8 Pointers

Objectives

In this chapter you'll:

- Learn what pointers are.
- Declare and initialize pointers.
- Use the address (&) and indirection (*) pointer operators.
- Learn the similarities and differences between pointers and references.
- Use pointers to pass arguments to functions by reference.
- Use built-in arrays.
- Use `const` with pointers.
- Use operator `sizeof` to determine the number of bytes that store a value of a particular type.
- Understand pointer expressions and pointer arithmetic.
- Understand the close relationships between pointers and built-in arrays.
- Use pointer-based strings.
- Use C++11 capabilities, including `nullptr` and Standard Library functions `begin` and `end`.

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.

ANS: address.

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

ANS: `nullptr`, an address.

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

ANS: `0`.

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.

ANS: 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) A pointer that is declared to be of type `void*` can be dereferenced.

ANS: 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) A pointer of one type can't be assigned to one of another type without a cast operation.

ANS: 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 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`.

ANS:
```
double numbers[size]{0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
```

b) Declare a pointer `nPtr` that points to a variable of type `double`.

ANS:
```
double* nPtr;
```

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.

ANS:
```
cout << fixed << showpoint << setprecision(1);
for (size_t i{0}; i < size; ++i) {
   cout << numbers[i] << ' ';
}
```

d) Write two separate statements that each assign the starting address of built-in array `numbers` to the pointer variable `nPtr`.

ANS:
```
nPtr = numbers;
nPtr = &numbers[0];
```

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

ANS:
```
cout << fixed << showpoint << setprecision(1);
for (size_t j{0}; j < size; ++j) {
   cout << *(nPtr + j) << ' ';
}
```

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.

ANS:
```
cout << fixed << showpoint << setprecision(1);
for (size_t k{0}; k < size; ++k) {
   cout << *(numbers + k) << ' ';
}
```

g) Use a `for` statement to display the elements of built-in array `numbers` using pointer/subscript notation with pointer `nPtr`.

ANS:
```
cout << fixed << showpoint << setprecision(1);
for (size_t m{0}; m < size; ++m) {
   cout << nPtr[m] << ' ';
}
```

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`.

ANS:
```
numbers[3]
*(numbers + 3)
nPtr[3]
*(nPtr + 3)
```

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?

ANS: The address is `1002500 + 8 * 8 = 1002564`. The value is `8.8`.

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?

ANS: 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 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`.

ANS: `double* doublePtr{nullptr};`

b) Assign the address of variable `number1` to pointer variable `doublePtr`.

ANS: `doublePtr = &number1;`

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

ANS: `cout << "The value of *fPtr is " << *doublePtr << endl;`

d) Assign the value of the object pointed to by `doublePtr` to variable `number2`.

ANS: `number2 = *doublePtr;`

e) Display the value of `number2`.

ANS: `cout << "The value of number2 is " << number2 << endl;`

f) Display the address of `number1`.

ANS: `cout << "The address of number1 is "  << &number1 << endl;`

g) Display the address stored in `doublePtr`. Is the address the same as that of `number1`?

ANS: `cout << "The address stored in fPtr is " << doublePtr << endl;`
Yes, the value is the same.

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.

ANS: `void exchange(double* x, double* y)`

b) Write the function prototype without parameter names for the function in part (a).

ANS:
```
void exchange(double*, double*);
```

c) Write two statements that each initialize the built-in array of `char`s named `vowel` with the string of vowels, `"AEIOU"`.

ANS:
```
char vowel[]{"AEIOU"};
char vowel[]{'A', 'E', 'I', 'O', 'U', '\0'};
```

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[5]{1, 2, 3, 4, 5};
```

a)
```
++zPtr;
```

ANS: *Error:* `zPtr` has not been initialized.

Correction: Initialize `zPtr` with `zPtr = z;` (Parts *b–e* depend on this correction.)

b)
```
// use pointer to get first value of a built-in array
number = zPtr;
```

ANS: *Error:* The pointer is not dereferenced.

Correction: Change the statement to `number = *zPtr;`

c)
```
// assign built-in array element 2 (the value 3) to number
number = *zPtr[2];
```

ANS: *Error:* `zPtr[2]` is not a pointer and should not be dereferenced.

Correction: Change `*zPtr[2]` to `zPtr[2]`.

d)
```
// display entire built-in array z
for (size_t i{0}; i <= 5; ++i) {
   cout << zPtr[i] << endl;
}
```

ANS: *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)
```
++z;
```

ANS: *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.

ANS: True.

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.

ANS: False. An array name cannot be modified to point to a different location in memory because an array name is a constant pointer to the first element of the array.

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`.

ANS: `unsigned int values[SIZE]{2, 4, 6, 8, 10};`

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

ANS: `unsigned int* vPtr;`

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

ANS:
```
for (int i{0}; i < SIZE; ++i) {
      cout << setw(4) << values[i];
   }
```

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

ANS: `vPtr = values;` and `vPtr = &values[0];`

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

ANS:
```
for (int i{0}; i < SIZE; ++i) {
      cout << setw(4) << *(vPtr + i);
   }
```

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.

ANS:
```
for (int i{0}; i < SIZE; ++i) {
      cout << setw(4) << *(values + i);
   }
```

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

ANS:
```
for (int i{0}; i < SIZE; ++i) {
      cout << setw(4) << vPtr[i];
   }
```

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.

ANS: `values[4]`, `*(values + 4)`, `vPtr[4]`, `*(vPtr + 4)`

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

ANS: The address of the location pertaining to `values[3]` (i.e., 1002506). 8.

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

ANS: The address of where `values` begins in memory (i.e., 1002500). 2.

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`.

ANS: `long* longPtr;`

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

ANS: `longPtr = &value1;`

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

ANS: `cout << *longPtr << '\n';`

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

ANS: `value2 = *longPtr;`

e) Display the value of `value2`.

ANS: `cout << value2 << '\n';`

f) Display the address of `value1`.

ANS: `cout << &value1 << '\n';`

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

ANS: `cout << longPtr << '\n';` yes.

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.

ANS: `void zero(long bigIntegers[], unsigned int size)` or

`void zero(long *bigIntegers, unsigned int size)`

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

ANS: `void zero(long bigIntegers[], unsigned int size);` or

`void zero(long *bigIntegers, unsigned int size);`

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.

ANS: `int add1AndSum(int oneTooSmall[], unsigned int size)` or

`int add1AndSum(int *oneTooSmall, unsigned int size)`

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

ANS: `int add1AndSum(int oneTooSmall[], unsigned int size);` or

`int add1AndSum(int *oneTooSmall, unsigned int size);`

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;
```

ANS: Pointer `number` does not "point" to a valid address—assigning a valid address of an `int` to `number` would correct this the problem. Also, `number` is not dereferenced in the output statement.

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

ANS: A pointer of type `double` cannot be directly assigned to a pointer of type `long`.

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

ANS: Variable `y` is not a pointer, and therefore cannot be assigned to `x`. Change the assignment statement to `x = &y;` or declare `y` as a pointer.

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

ANS: `s` is not a modifiable value. Attempting to use operator `++` is a syntax error. Changing to `[]` notation corrects the problem as in:

```
for (int t{0}; s[t] != '\0'; ++t)
   cout << s[t] << ' ';
```

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

ANS: A `void *` pointer cannot be dereferenced.

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

ANS: `xPtr` is not a pointer and therefore cannot be assigned an address. Change `xPtr`'s type to `double*` to correct the problem. The `cout` statement display's the address to which `xPtr` points (once the previous correction is made)—this is not an error, but normally you'd output the value of what the pointer points to, not the address stored in the pointer.

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

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. 8.1 | What does this program do?

ANS:

```
Enter two strings: string1 string2
string1string2
```

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

```cpp
// 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. 8.2 | What does this program do?

ANS:

```
Enter a string: length
6
```

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 simple 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![1]

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

[1] 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.

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. 8.3.

Operation code	Meaning
Input/output operations	
`const int read{10};`	Read a word from the keyboard into a specific location in memory.
`const int write{11};`	Write a word from a specific location in memory to the screen.
Load and store operations	
`const int load{20};`	Load a word from a specific location in memory into the accumulator.
`const int store{21};`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations	
`const int add{30};`	Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).
`const int subtract{31};`	Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).
`const int divide{32};`	Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).
`const int multiply{33};`	Multiply a word from a specific location in memory by the word in the accumulator (leave result in accumulator).
Transfer-of-control operations	
`const int branch{40};`	Branch to a specific location in memory.
`const int branchneg{41};`	Branch to a specific location in memory if the accumulator is negative.
`const int branchzero{42};`	Branch to a specific location in memory if the accumulator is zero.
`const int halt{43};`	Halt—the program has completed its task.

Fig. 8.3 | Simpletron Machine Language (SML) operation codes.

The last two digits of an SML instruction are the *operand*—the address of the memory location containing the word to which the operation applies.

Now let's consider two simple SML programs. The first (Fig. 8.4) reads two numbers from the keyboard and computes and displays their sum. The instruction `+1007` reads the first number from the keyboard and places it into location `07` (which has been initialized to zero). Instruction `+1008` reads the next number into location `08`. The *load* instruction, `+2007`, places (copies) the first number into the accumulator, and the *add* instruction, `+3008`, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator.* The *store* instruction, `+2109`, places (copies) the result back into memory location `09`. Then the *write* instruction, `+1109`, takes the number and displays it (as a signed four-digit decimal number). The *halt* instruction, `+4300`, terminates execution.

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Fig. 8.4 | SML Example 1.

The SML program in Fig. 8.5 reads two numbers from the keyboard, then determines and displays the larger value. Note the use of the instruction `+4107` as a conditional transfer of control, much the same as C++'s `if` statement.

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to `07`)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 8.5 | SML Example 2.

Now write SML programs to accomplish each of the following tasks:

a) Use a sentinel-controlled loop to read positive numbers and compute and display their sum. Terminate input when a negative number is entered.

ANS:

```
00   +1009    (Read Value)
01   +2009    (Load Value)
02   +4106    (Branch negative to 06)
03   +3008    (Add Sum)
04   +2108    (Store Sum)
05   +4000    (Branch 00)
06   +1108    (Write Sum)
07   +4300    (Halt)
08   +0000    (Storage for Sum)
09   +0000    (Storage for Value)
```

b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and display their average.

ANS:

```
00   +2018    (Load Counter)
01   +3121    (Subtract Termination)
02   +4211    (Branch zero to 11)
03   +2018    (Load Counter)
04   +3019    (Add Increment)
05   +2118    (Store Counter)
06   +1017    (Read Value)
07   +2016    (Load Sum)
08   +3017    (Add Value)
09   +2116    (Store Sum)
10   +4000    (Branch 00)
11   +2016    (Load Sum)
12   +3218    (Divide Counter)
13   +2120    (Store Result)
14   +1120    (Write Result)
15   +4300    (Halt)
16   +0000    (Variable Sum)
17   +0000    (Variable Value)
18   +0000    (Variable Counter)
19   +0001    (Variable Increment)
20   +0000    (Variable Result)
21   +0007    (Variable Termination)
```

c) Read a series of numbers, and determine and display the largest number. The first number read indicates how many numbers should be processed.

ANS:

```
00   +1017    (Read Endvalue)
01   +2018    (Load Counter)
02   +3117    (Subtract Endvalue)
03   +4215    (Branch zero to 15)
04   +2018    (Load Counter)
05   +3021    (Add Increment)
06   +2118    (Store Counter)
07   +1019    (Read Value)
08   +2020    (Load Largest)
09   +3119    (Subtract Value)
10   +4112    (Branch negative to 12)
11   +4001    (Branch 01)
12   +2019    (Load Value)
13   +2120    (Store Largest)
14   +4001    (Branch 01)
```

```
15   +1120    (Write Largest)
16   +4300    (Halt)
17   +0000    (Variable EndValue)
18   +0000    (Variable Counter)
19   +0000    (Variable Value)
20   +0000    (Variable Largest)
21   +0001    (Variable Increment)
```