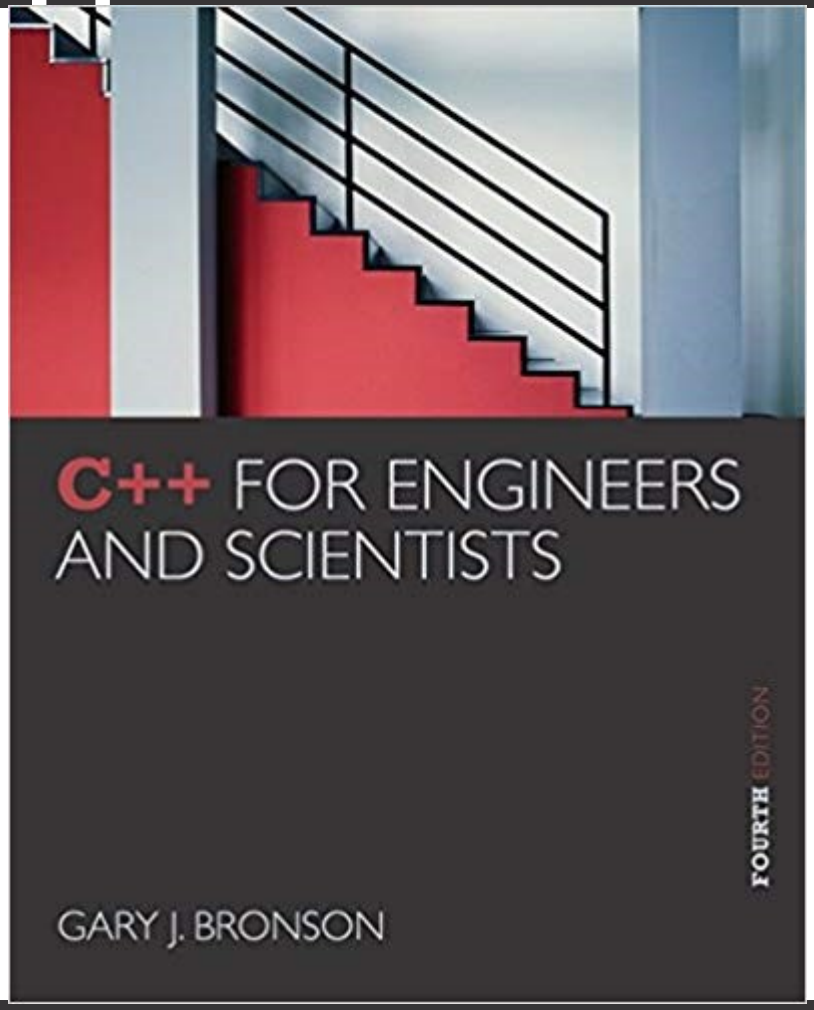
ELEG 1043
Computer Applications in Engineering



Chapter 10: Pointers

C++ FOR ENGINEERS AND SCIENTISTS

Acknowledgement

- Some of the slides or images are from various sources. The copyright of those materials belongs to their original owners.

Objectives

- In this chapter you will learn about:
  - Addresses and pointers
  - Array names as pointers
  - Pointer arithmetic
  - Passing addresses
  - Common programming errors

Addresses and Pointers

- The address operator, `&`, accesses a variable's address in memory
- The address operator placed in front of a variable's name refers to the address of the variable

  `&num` means the address of `num`

Storing Addresses

- Addresses can be stored in a suitably declared variable

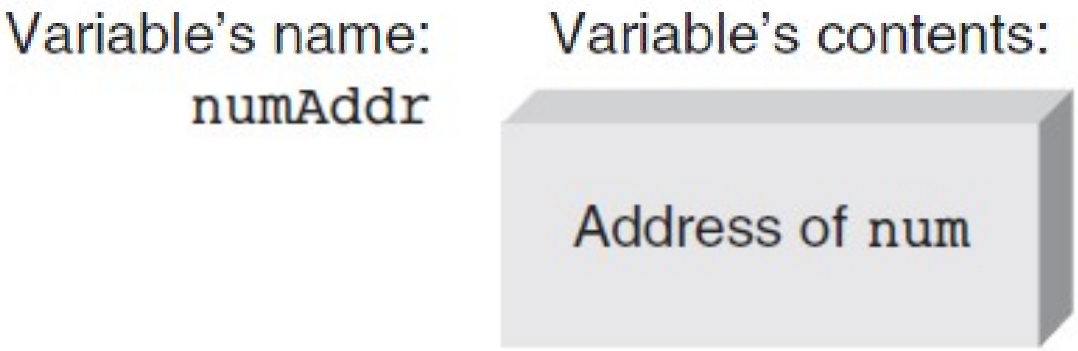


Figure 10.2 Storing `num`'s address in `numAddr`

Storing Addresses (continued)

- Example statements store addresses of the variable `m`, `list`, and `ch` in the variables `d`, `tabPoint`, and `chrPoint`

```
d = &m;
tabPoint = &list;
chrPoint = &ch;
```

- `d`, `tabPoint`, and `chrPoint` are called pointer variables or pointers

Storing Addresses (continued)

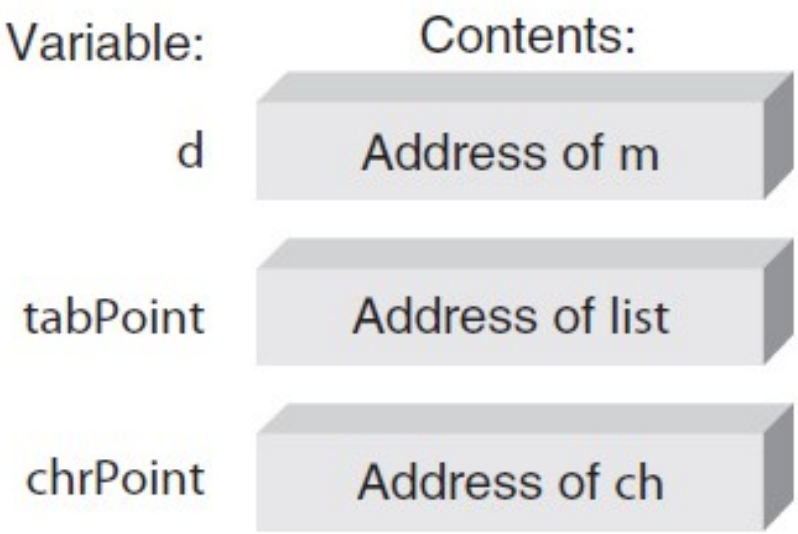


Figure 10.3 Storing more addresses

Using Addresses

- To use a stored address, C++ provides the **indirection operator**, *
- The * symbol, when followed by a pointer, means "the variable whose address is stored in"

  `*numAddr` means the variable whose address is stored in `numAddr`

Using Addresses (continued)

- The address contained in the pointer is then used to get the variable's contents
- Since this is an indirect way of getting to the final value, the term **indirect addressing** is used to describe it

Declaring Pointers

- Like all variables, pointers must be declared before they can be used to store an address
- When declaring a pointer variable, C++ requires specifying the type of the variable that is pointed to
  - Example: `int *numAddr;`
- To understand pointer declarations, reading them "backward" is helpful
  - Start with the indirection operator, `*`, and translate it as "the variable whose address is stored in" or "the variable pointed to by"

Declaring Pointers (continued)

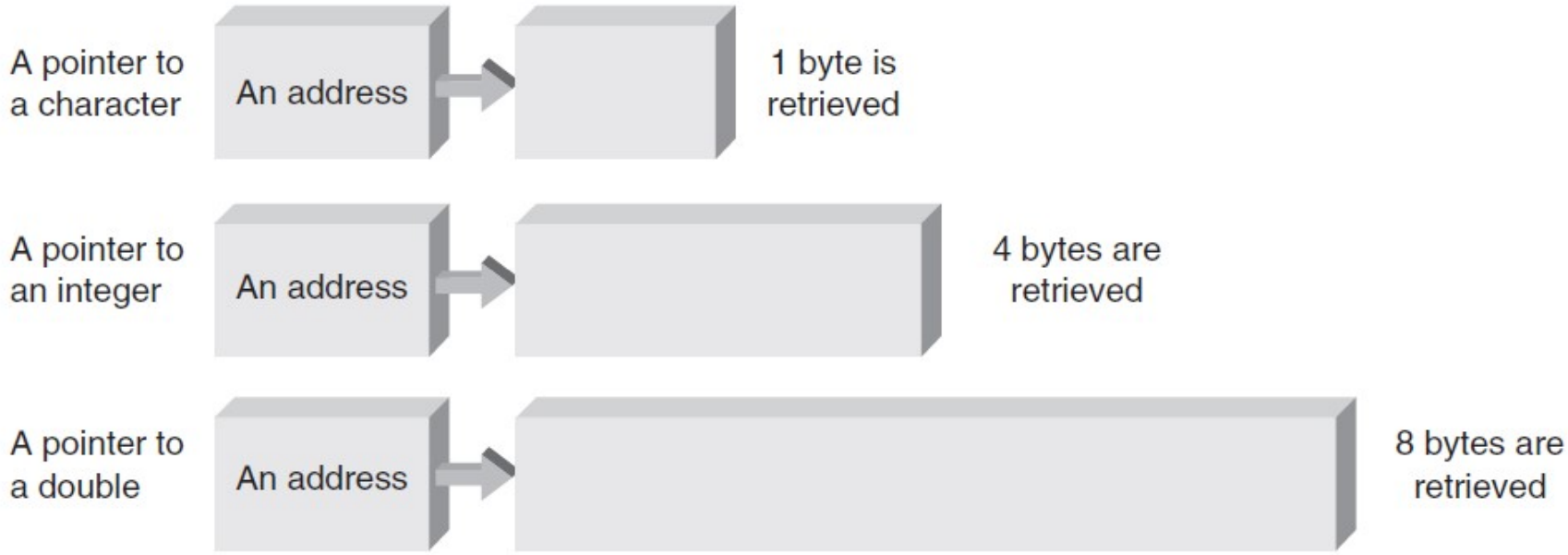


Figure 10.5 Addressing different data types by using pointers

References and Pointers

- A **reference** is a named constant for an address
  - The address named as a constant cannot be changed
- A pointer variable's value address can be changed
- For most applications, using references rather than pointers as arguments to functions is preferred

Reference Variables

- After a variable has been declared, it can be given additional names by using a **reference variable**
- The form of a reference variable is:
  - *dataType& newName = existingName;*
- Example: `double& sum = total;`

Array Names as Pointers

- There is a direct and simple relationship between array names and pointers

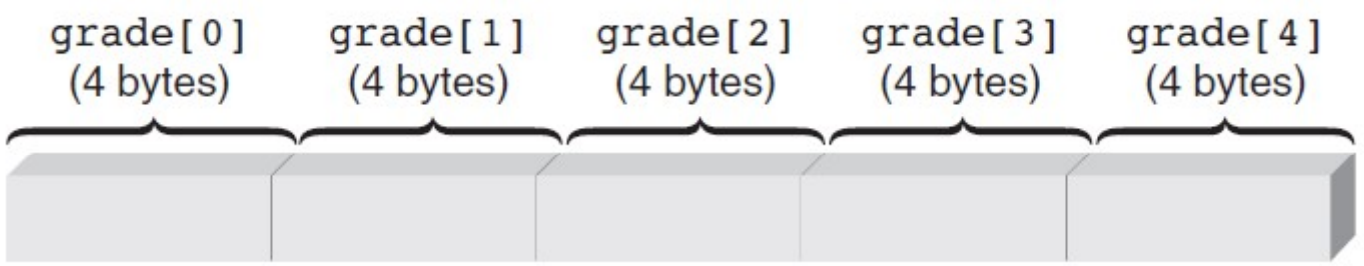


Figure 10.9 The `grade` array in storage

- Using subscripts, the fourth element in grade is referred to as grade[3], address calculated as:

```
&grade[3] = &grade[0] + (3 *
sizeof(int))
```

Array Names as Pointers (continued)

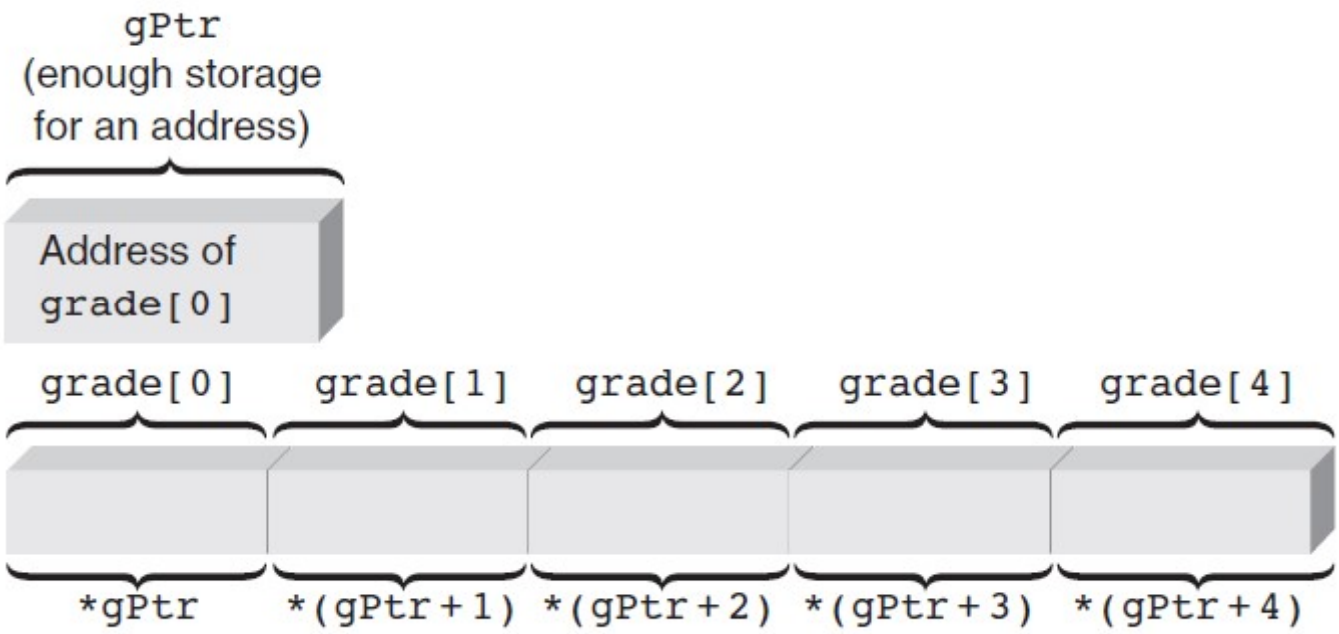


Figure 10.13 The relationship between array elements and pointers

Array Names as Pointers (continued)

Array Element	Subscript Notation	Pointer Notation
Element 0	`grade[0]`	`*gPtr` or `(gPtr + 0)`
Element 1	`grade[1]`	`*(gPtr + 1)`
Element 2	`grade[2]`	`*(gPtr + 2)`
Element 3	`grade[3]`	`*(gPtr + 3)`
Element 4	`grade[4]`	`*(gPtr + 4)`

Table 10.1 Array Elements Can Be Referenced in Two Ways

Exercise 1

- Replace each of the following references to a subscripted variable with a **pointer** reference
  - `year[10]`
  - `seconds[30]`
  - `students[0]`

Exercise 2

- Replace each of the following pointer references with a subscript (index) reference
  - `*(year + 2)`
  - `*(seconds + 20)`
  - `*(students)`

Dynamic Array Allocation

- After memory locations have been reserved for a variable, these locations are fixed for the life of that variable, whether or not they are used
- An alternative to fixed or static allocation is **dynamic allocation** of memory
- Using dynamic allocation, the amount of storage to be allocated is determined or adjusted at run time

Dynamic Array Allocation (continued)

- **`new`** and **`delete`** operators provide the dynamic allocation mechanisms in C++

Operator Name	Description
`new`	Reserves the number of bytes requested by the declaration. Returns the address of the first reserved location or `NULL` if not enough memory is available.
`delete`	Releases a block of bytes reserved previously. The address of the first reserved location must be passed as an argument to the operator.

Table 10.2 The `new` and `delete` Operators (Require the `new` Header File)

Dynamic Array Allocation (continued)

- Dynamic storage requests for scalar variables or arrays are made as part of a declaration or an assignment statement
  - Example:

```
int *num = new int;          // scalar
delete num;
```

  - Example:

```
int *grades = new int[200]; // array
delete[] grades;
```

    - Reserves memory area for 200 integers
    - Address of first integer in array is value of pointer variable `grades`

Pointer Arithmetic

- Pointer variables, like all variables, contain values
- The value stored in a pointer is a memory address
- By adding or subtracting numbers to pointers you can obtain different addresses
- Pointer values can be compared using relational operators (==, <, >, etc.)

Pointer Arithmetic (continued)

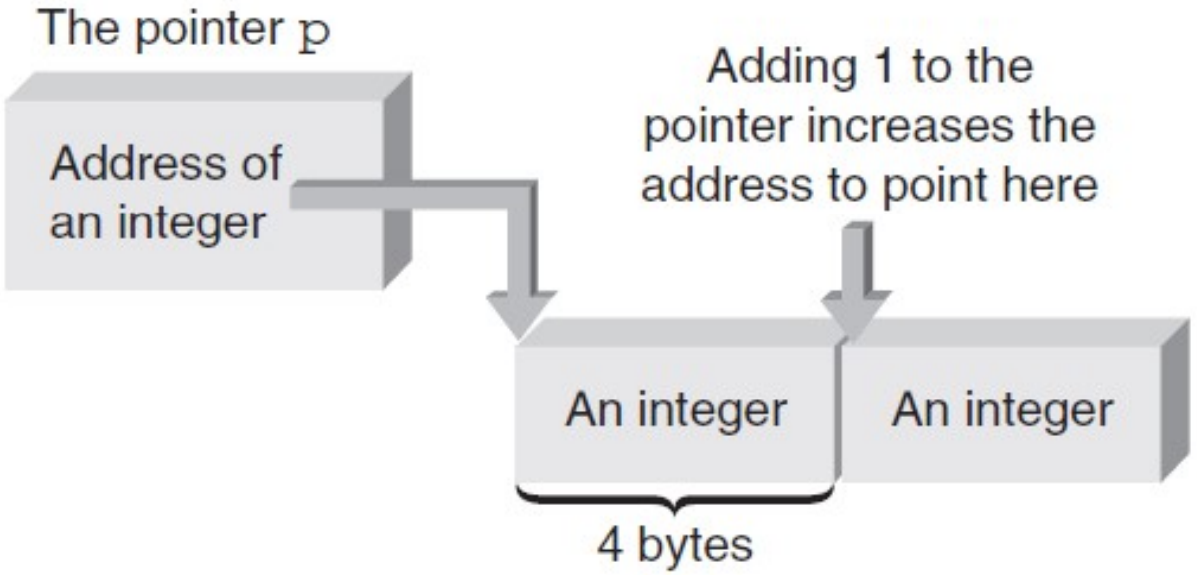


Figure 10.16 Increments are scaled when used with pointers

Pointer Arithmetic (continued)

- Increment and decrement operators can be applied as both prefix and postfix operators

```
*ptNum++     // use the pointer and then increment it
*++ptNum     // increment the pointer before using it
*ptNum--     // use the pointer and then decrement it
*--ptNum     // decrement the pointer before using it
```

- Of the four possible forms, `*ptNum++` is most common
  - Allows accessing each array element as the address is "marched along" from starting address to address of last array element

Pointer Initialization

- Pointers can be initialized with they are declared

```
int *ptNum = &miles;
```

- Pointers to arrays can also be initialized when they are declared

```
double *zing = &volts[0];
```

Passing Addresses

- Reference pointers can be used to pass addresses through reference parameters
  - Implied use of an address
- Pointers can be used explicitly to pass addresses with references
  - Explicitly passing references with the address operator is called **pass by reference**
  - Called function can reference, or access, variables in the calling function by using the passed addresses

Passing Addresses (continued)

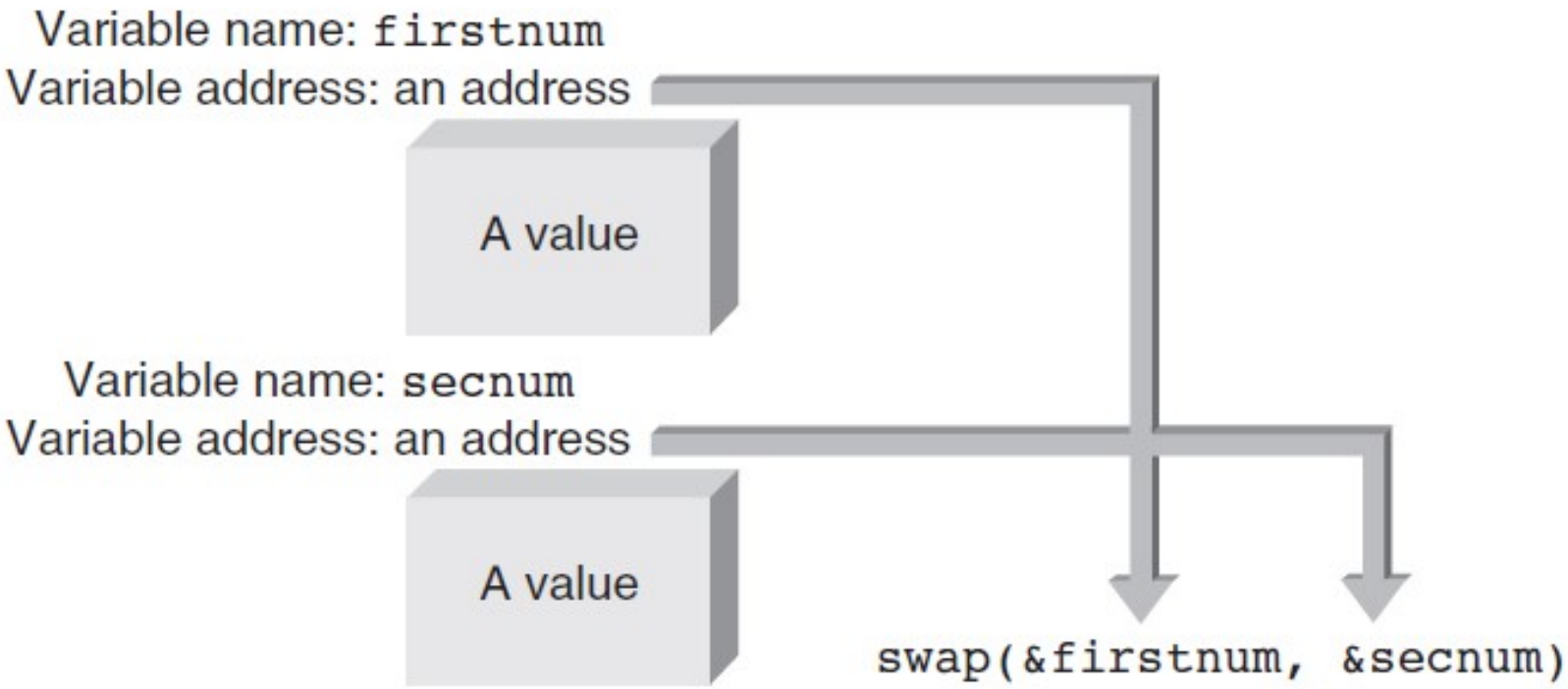


Figure 10.17 Explicitly passing addresses to swap()

Passing Addresses (continued)

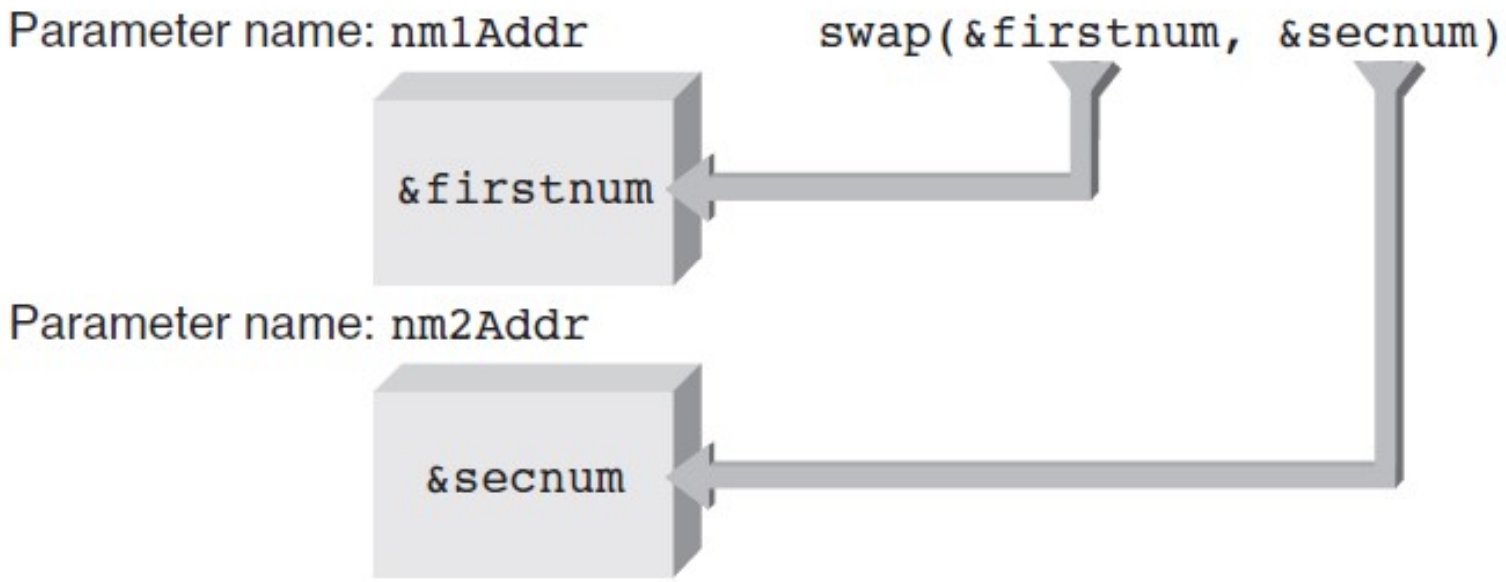


Figure 10.18 Storing addresses in parameters

Passing Arrays

- When an array is passed to a function, its address is the only item actually passed
  - "Address" means the address of the first location used to store the array
  - First location is always element zero of the array

Advanced Pointer Notation

- You can access multidimensional arrays by using pointer notation
  - Notation becomes more cryptic as array dimensions increase
- Sample declaration:

```
int nums[2][3] = { {16,18,20},
                   {25,26,27} };
```

Advanced Pointer Notation (continued)

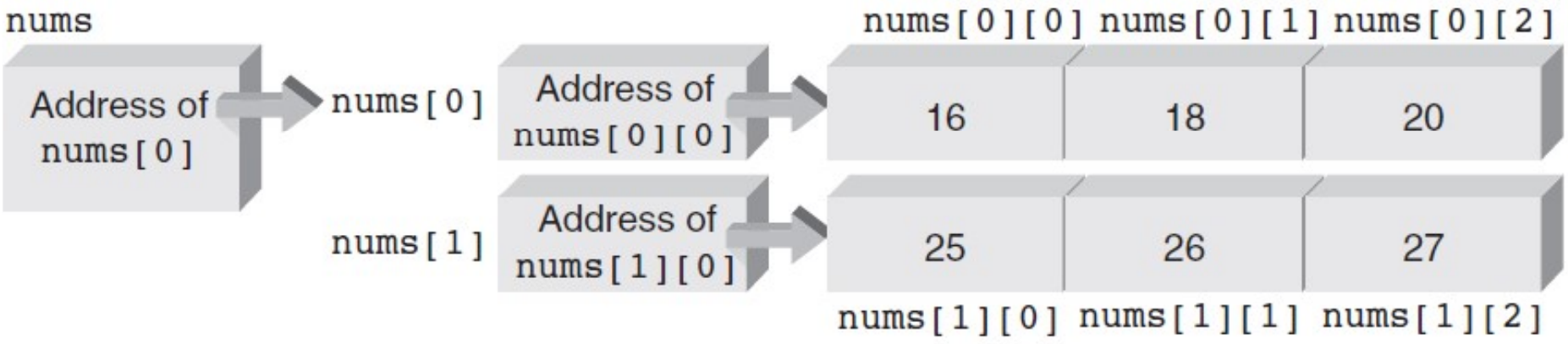


Figure 10.24 Storage of the `nums` array and associated pointer constants

Advanced Pointer Notation (continued)

- Ways to view two-dimensional arrays
  - As an array of rows
    - `nums[0]` is address of first element in the first row
    - Variable pointed to by `nums[0]` is `nums[0][0]`
- The following notations are equivalent:

Pointer Notation	Subscript Notation	Value
`*nums[0]`	`nums[0][0]`	16
`*(nums[0] + 1)`	`nums[0][1]`	18
`*(nums[0] + 2)`	`nums[0][2]`	20
`*nums[1]`	`nums[1][0]`	25
`*(nums[1] + 1)`	`nums[1][1]`	26
`*(nums[1] + 2)`	`nums[1][2]`	27

Advanced Pointer Notation (continued)

- You can replace `nums[0]` and `nums[1]` with pointer notations

  `*nums` is `nums[0]`

  `*(nums + 1)` is `nums[1]`

Pointer Notation	Subscript Notation	Value
`*(*nums)`	`nums[0][0]`	16
`*(*nums + 1)`	`nums[0][1]`	18
`*(*nums + 2)`	`nums[0][2]`	20
`*(*(nums + 1))`	`nums[1][0]`	25
`*(*(nums + 1) + 1)`	`nums[1][1]`	26
`*(*(nums + 1) + 2)`	`nums[1][2]`	27

Advanced Pointer Notation (continued)

- Same notation applies when a two-dimensional array is passed to function
- The `calc` function requires two dimensional array `nums` as a parameter
  - Suitable declarations
    - `calc(int pt[2][3])`
    - `calc(int pt[][3])`
    - `calc(int (*pt)[3])`
      - Parentheses are required
      - Without parentheses would evaluate to `int *pt[3]`, array of three pointers to integers

Common Programming Errors

- Attempting to store address in a variable not declared as pointer
- Using pointer to access nonexistent array elements
- Forgetting to use bracket set, `[]`, after delete operator
- Initialized pointer variables incorrectly

Common Programming Errors (continued)

- When an address is required, any of the following can be used:
  - A pointer variable name
  - A pointer argument name
  - A non-pointer variable name preceded by the address operator
  - A non-pointer variable argument name preceded by the address operator

Summary

- Every variable has a data type, an address, and a value
- In C++, obtain the address of variable by using the address operator, &
- A pointer is a variable used to store the address of another variable
  - Must be declared
  - Use indirection operator, *, to declare the pointer variable and access the variable whose address is stored in pointer

Summary (continued)

- Array name is a pointer constant
- Arrays can be created dynamically as program is executing
- Arrays are passed to functions as addresses
- When a one-dimensional array is passed to a function, the function's parameter declaration can be an array declaration or a pointer declaration
- Pointers can be incremented, decremented, compared, and assigned