HZ BOOKS
华章教育

全美经典
学习指导系列

数据结构习题与解答

(英文版)

C++ 语言描述

DATA STRUCTURES WITH C++

455 solved problems step-by-step

Ideal for independent study!

Conforms to the new ANSI/ISO Standard for C++

Download solutions to all examples and problems from author's web page

全球销售超过 3000 万册!

(美) John R. Hubbard 著

Mc Graw Hill

机械工业出版社
China Machine Press

全美经典
学习指导系列

数据结构习题与解答

C++ 语言描述

（英文版）

DATA STRUCTURES WITH c++

（美）John R. Hubbard 著

机械工业出版社
China Machine Press

本书版权登记号：图字：01-2002-3652

图书在版编目（CIP）数据

数据库结构习题与解答：C++语言描述/（美）哈伯特（Hubbard, J. R.）著. －北京：机械工业出版社，2002.8

（全美经典学习指导系列）

书名原文：Data Structures With C++

ISBN 7-111-10579-6

Ⅰ. 数… Ⅱ. 哈… Ⅲ. ① 数据结构－习题－英文 ② C语言－程序设计－习题－英文 Ⅳ. TP311.12-44

中国版本图书馆CIP数据核字（2002）第051855号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：华 章

北京忠信诚印刷厂印刷 · 新华书店北京发行所发行

2002年8月第1版第1次印刷

787mm × 1092mm 1/16 · 26.25印张

印数：0 001－3 000册

定价：40.00元

Preface

Like all Schaum's Outline Series books, this is intended to be used primarily for self study, preferably in conjunction with a regular course in data structures using the C++ programming language.

The book includes over 200 examples and problems. The author firmly believes that the principles of data structures can be learned from a well-constructed collection of examples with complete explanations. This book is designed to provide that support.

Source code for all the examples and problems in this book may be downloaded from the author's Web sites: `http://www.richmond.edu/~hubbard`, `http://www.jhubbard.net`, or `http://www.projectEuclid.net`. These sites also contain any corrections and addenda for the book.

I wish to thank all my friends, colleagues, students, and the McGraw-Hill staff who have helped me with the critical review of this manuscript. Special thanks to my wife, Anita Hubbard for her advice, encouragement, and supply of creative problems for this book. Many of the original ideas used here are hers.

JOHN R. HUBBARD
Richmond, Virginia

Contents

Chapter 1

Review of C++

This chapter reviews the essential features of C++. For more detail see the books **[Stroustrup2]** and **[Hubbard1]** listed in Appendix A.

1.1 THE STANDARD C++ PROGRAMMING LANGUAGE

The C++ programming language was invented by Bjarne Stroustrup in 1980 while he was building a distributed computing system. He based it upon the C programming language which had been invented in 1972 by Dennis Ritchie at Bell Labs. The name C was used because the language was a successor to a language named B, a typeless programming language invented by Ken Thompson as a successor to a language named BCPL (Basic Combined Programming Language) which was invented by Martin Richards in 1967. The name C++ was used to suggest an incremented C. Stroustrup incremented C by adding classes. Indeed, he first named the language "C with Classes." The classes feature, which facilitates object-oriented programming, came from the Simula programming language, developed in the early 1960s.

In 1998, the C++ programming language was standardized by the International Standards Organization (ISO) and by the American National Standards Institute (ANSI). This new standard includes the Standard Template Library (STL) developed originally by Alexander Stepanov in 1979. The term "Standard C++" refers to this standardized version of the language.

EXAMPLE 1.1 The Standard C++ "Hello World" Program

```
#include <iostream>   // defines the std::cout and std::endl objects
int main()
{ // prints "Hello, World!"
  std::cout << "Hello, World!" << std::endl;
}
Hello, World!
```

The *preprocessor directive* on the first line tells the C++ compiler to include the definitions from the *standard header* file `iostream` that is part of the *Standard C++ Library*. It defines the `cout` object and the `endl` object in the `std` namespace. The *scope resolution operator* `::` is used to indicate the location of those definitions.

EXAMPLE 1.2 Using the Standard `std` Namespace

```
#include <iostream>   // defines the std::cout and std::endl objects
using namespace std;  // renders the std:: prefix unnecessary
int main()
{ // prints "Hello, World!"
  cout << "Hello, World!" << endl;
}
Hello, World!
```

The using declaration on the second line adds the name std to the local scope, obviating the std:: scope resolution prefix on the cout and endl objects.

All the remaining programs in this book are assumed to begin with the following two lines:

```
#include <iostream>
using namespace std;
```

If you are using a pre-Standard compiler, use this single line instead:

```
#include <iostream.h>
```

EXAMPLE 1.3 The Quadratic Formula

```
#include <cmath>  // defines the sqrt() function
int main()
{ // implements the quadratic formula
  double a, b, c;
  cout << "Enter the coefficients of a quadratic equation:\n";
  cout << "\ta: ";  cin >> a;
  cout << "\tb: ";  cin >> b;
  cout << "\tc: ";  cin >> c;
  cout << "The equation is: " << a << "*x*x + " << b
       << "*x + " << c << " = 0\n";
  double d = b*b - 4*a*c;  // discriminant
  double sqrtd = sqrt(d);
  double x1 = (-b + sqrtd)/(2*a);
  double x2 = (-b - sqrtd)/(2*a);
  cout << "The solutions are:\n";
  cout << "\tx1 = " << x1 << endl;
  cout << "\tx2 = " << x2 << endl;
  cout << "Check:\ta*x1*x1 + b*x1 + c = " <<  a*x1*x1 + b*x1 + c;
  cout << "\n    \ta*x2*x2 + b*x2 + c = " <<  a*x2*x2 + b*x2 + c;
}
```

On the first run we input 2, 1, and –3 to solve the quadratic equation $2x^2 + x - 3 = 0$:

```
Enter the coefficients of a quadratic equation:
        a: 2
        b: 1
        c: -3
The equation is: 2*x*x + 1*x + -3 = 0
The solutions are:
        x1 = 1
        x2 = -1.5
Check:  a*x1*x1 + b*x1 + c = 0
        a*x2*x2 + b*x2 + c = 0
```

The program computes the correct solutions, 1 and –1.5, and then checks them by substituting them back into the quadratic expression to get 0.

On the second run we input 2, 1, and 3 to solve the quadratic equation $2x^2 + x + 3 = 0$:

The program outputs the symbol nan for the solutions x_1 and x_2 and for the resulting check calculation. That symbol stands for "not a number." It resulted from the fact that the discriminant $d = b^2 - 4ac < 0$. Consequently, the call sqrt(d) to the square root function failed, returning the value nan. That is a valid value for float and double variables. But it is *idempotent*: when combined arithmetically with any other value, the resulting value is also nan.

```
Enter the coefficients of a quadratic equation:
        a: 2
        b: 1
        c: 3
The equation is: 2*x*x + 1*x + 3 = 0
The solutions are:
        x1 = nan
        x2 = nan
Check:  a*x1*x1 + b*x1 + c = nan
        a*x2*x2 + b*x2 + c = nan
```

1.2 CONDITIONALS

We can improve the program in Example 1.3 by using an `if` statement to handle the negative discriminant case separately.

EXAMPLE 1.4 A More Robust Implementation of the Quadratic Formula

```
#include <cmath>  // defines the function sqrt() function
int main()
{ // implements the quadratic formula
  double a, b, c;
  cout << "Enter the coefficients of a quadratic equation:" << endl;
  cout << "\ta: ";  cin >> a;
  cout << "\tb: ";  cin >> b;
  cout << "\tc: ";  cin >> c;
  cout << "The equation is: " << a << "*x*x + " << b
       << "*x + " << c << " = 0\n";
  double d = b*b - 4*a*c;  // discriminant
  if (d < 0)
  { cout << "The discriminant, d = " << d
         << " < 0, so there are no real solutions.\n";
    return 0;
  }
  double sqrtd = sqrt(d);
  double x1 = (-b + sqrtd)/(2*a);
  double x2 = (-b - sqrtd)/(2*a);
  cout << "The solutions are:\n";
  cout << "\tx1 = " << x1 << endl;
  cout << "\tx2 = " << x2 << endl;
  cout << "Check:";
  cout << "\ta*x1*x1 + b*x1 + c = " <<  a*x1*x1 + b*x1 + c;
  cout << "\n\ta*x2*x2 + b*x2 + c = " <<  a*x2*x2 + b*x2 + c;
}
```

On the same input, 2, 1, and 3, to attempt to solve the quadratic equation $2x^2 + x + 3 = 0$, this version gives more informative output. When the discriminant $d < 0$, the program prints a diagnostic message and then exits by the `return 0` statement.

```
Enter the coefficients of a quadratic equation:
        a: 2
        b: 1
        c: 3
The equation is: 1*x*x + 2*x + 3 = 0
The discriminant, d = -8 < 0, so there are no real solutions.
```

The `if` statement is a *conditional*; its action depends upon the value of a condition, which is a boolean expression. C++ also has a `switch` statement. Its action depends upon the value of an integer expression.

EXAMPLE 1.5 A Simple Calculator Simulation

```
int main()
{ // performs arithmetic on integers
  int m, n;
  cout << "Enter two integers: ";  cin >> m >> n;
  char op;
  cout << "Enter an operator (+,-,*,/,%): ";  cin >> op;
  cout << "\t" << m << op << n << " = ";
  switch (op)
  { case '+': cout << m + n; break;
    case '-': cout << m - n; break;
    case '*': cout << m * n; break;
    case '/': cout << m / n; break;
    case '%': cout << m % n;
  }
}
```

The op variable holds one character. Since `char` is an integral type, it can be used to control the `switch` statement.

```
Enter two integers: 30 7
Enter an operator (+,-,*,/,%): %
        30%7 = 2
```

In this run, the value of op is the character `'%'`, so the statements that follow `case '/':` execute.

Note the need for `break` statements within the cases of the `switch` statement. Without them, control would "fall through," executing all the cases after the one selected.

In addition to the `if` and the `switch` statements, C++ also has the *conditional expression operator* for conditional execution. Its syntax is

`( condition ? value1 : value2 )`

Its value is ***`value1`*** if ***`condition`*** is true, and ***`value2`*** if ***`condition`*** is false.

EXAMPLE 1.6 The Conditional Expression Operator

```
int main()
{ // prints the maximum of two given integers
  int m, n;
  cout << "Enter two integers: ";  cin >> m >> n;
  cout << "Their maximum is " << ( m>n ? m : n );
}
```

```
Enter two integers: 44 33
Their maximum is 44
```

1.3 OPERATORS

An *operator* is a function that takes one or more expressions as input and returns an expression that uses a special infix symbol instead of the usual functional notation. For example, the operator "+" is written "22 + 44" instead of "+(22, 44)". The values that the operator operates on are called its *operands*. The operands of "22 + 44" are 22 and 44.

The five *arithmetic operators* are: `+`, `-`, `*`, `/`, `%`. These are all *binary operators*, which means that they have two operands. The `+` and `-` operators also have unary versions, meaning only one operand.

The arithmetic operators can be combined with the standard assignment operator (`=`) to produce five more *assignment operators*: `+=`, `-=`, `*=`, `/=`, `%=`. For example,

```
x *= y;
```

means multiply `x` by `y`.

The six *relational operators* are: `<`, `>`, `<=`, `<=`, `==`, `!=`. These have the same meanings as the corresponding mathematical operators $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$.

Don't confuse the equality operator `==` with the assignment operator `=`. In mathematics, both operators are represented by the equals sign =. In C++, these two operators have very different effects. The equality operator `==` tests for equality of expressions; it changes no values and returns either `true` or `false`. The assignment operator `=` assigns the vlue of the expression on its right to the object on its left and returns that value. Using the assignment operator in a condition is one of the most common errors made by C++ programmers:

```
if (n = 0) ++k;  // ERROR: assignment operator used by mistake
```

The expression `(n = 0)` evaluates to 0, which is then interpreted to mean `false`. The author of that code probably meant to write

```
if (n == 0) ++k;  // correct usage
```

The three *logical operators* are: `&&`, `||`, `!`. These are also called *boolean operators* because both their operands and their resulting values are *boolean expressions* (expressions of type `bool`).

The `&&` and `||` operators allow for "short circuiting," which means that their second operand is not evaluated unless necessary. For example,

```
if (x == 0 || y/x > 1) ++k;  // OK: will not crash if x is 0
```

Evaluating the expression `y/x > 1` would generate a run-time error (program "crash") if the value of `x` were 0. But if it is, the first operand will evaluate to `true`, causing the second operand to be ignored. Similarly,

```
if (x != 0 && y/x > 1) ++k;  // OK: will not crash if x is 0
```

is also safe because here the second operand will be ignored if the first evaluates to `false`.

Every operator expression has a value and a type. For example, the value of the expression

```
n = 44
```

is 44 and has type `int`. That allows operators to be chained, like this:

```
k = m += n = 44;
```

This means: (1) assign 44 to `n`; (2) add that value (44) to `m`; assign that value to `k`. Chaining assignment operators works from right to left. On the other hand, chaining arithmetic operators and chaining input/output operators work from left to right:

```
z = 88 - x + y;                  // the "-" is evaluated before the "+"
cout << x << ", " << y << ", " << z << "\n";
```

Operators follow standard precedence rules that determine the order of evaluation when several operators are used in the same expression. For example, `*` has higher precedence than `+`, so in the expression `x + y * z`, the expression `y * z` will be evaluated first. And `<` has higher precedence than `||`, so in the expression `(x<4 || y<8)`, the expressions `x<4` and `y<8` will be evaluated first. The following table groups all the C++ operators according to their precedence levels, from highest (`::`) to lowest (`,`).

Precedence of Operators

```
::
., ->, [], (), ++ (post-increment), --(post-decrement), typeid,
dynamic_cast, static_cast, reinterpret_cast, const_cast
~, !, +, - (unary), ++ (pre-increment), --(pre-decrement), new,
delete, & (reference), * (dereference), sizeof
.*, ->*
* (multiply), /, %
+, - (binary)
<<, >>
<, >, <=, >=
==, !=
& (bitwise AND)
^  (bitwise XOR)
| (bitwise OR)
&&
||
=, +=, -=, *=,  /=, %=,  <<=, >>=, &=,  |=, ^=
?:
throw
,
```

These precedence rules can be overridden by using parentheses. For example, in the expression `x * (y + z)`, the expression `y + z` will be evaluated first.

1.4 ITERATION

To *iterate* means to repeat one or more statements until a certain condition is true. This process is called *iteration*. C++ has four mechanisms for producing iteration: the `goto` statement, the `do` statement, the `while` statement, and the `for` statement.

EXAMPLE 1.7 Using a `goto` Loop to Sum Reciprocals

This program computes and prints the sum 1 + 1/2 + 1/3 + · · · + 1/100.

```
int main()
{ const int N=100;
  double sum=0.0;
  int x=1;
  repeat: sum += 1.0/x++;
  if (x<=N) goto repeat;
  cout << "The sum of the first " << N << " reciprocals is " << sum;
}
The sum of the first 100 reciprocals is 5.18738
```

The expression `repeat:` on the fifth line is called a *label.* It locates a line in the program to which execution control can be diverted, as it is on the next line, by means of a `goto` statement. The effect here is that the statement

```
sum += 1.0/x++;
```

is executed repeatedly as long as the condition `(x<=N)` is true. Thus, the values 1.0/1, 1.0/2, 1.0/3, · · · are added to `sum` until the value of `x` exceeds 100.

Note the use of the `const` keyword on the second line. This specifies that the integer N is a constant and prevents its value from being changed. (C++ programmers usually capitalize all the letters of a constant identifier.)

EXAMPLE 1.8 Using a do...while Loop to Sum Reciprocals

```
int main()
{ const int N=100;
  double sum=0.0;
  int x=1;
  do sum += 1.0/x++;
  while (x <= N);
  cout << "The sum of the first " << N << " reciprocals is " << sum;
}
The sum of the first 100 reciprocals is 5.18738
```

This program is the same as the program in Example 1.7 except that the keywords **`do`** and **`while`** are used in place of the **`repeat`** label and the **`goto`** keyword. The effect is the same.

EXAMPLE 1.9 Using a while Loop to Sum Reciprocals

```
int main()
{ const int N=100;
  double sum=0.0;
  int x=1;
  while (x <= N)
    sum += 1.0/x++;
  cout << "The sum of the first " << N << " reciprocals is " << sum;
}
The sum of the first 100 reciprocals is 5.18738
```

This program is the same as the program in Example 1.8 except that the keyword `do` is not used and the `while` condition is placed ahead of the statement to be repeated. The effect is the same.

EXAMPLE 1.10 Using a for Loop to Sum Reciprocals

```
int main()
{ const int N=100;
  double sum=0.0;
  for (int x=1; x <= N; x++)
    sum += 1.0/x;
  cout << "The sum of the first " << N << " reciprocals is " << sum;
}
```

```
The sum of the first 100 reciprocals is 5.18738
```

This program is the same as the program in Example 1.9 except that the keyword `while` is replaced with the keyword `for` and the three expressions `int x=1`, `x <= N`, and `x++` are placed together in a control descriptor delimited by parentheses. The effect is the same.

Like any other statement or block of statements, a loop may be inserted in another loop. The result is called *nested loops*.

EXAMPLE 1.11 Using Nested for Loops to Print a Triangle of Stars

```
int main()
{ const int N=10;
  for (int i=0; i<N; i++)
  { for (int j=0; j<2*N; j++)
      if (j<N-i || j>N+i) cout << " ";
      else cout << "*";
    cout << "\n";
  }
}
```

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
 *******************
```

The outer `for` loop prints one line on each iteration. The inner `for` loop prints one character, either a blank or a star, on each iteration.

1.5 FUNCTIONS

A *function* is a subprogram that can be called (invoked) from another function and can return a value to it. Every C++ program is required to begin with the `main()` function. Relegating separate tasks to separate functions is a fundamental programming technique that leads to simpler and more efficient programs.

EXAMPLE 1.12 Using a Separate Function

```
void printRow(const int, const int);  // prototype

int main()
{ const int N=10;
  for (int i=0; i<N; i++)
    printRow(i,N);
}
void printRow(const int row, const int N)  // implementation
{ for (int j=0; j<2*N; j++)
    if (j<N-row || j>N+row) cout << " ";
    else cout << "*";
  cout << "\n";
}
```

This program has the same results as that in Example 1.11. It has relegated the task of printing one row to the separate `printRow()` function. This is a `void` function because it does not return anything to `main()`. It has two `int` parameters: `row` and `N`. They are passed values from the arguments `i` and `N`.

The function is declared by the one-line prototype above `main()`, and it is defined by its complete implementation below `main()`. Note that the prototype omits the parameter names (optional) and ends with a semicolon (required).

Notice that both parameters are declared to be `const`. It is good programming practice to declare as `const` any object that is intended not to be changed.

EXAMPLE 1.13 A `power()` Function

```
double power(const double, const int);

int main()
{ cout << "power(2,0) = " << power(2,0) << "\n";
  cout << "power(2,1) = " << power(2,1) << "\n";
  cout << "power(2,2) = " << power(2,2) << "\n";
  cout << "power(2,3) = " << power(2,3) << "\n";
  cout << "power(2,-3) = " << power(2,-3) << "\n";
  cout << "power(2.01,3) = " << power(2.01,3) << "\n";
}

double power(const double x, int n)
{ double y=1.0;
  for (int i=0; i<n; i++)   // if n>0, y = x*x*...*x (n times)
    y *= x;
  for (int i=0; i>n; i--)   // if n<0, y = 1/x*x*...*x (n times)
    y /= x;
  return y;
}
```

```
power(2,0) = 1
power(2,1) = 2
power(2,2) = 4
power(2,3) = 8
power(2,-3) = 0.125
power(2.01,3) = 8.1206
```