PROGRAMMING

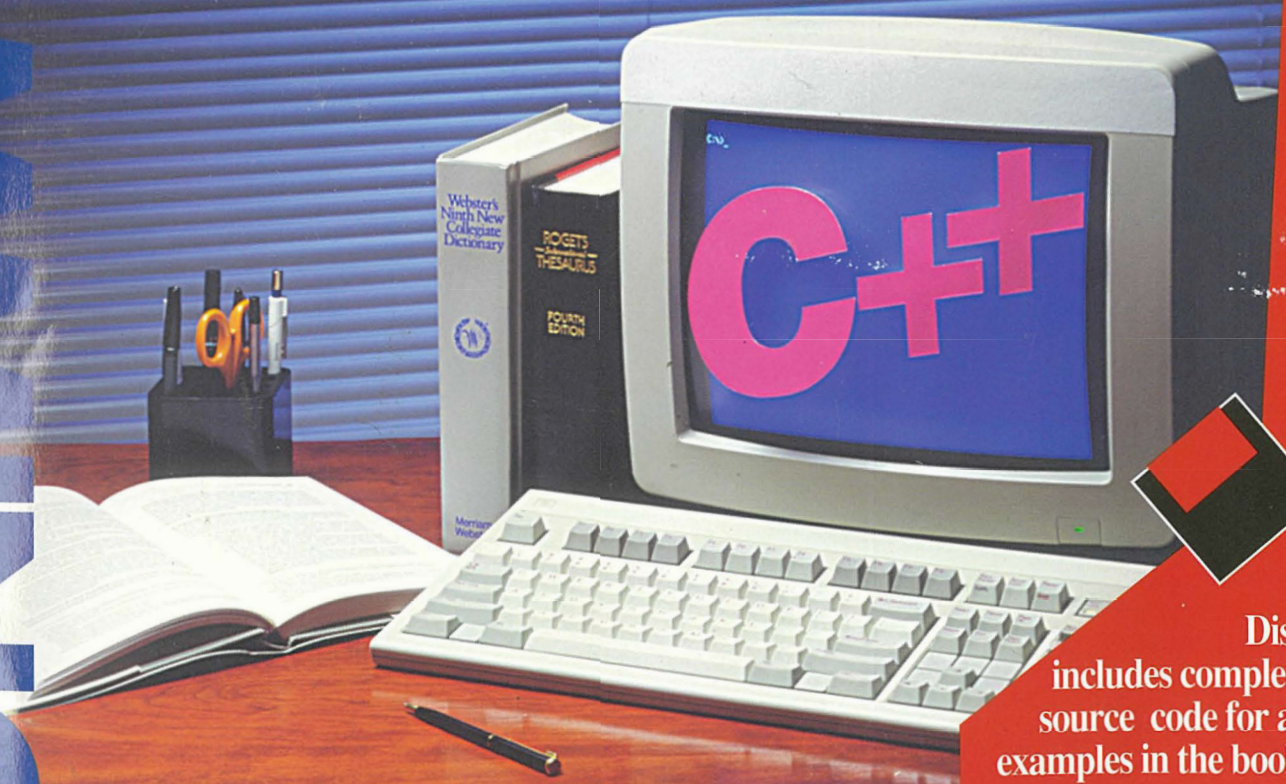
SAMS PUBLISHING

C++ Programming 101

Actively learn C++ with step-by-step instructions

Short, productive lessons for the true beginner

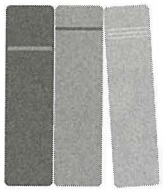
Packed with dozens of examples that you can try right in the book



Disk includes complete source code for all examples in the book, PLUS dozens of bonus exercises and examples

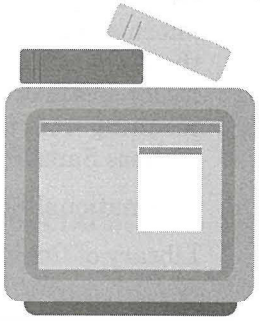
Greg Perry

SAP Exhibit 1006
SAP v. Express Mobile, Inc.
IPR2022-00518
Page 00001



C++ Programming 101

C++ PROGRAMMING 101



Greg Perry

SAMS PUBLISHING

This book is for Dr. Roger Wainwright, my first and only "real" computer teacher.

Copyright© 1992 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. For information, address Sams Publishing, 11711 N. College Ave., Carmel, IN 46032.

International Standard Book Number: 0-672-30200-4

Library of Congress Catalog Card Number: 92-62272

00 10 9

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit, the number of the book's printing. For example, a printing code of 92-1 shows that the first printing of the book occurred in 1992.

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Composed in New Century Schoolbook and MCPdigital by Prentice Hall Computer Publishing

Printed in the United States of America

Publisher
Richard K. Swadley

Acquisitions Manager
Jordan Gold

Acquisitions Editor
Stacy Hiquet

Development Editor
Rosemarie Graham

Editors
Gayle L. Johnson
Keith Davenport
Mary Corder
Tad Ringo

Editorial Coordinators
Rebecca S. Freeman
Bill Whitmer

Editorial Assistant
Lori Kelley

Technical Editor
Timothy C. Moore

Cover Designer
Jean Bisesi

Director of Production and Manufacturing
Jeff Valler

Production Manager
Corinne Walls

Imprint Manager
Matthew Morrill

Book Designer
Michele Laseau

Production Analyst
Mary Beth Wakefield

Proofreading/Indexing Coordinator
Joelynn Gifford

Graphics Image Specialists
Dennis Sheehan
Sue VandeWalle
Jerry Ellis

Production
Katy Bodenmiller
Christine Cook
Lisa Daugherty
Carla Hall-Batton
Howard Jones
John Kane
Sean Medlock
Linda Quigley
Michelle Self
Susan Shepard
Greg Simsic
Angie Trzepacz
Alyssa Yesh

Indexer
Loren Malloy

Overview

Introduction xxv

Classroom 1 Looking at C++

Lessons

1 Introduction to C++ 3
2 The C++ Program 11
3 C++ Comments 19
4 The C++ Program's Format 29

Classroom 2 Working with Data

Lessons

5 C++ Numeric Data 45
6 Characters, Strings, and Character Arrays 65
7 Preprocessor Directives 85
8 I/O with `cout` and `cin` 99

Classroom 3 Operators

Lessons

9 Math Operators and Precedence 115
10 Relational Logic and Operators 137
11 Logical Operators: Combining Relational Operators 157
12 Additional C++ Operators 171
13 The Bitwise Operators 187

Classroom 4 Controlling Program Flow

Lessons

14 Looping with `while` 205

15 The `for` Loop 227

16 More About Loops 247

17 Controlling with `switch` 261

Classroom 5 Scope and Functions

Lessons

18 Using C++ Functions 285

19 Variable Scope 307

20 Passing Values 333

21 Return Values and Prototypes 351

22 Default Arguments and Function Overloading 369

Classroom 6 Supplied Functions

Lessons

23 Input and Output Functions 385

24 C++ Built-In Functions 403

Classroom 7 Arrays and Pointers

Lessons

25 Array Processing 423

26 Working with Arrays 441

27 Multidimensional Arrays 455

28 Introduction to Pointers 469

29 Pointers and Arrays 481

Classroom 8 Structures, Files, and OOP

Lessons

30 Structures 503

31 Arrays of Structures 521

32 Sequential Files 539

33 Random-Access Files 557

34 Introduction to OOP 571

Appendixes

A C++ Keywords 583

B Memory and Numbering Systems 585

C ASCII Table 595

D C++ Operator Precedence Table 605

Index 609

Introduction xxv

Classroom 1 Looking at C++

Lessons

1 Introduction to C++ 3
- Introduction to Programming 4
- The Programming Process 4
- Exterminating Pests 6
- C++ Compared with Other Languages 7
- C++ and Microcomputers 9

2 The C++ Program 11
- Your First C++ Program 11
- C++ Special Characters 12
- Freeform Style 13
- Uppercase and Lowercase 14
- Longer Programs 16
- Review Questions 17

3 C++ Comments 19
- The Importance of Comments 19
- A Program's Comments 20
- The Syntax of C++ Comments 21
- Comments Are For You Too 24
- Use Good Judgment 25
- C-Style Comments 26
- Review Questions 27
- Review Exercises 27

4 The C++ Program's Format 29
- The Format of Every C++ Program 29
- Dissecting a Program 30
  - The Program 31
  - Braces and `main()` 32
  - Data 34

Assigning Values to Variables 37
Output to the Screen 38
Returning to the Operating System 38
Program Structure 39
Review Questions 40
Review Exercises 40

Classroom 2 Working with Data

Lessons

5 C++ Numeric Data 45
Data and Information 45
Data in a C++ Program 47
Characteristics of Variables 48
Naming Variables 48
Variable Types 50
Defining Variables 50
Looking at Data Types 52
Literals 56
Assigning Integer Literals 56
Long, Unsigned, and Floating-Point Literals 57
Putting Values in Variables 58
Constant Variables 61
Review Questions 63
Review Exercises 63

6 Characters, Strings, and Character Arrays 65
Character Literals and Variables 66
String Literals 67
String-Literal Endings 68
String Lengths 69
Special Character Literals 71
Math with C++ Characters 74
Character Array Variables 74
Character Arrays Versus Strings 78
Review Questions 83
Review Exercise 84

7 Preprocessor Directives 85
Looking at Preprocessor Directives 85
The `#include` Directive 86
The `#define` Directive 90
Really Change Things 94
Do Not Overdo `#define` 95

Review Questions 96
Review Exercises 97

8 I/O with `cout` and `cin` 99

Comparing `cout` and `cin` 99
Output Options 101
Output Manipulators 102
Input Options 106
Getting Lines of Data 109
Review Questions 110
Review Exercises 111

Classroom 3 Operators

Lessons

9 Math Operators and Precedence 115

Primary Math Operators 115
Unary Operators 117
Division and Modulus 118
Order of Precedence 121
Using Parentheses 124
Assignment Statements 127
Multiple Assignments 127
Compound Assignments 128
Associativity of Compound Assignments 129
Mixing Data Types in Calculations 130
Typecasting 133
Review Questions 135
Review Exercises 136

10 Relational Logic and Operators 137

Defining Relational Operators 138
The `if` Statement 141
Expressions as the Condition 143
Data Validation 147
The `else` Statement 150
Review Questions 155
Review Exercises 156

11 Logical Operators: Combining Relational Operators 157

The Logical Operators 158
Using the Logical Operators 159
Logical Issues 166
C++'s Logical Efficiency 166

IPR2022-00518 Page 00015

The Precedence of Logical Operators 167
Review Questions 168
Review Exercises 169

12 Additional C++ Operators 171

The Conditional Operator 171
The Increment and Decrement Operators 176
Increment and Decrement Efficiency 177
Using Increment and Decrement in Expressions 177
The `sizeof` Operator 181
The Comma Operator 183
Review Questions 185
Review Exercises 186

13 The Bitwise Operators 187

The Bitwise Operators 188
Bitwise 1's Complement 192
Using the Bitwise Operators 193
Compound Bitwise Operators 196
Bitwise Shift Operators 197
Compound Bitwise Shift Operators 200
Review Questions 202
Review Exercises 202

Classroom 4 Controlling Program Flow

Lessons

14 Looping with `while` 205

The `while` Loop 206
The Concept of Loops 208
The `do-while` Loop 213
`if` Versus the `while` Loop 215
The `exit()` Function and the `break` Statement 216
Counters and Totals 219
Producing Totals 223
Review Questions 225
Review Exercises 226

15 The `for` Loop 227

The `for` Statement 228
Another `for` Test 236
Nested `for` Loops 239
Review Questions 244
Review Exercises 245

IPR2022-00518 Page 00016

16 More About Loops 247
- Timing Loops 248
- The `break` and `for` Statements 250
- The `continue` Statement 254
- Review Questions 259
- Review Exercises 260

17 Controlling with `switch` 261
- The `switch` Statement 262
  - The Importance of `break` 268
  - Character `case` Values 271
- The `goto` Statement 273
- Review Questions 279
- Review Exercises 281

Classroom 5 Scope and Functions

Lessons

18 Using C++ Functions 285
- Function Basics 286
- Breaking Problems Down 287
- More Function Basics 291
- Calling and Returning Functions 293
- Review Questions 304

19 Variable Scope 307
- Global Versus Local Variables 307
  - Defining Variable Scope 308
  - Differentiating Between Local and Global 311
  - Use Global Variables Sparingly 317
  - The Need for Passing Variables 318
- Automatic Versus Static Variables 324
- Three Issues of Parameter Passing 329
- Review Questions 329
- Review Exercises 330

20 Passing Values 333
- Passing by Value (Passing by Copy) 334
- Passing by Reference 339
  - Variable Addresses 339
  - Sample Program 340
  - Passing Nonarrays by Reference 345
- Review Questions 348
- Review Exercises 349

21 Return Values and Prototypes 351
Function Return Values 352
Function Prototypes 360
Prototype for Safety 362
Prototype All Functions 363
Review Questions 366
Review Exercises 366

22 Default Arguments and Function Overloading 369
Default Argument Lists 370
Multiple Default Arguments 371
Overloaded Functions 375
Name-Mangling 379
Review Questions 380
Review Exercises 381

Classroom 6 Supplied Functions

Lessons

23 Input and Output Functions 385
Stream and Character I/O 386
The Newline Special Character: \n 386
Standard Devices 387
Redirecting Devices from MS-DOS 388
Printing Formatted Output to the Printer 390
Character I/O Functions 391
The `get()` and `put()` Functions 392
The `getch()` and `putch()` Functions 397
A Better Way to Input Strings 400
Review Questions 401
Review Exercises 402

24 C++ Built-In Functions 403
Character Functions 403
Character Testing Functions 404
Alphabetic and Digital Testing 404
Special Character-Testing Functions 408
Character Conversion Functions 408
String Functions 410
Useful String Functions 410
Converting Strings to Numbers 412

IPR2022-00518 Page 00018

Numeric Functions 413
Useful Mathematical Functions 414
Trigonometric Functions 416
Logarithmic Functions 416
Random-Number Processing 417
Review Questions 419
Review Exercises 419

Classroom 7 Arrays and Pointers

Lessons

25 Array Processing 423
Array Basics 424
The Size of Arrays 426
Array Boundaries 427
Initializing Arrays 429
Initializing Elements at Declaration Time 430
Initializing Elements in the Program 436
Out-of-Order Arrays 438
Review Questions 439
Review Exercises 440

26 Working with Arrays 441
Searching Arrays 442
Searching for Values 443
Sorting Arrays 446
Advanced Referencing of Arrays 449
Review Questions 454
Review Exercises 454

27 Multidimensional Arrays 455
Multidimensional Array Basics 456
Reserving Multidimensional Arrays 458
Mapping Arrays to Memory 459
Initializing Multidimensional Arrays 461
Tables and `for` Loops 463
Review Questions 466
Review Exercises 467

28 Introduction to Pointers 469
Introduction to Pointer Variables 470
Declaring Pointers 470
Assigning Values to Pointers 472
Pointers and Parameters 474

IPR2022-00518 Page 00019

Review of Pointer Storage 475
Arrays of Pointers 477
Review Questions 479

29 Pointers and Arrays 481

Array Names as Pointers 482
Internal Locations 483
Pointer Advantages 484
Using Character Pointers 487
Pointer Arithmetic 490
Arrays of Strings 494
Review Questions 497
Review Exercises 498

Classroom 8 Structures, Files, and OOP

Lessons

30 Structures 503

Introduction to Structures 503
Defining Structures 507
Initializing Structure Data 510
Nested Structures 516
Review Questions 518
Review Exercises 519

31 Arrays of Structures 521

Declaring Arrays of Structures 522
Referencing the Array Structures 524
Building a Structure Array 525
Arrays as Members 530
Seeing the Steps 533
Review Questions 536
Review Exercises 537

32 Sequential Files 539

Why Use a Disk? 540
Types of Disk File Access 541
Sequential File Concepts 542
Opening and Closing Files 543
Writing to a File 547
Writing to a Printer 549
Adding to a File 550
Reading from a File 551
Review Questions 554
Review Exercises 555

IPR2022-00518 Page 00020

33 Random-Access Files 557
Random File Records 558
Opening Random-Access Files 559
The `seekg()` Function 561
Other Helpful I/O Functions 567
Review Questions 569
Review Exercises 570

34 Introduction to OOP 571
OOP Overview 571
Data with Functions 573
Executing the Member Functions 576
Data Hiding with Classes 577
Redefining Access 579
Your OOP Future 580
Where Do You Go From Here? 581
Review Questions 581
Review Exercises 582

Appendixes

A C++ Keywords 583

B Memory and Numbering Systems 585
Computer Memory 586
8 Bits Make a Byte 586
Binary Numbers 588
Hexadecimal Numbers 590
Why Learn Hexadecimal? 591
How Binary and Addressing Relate to C++ 592

C ASCII Table 595

D C++ Precedence Table 605

Index 609

IPR2022-00518 Page 00021

Review Questions

1. Where did the bubble sort get its name?
2. Are the following values sorted in ascending or descending order?

   33 55 78 78 90 102 435 859 976 4092

3. How does C++ interpret the name of an array?
4. Consider the following array definition:

```
char teams[] = {'E','a','g','l','e','s','\0','R',
                'a','m','s','\0'};
```

   What is printed with each of these statements? (Answer "invalid" if the `cout` is illegal.)

   (a) `cout << teams;`
   (b) `cout << teams + 7;`
   (c) `cout << (teams + 3);`
   (d) `cout << teams[0];`
   (e) `cout << (teams + 0)[0];`
   (f) `cout << (teams + 5);`

Review Exercises

1. Write a program to store six of your friends' ages in a single array. Assign the ages in random order. Print the ages, from low to high, on-screen.
2. Modify the program in Exercise 1 to print the ages in descending order.
3. Using the new approach of subscripting arrays, rewrite the programs in Exercises 1 and 2. Always put a 0 in the subscript brackets, modifying the address instead (use `(ages + 3)[0]` rather than `ages[3]`).

Lesson

Multi-Dimensional Arrays

To explain how to access and manipulate C++ arrays of more than one dimension.

This lesson teaches you how to store tables of data in your C++ program. Some data fits in lists, such as the data discussed in the preceding two lessons, and other data is better suited for tables of information. This lesson takes arrays one step further. The previous lessons introduced single-dimensional arrays—arrays that have only one subscript and represent lists of values.

This lesson introduces arrays of more than one dimension, called *multidimensional arrays.* Multidimensional arrays, sometimes called *tables* or *matrices,* have at least two dimensions (rows and columns). Often they have more than two. If you understand single-dimensional arrays, you should have no trouble understanding arrays that have more than one dimension.

Multidimensional Array Basics

A multidimensional array has more than one subscript. Whereas a single-dimensional array is a list of values, a multidimensional array simulates a table of values, or multiple tables of values. The most commonly used table is a two-dimensional table (an array with two subscripts).

C++ does not store multidimensional arrays in the same way as other programming languages. In C++, a multidimensional array is nothing more than an *array of arrays*. Most of the time, however, you can treat a C++ multidimensional array as you would multidimensional arrays in other programming languages.

Suppose that a football team wanted to track its players' total yardage. The team has eight games scheduled, and there are six rushing players on the team. Table 27.1 shows the team's yardage record.

Table 27.1. A football team's yardage record.

Player Name	*Game 1*	*Game 2*	*Game 3*	*Game 4*	*Game 5*	*Game 6*	*Game 7*	*Game 8*
Downing	1	0	2	1	0	0	0	0
Edwards	0	3	6	4	6	4	5	3
Franks	2	2	3	2	1	0	2	3
Grady	1	3	2	0	1	5	2	1
Howard	3	1	1	1	2	0	1	0
Williams	2	3	1	0	1	2	1	2

Do you see that the football table is a two-dimensional table? It has rows (the first dimension) and columns (the second dimension). Therefore, this is a two-dimensional table with six rows and eight columns. (Generally, the number of rows is specified first.)

Each row has a player's name, and each column has a game number, but these are not part of the actual data. The data consists of only 48 values (six rows by eight columns). The data in a two-dimensional table always is the same type of data; in this case, every value is an integer. If it were a table of salaries, every element would be a floating-point or a double floating-point number.

IPR2022-00518 Page 00024

How many elements would a 10-by-15 multidimensional table have?

The number of dimensions, in this case two, corresponds to the dimensions in the physical world. The single-dimensional array is a line, or list of values. Two dimensions represent both length and width. You write on a piece of paper in two dimensions; two dimensions represent a flat surface. Three dimensions represent width, length, and depth. You have seen 3-D movies. Not only do the images have width and height, but they also seem to have depth. Figure 27.1 shows what a three-dimensional array looks like if it has a depth of four, six rows, and three columns. Notice that a three-dimensional table resembles a cube.

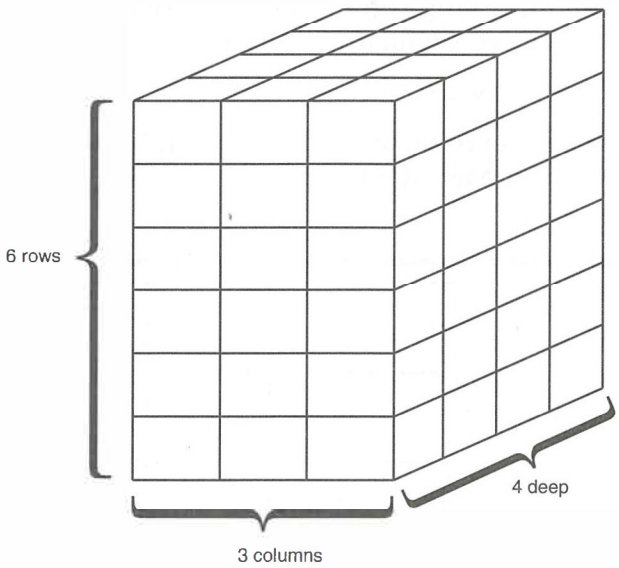


Figure 27.1. A representation of a three-dimensional table (a cube).

It is difficult to visualize more than three dimensions. However, you can think of each dimension after three as another occurrence. In other words, a list of one player's season yardage record can be stored in an array. The team's yardage record is two-dimensional (as shown in Table 27.1). The league, made of up several teams' yardage records, would represent a three-dimensional table. Each team (the depth of the table) has rows and columns of yardage data. Each additional league becomes an additional dimension (another set of data).

C++ enables you to store several dimensions, although "real-world" data rarely requires more than two or three.

Reserving Multidimensional Arrays

When you define storage for a multidimensional array, you must inform C++ that the array has more than one dimension by putting more than one subscript in brackets after the array name. You must include a separate number in brackets for each dimension in the table. For example, to reserve the team data from Table 27.1, you use the following multidimensional array declaration:

```
int teams[8][6];   // Reserves a two-dimensional table
```

Notice that C++ requires you to enclose each dimension in brackets. Most other programming languages do not require two sets of brackets. Do not reserve multidimensional array storage like this:

```
int teams[8,6];   // Invalid table declaration
```

Properly reserving the `teams` table produces a table with 48 elements. Figure 27.2 shows what each element's subscript looks like.

Figure 27.2. Subscripts for the football team table.

columns

[0] [0]	[0] [1]	[0] [2]	[0] [3]	[0] [4]	[0] [5]
[1] [0]	[1] [1]	[1] [2]	[1] [3]	[1] [4]	[1] [5]
[2] [0]	[2] [1]	[2] [2]	[2] [3]	[2] [4]	[2] [5]
[3] [0]	[3] [1]	[3] [2]	[3] [3]	[3] [4]	[3] [5]
[4] [0]	[4] [1]	[4] [2]	[4] [3]	[4] [4]	[4] [5]
[5] [0]	[5] [1]	[5] [2]	[5] [3]	[5] [4]	[5] [5]
[6] [0]	[6] [1]	[6] [2]	[6] [3]	[6] [4]	[6] [5]
[7] [0]	[7] [1]	[7] [2]	[7] [3]	[7] [4]	[7] [5]

rows

If you had to track three teams, each with 15 players and 10 games, the three-dimensional table would be created as follows:

```
int teams[3][15][10]; // Reserves a three-dimensional
                      // table
```

When creating a two-dimensional table, always put the maximum number of rows first and the maximum number of columns second. C++ always uses 0 as the starting subscript of each dimension. The last element, the lower-right element of the `teams` table, is `teams[7][5]`.

A programmer wants to print the first and last values of a four-dimensional table named `MyData` that she defined with this statement:

```
double MyData[2][10][3][60];    // Defines the table
```

After filling the table with values, she writes the following `cout` statements, but they do not produce the first or the last values in the table as she wants to do. What is she doing wrong?

```
cout << MyData[1][1][1][1] << "\n";  // Prints the first value
cout << MyData[2][10][3][60] << "\n";  // Prints the last value
```

How would you define an array named `utilities` that holds five years' worth of monthly utility bill payments from your four utility companies?

Mapping Arrays to Memory

C++ approaches multidimensional arrays a little differently than most programming languages do. When you use subscripts, you do not have to understand the internal representation of multidimensional arrays. However, most C++ programmers think that a deeper understanding of these arrays is important, especially when programming advanced applications.

Remember, a two-dimensional array is actually an *array of arrays*. You program multidimensional arrays as though they were tables with

IPR2022-00518 Page 00027

rows and columns. A two-dimensional array is actually a single-dimensional array, but each of its elements is not an integer, floating-point, or character, but another array.

Knowing that a multidimensional array is an array of other arrays is critical when passing and receiving such arrays. C++ passes all arrays, including multidimensional arrays, by address. Suppose that you were using an integer array called `scores`, reserved as a 5-by-6 table. You could pass `scores` to a function called `print_it()`, as follows:

```
print_it(scores);   // Passes table to a function
```

The function `print_it()` must identify the type of parameter being passed to it. The `print_it()` function also must recognize that the parameter is an array. If `scores` were single-dimensional, you could receive it as

```
print_it(int scores[])   // Works only if scores
                         // is single-dimensional
```

or

```
print_it(int scores[10])   // Assuming that scores
                           // has 10 elements
```

If `scores` were a multidimensional table, you would have to designate each pair of brackets and put the maximum number of subscripts in its brackets, as in

```
print_it(int scores[5][6])   // Informs print_it() of
                             // the array's dimensions
```

or

```
print_it(int scores[][6])   // Informs print_it() of
                            // the array's dimensions
```

Notice that you do not have to explicitly state the maximum subscript on the first dimension when receiving multidimensional arrays, but you must designate the second. If `scores` were a three-dimensional table, dimensioned as 10-by-5-by-6, you would receive it with `print_it()` as

```
print_it(int scores[][5][6])   // Only first dimension
                               // is optional
```

or

```
print_it(int scores[10][5][6])   // Inform print_it() of
                                 // array's dimensions
```

Write a statement that passes a four-dimensional table named `Bills`, defined with this statement:

```
double Bills[3][5][2][7];
```

to a function named `PayThem()`.

Write the first line of `Bills` that receives the `Bills` table.

Initializing Multidimensional Arrays

C++ enables you to initialize a multidimensional array when you define it. As with single-dimensional arrays, you initialize multidimensional arrays with braces that designate dimensions. Because a multidimensional array is an array of arrays, you can nest braces when you initialize them.

The following three array definitions fill the three arrays `ara1`, `ara2`, and `ara3`, as shown in Figure 27.3:

```
int ara1[5] = {8, 5, 3, 25, 41};    // Single-dimensional array
int ara2[2][4] = {{4, 3, 2, 1},{1, 2, 3, 4}};
int ara3[3][4] = {{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
```

Notice that the multidimensional arrays are stored in row order. In `ara3`, the first row receives the first four elements of the definition (1, 2, 3, and 4).

Fill in the boxes of Figure 27.4 to show how the following statement initializes a table.

```
int Table[4][2] = {2, 1, 5, 4, 6, 7, 4, 2};    // Fill up Table
```

To make a multidimensional array initialization look more like the array's subscripts, some programmers like to show how arrays are filled. Because C++ programs are free-form, you can initialize `ara2` and `ara3` as

```
int ara2[2][4] = {{4, 3, 2, 1},    // Does exactly the same
                  {1, 2, 3, 4}};   // thing as before
```

and

```
int ara3[3][4] = {{1, 2, 3, 4},
                  {5, 6, 7, 8},
                  {9,10, 11, 12};   // Visually more
                                    // obvious
```

Figure 27.3. After initializing a table.

	[0]	[1]	[2]	[3]	[4]
ara 1	8	5	3	25	41

ara 2

4	3	2	1
1	2	3	4

ara 3

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

Figure 27.4. Fill in the values for the `Table` *elements.*

Table

You can initialize a multidimensional array as if it were single-dimensional in C++. You must keep track of the row order if you do this. For instance, the following two definitions also reserve storage for and initialize `ara2` and `ara3`:

```
int ara2[2][4] = {4, 3, 2, 1, 1, 2, 3, 4};
int ara3[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

There is no difference between initializing `ara2` and `ara3` with or without the nested braces. The nested braces seem to show the dimensions and how C++ fills them a little better, but the choice of using nested braces is yours.

Rewrite the following array definition to show its two dimensions in the list of values that initialize it. (Break the list of values into a row-and-column appearance using more braces.)

```
char Initials[3][3] = {9,8,7,6,5,4,3,2,1};
```

TIP: Multidimensional arrays (unless they are global) are not initialized to specific values unless you assign them values at declaration time or in the program. As with single-dimensional arrays, if you initialize one or more of the elements, but not all of them, C++ fills the rest with zeros. If you want to fill an entire multidimensional array with zeros, you can do so with the following:

```
float sales[3][4][7][2] = {0};    // Fills all sales
                                  // with zeros
```

Tables and `for` Loops

As the following examples show, nested `for` loops are useful when you want to loop through every element of a multidimensional table.

For instance, this section of code

```
for (row = 0; row < 2; row++)
   { for (col = 0; col < 3; col++)
```


IPR2022-00518 Page 00031

```
        { cout << row << " " << col "\n"; }
    }
```

produces the following output:

```
0    0
0    1
0    2
1    0
1    1
1    2
```

These numbers are the subscripts, in row order, for a two-row by three-column table dimensioned with

```
int table[2][3];
```

Notice that there are as many `for` loops as there are subscripts in the array (two). The outside loop represents the first subscript (the rows), and the inside loop represents the second subscript (the columns). The nested `for` loop steps through each element of the table.

You can use `cin`, `gets()`, `get`, and other input functions to fill a table, and you also can assign values to the elements when declaring the table. More often, the data comes from data files on the disk. Regardless of what method is used to store the values in multidimensional arrays, nested `for` loops are excellent control statements to step through the subscripts.

Suppose that you used the following statement to reserve enough memory elements for a television station's ratings (A through D) for one week:

```
char ratings[7][48];
```

These statements reserve enough elements to hold seven days (the rows) of ratings for each 30-minute time slot (48 of them in a day). Every element in a table is always the same type. In this case, each element is a character variable. Some are initialized with the following assignment statements:

```
shows[3][12] = 'B';    // Stores B in 4th row, 13th column
shows[1][5] = 'A' ;    // Stores C in 2nd row, 6th column
shows[6][20] = getch();    // Stores the letter that the
                           // user types
```

The following complete program defines, initializes, and prints the values from a two-dimensional table. Suppose that a video store sells two grades of video tapes, High Grade and Regular. Each grade of tape comes in one of four lengths: 2-hour, 4-hour, 6-hour, or 8-hour. The tape inventory is well-suited for a two-dimensional table. The store determined that the tapes should have the following retail prices:

	2-Hour	*4-Hour*	*6-Hour*	*8-Hour*
High Grade	2.39	2.85	3.29	3.99
Regular	1.75	2.19	2.69	2.95

The store wants to store the price of each tape in a table for easy access. Typically, you use a nested `for` loop to print rows and columns.

To add the titles, simply print a row of titles before the first row of values, then print a new column title before each column. The following program stores the tape table, initializes it, and prints the output to the screen.

```
// Filename: TAPE.CPP
// Initializes and prints tape prices
// in a table format with titles
#include <iostream.h>
#include <iomanip.h>
void main()
{
   float tapes[2][4];   // Table of tape prices
   int row, col;

   tapes[0][0] = 2.39;   // Row 1, column 1
   tapes[0][1] = 2.85;   // Row 1, column 2
   tapes[0][2] = 3.29;   // Row 1, column 3
   tapes[0][3] = 3.99;   // Row 1, column 4
   tapes[1][0] = 1.75;   // Row 2, column 1
   tapes[1][1] = 2.19;   // Row 2, column 2
   tapes[1][2] = 2.69;   // Row 2, column 3
   tapes[1][3] = 2.95;   // Row 2, column 4

   // Prints the column titles
   cout << "\t\t2-Hour\t\t4-Hour\t\t6-Hour\t\t8-Hour\n";

   // Prints the prices
   for (row = 0; row < 2; row++)
```

```
        { if (row == 0)
         { cout << "High Grade\t"; }
          else
            { cout << "Regular\t\t"; }
          for (col=0; col<4; col++)   // Prints the current row
              { cout << setprecision(2) << "$" << tapes[row][col]
                     << "\t\t";
             }
    }

    return;
}
```

Here is the program's output:

```
              2-Hour      4-Hour      6-Hour      8-Hour
High Grade    $2.39       $2.85       $3.29       $3.99
Regular       $1.75       $2.19       $2.69       $2.95
```

Review Questions

1. What statement reserves a two-dimensional table of integers called `scores` with five rows and six columns?
2. What statement reserves a three-dimensional table of four character arrays called `initials` with 10 rows and 20 columns?
3. In the following statement, which subscript (first or second) represents rows and which represents columns?

   ```
   int weights[5][10];
   ```

4. How many elements are reserved with the following statement?

   ```
   int ara[5][6];
   ```

5. The following table of integers is called `ara`:

4	1	3	5	9
10	2	12	1	6
25	42	2	91	8

What values do the following elements contain?

(a) `ara[2][2]`

(b) `ara[0][1]`

(c) `ara[2][3]`

(d) `ara[2][4]`

6. What control statement is best for stepping through multidimensional arrays?

7. Notice the following section of a program:

```
int grades[3][5] = {80,90,96,73,65,67,90,68,92,84,70,
                    55,95,78,100};
```

What are the values of the following?

(a) `grades[2][3]`

(b) `grades[2][4]`

(c) `grades[0][1]`

Review Exercises

1. Write a program that stores and prints the numbers from 1 to 21 in a 3-by-7 table. (*Hint:* Remember that C++ begins subscripts at 0.)

2. Write a program that reserves storage for three years' worth of sales data for five salespeople. Use assignment statements to fill the table with data, then print it, one value per line.

3. Instead of using assignment statements, use the `cin` function to fill the salespeople data from Exercise 2.

4. Write a program that tracks the grades for five classes, each having 10 students. Input the data using the `cin` function. Print the table in its native row-and-column format.

5. Rewrite the TAPE.CPP program presented in this lesson so that it prints a title at the top of the table that says "Tape Inventory," and add a third row to the table for a new Medium grade of tape.

IPR2022-00518 Page 00035

Take an active role in learning! *C++ Programming 101*'s interactive, workbook-style approach teaches you how to program in C++ from the ground up.

Each lesson begins with a clear objective, so you know what you should glean from that assignment. Step by easy step you are guided through the process of writing clear, workable code. The workbook section in each lesson helps you through the thought processes needed, and debugging techniques help you accomplish the task. Before long, you graduate to working with strings, loops, and operators like an experienced C++ programmer. The book ends with an introduction to object-oriented programming.

Greg Perry is fast becoming one of the most sought-after speakers and writers in the programming field. He is known for being able to take programming topics and bring them down to the beginner's level. He currently is a professor of computer science at Tulsa Junior College.

Disk Features:

Complete source code for all examples in the book

Bonus exercises and examples

Answers to questions and exercises

Rely on These Easy to Follow Sections!

- **Try Th**
  Examples
- **Finish**
  Fill in the
  your prog
- **Find the Bug**—
  Learn how to debug your programs
- **Still Confused?**—
  Work through the tough topics and understand C++

PRICE $29.95
C++ PROGRAMMING 101 W/DISK
215
PERRY G 3573 Prog C++ J080
736072 QP 2 103100 MMGZ 9
709-078 571841

GUARANTEED TIMELY & ACCURATE
Approved for Technical Accuracy by Tim Moore

INFORMATION CENTER
Beginning/Intermediate
How To/Reference
IBM Compatible
Programming
C++

SAMS PUBLISHING

$29.95 US / $37.95 CAN

ISBN 0-672-30200-4
9 780672 302008 90000