Chapter 7: (Downplaying) Pointers in Modern C++

C++ How to Program: An Objects-Natural Approach, 11/e

Presented by
Paul Deitel, CEO, Deitel & Associates, Inc.

Chapter 7: (Downplaying) Pointers in Modern C++

- Variables that store the addresses of other variables
- Pointer declaration/initialization
- Address (&) and indirection (*) pointer operators
- Pointers vs. references
- Pass-by-reference with pointers
- Built-in pointer-based arrays
- Pointer-based strings.
- Use `const` with pointers and the data they point to

Chapter 7: (Downplaying) Pointers in Modern C++

- Operator `sizeof`
- Pointer expressions and pointer arithmetic
- C++11's `nullptr` for pointers to nothing
- Revisit C++11's `begin` and `end` library functions
- C++ Core Guidelines for avoiding pointers
- C++20's to_`array` function
  - Convert built-in arrays and initializer lists to `std::array`s
- C++20's class template `span`
  - Views into built-in arrays, `std::array`s and `std::vector`s

Downplaying Pointers

- Powerful but challenging to work with and error-prone.
- Various Modern C++ features eliminate the need for most pointers
- New software-development projects:
  - use references rather than pointers,
  - use `std::array` and `std::vector` rather than built-in pointer-based arrays
  - use `std::string` objects to pointer-based C-strings

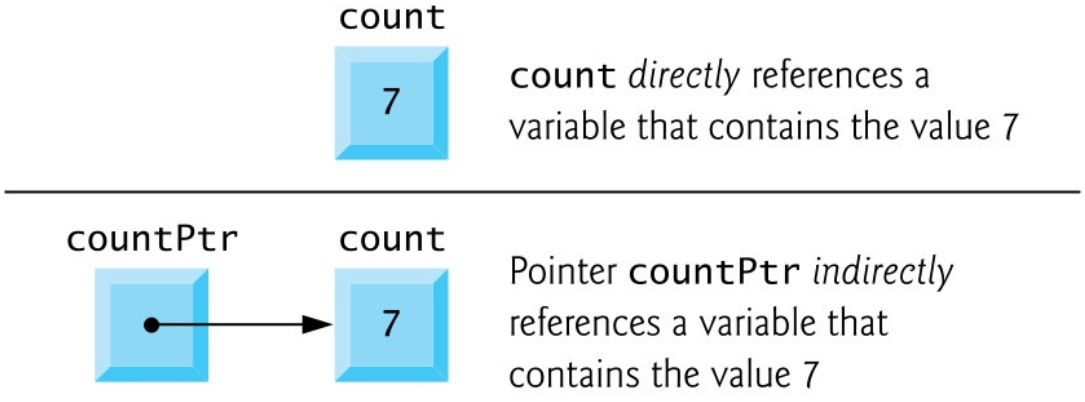
Sometimes Pointers Are Still Required

- Legacy code
- Required for
  - dynamic data structures—though most programmers will use the C++ standard library's existing dynamic containers
  - command-line arguments
  - pass arguments by reference if there's a possibility of a `nullptr`

C++20 Features for Avoiding Pointers

- C++20 adds two features for making programs safer/more robust:
  - Function `to_array` converts a pointer-based array to a more robust `std::array`
  - spans – safer way to pass built-in arrays to functions
    - Iterable, so you can use them with range-based `for`
    - Also can use them with standard library container-processing algorithms
- Key takeaway
  - Avoid using pointers, pointer-based arrays and pointer-based strings whenever possible
  - If you must use them, take advantage of `to_array` and spans.

7.2 Pointer Variable Declarations and Initialization



7.2 Pointer Variable Declarations and Initialization

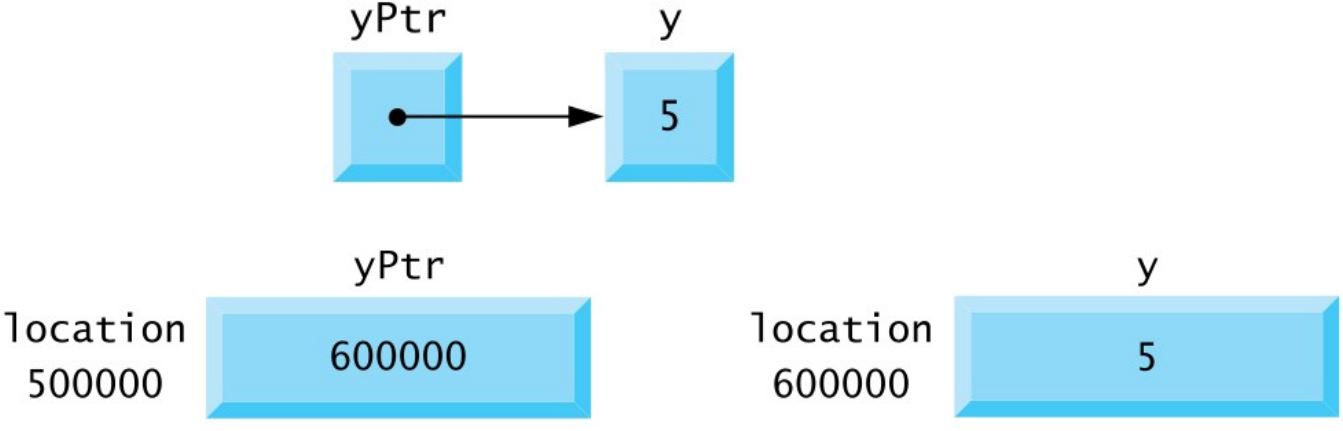
- `int* countPtr; // uninitialized "dangling pointer"`
- C++11 `nullptr`
  - `int* countPtr{nullptr}; // pointer to nothing`
- Null pointers before C++11
  - `0`
  - `NULL`

7.3 Pointer Operators

- Address operator &
- Indirection operator *

7.3 Pointer Operators

- `int y{5}; // declare variable y`
- `int* yPtr{nullptr}; // declare pointer variable yPtr`
- `yPtr = &y; // assign address of y to yPtr`



7.3 Pointer Operators

- Applying unary * operator to a pointer results in an *lvalue* representing the object to which its pointer operand points
  - Known as the indirection or dereferencing operator
- Following statement displays y's value (5):
  - `std::cout << *yPtr << '\n';`
- Using * in this manner is called dereferencing a pointer

7.4 Pass-by-Reference with Pointers

- Three ways to pass arguments to a function
  - pass-by-value
  - pass-by-reference with a reference argument and
  - pass-by-reference with a pointer argument (sometimes called pass-by-pointer)

7.5 Built-In Arrays

- Similar to `std::arrays`—fixed-size data structures
- Common in legacy C++ code
- New apps should use `std::array` and `std::vector`
- `std::array`/`std::vector` objects always know size
  - not so for built-in arrays
- If built-in arrays are required:
  - C++20 `to_array` function to convert to `std::arrays`
  - Process as C++20 spans

7.5.1 Declaring and Accessing a Built-In Array

- Five element built-in array of `int`s named `c`, use
  - `int c[5]; // c is a built-in array of 5 integers`
- Access elements via `[ ]`
  - Does not provide bounds checking

7.5.2 Initializing Built-In Arrays

- `int n[5]{50, 20, 30, 10, 40};`
- If you provide fewer initializers than the number of elements, the remaining elements are value initialized
  - Fundamental numeric types are set to 0
  - bools are set to false
  - pointers are set to nullptr
  - objects receive their default initialization
- Too many initializers is a compilation error
- `int n[]{50, 20, 30, 10, 40};`

7.5.3 Passing Built-In Arrays to Functions

- Built-in array's name is implicitly convertible to a const or non-const pointer to the built-in array's first element
  - decaying to a pointer
- Array name `n` is equivalent to `&n[0]`
- For built-in arrays, a called function can modify all the elements
  - Unless the parameter is declared `const`

7.5.4 Declaring Built-In Array Parameters

- `int sumElements(const int values[], size_t numberOfElements)`
  - Built-in arrays don't know their own size
- `int sumElements(const int* values, size_t numberOfElements)`
- Function must "know" when it's receiving a built-in array vs. a single variable being passed by reference
- C++ Core Guidelines
  - **Do not to pass built-in arrays to functions**
  - Pass C++20 spans
    - Maintain a **pointer to the array's first element and the array's size**

7.5.5 C++11: Standard Library Functions `begin` and `end`

- `sort(begin(colors), end(colors)); // sort contents of colors`
- `sort` (and many other C++ Standard Library functions) also can be applied to built-in arrays
  - `sort(begin(n), end(n)); // sort contents of built-in array n`
  - works only in the scope that originally defines the array

7.5.6 Built-In Array Limitations

- Cannot be compared using the relational and equality operators
  - For built-in arrays named `array1` and `array2`, the following condition would always be false
    - `array1 == array2`
- They cannot be assigned to one another
- They don’t know their own size
- They don’t provide automatic bounds checking

7.6 Using C++20 `to_array` to Convert a Built-in Array to a `std::array`

- C++ legacy code often contains built-in arrays
- C++ Core Guidelines—prefer `std::arrays` and `std::vectors` to built-in arrays
  - safer and do not decay to pointers when you pass them to functions
- `std::to_array` function (header `<array>`) creates a `std::array` from a built-in array
- Figure 7.6 demonstrates `to_array`.

7.7 Using `const` with Pointers and the Data Pointed To

- Four ways to pass a pointer to a function:
  - a nonconstant pointer to nonconstant data,
  - a nonconstant pointer to constant data,
  - a constant pointer to nonconstant data and
  - a constant pointer to constant data
- Each combination provides a different level of access privilege
- The highest access is a nonconstant pointer to nonconstant data (e.g., `int* ptr;`)

7.8 `sizeof` Operator

- Compile-time unary operator
- Determines the size in bytes of a built-in array, type, variable or constant
- When applied to a built-in array's name, returns the total number of bytes in the built-in array as a value of type `size_t`

7.9 Pointer Expressions and Pointer Arithmetic

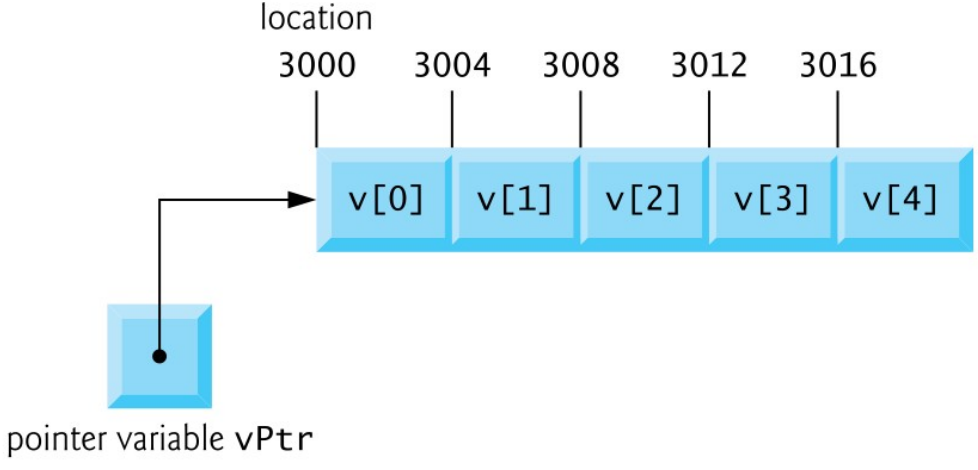
- Arithmetic operations that may be performed on pointers
- Pointer arithmetic is appropriate only for pointers that point to built-in array elements
- Likely to encounter pointer arithmetic in legacy code
- C++ Core Guidelines – A pointer should refer only to a single object (not an array)
- C++ Core Guidelines – Do not use pointer arithmetic because it’s highly error prone

7.9 Pointer Expressions and Pointer Arithmetic

- Valid pointer arithmetic operations:
  - incrementing (++) or decrementing (--),
  - adding an integer to a pointer (+ or +=) or subtracting an integer from a pointer (- or -=), and
  - subtracting one pointer from another of the same type
    - Only for two pointers that point to elements of the same built-in array

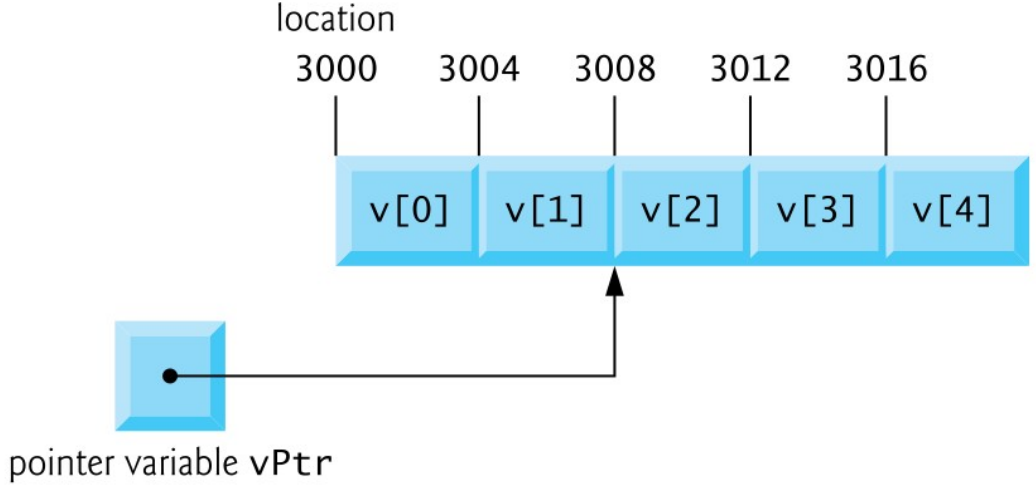
7.9 Pointer Expressions and Pointer Arithmetic

- `int v[5];`
- `int* vPtr{v};`



7.9 Pointer Expressions and Pointer Arithmetic

- `vPtr += 2;`



7.9.3 Pointer Assignment

- A pointer can be assigned to another pointer if both pointers are of the same type
- Exception: Pointer to void (i.e., `void*`)
  - Capable of representing any pointer type
  - Any pointer can be assigned to a `void*`
  - A pointer of type **void*** cannot be assigned directly to a pointer of another type
    - Must cast to the proper pointer type (generally via a `reinterpret_cast`

7.9.4 Cannot Dereference a `void*`

- The allowed operations on `void*` pointers are:
  - comparing `void*` pointers with other pointers,
  - casting `void*` pointers to other pointer types and
  - assigning addresses to `void*` pointers.
- All other operations on `void*` pointers are compilation errors.

7.10 Objects-Natural Case Study: C++20 spans—Views of Contiguous Container Elements

- A span (header) enables programs to view contiguous elements of a container
  - built-in array, `std::array`, `std::vector`
- "Sees" the container's elements but does not have its own copy of those elements
- **C++ Core Guidelines**
  - Pass built-in arrays to functions as spans, which contain both a pointer to the array's first element and the array's size
  - Pass a span by value because it's just as efficient as passing a pointer and size separately

7.11 A Brief Intro to Pointer-Based Strings

- C-style, pointer-based strings
- `std::string` is preferred
- There are some cases in which C-strings are required
  - Command-line arguments
- Frequently appear in legacy C and C++ programs

Pointer-Based Strings

- C-string is a built-in array of characters ending with a null character (`'\0'`), which marks where the string terminates in memory
- Accessed via a pointer to its first character
- `sizeof` for a string literal (which is a C-string) is the length of the string, **including** the terminating `'\0'`

String Literals as Initializers

- A string literal may be used as an initializer in the declaration of either a built-in array of chars or a variable of type const char
  - `char color[]{"blue"};`
  - `const char* colorPtr{"blue"};`
- The first declaration above also may be
  - `char color[]{'b', 'l', 'u', 'e', '\0'};`
- String literals are immutable

Problems with C-Strings

- Not allocating sufficient space in a built-in array of chars to store the null character that terminates a string
- Creating or using a C-string that does not contain a terminating null character
  - Output would walk through memory until a null character is encountered or the program crashes
- For a built-in array of chars, be sure that it's large enough to hold the largest string that will be stored
- Strings longer than the built-in arrays of chars in which they're stored overwrite subsequent locations
- Lead to logic errors, program crashes or security breaches

7.11.1 Command-Line Arguments

- There are cases in which built-in arrays and C-strings must be used
- Command-line arguments are often passed to applications to specify configuration options, file names to process and more
  - On Windows the following command uses the `/p` argument to list current folder's contents, pausing after each screen full
    - `dir /p`
  - On Linux or macOS, the following command uses the `-la` argument to list the current folder's contents with details about each file and folder
    - `ls -la`
- Passed into a C++ program as C-strings

7.11.2 Revisiting C++20's `to_array` Function

- `to_array` recognizes a C-string and initializes the `std::array` with the individual characters

7.12 Looking Ahead to Other Pointer Topics

- "Runtime polymorphic processing"
- Dynamic memory management with pointers
  - Create and destroy objects as needed
  - Source of subtle errors, such as "memory leaks"
  - See how "smart pointers" can automatically manage memory
- A function's name is a pointer to its implementation
  - Can be passed into other functions