution in C++ in more depth. It compares virtual and non virtual resolution, and shows how virtual resolution compares to the C techniques of using function pointers to achieve code generality. The linked list class introduced earlier is recoded to make full use of inheritance and virtual resolution.

Chapter 9 discusses a collection of issues all having to do with managing memory in C++. We discuss the relationship between memory allocation and memory construction, between deallocation and destruction. We show how the

C++ programmer can take full control over allocation, construction, deallocation, and destruction. We discuss related issues such as reference and constant variable types and assignment operators.

Chapter 10 shows how the most popular C++ precompiler actually works. We look at the C code the precompiler emits, and compare this code to our own versions of C classes. This chapter gives some valuable insight into why C++ works the way it does, and why it has some of the problems it has.

Chapter 11 discusses some of the problems with C++. This is not to denigrate the language, only to point out some of the tradeoffs the language makes.

Chapter 12 gives a full, complex example coded in C++. The example is a text processing program. It is difficult to appreciate how C++ is used in a real programming environment without looking at a real problem. This chapter solves a problem, a real problem, with a nontrivial solution. This chapter includes over 18 class definitions and 1700 lines of code. By looking at this code in detail, we can appreciate the complexities of trying to apply object-oriented programming, and the design issues one typically faces.

Finally, an epilogue. This gives an overview of what this book has not covered, and points the reader in some directions for following up on areas of interest.

Acknowledgments

I owe a great deal to a great many people for their support in writing this book.

I especially thank my wife, Alice Sessions, who has not only supported the effort emotionally, but spent many hours at the word processor entering and editing text. She also found many of the opening chapter quotations.

Other members of my family have taken a keen interest in this work. My daughter Emily critiqued quotations and helped choose the title. My son Michael kept reminding me to "work on the book."

The book has benefited greatly from some very thorough reviews by some very knowledgeable people. I am grateful to Stephen Dewhurst of Glockenspiel, Doug Lea of SUNY at Oswego, and Clovis Tondo of IBM at Boca Raton for their many helpful suggestions.

I appreciate the support of IBM in this writing. IBM has allowed me to use their hardware and software, and has provided an environment which greatly nourishes the creative process. Tony Dvorak, Mike Kiehl, Larry Loucks, and others have encouraged and supported publishing activity. My co-workers have been very helpful. Hari Madduri and Craig Becker especially have provided me with valuable in depth critiques of earlier revisions of this manuscript. Mike Conner was the first to point out to me the problems of C++ library version incompatibility, one of the topics in the C++ Problems chapter.

Although IBM has kindly supported this effort, it has exerted no editorial control. The views expressed here reflect those of the author, and not necessarily those of IBM.

Prentice Hall is, as always, a pleasure to work with. I am greatly indebted to my editor Greg Doench for his encouragement, and to his assistant Joan Magrabi for coordinating most of the activity of this book.

Clovis Tondo of IBM Boca Raton designed this book and prepared it for typesetting. The final camera ready copy was printed with the Chelgraph IBX typesetter by TYPE 2000, 16 Madrona Avenue, Mill Valley, California 94941.

It seems only appropriate to thank the many writers' hangouts of Austin, Texas where so much of this book was written and edited. These are all spots, where, for the price of a expresso or an inexpensive meal, one can take over a table, spread out a ream or two of paper, and lose oneself for hours in the process of writing. These establishments include Martin Brothers Cafe, Chez Fred, Kerbey Lane Cafe, La Zona Rosa, Texas French Bread Bakery, Elephant Club, University of Texas Student Union, Upper Crust Bakery, Campus Cafe, and of course, the quintessential Austin writer's hangout, Captain Quackenbush's Intergalactic Dessert and Expresso Emporium. Thanks to all of you for your tolerance, and to the many other establishments I have yet to discover.

Finally, I thank the many publishers who have kindly consented to allow me to reprint from these copyrighted materials:

Contents

Tzu-kung asked how to become Good. The Master said, A craftsman, if he means to do good work, must first sharpen his tools.

- The Analects of Confucius
Translated by Arthur Waley

Chapter 1

C Refresher

1.1 Introduction

Although this book assumes readers have a working knowledge of the C programming language, some of the more advanced features of the language may be unfamiliar to some readers. This chapter reviews some important features of the language which we depend on in developing the techniques of this book.

If you have been using C extensively in a production environment, you may have no need of this chapter at all. If you have just finished your first course in the language, you may want to study this chapter carefully. If you fall someplace in the middle, as most readers will, browse through the chapter and study those sections which seem new to you. As you continue in this book, return to this chapter on an "as needed" basis.

If you find yourself unable to understand C programming techniques not covered in this chapter, unable to understand the material in this chapter, or unable to complete the exercises at the end of this chapter, you may need to review a C introductory text.

1.2 typedef

C provides a standard collection of types. When a variable is declared, it can be declared to be any of the built in C types. For example,

```
int size;
```

declares a variable `size` to be the standard C type `int`. We can also declare new types using the `typedef` construct. These new types can then be used in variable declarations such as

```
name myName;
name yourName;
```

The general rule for using a `typedef` to define a desired type is

1. Define a variable of the desired type.
2. Place `typedef` in the front of the line.

For example,

```
char name[100];
```

declares a variable which is a 100 character array.

```
typedef char name[100];
```

declares a type which is a 100 character array. The lines

```
name myName;
name yourName;
```

then declare two variables of type `name` which, based on our `typedef`, are 100 character arrays. These declarations are exactly equivalent to

```
char myName[100];
char yourName[100];
```

but have two advantages. First, the declarations are simpler. Second, type changes are easier.

By collecting `typedefs`, in a small number of header files, we can update our types in one location and propagate them quickly throughout the system. Suppose, for example, we have 20 variables of type `name` scattered throughout our system, and we then discover that we need 110 characters instead of 100 for a name. We can update every variable of type `name` by making this one change

```
typedef char name[110];    /* Changed from 100 */
```

Without the `typedef`, we must hunt through possibly hundreds of declarations like

```
char myName[100];
char who[110];
char what[100];
char where[98];
```

and decide on an individual basis which of these variables were meant to hold names and therefore need updating, a time consuming and error prone process.

1.3 Structures and Structure Pointers

Programs are often responsible for coordinating large amounts of data. One way of managing the complexity of data is to package together related data items into what is called a structure. For example, we could define an employee structure that contains an employee name, address, social security number, and manager name. A program which manipulates ten thousand employee names, ten thousand employee addresses, ten thousand social security numbers, and ten thousand manager names is a complicated program. A program which manipulates ten thousand employee structures is a simple program. The volume of data is similar for both programs, but the latter manages the complexity of the data by using structures.

The term *structure* is commonly used to refer to both the definition and the allocation of data structures. The definition of a structure defines the size and contents of a given structure, without actually allocating memory. The allocation takes an existing definition and allocates memory for such a structure. The definition of a structure is done exactly once per structure type. The allocation may be done any number of times, including zero.

A structure is defined using the syntax

```
struct structureName {
  type1 item1;
  type2 item2;
  etc.
};
```

The definition of our employee structure looks like

```
struct employeeStructure {
  char name[100];
  char address[100];
  char ssnum[20];
  char manager[100];
};
```

A structure is allocated using the syntax

```
struct definedStructureName thisStructureName;
```

We can define an instance of the `employeeStructure` named `mary` by

```
struct employeeStructure mary;
```

Once a structure has been allocated, we refer to its elements using this syntax

```
structure.item
```

For example, we could print `mary`'s name by

```
printf("Name: %s\n", mary.name);
```

We can also define variables which contain the addresses of structures. The syntax for this is

```
struct structName *varName;
```

so we could have

```
struct employeeStructure mary;        /* Allocate mary */
struct employeeStructure sam;         /* Allocate sam */
struct employeeStructure *currentEmp; /* Allocate Pointer */
currentEmp = &mary;                   /* Set Pointer to mary */
```

Logically, you would expect to be able to refer to a member of a structure being pointed to by this syntax:

```
(*varName).item
```

or in this case,

```
(*currentEmp).name
```

but C provides this more convenient equivalent syntax

```
varName->item
```

or in this case,

```
currentEmp->name
```

When we pass a structure into a function, we almost always pass in the address of the structure, and receive it as a pointer. The following types of code fragments are very common.

```
  struct employeeStructure mary;
  struct employeeStructure sam;
  /* ... */
  printEmp(&mary);
  printEmp(&sam);
}
void printEmp(employee *thisEmp)
{
/* ... */
```

We also frequently see structures `typedef`ed. The following statement

```
typedef struct employeeStructure employee;
```

defines `employee` to be a valid C type, in that it can be used to define other variables. With this `typedef`, we can replace these lines

```
struct employeeStructure mary;        /* Allocate mary */
struct employeeStructure sam;         /* Allocate sam */
struct employeeStructure *currentEmp; /* Allocate Pointer */
currentEmp = &mary;                   /* Set Pointer to mary */
```

by these

```
employee mary;        /* Allocate mary */
employee sam;         /* Allocate sam */
employee *currentEmp; /* Allocate Pointer */
currentEmp = &mary;   /* Set Pointer to mary */
```

The following program shows all of these techniques in use.

```
#include <stdio.h>
#include <stdlib.h>

/* Define an employee structure.
   ----------------------------- */
   struct employeeStructure {
     char name[100];
     char address[100];
     char ssnum[20];
     char manager[100];
   };
   typedef struct employeeStructure employee;

/* Function declarations.
   ----------------------- */
   void printEmp(employee *thisEmp);

int main()
{
/* Allocate memory for mary and sam.
   ---------------------------------- */
   employee mary;
   employee sam;
```