```
The original value of number is 5
The new value of number is 125
```

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value. (Part 2 of 2.)

8.4 Pass-by-Reference with Pointers (cont.)

Insight: All Arguments Are Passed By Value

- In C++, *all* arguments are *always* passed by value.
- Passing a variable by reference with a pointer *does not actually pass anything by reference*—a pointer to that variable is *passed by value* and is *copied* into the function's corresponding pointer parameter.
- The called function can then access that variable in the caller simply by dereferencing the pointer, thus accomplishing *pass-by-reference*.

8.4 Pass-by-Reference with Pointers (cont.)

Graphical Analysis of Pass-By-Value and Pass-By-Reference

- Figures 8.8–8.9 analyze graphically the execution of the programs in Fig. 8.6 and Fig. 8.7, respectively.
- In the diagrams, the values in blue rectangles above a given expression or variable represent the value of that expression or variable.
- Each diagram's right column shows functions `cubeByValue` (Fig. 8.6) and `cubeByReference` (Fig. 8.7) *only* when they're executing.

Step 1: Before `main` calls `cubeByValue`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

Step 2: After `cubeByValue` receives the call:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

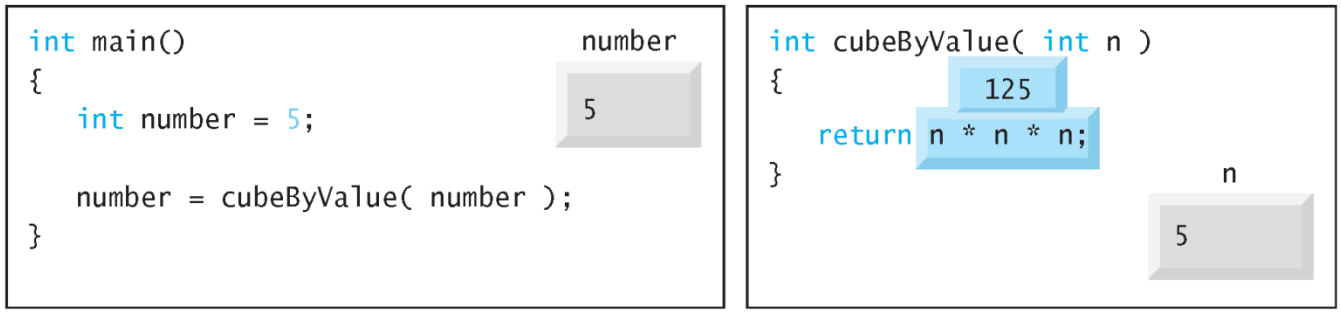
number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Fig. 8.8 | Pass-by-value analysis of the program of Fig. 8.6. (Part 1 of 3.)

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

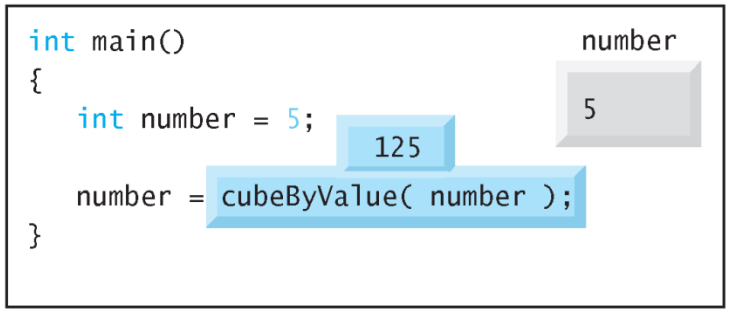


Fig. 8.8 | Pass-by-value analysis of the program of Fig. 8.6. (Part 2 of 3.)

Step 5: After `main` completes the assignment to `number`:

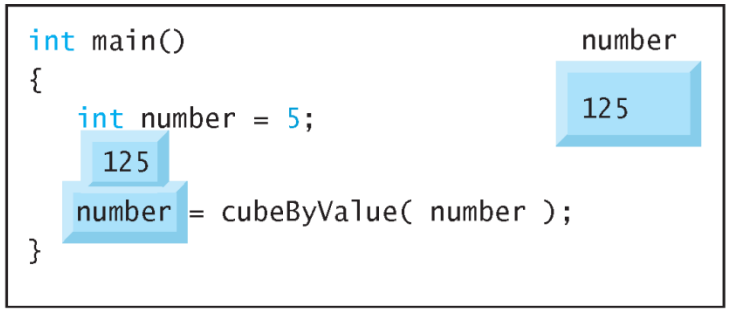
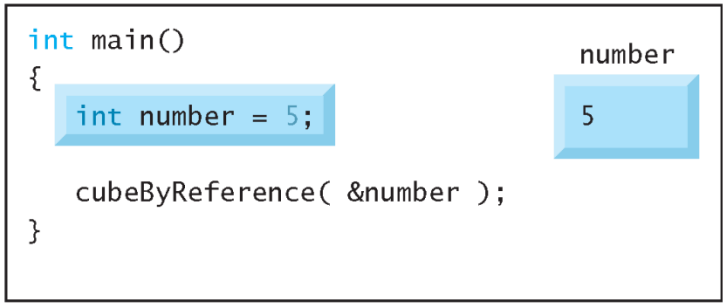


Fig. 8.8 | Pass-by-value analysis of the program of Fig. 8.6. (Part 3 of 3.)

Step 1: Before `main` calls `cubeByReference`:



Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:

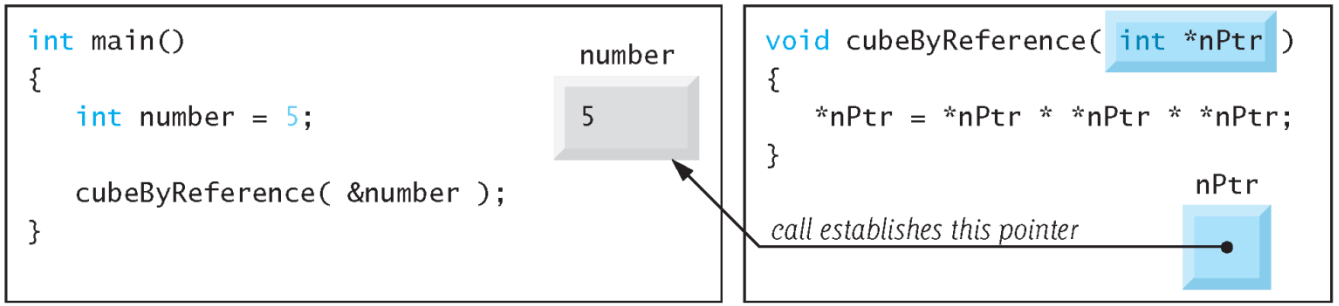
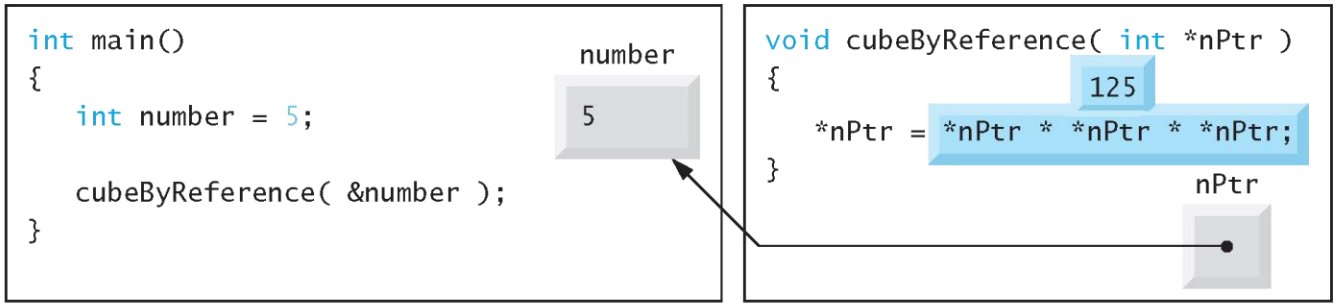


Fig. 8.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 8.7. (Part 1 of 3.)

Step 3: Before `*nPtr` is assigned the result of the calculation `5 * 5 * 5`:



Step 4: After `*nPtr` is assigned 125 and before program control returns to `main`:

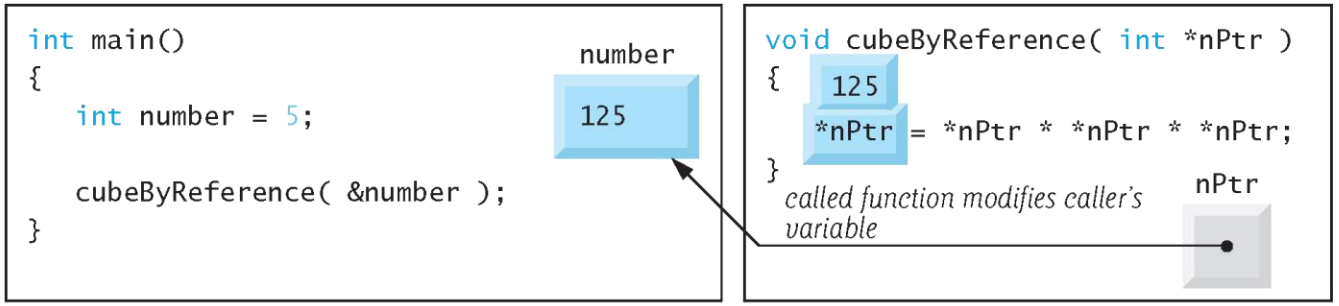


Fig. 8.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 8.7. (Part 2 of 3.)

Step 5: After `cubeByReference` returns to `main`:

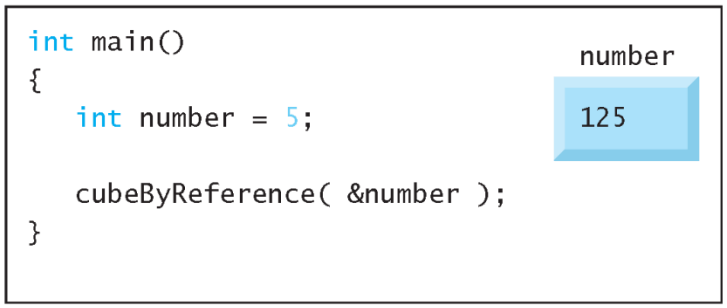


Fig. 8.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 8.7. (Part 3 of 3.)

8.5 Built-In Arrays

- Here we present *built-in arrays*, which are also *fixed-size* data structures.

Declaring a Built-In Array

- To specify the type of the elements and the number of elements required by a built-in array, use a declaration of the form:

  type arrayName[arraySize];

- The compiler reserves the appropriate amount of memory.
- The *arraySize* must be an integer constant greater than zero.
- For example, to tell the compiler to reserve 12 elements for built-in array of `int`s named `c`, use the declaration

```
// c is a built-in array of 12 integers
int c[ 12 ];
```

8.5 Built-In Arrays (cont.)

Accessing a Built-In Array's Elements

- As with array objects, you use the subscript (`[]`) operator to access the individual elements of a built-in array.

8.5 Built-In Arrays (cont.)

Initializing Built-In Arrays

- You can initialize the elements of a built-in array using an initializer list. For example,

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

- creates a built-in array of five `int`s and initializes them to the values in the initializer list.
- If you provide fewer initializers
  - the number of elements, the remaining elements are value initialized—fundamental numeric types are set to `0`, `bool`s are set to false, pointers are set to `nullptr` and class objects are initialized by their default constructors.
- If you provide too many initializers a compilation error occurs.

8.5 Built-In Arrays (cont.)

- If a built-in array's size is *omitted* from a declaration with an initializer list, the compiler sizes the built-in array to the number of elements in the initializer list.
- For example,

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

- creates a five-element array.

Error-Prevention Tip 8.3

Always specify a built-in array's size, even when providing an initializer list. This enables the compiler to ensure that you do not provide too many initializers.

8.5 Built-In Arrays (cont.)

Passing Built-In Arrays to Functions

- *The value of a built-in array's name is implicitly convertible to the address of the built-in array's first element.*
  - So `arrayName` is implicitly convertible to `&arrayName[0]`.
- You don't need to take the address (&) of a built-in array to pass it to a function—you simply pass the built-in array's name.
- For built-in arrays, the called function can modify *all* the elements of a built-in array in the caller—unless the function precedes the corresponding built-in array parameter with `const` to indicate that the elements should *not* be modified.