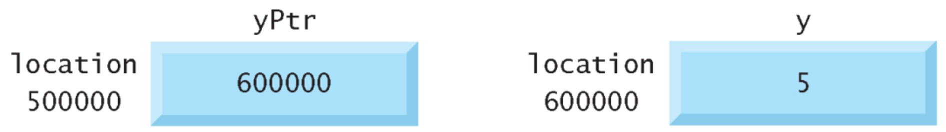

Fig. 8.3 | Representation of `y` and `yPtr` in memory.

8.3 Pointer Operators (cont.)

Indirection (*) Operator

- The unary * operator—commonly referred to as the indirection operator or dereferencing operator—*returns an lvalue representing the object to which its pointer operand points.*
  - Called dereferencing a pointer
- A *dereferenced pointer* may also be used on the *left* side of an assignment.

Common Programming Error 8.2

Dereferencing an uninitialized pointer results in undefined behavior that could cause a fatal execution-time error. This could also lead to accidentally modifying important data, allowing the program to run to completion, possibly with incorrect results.

Error-Prevention Tip 8.2

Dereferencing a null pointer results in undefined behavior and typically is a fatal execution-time error, so you should ensure that a pointer is not null before dereferencing it.

8.3 Pointer Operators (cont.)

Using the Address (&) and Indirection (*) Operators

- The program in Fig. 8.4 demonstrates the & and * pointer operators.

```cpp
// Fig. 8.4: fig08_04.cpp
// Pointer operators & and *.
#include <iostream>
using namespace std;

int main()
{
   int a = 7; // assigned 7 to a
   int *aPtr = &a; // initialize aPtr with the address of int variable a

   cout << "The address of a is " << &a
      << "\nThe value of aPtr is " << aPtr;
   cout << "\n\nThe value of a is " << a
      << "\nThe value of *aPtr is " << *aPtr << endl;
} // end main
```

```
The address of a is 002DFD80
The value of aPtr is 002DFD80

The value of a is 7
The value of *aPtr is 7
```

Fig. 8.4 | Pointer operators & and *.

8.3 Pointer Operators (cont.)

Precedence and Associativity of the Operators Discussed So Far

- Figure 8.5 lists the precedence and associativity of the operators introduced to this point.
- The address (&) and dereferencing operator (*) are *unary operators* on the fourth level.

Operators	Associativity	Type
`::` `()`	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
`()` `[]` `++` `--` `static_cast<`*type*`>(`*operand*`)`	left to right	postfix
`++` `--` `+` `-` `!` `&` `*`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional

Fig. 8.5 | Operator precedence and associativity of the operators discussed so far. (Part 1 of 2.)

Operators	Associativity	Type
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 8.5 | Operator precedence and associativity of the operators discussed so far. (Part 2 of 2.)

8.4 Pass-by-Reference with Pointers

- There are three ways in C++ to pass arguments to a function—pass-by-value, pass-by-reference with reference arguments and pass-by-reference with pointer arguments.
- Here, we explain pass-by-reference with pointer arguments.
- Pointers, like references, can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.
- You can use pointers and the indirection operator (*) to accomplish pass-by-reference.
- When calling a function with an argument that should be modified, the *address* of the argument is passed.

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

An Example of Pass-By-Value

- Figure 8.6 and Fig. 8.7 present two versions of a function that cubes an integer.

```cpp
// Fig. 8.6: fig08_06.cpp
// Pass-by-value used to cube a variable's value.
#include <iostream>
using namespace std;

int cubeByValue( int ); // prototype

int main()
{
   int number = 5;

   cout << "The original value of number is " << number;

   number = cubeByValue( number ); // pass number by value to cubeByValue
   cout << "\nThe new value of number is " << number << endl;
} // end main

// calculate and return cube of integer argument
int cubeByValue( int n )
{
   return n * n * n; // cube local variable n and return result
} // end function cubeByValue
```

Fig. 8.6 | Pass-by-value used to cube a variable's value. (Part 1 of 2.)

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

Fig. 8.6 | Pass-by-value used to cube a variable's value. (Part 2 of 2.)

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

An Example of Pass-By-Reference with Pointers

- Figure 8.7 passes the variable `number` to function `cubeByReference` using pass-by-reference with a pointer argument—the address of `number` is passed to the function.
- The function uses the dereferenced pointer to cube the value to which `nPtr` points.
  - This *directly* changes the value of `number` in `main`.

```cpp
// Fig. 8.7: fig08_07.cpp
// Pass-by-reference with a pointer argument used to cube a
// variable's value.
#include <iostream>
using namespace std;

void cubeByReference( int * ); // prototype

int main()
{
   int number = 5;

   cout << "The original value of number is " << number;

   cubeByReference( &number ); // pass number address to cubeByReference

   cout << "\nThe new value of number is " << number << endl;
} // end main

// calculate cube of *nPtr; modifies variable number in main
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
} // end function cubeByReference
```

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value. (Part 1 of 2.)