Section 8.9 Relationship Between Pointers and Built-In Arrays

- Pointers that point to built-in arrays can be subscripted exactly as built-in array names can.
- In **pointer/offset notation** (p. 362), if the pointer points to the first element of a built-in array, the offset is the same as an array subscript.
- All subscripted array expressions can be written with a pointer and an offset, using either the built-in array's name as a pointer or using a separate pointer that points to the built-in array.

Section 8.10 Pointer-Based Strings (Optional)

- A **character constant** (p. 364) is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set.
- A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, /and $.
- **String literals** or **string constants** (p. 365) are written in double quotation marks.
- A pointer-based string is a built-in array of `char`s ending with a **null character** (`'\0'`; p. 365), which marks where the string terminates in memory. A string is accessed via a pointer to its first character.
- The result of `sizeof` for a string literal is the length of the string including the terminating null character.
- A string literal may be used as an initializer for a built-in array of `char`s or a variable of type `const char*`.
- You should always declare a pointer to a string literal as `const char*`.
- When declaring a built-in array of `char`s to contain a C string, the built-in array must be large enough to store the C string and its terminating null character.
- If a string is longer than the built-in array of `char`s in which it's to be stored, characters beyond the end of the built-in array will overwrite data in memory following the built-in array, leading to logic errors.
- You can access individual characters in a string directly with array subscript notation.
- A string can be read into a built-in array of `char`s using stream extraction with `cin`. Characters are read until a whitespace character or end-of-file indicator is encountered. The `setw` stream manipulator should be used to ensure that the string read into a built-in array of `char`s does not exceed the size of the built-in array.
- The `cin` object provides the member function **`getline`** (p. 366) to input an entire line of text into a built-in array of `char`s. The function takes three arguments—a built-in array of `char`s in which the line of text will be stored, a length and a delimiter character. The third argument has `'\n'` as a default value.
- A built-in array of `char`s representing a null-terminated string can be output with `cout` and `<<`. The characters of the string are output until a terminating null character is encountered.

Self-Review Exercises

8.1 Answer each of the following:

a) A pointer is a variable that contains as its value the ________ of another variable.

b) A pointer should be initialized to ________ or ________.

c) The only integer that can be assigned directly to a pointer is ________.

8.2 State whether each of the following is *true* or *false*. If the answer is *false*, explain why.

a) The address operator `&` can be applied only to constants and to expressions.

b) A pointer that is declared to be of type `void*` can be dereferenced.
c) A pointer of one type can't be assigned to one of another type without a cast operation.

8.3 For each of the following, write C++ statements that perform the specified task. Assume that double-precision, floating-point numbers are stored in eight bytes and that the starting address of the built-in array is at location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.

a) Declare a built-in array of type `double` called `numbers` with 10 elements, and initialize the elements to the values `0.0, 1.1, 2.2, …, 9.9`. Assume that the constant `size` has been defined as `10`.
b) Declare a pointer `nPtr` that points to a variable of type `double`.
c) Use a `for` statement to display the elements of built-in array `numbers` using array subscript notation. Display each number with one digit to the right of the decimal point.
d) Write two separate statements that each assign the starting address of built-in array `numbers` to the pointer variable `nPtr`.
e) Use a `for` statement to display the elements of built-in array `numbers` using pointer/offset notation with pointer `nPtr`.
f) Use a `for` statement to display the elements of built-in array `numbers` using pointer/offset notation with the built-in array's name as the pointer.
g) Use a `for` statement to display the elements of built-in array `numbers` using pointer/subscript notation with pointer `nPtr`.
h) Refer to the fourth element of built-in array `numbers` using array subscript notation, pointer/offset notation with the built-in array's name as the pointer, pointer subscript notation with `nPtr` and pointer/offset notation with `nPtr`.
i) Assuming that `nPtr` points to the beginning of built-in array `numbers`, what address is referenced by `nPtr + 8`? What value is stored at that location?
j) Assuming that `nPtr` points to `numbers[5]`, what address is referenced by `nPtr` after `nPtr -= 4` is executed? What's the value stored at that location?

8.4 For each of the following, write a statement that performs the specified task. Assume that `double` variables `number1` and `number2` have been declared and that `number1` has been initialized to `7.3`.

a) Declare the variable `doublePtr` to be a pointer to an object of type `double` and initialize the pointer to `nullptr`.
b) Assign the address of variable `number1` to pointer variable `doublePtr`.
c) Display the value of the object pointed to by `doublePtr`.
d) Assign the value of the object pointed to by `doublePtr` to variable `number2`.
e) Display the value of `number2`.
f) Display the address of `number1`.
g) Display the address stored in `doublePtr`. Is the address the same as that of `number1`?

8.5 Perform the task specified by each of the following statements:

a) Write the function header for a function called `exchange` that takes two pointers to double-precision, floating-point numbers `x` and `y` as parameters and does not return a value.
b) Write the function prototype without parameter names for the function in part (a).
c) Write two statements that each initialize the built-in array of `char`s named `vowel` with the string of vowels, `"AEIOU"`.

8.6 Find the error in each of the following program segments. Assume the following declarations and statements:

```
int* zPtr; // zPtr will reference built-in array z
int number;
int z[ ]{[illegible], [illegible], [illegible], [illegible], [illegible]};
```

a) `++zPtr;`

b)
```
// use pointer to get first value of a built-in array
number = zPtr;
```
c)
```
// assign built-in array element 2 (the value 3) to number
number = *zPtr[2];
```
d)
```
// display entire built-in array z
for (size_t i{0}; i <= 5; ++i) {
   cout << zPtr[i] << endl;
}
```
e)
```
++z;
```

Answers to Self-Review Exercises

8.1 a) address. b) `nullptr`, an address. c) `0`.

8.2 a) False. The operand of the address operator must be an *lvalue*; the address operator cannot be applied to literals or to expressions that result in temporary values.

b) False. A pointer to `void` cannot be dereferenced. Such a pointer does not have a type that enables the compiler to determine the type of the data and the number of bytes of memory to which the pointer points.

c) False. Pointers of any type can be assigned to `void` pointers. Pointers of type `void` can be assigned to pointers of other types only with an explicit type cast.

8.3 a)
```
double numbers[size]{0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
```
b)
```
double* nPtr;
```
c)
```
cout << fixed << showpoint << setprecision(1);
for (size_t i{0}; i < size; ++i) {
   cout << numbers[i] << ' ';
}
```
d)
```
nPtr = numbers;
nPtr = &numbers[0];
```
e)
```
cout << fixed << showpoint << setprecision(1);
for (size_t j{0}; j < size; ++j) {
   cout << *(nPtr + j) << ' ';
}
```
f)
```
cout << fixed << showpoint << setprecision(1);
for (size_t k{0}; k < size; ++k) {
   cout << *(numbers + k) << ' ';
}
```
g)
```
cout << fixed << showpoint << setprecision(1);
for (size_t m{0}; m < size; ++m) {
   cout << nPtr[m] << ' ';
}
```
h)
```
numbers[3]
*(numbers + 3)
nPtr[3]
*(nPtr + 3)
```
i) The address is `1002500 + 8 * 8 = 1002564`. The value is `8.8`.

j) The address of `numbers[5]` is `1002500 + 5 * 8 = 1002540`. The address of `nPtr -= 4` is `1002540 - 4 * 8 = 1002508`. The value at that location is `1.1`.

8.4 a)
```
double* doublePtr{nullptr};
```
b)
```
doublePtr = &number1;
```

c) `cout <<` [illegible] `<< *doublePtr << endl;`

d) `number2 = *doublePtr;`

e) `cout <<` [illegible] `<< number2 << endl;`

f) `cout <<` [illegible] `<< &number1 << endl;`

g) `cout <<` [illegible] `<< doublePtr << endl;`
Yes, the value is the same.

8.5 a) `void exchange(double* x, double* y)`

b) `void exchange(double*, double*);`

c) `char vowel[]{`[illegible]`};`
`char vowel[]{`[illegible]`,` [illegible]`,` [illegible]`,` [illegible]`,` [illegible]`,` [illegible]`};`

8.6 a) *Error:* zPtr has not been initialized.
Correction: Initialize zPtr with zPtr = z; (Parts *b–e* depend on this correction.)

b) *Error:* The pointer is not dereferenced.
Correction: Change the statement to number = *zPtr;

c) *Error:* zPtr[2] is not a pointer and should not be dereferenced.
Correction: Change *zPtr[2] to zPtr[2].

d) *Error:* Referring to an out-of-bounds built-in array element with pointer subscripting.
Correction: To prevent this, change the relational operator in the for statement to < or change the 5 to a 4.

e) *Error:* Trying to modify a built-in array's name with pointer arithmetic.
Correction: Use a pointer variable instead of the built-in array's name to accomplish pointer arithmetic, or subscript the built-in array's name to refer to a specific element.

Exercises

8.7 *(True or False)* State whether the following are *true* or *false*. If *false*, explain why.

a) Two pointers that point to different built-in arrays cannot be compared meaningfully.

b) Because the name of a built-in array is implicitly convertible to a pointer to the first element of the built-in array, built-in array names can be manipulated in the same manner as pointers.

8.8 *(Write C++ Statements)* For each of the following, write C++ statements that perform the specified task. Assume that unsigned integers are stored in four bytes and that the starting address of the built-in array is at location 1002500 in memory.

a) Declare an unsigned int built-in array values with five elements initialized to the even integers from 2 to 10. Assume that the constant size has been defined as 5.

b) Declare a pointer vPtr that points to an object of type unsigned int.

c) Use a for statement to display the elements of built-in array values using array subscript notation.

d) Write two separate statements that assign the starting address of built-in array values to pointer variable vPtr.

e) Use a for statement to display the elements of built-in array values using pointer/offset notation.

f) Use a for statement to display the elements of built-in array values using pointer/offset notation with the built-in array's name as the pointer.

g) Use a for statement to display the elements of built-in array values by subscripting the pointer to the built-in array.

h) Refer to the fifth element of values using array subscript notation, pointer/offset notation with the built-in array name's as the pointer, pointer subscript notation and pointer/offset notation.

i) What address is referenced by vPtr + 3? What value is stored at that location?

j) Assuming that `vPtr` points to `values[ 4 ]`, what address is referenced by `vPtr -= 4`? What value is stored at that location?

8.9 *(Write C++ Statements)* For each of the following, write a single statement that performs the specified task. Assume that `long` variables `value1` and `value2` have been declared and `value1` has been initialized to `200000`.

a) Declare the variable `longPtr` to be a pointer to an object of type `long`.

b) Assign the address of variable `value1` to pointer variable `longPtr`.

c) Display the value of the object pointed to by `longPtr`.

d) Assign the value of the object pointed to by `longPtr` to variable `value2`.

e) Display the value of `value2`.

f) Display the address of `value1`.

g) Display the address stored in `longPtr`. Is the address displayed the same as `value1`'s?

8.10 *(Function Headers and Prototypes)* Perform the task in each of the following:

a) Write the function header for function `zero` that takes a long integer built-in array parameter `bigIntegers` and a second parameter representing the array's size and does not return a value.

b) Write the function prototype for the function in part (a).

c) Write the function header for function `add1AndSum` that takes an integer built-in array parameter `oneTooSmall` and a second parameter representing the array's size and returns an integer.

d) Write the function prototype for the function described in part (c).

8.11 *(Find the Code Errors)* Find the error in each of the following segments. If the error can be corrected, explain how.

a)
```
int* number;
cout << number << endl;
```

b)
```
double* realPtr;
long* integerPtr;
integerPtr = realPtr;
```

c)
```
int* x, y;
x = y;
```

d)
```
char s[]{"this is a character array"};
for (; *s != '\0'; ++s) {
   cout << *s << ' ';
}
```

e)
```
short* numPtr, result;
void* genericPtr{numPtr};
result = *genericPtr + 7;
```

f)
```
double x = 19.34;
double xPtr{&x};
cout << xPtr << endl;
```

8.12 *(Simulation: The Tortoise and the Hare)* In this exercise, you'll re-create the classic race of the tortoise and the hare. You'll use random number generation to develop a simulation of this memorable event.

Our contenders begin the race at "square 1" of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

There is a clock that ticks once per second. With each tick of the clock, your program should use function `moveTortoise` and `moveHare` to adjust the position of the animals according to the rules in Fig. 8.18. These functions should use pointer-based pass-by-reference to modify the position of the tortoise and the hare.

An[illegible]	[illegible]	[illegible]e time	[illegible]
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Fig. 8.18 | Rules for moving the tortoise and the hare.

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the "starting gate"). If an animal slips left before square 1, move the animal back to square 1.

Generate the percentages in Fig. 8.18 by producing a random integer i in the range $1 \le i \le 10$. For the tortoise, perform a "fast plod" when $1 \le i \le 5$, a "slip" when $6 \le i \le 7$ or a "slow plod" when $8 \le i \le 10$. Use a similar technique to move the hare.

Begin the race by displaying

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

For each tick of the clock (i.e., each iteration of a loop), display a 70-position line showing the letter T in the tortoise's position and the letter H in the hare's position. Occasionally, the contenders land on the same square. In this case, the tortoise bites the hare and your program should display OUCH!!! beginning at that position. All positions other than the T, the H or the OUCH!!! (in case of a tie) should be blank.

After displaying each line, test whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation. If the tortoise wins, display TORTOISE WINS!!! YAY!!! If the hare wins, display Hare wins. Yuch. If both animals win on the same clock tick, you may want to favor the tortoise (the "underdog"), or you may want to display It's a tie. If neither animal wins, perform the loop again to simulate the next tick of the clock.

8.13 *(What Does This Code Do?)* What does this program do?

```
// Ex. 8.13: ex08_13.cpp
// What does this program do?
#include <iostream>
using namespace std;

void mystery1(char*, const char*); // prototype
```

Fig. 8.19 | What does this program do? (Part 1 of 2.)

```
int main() {
   char string1[80];
   char string2[80];

   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   mystery1(string1, string2);
   cout << string1 << endl;
}

// What does this function do?
void mystery1(char* s1, const char* s2) {
   while (*s1 != '\0') {
      ++s1;
   }

   for (; (*s1 = *s2); ++s1, ++s2) {
      ; // empty statement
   }
}
```

Fig. 8.19 | What does this program do? (Part 2 of 2.)

8.14 *(What Does This Code Do?)* What does this program do?

```
// Ex. 8.14: ex08_14.cpp
// What does this program do?
#include <iostream>
using namespace std;

int mystery2(const char*); // prototype

int main() {
   char string1[80];

   cout << "Enter a string: ";
   cin >> string1;
   cout << mystery2(string1) << endl;
}

// What does this function do?
int mystery2(const char* s) {
   unsigned int x;

   for (x = 0; *s != '\0'; ++s) {
      ++x;
   }

   return x;
}
```

Fig. 8.20 | What does this program do?

Special Section: Building Your Own Computer

In the next several problems, we take a temporary diversion away from the world of high-level-language programming. We "peel open" a simply hypothetical computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (using software-based *simulation*) on which you can execute your machine-language programs![2]

8.15 (***Machine-Language Programming***) Let's create a computer we'll call the Simpletron. As its name implies, it's a simple machine, but, as we'll soon see, it's a powerful one as well. The Simpletron runs programs written in the only language it directly understands, that is, Simpletron Machine Language, or SML for short.

The Simpletron contains an *accumulator*—a "special register" in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of *words*. A word is a signed four-digit decimal number, such as +3364, -1293, +0007, -0001, etc. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers 00, 01, …, 99.

Before running an SML program, we must *load*, or place, the program into memory. The first instruction (or statement) of every SML program is always placed in location 00. The simulator will start executing at this location.

Each instruction written in SML occupies one word of the Simpletron's memory; thus, instructions are signed four-digit decimal numbers. Assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron's memory may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the *operation code* that specifies the operation to be performed. SML operation codes are shown in Fig. 8.21.

Operation code	Meaning
Input/output operations	
`const int { };`	Read a word from the keyboard into a specific location in memory.
`const int { };`	Write a word from a specific location in memory to the screen.
Load and store operations	
`const int { };`	Load a word from a specific location in memory into the accumulator.
`const int { };`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations	
`const int { };`	Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).
`const int { };`	Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).

Fig. 8.21 | Simpletron Machine Language (SML) operation codes. (Part 1 of 2.)

[2] In Exercises 19.30–19.34, we'll "peel open" a simple hypothetical compiler that will translate statements in a simple high-level language to the machine language you use here. You'll write programs in that high-level language, compile them into machine language and run that machine language on your computer simulator.