# Pointers and Arrays

Arrays and pointers are closely related in the C++ programming language. You can address arrays as if they were pointers and address pointers as if they were arrays. Being able to store and access pointers and arrays gives you the ability to store strings of data in array elements. Without pointers, you could not store strings of data in arrays because there is no fundamental string data type in C++ (no string variables, only string literals).

This chapter introduces the following concepts:

- Array names and pointers
- Character pointers
- Pointer arithmetic
- Ragged-edge arrays of string data

This chapter introduces concepts you will use for much of your future programming in C++. Pointer manipulation is important to the C++ programming language.

## Array Names as Pointers

An array name is just a pointer, nothing more. To prove this, suppose you have the following array declaration:

```
int ara[5] = {10, 20, 30, 40, 50};
```

If you printed `ara[0]`, you would see `10`. Because you now fully understand arrays, this is the value you would expect.

An array name is a pointer.

But what if you were to print `*ara`? Would `*ara` print anything? If so, what? If you thought an error message would print because `ara` is not a pointer but an array, you would be wrong. An array name is a pointer. If you print `*ara`, you also would see `10`.

Recall how arrays are stored in memory. Figure 27.1 shows how `ara` would be mapped in memory. The array name, `ara`, is nothing more than a pointer pointing to the first element of the array. Therefore, if you dereference that pointer, you dereference the value stored in the first element of the array, which is `10`. Dereferencing `ara` is exactly the same thing as referencing to `ara[0]`, because they both produce the same value.

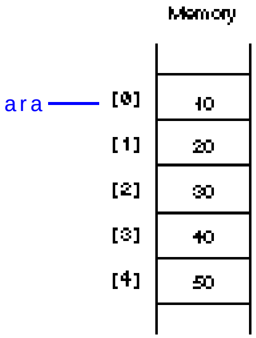


Figure 27.1. Storing the array called `ara` in memory.

You now see that you can reference an array with subscripts or with pointer dereferencing. Can you use pointer notation to print the third element of `ara`? Yes, and you already have the tools to do so. The following `cout` prints `ara[2]` (the third element of `ara`) without using a subscript:

```
cout << *(ara+2) ;              // Prints ara[2].
```

The expression `*(ara+2)` is not vague at all, if you remember that an array name is just a pointer that always points to the array's first element. `*(ara+2)` takes the address stored in `ara`, adds two to the address, and dereferences that location. The following holds true:

`ara+0` points to `ara[0]`

`ara+1` points to `ara[1]`

`ara+2` points to `ara[2]`

`ara+3` points to `ara[3]`

`ara+4` points to `ara[4]`

Therefore, to print, store, or calculate with an array element, you can use either the subscript notation or the pointer notation. Because an array name contains the address of the array's first element, you must dereference the pointer to get the element's value.

**Internal Locations**

C++ knows the internal data size requirements of characters, integers, floating-points, and the other data types on your computer. Therefore, because `ara` is an integer array, and because each element in an integer array consumes two to four bytes of storage, depending on the computer, C++ adds two or four bytes to the address if you reference arrays as just shown.

If you write `*(ara+3)` to refer to `ara[3]`, C++ would add six or twelve bytes to the address of `ara` to get the third element. C++ does not add an actual three. You do not have to worry about this, because C++ handles these internals. When you write `*(ara+3)`, you are actually requesting that C++ add three integer addresses to the address of `ara`. If `ara` were a floating-point array, C++ would add three floating-point addresses to `ara`.

## Pointer Advantages

An array name is a pointer constant.

Although arrays are actually pointers in disguise, they are special types of pointers. An array name is a *pointer constant*, not a pointer variable. You cannot change the value of an array name, because you cannot change constants. This explains why you cannot assign an array new values during a program's execution. For instance, even if `cname` is a character array, the following is not valid in C++:

```
cname = “Christine Chambers”;  // Invalid array assignment.
```

The array name, `cname`, cannot be changed because it is a constant. You would not attempt the following

```
5 = 4 + 8 * 21;                // Invalid assignment
```

because you cannot change the constant `5` to any other value. C++ knows that you cannot assign anything to `5`, and C++ prints an error message if you attempt to change `5`. C++ also knows an array name is a constant and you cannot change an array to another value. (You can assign values to an array only at declaration time, one element at a time during execution, or by using functions such as `strcpy()`.)

This brings you to the most important reason to learn pointers: pointers (except arrays referenced as pointers) are variables. You can change a pointer variable, and being able to do so makes processing virtually any data, including arrays, much more powerful and flexible.

### Examples

1. By changing pointers, you make them point to different values in memory. The following program demonstrates how to change pointers. The program first defines two floating-point values. A floating-point pointer points to the first variable, `v1`, and is used in the `cout`. The pointer is then changed so it points to the second floating-point variable, `v2`.

```
// Filename: C27PTRCH.CPP
// Changes the value of a pointer variable.
#include <iostream.h>
```

```
#include <iomanip.h>
void main()
{
   float v1=676.54;                          // Defines two
   float v2=900.18;                   // floating-point variables.
   float * p_v;             / Defines a floating-point pointer.

   p_v = &v1;                   // Makes pointer point to v1.
   cout << “The first value is “ << setprecision(2) <<
           *p_v << “\n”;                     // Prints 676.54.

   p_v = &v2;              // Changes the pointer so it
                           // points to v2.
   cout << “The second value is “ << setprecision(2) <<
           *p_v << “\n”;                     // Prints 900.18.
   return;
}
```

Because they can change pointers, most C++ programmers use pointers rather than arrays. Because arrays are easy to declare, C++ programmers sometimes declare arrays and then use pointers to reference those arrays. If the array data changes, the pointer helps to change it.

2. You can use pointer notation and reference pointers as arrays with array notation. The following program declares an integer array and an integer pointer that points to the start of the array. The array and pointer values are printed using subscript notation. Afterwards, the program uses array notation to print the array and pointer values.

   Study this program carefully. You see the inner workings of arrays and pointer notation.

```
// Filename: C27ARPTR.CPP
// References arrays like pointers and
// pointers like arrays.
#include <iostream.h>
void main()
{
   int ctr;
   int iara[5] = {10, 20, 30, 40, 50};
   int *iptr;

   iptr = iara;        // Make iptr point to array’s first
                       // element. This would work also:
                       // iptr = &iara[0];

   cout << “Using array subscripts:\n”;
   cout << “iara\tiptr\n”;
   for (ctr=0; ctr<5; ctr++)
      { cout << iara[ctr] << “\t” << iptr[ctr] << “\n”;  }

   cout << “\nUsing pointer notation:\n”;
   cout << “iara\tiptr\n”;
   for (ctr=0; ctr<5; ctr++)
      { cout << *(iara+ctr) << “\t” << *(iptr+ctr) << “\n”;  }

 return;
}
```

Here is the program’s output:

```
Using array subscripts:
iara     iptr
10       10
20       20
30       30
40       40
50       50

Using pointer notation:
iara     iptr
10       10
20       20
30       30
40       40
50       50
```

## Using Character Pointers

The ability to change pointers is best seen when working with character strings in memory. You can store strings in character arrays, or point to them with character pointers. Consider the following two string definitions:

```
char cara[] = “C++ is fun”;     // An array holding a string

char *cptr = “C++ By Example”;    // A pointer to the string
```

Character pointers can point to the first character of a string.

Figure 27.2 shows how C++ stores these two strings in memory. C++ stores both in basically the same way. You are familiar with the array definition. When assigning a string to a character pointer, C++ finds enough free memory to hold the string and assign the address of the first character to the pointer. The previous two string definition statements do almost exactly the same thing; the only difference between them is that the two pointers can easily be exchanged (the array name and the character pointers).

Because `cout` prints strings starting at the array or pointer name until the null zero is reached, you can print each of these strings with the following `cout` statements:

```
cout << “String 1: “ << cara << “\n”;

cout << “String 2: “ << cptr << “\n”;
```

You print strings in arrays and pointed-to strings the same way. You might wonder what advantage one method of storing strings has over the other. The seemingly minor difference between these stored strings makes a big difference when you change them.

Suppose you want to store the string `Hello` in the two strings. You cannot assign the string to the array like this:

```
cara = “Hello”;                                   // Invalid
```

Because you cannot change the array name, you cannot assign it a new value. The only way to change the contents of the array is by assigning the array characters from the string an element at a time, or by using a built-in function such as `strcpy()`. You can, however, make the character array point to the new string like this:

```
cptr = “Hello”;              // Change the pointer so
                            // it points to the new string.
```

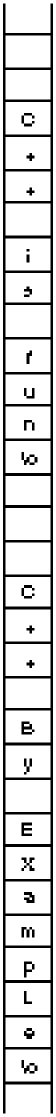


Figure 27.2. Storing two strings: One in an array and one pointed to by a pointer variable.

**TIP:** If you want to store user input in a string pointed to by a pointer, first you must reserve enough storage for that input string. The easiest way to do this is to reserve a character array, then assign a character pointer to the beginning element of that array like this:

```
char input[81];          // Holds a string as long as
                         // 80 characters.
char *iptr=input;        // Also could have done this:
                         // char *iptr=&input[0];
```

Now you can input a string by using the pointer:

```
gets(iptr);            // Make sure iptr points to
                       // the string typed by the user.
```

You can use pointer manipulation, arithmetic, and modification on the input string.

## Examples

1. Suppose you want to store your sister’s full name and print it. Rather than using arrays, you can use a character pointer. The following program does just that.

```
// Filename: C27CP1.CPP
// Stores a name in a character pointer.
#include <iostream.h>
void main()
{
   char *c=”Bettye Lou Horn”;

   cout << “My sister’s name is “ << c << “\n”;
   return;
}
```

This prints the following:

```
My sister’s name is Bettye Lou Horn
```

2. Suppose you must change a string pointed to by a character pointer. If your sister changed her last name to Henderson, your program can show both strings in the following manner:

*Identify the program and include the I/O header file. This program uses a character pointer,* `c`*, to point to a string literal in memory. Point to the string literal, and print the string. Make the character-pointer point to a new string literal, then print the new string.*

```
// Filename: C27CP2.CPP
// Illustrates changing a character string.
#include <iostream.h>
void main()
{
   char *c=”Bettye Lou Horn”;

   cout << “My sister’s maiden name was “ << c << “\n”;

   c = “Bettye Lou Henderson”;  // Assigns new string to c.

   cout << “My sister’s married name is “ << c << “\n”;
   return;
}
```

The output is as follows:

```
My sister’s maiden name was Bettye Lou Horn
My sister’s married name is Bettye Lou Henderson
```

3. Do not use character pointers to change string constants. Doing so can confuse the compiler, and you probably will not get the results you expect. The following program is similar to those you just saw. Rather than making the character pointer point to a new string, this example attempts to change the contents of the original string.

```
// Filename: C27CP3.CPP
// Illustrates changing a character string improperly.
#include <iostream.h>
void main()
```

```
{
   char *c=”Bettye Lou Horn”;

   cout << “My sister’s maiden name was “ << c << “\n”;

   c += 11;              // Makes c point to the last name
                         //   (the twelfth character).
   c = “Henderson”;           // Assigns a new string to c.

   cout << “My sister’s married name is “ << c << “\n”;
   return;
}
```

The program seems to change the last name from Horn to Henderson, but it does not. Here is the output of this program:

```
My sister’s maiden name was Bettye Lou Horn
My sister’s married name is Henderson
```

Why didn’t the full string print? Because the address pointed to by `c` was incremented by 11, `c` still points to `Henderson`, so that was all that printed.

4. You might guess at a way to fix the previous program. Rather than printing the string stored at `c` after assigning it to `Henderson`, you might want to decrement it by 11 so it points to its original location, the start of the name. The code to do this follows, but it does not work as expected. Study the program before reading the explanation.

```
// Filename: C27CP4.C
// Illustrates changing a character string improperly.
#include <iostream.h>
void main()
{
   char *c=”Bettye Lou Horn”;

   cout << “My sister’s maiden name was “ << c << “\n”;

   c += 11;                   // Makes c point to the last
                              // name (the twelfth character).
   c = “Henderson”;           // Assigns a new string to c.
   c -= 11;                        // Makes c point to its
                                 // original location (???).

   cout << “My sister’s married name is “ << c << “\n”;
   return;
}
```

This program produces garbage at the second `cout`. There are actually two string literals in this program. When you first assign `c` to `Bettye Lou Horn`, C++ reserves space in memory for the constant string and puts the starting address of the string in `c`.

When the program then assigns `c` to `Henderson`, C++ finds room for *another* character constant, as shown in Figure 27.3. If you subtract 11 from the location of `c`, after it points to the new string `Henderson`, `c` points to an area of memory your program is not using. There is no guarantee that printable data appears before the string constant `Henderson`. If you want to manipulate parts of the string, you must do so an element at a time, just as you must with arrays.

## Pointer Arithmetic

You saw an example of pointer arithmetic when you accessed array elements with pointer notation. By now you should be comfortable with the fact that both of these array or pointer references are identical:

```
ara[sub] and *(ara + sub)
```

You can increment or decrement a pointer. If you increment a pointer, the address inside the pointer variable increments. The pointer does not always increment by one, however.

Suppose `f_ptr` is a floating-point pointer indexing the first element of an array of floating-point numbers. You could initialize `f_ptr` as follows:

```
float fara[] = {100.5, 201.45, 321.54, 389.76, 691.34};
f_ptr = fara;
```

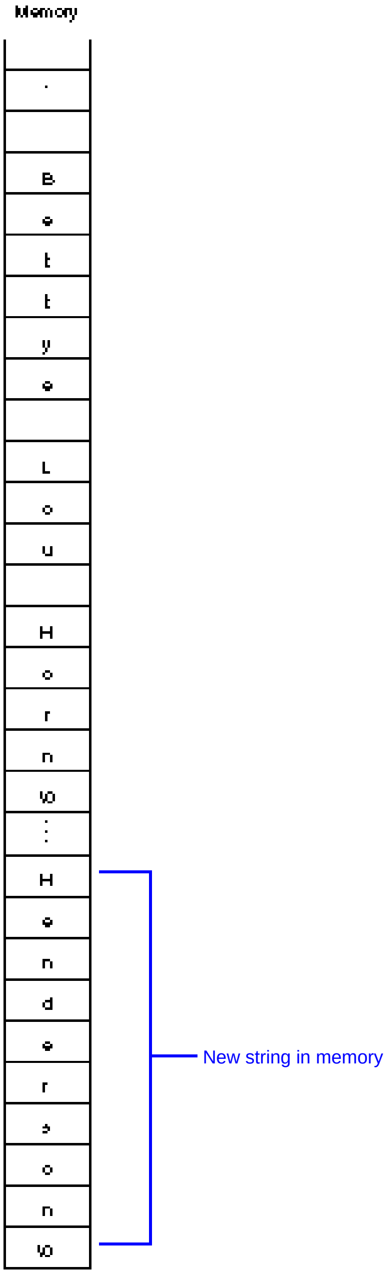


Figure 27.3. Two string constants appear in memory because two string constants are used in the program.

Figure 27.4 shows what these variables look like in memory. Each floating-point value in this example takes four bytes of memory.

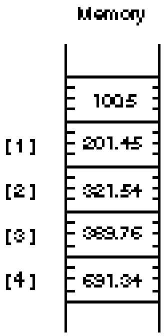


Figure 27.4. A floating-point array and a pointer.

Incrementing a pointer can add more than one byte to the pointer.

If you print the value of `*f_ptr`, you see 100.5. Suppose you increment `f_ptr` by one with the following statement:

```
f_ptr++;
```

C++ does not add one to the address in `f_ptr`, even though it seems as though one should be added. In this case, because floating-point values take four bytes each on this machine, C++ adds four to `f_ptr`. How does C++ know how many bytes to add to `f_ptr`? C++ knows from the pointer's declaration how many bytes of memory pointers take. This is why you have to declare the pointer with the correct data type.

After incrementing `f_ptr`, if you were to print `*f_ptr`, you would see `201.45`, the second element in the array. If C++ added only one to the address in `f_ptr`, `f_ptr` would point only to the second byte, `100.5`. This would output garbage to the screen.

**NOTE:** When you increment a pointer, C++ adds one full data-type size (in bytes) to the pointer, not one byte. When you decrement a pointer, C++ subtracts one full data type size (in bytes) from the pointer.

## Examples

1. The following program defines an array with five values. An integer pointer is then initialized to point to the first element in the array. The rest of the program prints the dereferenced value of the pointer, then increments the pointer so it points to the next integer in the array.

   Just to show you what is going on, the size of integer values is printed at the bottom of the program. Because (in this case) integers take two bytes, C++ increments the pointer by two so it points to the next integer. (The integers are two bytes apart from each other.)

```
// Filename: C27PTI.CPP
// Increments a pointer through an integer array.
#include <iostream.h>
void main()
{
   int iara[] = {10,20,30,40,50};
   int *ip = iara;              // The pointer points to
                                // The start of the array.
   cout << *ip << “\n”;
   ip++;                           // Two are actually added.
   cout << *ip << “\n”;
   ip++;                           // Two are actually added.
   cout << *ip << “\n”;
   ip++;                           // Two are actually added.
   cout << *ip << “\n”;
   ip++;                           // Two are actually added.
   cout << *ip << “\n\n”;
   cout << “The integer size is “ << sizeof(int);
   cout << “ bytes on this machine \n\n”;
   return;
}
```

Here is the output from the program:

```
10
20
30
40
50

The integer size is two bytes on this machine
```

2. Here is the same program using a character array and a character pointer. Because a character takes only one byte of storage, incrementing a character pointer actually adds just one to the pointer; only one is needed because the characters are only one byte apart.

```
// Filename: C27PTC.CPP
// Increments a pointer through a character array.
#include <iostream.h>
void main()
{
   char cara[] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’};
   char *cp = cara;             // The pointers point to
                                // the start of the array.
   cout << *cp << “\n”;
   cp++;                           // One is actually added.
   cout << *cp << “\n”;
   cp++;                           // One is actually added.
   cout << *cp << “\n”;
   cp++;                           // One is actually added.
   cout << *cp << “\n”;
   cp++;                           // One is actually added.
   cout << *cp << “\n\n”;
   cout << “The character size is “ << sizeof(char);
   cout << “ byte on this machine\n”;
   return;
}
```

3. The next program shows the many ways you can add to, subtract from, and reference arrays and pointers. The program defines a floating-point array and a floating-point pointer. The body of the program prints the values from the array using array and pointer notation.

```
// Filename: C27ARPT2.CPP
// Comprehensive reference of arrays and pointers.
#include <iostream.h>
void main()
{
   float ara[] = {100.0, 200.0, 300.0, 400.0, 500.0};
   float *fptr;                   // Floating-point pointer.

   // Make pointer point to array’s first value.
   fptr = &ara[0];            // Also could have been this:
                              // fptr = ara;

   cout << *fptr << “\n”;                  // Prints 100.0
   fptr++;         // Points to next floating-point value.
   cout << *fptr << “\n”;                  // Prints 200.0
   fptr++;         // Points to next floating-point value.
   cout << *fptr << “\n”;                  // Prints 300.0
   fptr++;         // Points to next floating-point value.
   cout << *fptr << “\n”;                  // Prints 400.0
   fptr++;         // Points to next floating-point value.
   cout << *fptr << “\n”;                  // Prints 500.0

   fptr = ara;            // Points to first element again.
   cout << *(fptr+2) << “\n”;         // Prints 300.00 but
                                    // does not change fptr.

   // References both array and pointer using subscripts.
   cout << (fptr+0)[0] << “ “ << (ara+0)[0] << “\n”;
   // 100.0  100.0
   cout << (fptr+1)[0] << “ “ << (ara+1)[0] << “\n”;
   // 200.0  200.0
   cout << (fptr+4)[0] << “ “ << (ara+4)[0] << “\n”;
   // 500.0  500.0
   return;
}
```

The following is the output from this program:

```
100.0
200.0
300.0
400.0
```

```
500.0
300.0
100.0  100.0
200.0  200.0
500.0  500.0
```

## Arrays of Strings

You now are ready for one of the most useful applications of character pointers: storing arrays of strings. Actually, you cannot store an array of strings, but you can store an array of character pointers, and each character pointer can point to a string in memory.

An array that a character pointer defines is a ragged-edge array.

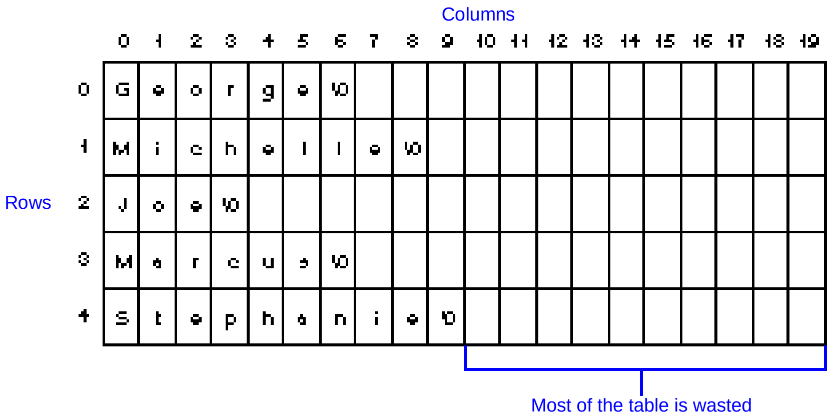
By defining an array of character pointers, you define a *ragged-edge array*. A ragged-edge array is similar to a two-dimensional table, except each row contains a different number of characters (instead of being the same length).

The word *ragged-edge* derives from the use of word processors. A word processor typically can print text fully justified or with a ragged-right margin. The columns of this paragraph are fully justified, because both the left and the right columns align evenly. Letters you write by hand and type on typewriters (remember what a typewriter is?) generally have ragged-right margins. It is difficult to type so each line ends in exactly the same right column.

All two-dimensional tables you have seen so far have been fully justified. For example, if you declared a character table with five rows and 20 columns, each row would contain the same number of characters. You could define the table with the following statement:

```
char names[5][20]={ {“George”},
                    {“Michelle”},
                    {“Joe”},
                    {“Marcus”},
                    {“Stephanie”} };
```

This table is shown in Figure 27.5. Notice that much of the table is wasted space. Each row takes 20 characters, even though the data in each row takes far fewer characters. The unfilled elements contain null zeros because C++ nullifies all elements you do not initialize in arrays. This type of table uses too much memory.



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | G | e | o | r | g | e | \0 | | | | | | | | | | | | | |
| 1 | M | i | c | h | e | l | l | e | \0 | | | | | | | | | | | |
| 2 | J | o | e | \0 | | | | | | | | | | | | | | | | |
| 3 | M | a | r | c | u | s | \0 | | | | | | | | | | | | | |
| 4 | S | t | e | p | h | a | n | i | e | \0 | | | | | | | | | | |

Figure 27.5. A fully justified table.

To fix the memory-wasting problem of fully justified tables, you should declare a single-dimensional array of character pointers. Each pointer points to a string in memory, and the strings do not have to be the same length.

Here is the definition for such an array:

```
char *names[5]={ {“George”},
                 {“Michelle”},
                 {“Joe”},
                 {“Marcus”},
                 {“Stephanie”} };
```

This array is single-dimensional. The definition should not confuse you, although it is something you have not seen. The asterisk before `names` makes this an array of pointers. The data type of the pointers is character. The strings are not being assigned to the array elements, but they are being pointed to by the array elements. Figure 27.6 shows this array of pointers. The strings are stored elsewhere in memory. Their actual locations are not critical because each pointer points to the starting character. The strings waste no data. Each string takes only as much memory as needed by the string and its terminating zero. This gives the data its ragged-right appearance.

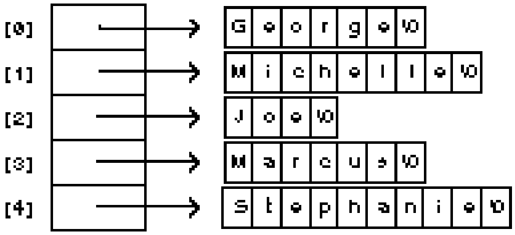


Figure 27.6. The array that points to each of the five strings.

To print the first string, you would use this `cout`:

```
cout << *names;              // Prints George
```

To print the second string, you would use this `cout`:

```
cout << *(names+1);          // Prints Michelle
```

Whenever you dereference any pointer element with the `*` dereferencing operator, you access one of the strings in the array. You can use a dereferenced element any place you use a string constant or character array (with `strcpy()`, `strcmp()`, and so on).

**TIP:** Working with pointers to strings is much more efficient than working directly with the strings. For instance, sorting a list of strings takes much time if they are stored as a fully justified table. Sorting strings pointed to by a pointer array is much faster. You swap only pointers during the sort, not entire strings.

## Examples

1. Here is a full program that uses the pointer array with five names. The `for` loop controls the `cout` function, printing each name in the string data. Now you can see why learning about pointer notation for arrays pays off!

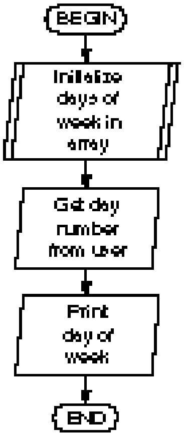
```
// Filename: C27PTST1.CPP
// Prints strings pointed to by an array.
#include <iostream.h>
```

```
void main()
{
   char *name[5]={ {“George”},    // Defines a ragged-edge
                   {“Michelle”},  // array of pointers to
                   {“Joe”},       // strings.
                   {“Marcus”},
                   {“Stephanie”} };
   int ctr;

   for (ctr=0; ctr<5; ctr++)
      { cout << “String #” << (ctr+1) <<
                “ is “ << *(name+ctr) << “\n”; }

   return;
}
```

The following is the output from this program:

```
String #1 is George
String #2 is Michelle
String #3 is Joe
String #4 is Marcus
String #5 is Stephanie
```

2. The following program stores the days of the week in an array. When the user types a number from 1 to 7, the day of the week that matches that number (with Sunday being 1) displays by dereferencing the pointer referencing that string.



```
// Filename: C27PTST2.CPP
// Prints the day of the week based on an input value.
#include <iostream.h>
void main()
{
   char *days[] = {“Sunday”,   // The seven separate sets
              “Monday”,        // of braces are optional.
              “Tuesday”,
              “Wednesday”,
              “Thursday”,
              “Friday”,
              “Saturday”};
   int day_num;
   do
     { cout << “What is a day number (from 1 to 7)? “;
       cin >> day_num;
     } while ((day_num<1) || (day_num>7));     // Ensures
                                    // an accurate number.


   day_num--;                       // Adjusts for subscript.
   cout << “The day is “ << *(days+day_num) << “\n”;
   return;
}
```

## Review Questions

The answers to the review questions are in Appendix B.

1. What is the difference between an array name and a pointer?

2. If you performed the following statement (assume `ipointer` points to integers that take four bytes of memory),

   `ipointer += 2;`

   how many bytes are added to `ipointer`?

3. Which of the following are equivalent, assuming `iary` is an integer array and `iptr` is an integer pointer pointing to the start of the array?

   a. `iary` and `iptr`

   b. `iary[1]` and `iptr+1`

   c. `iary[3]` and `*(iptr + 3)`

   d. `*iary` and `iary[0]`

   e. `iary[4]` and `*iptr+4`

4. Why is it more efficient to sort a ragged-edge character array than a fully justified string array?

5. Given the following array and pointer definition

```
int ara[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *ip1, *ip2;
```

which of the following is allowed?

a. `ip1 = ara;`

b. `ip2 = ip1 = &ara[3];`

c. `ara = 15;`

d. `*(ip2 + 2) = 15;  // Assuming ip2 and ara are equal.`

## Review Exercises

1. Write a program to store your family members' names in a character array of pointers. Print the names.
2. Write a program that asks the user for 15 daily stock market averages and stores those averages in a floating-point array. Using only pointer notation, print the array forward and backward. Again using only pointer notation, print the highest and lowest stock market quotes in the list.
3. Modify the bubble sort shown in Chapter 24, "Array Processing," so that it sorts using pointer notation. Add this bubble sort to the program in Exercise 2 to print the stock market averages in ascending order
4. Write a program that requests 10 song titles from the user. Store the titles in an array of character pointers (a ragged-edge array). Print the original titles, print the alphabetized titles, and print the titles in reverse alphabetical order (from *Z* to *A*).

## Summary

You deserve a break! You now understand the foundation of C++'s pointers and array notation. When you have mastered this section, you are on your way to thinking in C++ as you design your programs. C++ programmers know that C++'s arrays are pointers in disguise, and they program them accordingly.

Being able to use ragged-edge arrays offers two advantages: You can hold arrays of string data without wasting extra space, and you can quickly change the pointers without having to move the string data around in memory.

As you progress into advanced C++ concepts, you will appreciate the time you spend mastering pointer notation. The next chapter introduces a new topic called *structures*. Structures enable you to store data in a more unified manner than simple variables have allowed.