原版风暴 • 深入 C++ 系列

PEARSON Addison Wesley

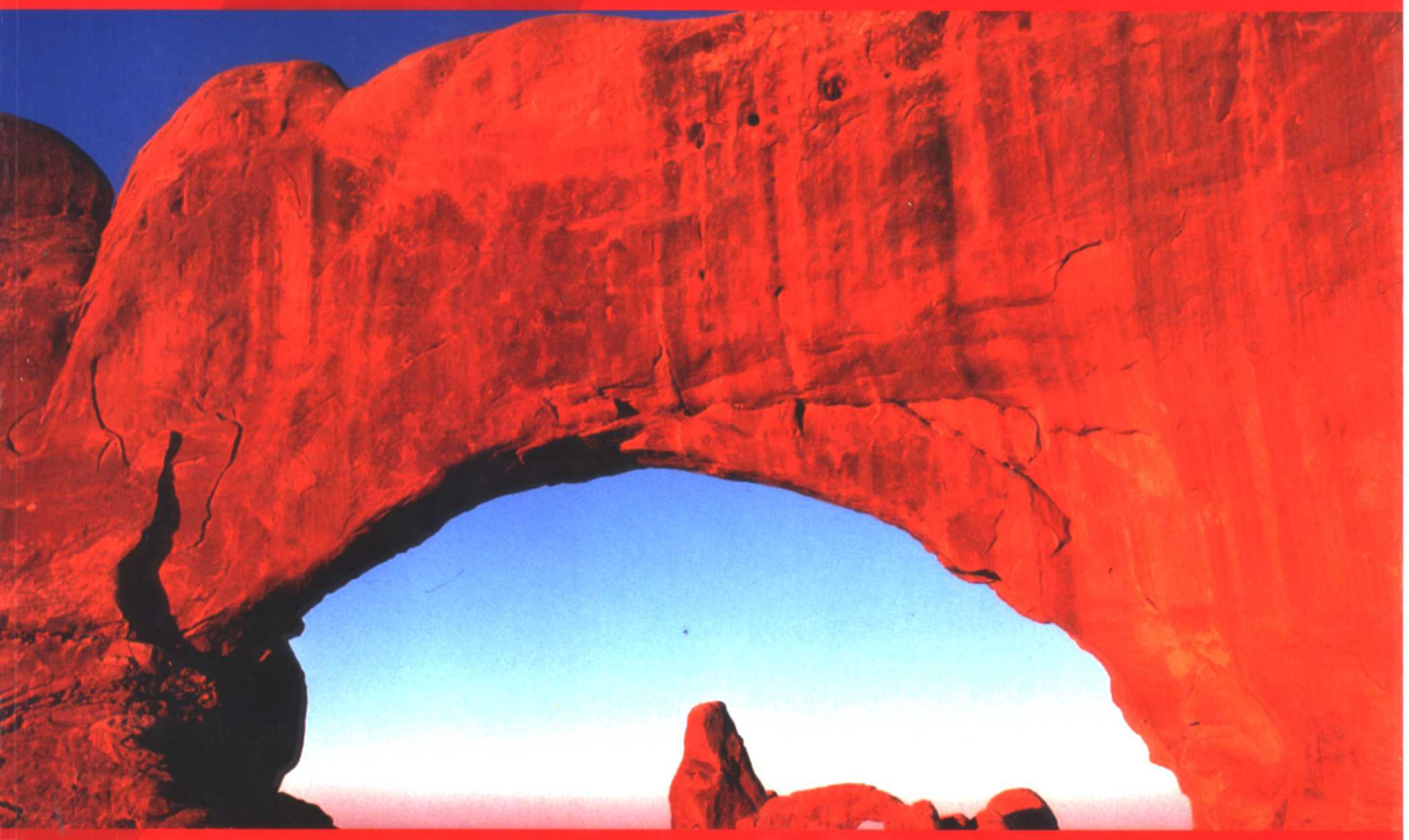
与《C++ Primer》配套使用

C++ Primer Answer Book

C++ Primer 题解

（影印版）

[美] Clovis L. Tondo, Bruce P. Leung 著



Pearson Education

中国电力出版社

www.infopower.com.cn

C++ Primer Answer Book

C++ Primer 题解

（影印版）

[美] Clovis L. Tondo, Bruce P. Leung 著

中国电力出版社

C++ Primer Answer Book (ISBN 0-201-30993-9)

北京市版权局著作合同登记号：图字：01-2003-1018

图书在版编目（CIP）数据

C++ Primer 题解 /（美）通都（Tondo, C.），（美）路昂（Leung, B.）著.
影印本. —北京：中国电力出版社，2003
（原版风暴·深入 C++系列）
ISBN 7-5083-1495-6
Ⅰ.C... Ⅱ.①通... ②路... Ⅲ.C 语言—程序设计—解题—英文 Ⅳ.TP312-44
中国版本图书馆 CIP 数据核字（2003）第 023084 号

责任编辑： 程瑞
丛 书 名： 原版风暴·深入C++系列
书　　名： C++ Primer题解（影印版）
编　　著： （美）Clovis L. Tondo, Bruce P. Leung
出版发行： 中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传真：（010）88423191
印　　刷： 北京地矿印刷厂
开　　本： 787×1092 1/16　**印　　张：** 27.5
书　　号： ISBN 7-5083-1495-6
版　　次： 2003年6月北京第一版
印　　次： 2003年6月第一次印刷
定　　价： 39.00 元

Foreword

For more than a decade, readers of C++ *Primer* pursued Stan down conference halls, cornering him in elevators, surrounding him at book signings, and inundating him with e-mail asking where, when, why, and — well then — why aren't the answers to the exercises available?

Well. Hmm. Eyes fixed on his shoelaces. "It's a tradition," he finally answered. "Within Bell Laboratories, I mean. No answers to the exercises. At least, not within the text." Silence. He knew what many people were thinking: yep, Kernighan and Ritchie (K&R) had an answer book. So did Bjarne. "Well. Hey." (Diffident shrug.) Stan never dreamed he'd ever see an independent answer book for the *Primer*.

Well, that was then. Now with great satisfaction — voilà — we introduce C++ *Primer Answer Book*!

The involvement of Clovis Tondo is something of a tradition for both C++ *Primer* and *Inside the Object Model*. He first reviewed the *Primer* in 1986 during an early draft of the first edition; his comments were both helpful and encouraging. (The value of encouragement for a newbie author should not be underestimated!) He followed that up with an excellent critique of the second edition as well as many helpful comments and encouragement on *Inside the Object Model*. (Hey, the value of encouragement for a veteran author should not be underestimated!) Finally, as a reviewer of the third edition, he provided an exceptionally in-depth reading.

When it was suggested to us that Dr. Tondo, together with Bruce Leung, might author an answer book for the third edition, it felt like a perfect fit, as if he had been preparing for just this book these past ten years. Of course, he had done *The C Answer Book* for the K&R text as well, so this makes for a second tradition. It's one that we are thrilled to be a part of.

The exercises within C++ *Primer* are intended (a) to reinforce the key elements of the material presented within the preceding section, (b) to provide a concrete set of design and programming tasks to allow the reader to exercise his or her new-won expertise, and in some few occasions (c) to provoke consideration of C++ as a language invention in which theoretical concerns at times take a back seat to practical considerations. These latter exercises do not call so much for an answer as a more or less well-articulated position.

Are all the exercises we presented in the *Primer* absolutely the best we could have done? Reading through the answers presented here, it is clear that a small number of the exercises could have undergone some revision. And so we appreciate the forbearance of the authors.

In one case, we accidentally requested that an `OrQuery` binary operator be derived from an abstract `UnaryOperator` class. Sorry! Stan was sure he wrote `NotQuery`! That was certainly the class he intended to have revised.

C++ Primer Answer Book, then, can be grouped into two primary categories. The first is answers to the exercises in which an item is either correct or incorrect. This material provides a useful summary of the salient points of each section in addition to providing the solutions. The second category covers solutions to the design and programming tasks. These are our favorites, in general, perhaps because they offer the most surprises. After all, we know category 1 pretty well. This latter category allowed the authors to exercise their expertise. The programs in Chapters 6, 12, and 17 are particularly impressive.

We think you'll both enjoy and learn from *C++ Primer Answer Book*. And we extend a warm thanks to the two authors for their work.

Stanley B. Lippman
Josée Lajoie

Preface

C++ has proven to be a popular programming language, and *C++ Primer* has proven to be just as popular among those wishing to learn the language. Learning a programming language, however, requires more than just reading about the language constructs. You must program, write your own code, and study code written by those who are more experienced with the use of the language.

To this end, Lippman and Lajoie (L&L) have provided exercises throughout *C++ Primer* to encourage their readers to test their understanding of the material. Our book provides the solutions to those exercises.

C++ Primer Answer Book is intended to be used in conjunction with *C++ Primer*. We assume that you have read the material in L&L preceding each exercise. We present our solution along with an explanation, but we do not repeat the material found in L&L. Only those concepts and constructs that have already been introduced are used in the solutions.

When the solution involves a complete program, we generally include the entire source code so that each solution stands on its own. All programs have been compiled using Microsoft Visual C++ Version 5.0. In some instances when the compiler did not meet the standard, a workaround was used and an explanation given.

We recommend that you use L&L to learn C++, work the exercises, and study the solutions presented here. We hope *C++ Answer Book* will aid you in understanding C++ while eliminating the frustration of being stuck without an answer to a problem.

Acknowledgments

Special thanks go to Stan Lippman and Josée Lajoie for having the faith to allow us to write this answer book and for their careful review of the text.

Sean Davey, Steve Edwards, S. Rollins Guild, Cay Horstmann, and Jeffrey Oldham provided many helpful comments on the manuscript.

Clovis thanks George Edmunds, Sam Hsu, Mohammad Ilyas, Mahesh Neelakanta, and Cyril Párkányi for their continued support; Sean Davey for the C++ review, for the LaTeX macros, and for his technical support; Andrew Nathanson for his friendship and software/hardware support; S. Rollins Guild for the C++ review and his friendship; A. Carlos

Tondo, Julia Mistrello, and Luiz and Elizabete T. Biavatti for helping our company succeed; and Caren E. Tondo for her love, patience, and sense of humor.

Bruce thanks Andrew Bellezza, Jodi Solomon, and Mary Walstrom for their friendship and encouragement; Zahira Ammarguellat, Luddy Harrison, Sandra Loosemore, and Cotton Seed — one couldn't ask for better co-workers; and Misty and Buddy for their boundless patience.

Last, but certainly not least, we thank the staff at Addison-Wesley. We are especially grateful to our associate editor, Debbie Lafferty, for her patience and knowing when and how hard to push; the production editor, Maureen Willard, for guiding us through the editing, proofs, and final pages; the production manager, John Fuller, for getting the macros approved and improved early in the process; and the freelancer, Diane Freed, for assisting with the production of this book. We appreciate your kind help.

Clovis L. Tondo
Bruce P. Leung

Contents

chapter one

Getting Started

Chapter 1 of the *C++ Primer* does not contain any exercises.

chapter two

A Tour of C++

Exercise 2.1

Why do you think the built-in array does not support the assignment of one array with another? What information is required to support this operation?

The name of a built-in array is a synonym for an address, so assigning one array to another is similar to assigning the constant 5 to the constant 3. This is syntactically correct but semantically generates an error message.

The language was not designed to handle array assignments. The compiler would have to know the length of the arrays at run-time to generate code to support the assignment of one array with another.

Exercise 2.2

What operations should a first class array support?

"Although C++ provides built-in support for an array type, that support is limited to the mechanics required to read and write individual elements. C++ does not support the *abstraction* of an array; there is no support for the operations one might wish to perform on an array, such as the assignment of one array to another, the comparison of two arrays for equality, or asking an array its size" (L&L, page 26). That is, a first class array support should allow us to treat the array as a unit as well as allow us to access the individual elements of the array.

The operations that a first class array should support include

- Array initialization
- Array assignment
- Comparison of arrays

- Array size
- Index range check

Exercise 2.3

Explain the difference between the four objects defined below.

```
(a) int ival = 1024;   (c) int *pi2 = new int( 1024 );
(b) int *pi = &ival;   (d) int *pi3 = new int[ 1024 ];
```

`(a) int ival = 1024;`

The object `ival` has static or automatic memory allocation. The declaration "instructs the compiler to allocate sufficient storage to hold any value of type `int`, associate the name `ival` with that storage, and then place an initial value of 1024 in that storage" (L&L, page 27).

`(b) int *pi = &ival;`

The object `pi` has a pointer type that may hold the memory address of an `int`. The *address-of* operator (`&`) returns the address of the `int` object `ival`. So `pi` is a pointer to an `int`, and it has been initialized with the address of the `int` object `ival`.

`(c) int *pi2 = new int( 1024 );`

The object `pi2` has a pointer type that may hold the memory address of an `int`. The `new` expression allocates a single unnamed object of type `int`, initializes that object to a value of 1024, and then returns the address of the object in memory, and `pi2` is initialized with that address.

`(d) int *pi3 = new int[ 1024 ];`

The object `pi3` has a pointer type that may hold the memory address of an `int`. The `new` expression allocates an array of 1024 integer elements (it does not initialize the elements of the array) and then returns the address of the first element of the array in memory, and `pi3` is initialized with that address.

The object `ival` is of type `int` and may hold any positive or negative integer value that can be contained in the underlying machine support of an `int`. So `(a)` holds a value, whereas the others (`(b)`, `(c)`, and `(d)`) are pointers that hold addresses.

Exercise 2.4

What does the following code fragment do? What is its significant error? (Note that the use of the subscript operator with the pointer `pia`, below, is correct; the reason we can do this is explained in Section 3.9.2.)

```
int *pi = new int( 10 );
int *pia = new int[ 10 ];
```

```
while ( *pi < 10 ) {
    pia[ *pi ] = *pi;
    *pi = *pi + 1;
}
delete pi;
delete [] pia;
```

The preceding code fragment attempts to initialize the array of `int`s that `pia` points to. `pi` points to an unnamed `int` object initially set to 10. The test expression in the `while` loop compares the object that `pi` points to against 10 and executes the body of the loop as long as the object has a value less than 10. The problem is that the test will fail the first time, and the body of the loop will not be executed at all. One possible way to fix the initialization of the array that `pia` points to is

```
int *pi = new int( 10 );
int *pia = new int[ 10 ];
int i = 0;

while ( i < 10 ) {
    pia[ i ] = *pi;
    i++;
}
```

The remainder of the code fragment correctly deallocates the unnamed `int` object that `pi` points to and then deallocates the array of `int`s that `pia` points to.

Exercise 2.5

The key feature of a C++ class is the separation of interface and implementation. The interface is the set of operations that users can apply to objects of the class. It consists of three parts: the name of the operations, their return values, and their parameter lists. Generally, that is all the user of the class is required to know. The private implementation consists of the algorithms and data necessary to support the public interface. Ideally, even though the class interface may grow, it does not change in ways incompatible with earlier versions during the lifetime of the class. The implementation, on the other hand, is free to evolve over the lifetime of the class. Choose one of the following abstractions and write the public interface for that class:

```
(a) Matrix      (c) Person      (e) Pointer
(b) Boolean     (d) Date        (f) Point
```

(a) Matrix

```
class Matrix {
  public:
```

```
                                // default constructor
    Matrix( int = defaultRowSize,
            int = defaultColumnSize );
    Matrix( const Matrix & );   // copy constructor
    ~Matrix();                  // destructor
                                // copy assignment operator
    Matrix &operator=( const Matrix & );
                                // equality operator
    bool operator==( const Matrix & ) const;
                                // inequality operator
    bool operator!=( const Matrix & ) const;
                                // indexing operators
    int &operator()( int, int );
    int operator()( int, int ) const;
    int getRowSize() const;     // number of rows
    int getColumnSize() const;  // number of columns
                                // iostream output operator
    friend ostream &operator<<( ostream &, const Matrix & );
                                // iostream input operator
    friend istream &operator>>( istream &, Matrix & );
  private:
    int **im;                   // implementation...
    int rows;                   // how many rows
    int cols;                   // how many columns
    static const int defaultRowSize = 5;
    static const int defaultColumnSize = 5;
};
```

For the class `Matrix` we provided two constructors:

1. The default constructor `Matrix()` (see L&L, page 698)
2. A copy constructor that expects another `Matrix` object as its argument (see L&L, page 700)

Destructors, unlike constructors, cannot take any argument. So we can create a single destructor for `Matrix`.

We provided a copy assignment operator

```
Matrix &operator=( const Matrix & );
```

and equality and inequality operators

```
bool operator==( const Matrix & ) const;
bool operator!=( const Matrix & ) const;
```

(see L&L, page 738). We also provided Fortran-style index operators

```
int &operator()( int, int );
int operator()( int, int ) const;
```

and iostream output and input operators:

```
friend ostream &operator<<( ostream &, const Matrix & );
friend ostream &operator>>( ostream &, Matrix & );
```

L&L have not yet discussed `friend` functions, but we included the << and >> operators to obtain a reasonably complete interface. See Section 13.5 in L&L.

Exercise 2.6

The words *constructor* and *destructor* are somewhat misleading in that these programmer-supplied functions neither construct nor destroy the objects of the class to which they are applied by the compiler automatically. When we write

```
int main() {
    IntArray myArray( 1024 );
    // ...
    return 0;
}
```

the memory necessary to maintain the data members of `myArray` is allocated prior to application of the constructor. The compiler, in effect, internally transforms our program as follows (note that this is not legal C++ code):

```
int main() {
    IntArray myArray;

    // Pseudo C++ Code -- apply constructor
    myArray.IntArray::IntArray( 1024 );
    // ...
    // Pseudo C++ Code -- apply destructor
    myArray.IntArray::~IntArray();

    return 0;
}
```

The constructors of a class serve primarily to initialize the data members of the class object. The destructor primarily frees any resources acquired by the class object during its lifetime. Define the set of constructors needed by the class you chose in Exercise 2.5. Does your class require a destructor?

The constructors are

```
Matrix( int = defaultRowSize, // default constructor
        int = defaultColumnSize );
Matrix( int, int, int * );    // constructor
Matrix( const Matrix & );     // copy constructor
```

See L&L, page 698, for a discussion of default constructors. On page 700, L&L address copy constructors. We added the other constructor to be able to initialize a `Matrix` with an array of `int`s (see Exercise 2.7).

We have a destructor because `Matrix` has a data member that is a pointer. Remember that if a class needs a copy constructor, a copy assignment operator, or a destructor, the class needs them all. And a class needs them all if the class has at least one pointer that addresses heap memory allocated within the class constructor and that must be freed by the destructor.

Exercise 2.7

In Exercises 2.5 and 2.6, you have specified nearly the complete public interface necessary for use of the class. (We may still need to define a copy assignment operator, but we'll ignore that fact for now — C++ provides default support for the assignment of one class object with another. The problem is that this default behavior is often inadequate. See Section 14.6 for a discussion.) Write a program to exercise the public interface of the class you defined in the two previous exercises. Is it easy or awkward to use? Do you wish to revise the specification? Can you do that and maintain compatibility?

When we implemented the class we added two private `init()` functions.

```
class Matrix {
  public:
                                // default constructor
    Matrix(int = defaultRowSize,
           int = defaultColumnSize);
    Matrix(int, int, int *);    // constructor
    Matrix(const Matrix &);     // copy constructor
    ~Matrix();                  // destructor
                                // assignment operator
    Matrix &operator =(const Matrix &);
    int &operator()(int, int);
    const int &operator()(int, int) const;
    friend ostream &operator <<(ostream &, const Matrix &);
  private:
    int **im;                   // implementation...
    int rows;                   // how many rows
    int cols;                   // how many columns
    static const int defaultRowSize;
    static const int defaultColumnSize;
    void initMatrix(int *);     // init with a row
    void initMatrix(int **);    // init with a matrix
    void initData(int r, int c);
};
```