

Chapter 10

Arrays and Strings

- 10.1 Arrays
- 10.2 One-Dimensional Arrays
 - 10.2.1 Accessing Array Elements
 - 10.2.2 Representation of Arrays in Memory
 - 10.2.3 Example: Finding the Maximum
 - 10.2.4 No Array-to-Array Assignments
 - 10.2.5 No Bounds Checking
- 10.3 Strings
 - 10.3.1 Reading a String from the Keyboard
 - 10.3.2 Some C++ Library Functions for Strings
 - 10.3.3 Using the Null Terminator

- 10.4 Two-Dimensional Arrays
- 10.5 Multidimensional Arrays
- 10.6 Array Initialization
 - 10.6.1 Character Array Initialization
 - 10.6.2 Multi-Dimensional Array Initialization
 - 10.6.3 Unsized Array Initializations
- 10.7 Arrays of Strings
- 10.8 An Example Using String Arrays
 - 10.8.1 Entering Information
 - 10.8.2 Displaying Database Contents
 - 10.8.3 Menu For User's Selection
 - 10.8.4 Main Function
 - 10.8.5 Putting It All Together
- 10.9 References

10.1 Arrays

An **array** is a *collection* of variables of the *same type* that are referred to by a *common name*.

Arrays offer a convenient means of grouping together several related variables, in one dimension or more dimensions:

• product part numbers:

```
int part numbers[] = {123, 326, 178, 1209};
```

• student scores:

```
int scores[10] = \{1, 3, 4, 5, 1, 3, 2, 3, 4, 4\};
```

• characters:



• names:

```
char names[][40] =
   {"Peter", "Mary", "Lisa", "John", "George-Simon"};
```

• 3D coordinates:

```
vector coordinates[4][3] =
  {{0, 0, 0},
   {1, 0, 1},
   {1, 0, 5}
  {4, 7, 9}};
```

First Back TOC Prev Next Last



10.2 One-Dimensional Arrays

A one-dimensional array is a list of related variables. The general form of a one-dimensional array declaration is:

type variable_name[size]

• *type*: base type of the array,

determines the data type of each element in

the array

• size: how many elements the array will hold

• *variable_name*: the name of the array

Examples:

```
int sample[10];
float float_numbers[100];
char last name[40];
```



10.2.1 Accessing Array Elements

An individual element within an array is accessed by use of an index.

An index describes the position of an element within an array.

Note: In C++ the first element has the index zero!

Example: Load the array **sample** with the numbers 0^2 through 9^2

```
#include <iostream.h>
int main()
  int sample[10]; // reserves for 10 integers
  int t;
  // initialize the array
  for(t=0; t<10; ++t) sample[t] = t*t;
  // display the array
  for(t=0; t<10; ++t)
    cout << sample[t] << ' ';</pre>
  return(0);}
```

10.2.2 Representation of Arrays in Memory

In C++, any array is mapped to a contiguous memory location.

All memory elements reside next to each other.

The lowest address corresponds to the first element, and the highest address to the last element.

Example:

Then the memory representation of array **a** looks like this:

10.2.3 Example: Finding the Maximum

```
#include <iostream.h>
int main()
  int i, max = 0;
  int list[100];
  // initialize the array with random values
  for(i=0; i<100; i++) list[i] = rand();
  // find maximum value
  for(i=0; i<100; i++)
    if(max < list[i]) max = list[i];</pre>
  cout << "Maximum value: " << max;</pre>
  return(0);
```



10.2.4 No Array-to-Array Assignments

You cannot assign one array to another in C++.

The following is illegal:

```
int a[10], b[10];

// do something
// assign all elements of array b to array a
a = b; // error -- illegal
```

Instead, you have to do the assignments for each element:

```
int i;

// assign all elements of array b to array a
for(i=0; i<10; i++) a[i] = b[i];</pre>
```



10.2.5 No Bounds Checking

C++ performs no bounds checking on arrays.

Nothing will stop you from overrunning the end of an array:

- → You will assign values to some other variables' data!!!
- → You might even write into a piece of the program code!!!

For example, you can compile and run the following program, even though the array crash is being overrun:

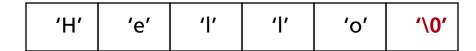
```
// An incorrect program. Do not execute!
int main()
{
  int crash[10], i;
  for(i=0; i<100; i++) crash[i] = i;
  return(1);
}</pre>
```



10.3 Strings

The most common use for one-dimensional arrays is to store **strings** of **characters**.

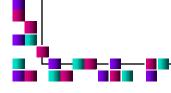
In C++, a **string** is defined as a character array terminated by a null symbol ('\0').



To declare an array **str** that could hold a **10**-character string, one would write:

```
char str[11];
```

Specifying the size as 11 makes room for the null at the end of the string.



Some examples of **string constants** in C++ are:

The **null string**, "", only contains the null terminator and represents the empty string.



10.3.1 Reading a String from the Keyboard

How to read a string entered from the keyboard?

Make an array, that will receive the string, the target of a cin stream.

The following program reads (part of) a string entered by the user:

```
#include <stdio.h>
int main()
{
  char str[80];

  cout << "Enter a string: ";
  cin >> str; // read string from keyboard
  cout << "Here is your string: ";
  cout << str;

  return(0);}</pre>
```

Problem: Entering the string "This is a test", the above program only returns "This", not the entire sentence.

Reason: The C++ input/output system stops reading a string when the first **whitespace** character is encountered.

Solution: Use another C++ library function, gets().

```
#include <iostream.h>
#include <cstdio.h>
int main()
{
   char str[80]; // long enough for user input?

   cout << "Enter a string: ";
   gets(str); // read a string from the keyboard
   cout << "Here is your string: ";
   cout << str << endl;
   return(0);}</pre>
```



10.3.2 Some C++ Library Functions for Strings

C++ supports a range of string-manipulation functions.

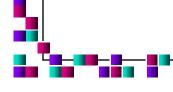
The most common are:

• strcpy(): copy characters from one string to another

• strcat(): concatenation of strings

• strlen(): length of a string

• strcmp(): comparison of strings



```
strcpy(to_string, from_string) — String Copy:
```

```
#include <iostream.h>
  #include <cstring.h>
  int main()
    char a[10];
    strcpy(a, "hello");
    cout << a;
    return(0);
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
          1
                        \0
                                           ?
 h
                             ?
      е
                    0
```

strlen(string) — String Length

strlen(str) returns the length of the string pointed to by str, i.e., the number of characters excluding the null terminator.

```
#include <iostream.h>
#include <cstdio.h>
#include <cstring.h>
int main()
  char str[80];
  cout << "Enter a string: ";</pre>
  gets(str);
  cout << "Length is: " << strlen(str);</pre>
  return(0);}
```

strcat(string_1, string_2) — Concatenation of Strings

The strcat() function appends s2 to the end of s1. String s2 is unchanged.

```
// includes ...
int main()
  char s1[21], s2[11];
  strcpy(s1, "hello");
  strcpy(s2, "there");
  strcat(s1, s2);
  cout << s1 << endl;
  cout << s2 << endl;
  return(0);
```

Output: hello there there



Note:

• The first string array has to be *large enough* to hold both strings:

To be on the safe side:
 strlen(s1concats2) >= strlen(s1) + strlen(s2)

strcmp(string_1, string_2) — Comparison of Strings

The strcmp(str_1, str_2) function compares two strings and returns the following result:

The strings are compared *lexicographically* (i.e., according to dictionary order):

Lexicographical order:

Given: Two words $a, b \in \Sigma^+$ over an alphabet Σ , and an ordering relation "<" on the elements of the alphabet.

$$a < b : \Leftrightarrow \exists (w \in \Sigma^*) \exists (x, y \in \Sigma) (\exists (v_1, v_2 \in \Sigma^*))$$
$$((a = wxv_1) \land (b = wxv_2) \land (x < y))$$

```
// Comparing strings
#include <iostream.h>
#include <cstring.h>
#include <cstdio.h>
int main()
  char str[80];
  cout << "Enter password: ";</pre>
  gets(str);
  if(strcmp(str, "password")) {
    // strings differ
    cout << "Invalid password.\n";}</pre>
  else cout << "Logged on.\n";
  return(0);
```

10.3.3 Using the Null Terminator

Operations on strings can be simplified using the fact that all strings are null-terminated.

```
// Convert a string to uppercase
// ... includes ...
int main()
  char str[80];
  int i;
  strcpy(str, "this is a test");
  for(i=0; str[i]; i++)
    str[i] = toupper(str[i]);
  cout << str; return(0); }</pre>
```



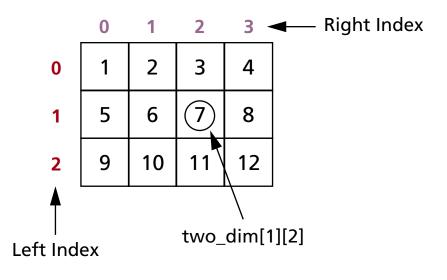
10.4 Two-Dimensional Arrays

A two-dimensional array is a list of one-dimensional arrays.

To declare a two-dimensional integer array **two_dim** of size 10,20 we would write:

```
int matrix[3][4];
```

This corresponds to a table with 3 rows and 4 columns (for example).



We can generate the array above by using this program:

```
#include <iostream.h>
int main()
  int row=3, col=4;
  int matrix[row][col];
  for(row=0; row < 3; ++row) {
    for(col=0; col < 4; ++col) {
      matrix[row][col] = (row*4) + col +1;
      cout << matrix[row][col] << ' ';</pre>
    cout << '\n';
  return(0);}
```



Memory Allocation for Two-Dimensional Arrays

Storage for all array elements is determined at compile time.

The memory used to hold an array is required the entire time that the array is in existence.

The following formula determines the number of bytes of memory that will be allocated:

bytes = rows * columns * number_of_bytes_in_type

For example, an integer array (with two-byte integers) with dimensions 100,100 would require 100 * 100 * 2 = 20,000 bytes.



10.5 Multidimensional Arrays

C++ allows arrays with more than two dimensions.

The general form of an N-dimensional array declaration is:

```
type array_name [size_1] [size_2] ... [size_N];
```

For example, the following declaration creates a $4 \times 10 \times 20$ character array, or a matrix of strings:

```
char string_matrix[4][10][20];
```

This requires 4 * 10 * 20 = 800 bytes.

If we scale the matrix by 10, i.e. to a 40 x 100 x 20 array, then 80,000 bytes are needed.





10.6 Array Initialization

The general form of array initialization is similar to that of other variables:

type array-name[size] = { list-of-values };

The *list-of-values* has to be a comma-separated list of **constants** that are type-compatible with the base type of the array.

In the following example, a 10-element float array is initialized with the numbers 1.0 through 10.0:

float
$$i[10] = \{1., 2., 3., 4., 5., 6, 7, 8, 9, 10\};$$

Therefore, i[0] will have the value 1.0, and i[9] will have the value 10.0.



10.6.1 Character Array Initialization

Character arrays that will hold strings allow a shorthand initialization that takes this form:

char array-name[size] = "string";

For example, the following code fragment initializes **str** to the phrase "hello":

```
char str[6] = "hello";
```

This is the same as writing

```
char str[6] =
    {'h', 'e', 'l', 'l', 'o', '\0'};
```

Remember that one has to make sure to make the array long enough to include the null terminator.



10.6.2 Multi-Dimensional Array Initialization

Multi-dimensional arrays are initialized the same way as onedimensional arrays.

For example, the following code fragment initializes an array squares with the numbers 1 through 10 and their squares:

```
int squares[9][2] =
{ 1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25,
    6, 36,
    7, 49,
    8, 64,
    9, 81 };
```

For better readability, especially for multi-dimensional arrays, one can use *subaggregate grouping* by adding braces accordingly.

The same declaration as above can also be written as:

```
int squares[10][2] =
  {1, 1},
  {2, 4},
  {3, 9},
  {4, 16},
  {5, 25},
  {6, 36},
  {7, 49},
  {8, 64},
  {9, 81},
  {10, 100}
```

The following program uses the **squares** array to find the root of a number entered by the user.

```
#include <iostream.h>
// declaration of squares array goes here
int main()
  int i, j;
  cout << "Enter a number 1<=i<=100: ";</pre>
  cin >> i;
  // look up i
  for(j=0; j<10; j++)
    if(squares[j][1] == i) {
      cout << "Root: " << squares[j][0];</pre>
      return(0);}
  cout << "No integer root."; return(0);}</pre>
```



10.6.3 Unsized Array Initializations

It is possible to let C++ automatically dimension the arrays through the use of *unsized arrays*.

```
char error_1[] = "Divide by 0\n";
char error_2[] = "End-of-File\n";
char error_3[] = "Access Denied\n";
```

C++ will automatically create arrays large enough to hold all the initializers present.

For a multi-dimensional array, all but the leftmost dimension have to be specified. So we can write:

```
char errors[][20] = {
   "Divide by 0\n",
   "End-of-File\n",
   "Access Denied\n" };
```



10.7 Arrays of Strings

An array of strings is a special form of a two-dimensional array.

- The size of the **left** index determines the **number** of strings.
- The size of the **right** index specifies the **maximum length** of each string.

For example, the following declares an array of 30 strings, each having a maximum length of 80 characters (with one extra character for the null terminator):

```
char string_array[30][81];
```



For accessing an <u>individual</u> string, one simply specifies only the left index:

```
firstString = string_array[0];
sixthString = string_array[5];
```

The following example calls the **gets()** function with the third string in the array:

```
gets(string_array[2]);
```

This program accepts lines of text entered at the keyboard and redisplays them after a blank line is entered.

```
// includes go here
int main()
  int t, i;
  char text[100][80];
  for(t=0; t<100; t++) {
    cout << t << ": ";
    gets(text[t]);
    if(!text[t][0]) break; // quit on blank line
  for(i=0; i<t; i++) // redisplay the strings</pre>
    cout << text[i] << '\n';
  return(0);}
```



10.8 An Example Using String Arrays

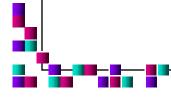
Arrays of strings are commonly used for handling tables of information.

One such application would be an employee database that stores

- the name
- telephone number
- hours worked per pay period, and
- hourly wage.

These data we could store in arrays:

```
char name[20][80]; // employee names
int phone[20]; // phone numbers
float hours[20]; // hours worked
float wage[20]; // wage
```



10.8.1 Entering Information

```
void enter()
  int i;
  for(i=0; i<20; i++) {
    cout << "Last name: ";</pre>
    cin >> name[i];
    cout << "Phone number: ";</pre>
    cin >> phone[i];
    cout << "Hours worked: ";
    cin >> hours[i];
    cout << "Wage: ";</pre>
    cin >> wage[i];
```

10.8.2 Displaying Database Contents

10.8.3 Menu For User's Selection

```
int menu()
  int choice;
  cout << "0. Quit\n";</pre>
  cout << "1. Enter information\n";</pre>
  cout << "2. Report information\n";</pre>
  cout << "\n";
  cout << "Choose one: ";
  cin >> choice;
  return choice;
```

10.8.4 Main Function

```
int main()
  int choice;
  do {
    choice = menu(); // get selection
    switch(choice) {
      case 0: break;
      case 1: enter(); break;
      case 2: report(); break;
      default: cout << "Try again.\n\n";</pre>
  } while( choice != 0);
  return(0);
```

10.8.5 Putting It All Together

```
#include <iostream.h>
// array declarations
int menu();
void enter();
void report();
int main() { ... }
int menu() { ... }
void enter() { ... }
void report() { ... }
```



10.9 References

- Schildt, H., C++ from the Ground Up, Berkeley, McGraw-Hill, 1998, Chapter 5.
- G. Blank and R. Barnes, *The Universal Machine*, Boston, MA: WCB/McGraw-Hill, 1998. Chapter 11.

