10.12 Converting between Types (cont.)

- The return type of an overloaded cast operator function is implicitly the type to which the object is being converted.
- If `s` is a class object, when the compiler sees the expression `static_cast< char * >( s )`, the compiler generates the call
    - `s.operator char *()`

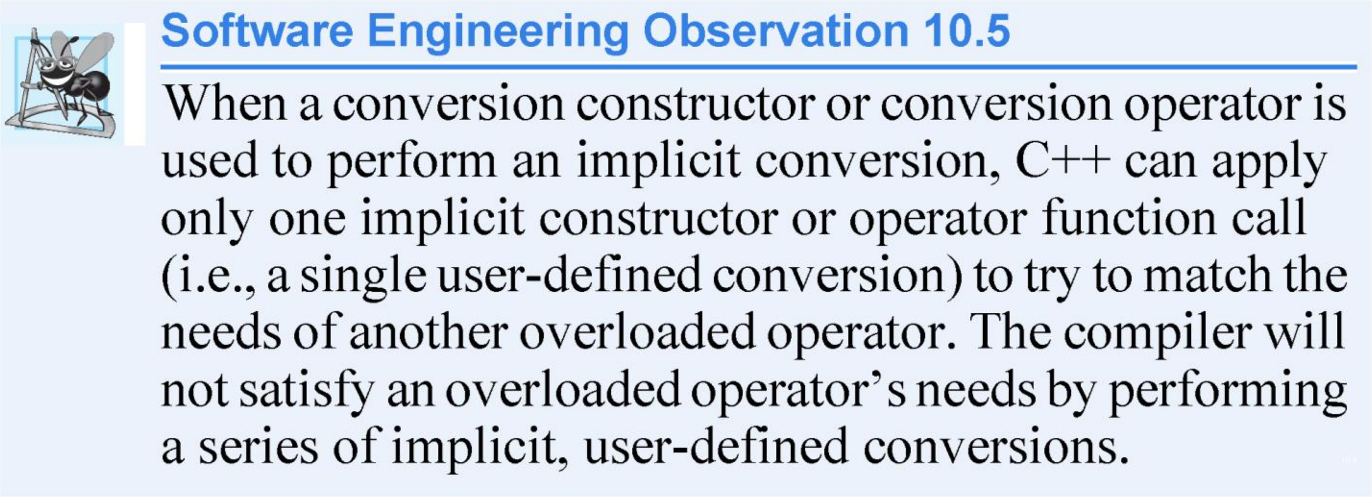
10.12 Converting between Types (cont.)

Overloaded Cast Operator Functions

- Overloaded cast operator functions can be defined to convert objects of user-defined types into fundamental types or into objects of other user-defined types.

Implicit Calls to Cast Operators and Conversion Constructors

- One of the nice features of cast operators and conversion constructors is that, when necessary, the compiler can call these functions *implicitly* to create *temporary objects*.



Software Engineering Observation 10.5

When a conversion constructor or conversion operator is used to perform an implicit conversion, C++ can apply only one implicit constructor or operator function call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator. The compiler will not satisfy an overloaded operator's needs by performing a series of implicit, user-defined conversions.

10.13 `explicit` Constructors and Conversion Operators

- Recall that we've been declaring as explicit every constructor that can be called with one argument.
- With the exception of copy constructors, any constructor that can be called with a *single argument* and is not declared explicit can be used by the compiler to perform an *implicit conversion*.
- The conversion is automatic and you need not use a cast operator.
- *In some situations, implicit conversions are undesirable or error-prone.*
- For example, our `Array` class in Fig. 10.10 defines a constructor that takes a single `int` argument.
- The intent of this constructor is to create an `Array` object containing the number of elements specified by the `int` argument.
- However, if this constructor were not declared `explicit` it could be misused by the compiler to perform an *implicit conversion*.

Common Programming Error 10.6

Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in ambiguous expressions that generate compilation errors or result in execution-time logic errors.

10.13 explicit Constructors and Conversion Operators (cont.)

- The program (Fig. 10.12) uses the Array class of Figs. 10.10–10.11 to demonstrate an improper implicit conversion.
- Line 13 calls function outputArray with the int value 3 as an argument.
- This program does not contain a function called outputArray that takes an int argument.
  - The compiler determines whether class Array provides a conversion constructor that can convert an int into an Array.
  - The compiler assumes the Array constructor that receives a single int is a conversion constructor and uses it to convert the argument 3 into a temporary Array object that contains three elements.
  - Then, the compiler passes the temporary Array object to function outputArray to output the Array's contents.

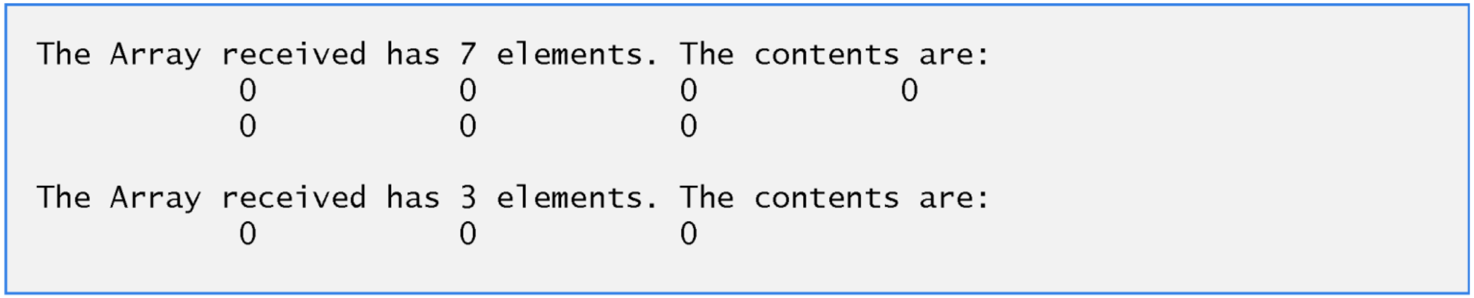
```cpp
// Fig. 10.12: fig10_12.cpp
// Single-argument constructors and implicit conversions.
#include <iostream>
#include "Array.h"
using namespace std;

void outputArray( const Array & ); // prototype

int main()
{
   Array integers1( 7 ); // 7-element Array
   outputArray( integers1 ); // output Array integers1
   outputArray( 3 ); // convert 3 to an Array and output Array’s contents
}  // end main

// print Array contents
void outputArray( const Array &arrayToOutput )
{
   cout << "The Array received has " << arrayToOutput.getSize()
      << " elements. The contents are:\n" << arrayToOutput << endl;
} // end outputArray
```

Fig. 10.12 | Single-argument constructors and implicit conversions. (Part 1 of 2.)



```
The Array received has 7 elements. The contents are:
           0           0           0           0
           0           0           0

The Array received has 3 elements. The contents are:
           0           0           0
```

Fig. 10.12 | Single-argument constructors and implicit conversions. (Part 2 of 2.)

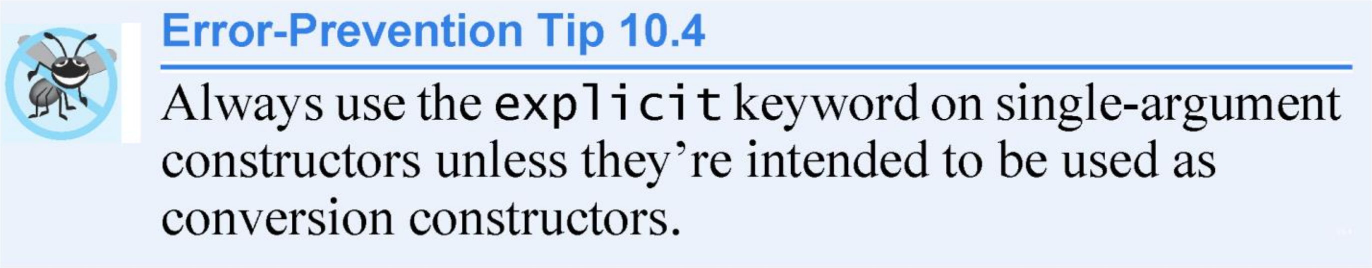
10.13 explicit Constructors and Conversion Operators (cont.)

Preventing Implicit Conversions with Single-Argument Constructors

- The reason we've been declaring every single-argument constructor preceded by the keyword explicit is to *suppress implicit conversions via conversion constructors when such conversions should not be allowed.*
- A constructor that is declared explicit cannot be used in an implicit conversion.
- In the example if Figure 10.13, we use the original version of Array.h from Fig. 10.10, which included the keyword explicit in the declaration of the *single-argument constructor* in line 14.

10.13 `explicit` Constructors and Conversion Operators (cont.)

- Figure 10.13 presents a slightly modified version of the program in Fig. 10.12.
- When this program is compiled, the compiler produces an error message indicating that the integer value passed to `outputArray` in line 13 cannot be converted to a `const Array &`.
- The compiler error message (from Visual C++) is shown in the output window.
- Line 14 demonstrates how the explicit constructor can be used to create a temporary `Array` of `3` elements and pass it to function



Error-Prevention Tip 10.4

Always use the `explicit` keyword on single-argument constructors unless they're intended to be used as conversion constructors.

```cpp
// Fig. 10.13: fig10_13.cpp
// Demonstrating an explicit constructor.
#include <iostream>
#include "Array.h"
using namespace std;

void outputArray( const Array & ); // prototype

int main()
{
   Array integers1( 7 ); // 7-element Array
   outputArray( integers1 ); // output Array integers1
   outputArray( 3 ); // convert 3 to an Array and output Array's contents
   outputArray( Array( 3 ) ); // explicit single-argument constructor call
} // end main

// print Array contents
void outputArray( const Array &arrayToOutput )
{
   cout << "The Array received has " << arrayToOutput.getSize()
      << " elements. The contents are:\n" << arrayToOutput << endl;
} // end outputArray
```

Fig. 10.13 | Demonstrating an `explicit` constructor. (Part 1 of 2.)

```
c:\books\2012\cpphtp9\examples\ch10\fig10_13\fig10_13.cpp(13): error C2664:
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
          Reason: cannot convert from 'int' to 'const Array'
          Constructor for class 'Array' is declared 'explicit'
```

Fig. 10.13 | Demonstrating an `explicit` constructor. (Part 2 of 2.)

10.13 explicit Constructors and Conversion Operators (cont.)

C++11: explicit Conversion Operators

- As of C++11, similar to declaring single-argument constructors explicit, you can declare conversion operators explicit to prevent the compiler from using them to perform implicit conversions.
- For example, the prototype:

```
explicit MyClass::operator char *()
const;
```

- declares MyClass's char * cast operator explicit.

10.14 Overloading the Function Call Operator ()

- Overloading the function call operator `()` is powerful, because functions can take an arbitrary number of comma-separated parameters.
- In a *customized* `String` class, for example, you could overload this operator to select a substring from a `String`—the operator's two integer parameters could specify the *start location* and the *length of the substring to be selected.*
- The `operator()` function could check for such errors as a *start location out of range* or a *negative substring length*.
- The overloaded function call operator must be a non-static member function and could be defined with the first line:

```
String String::operator()( size_t index, size_t length ) const
```

10.14 Overloading the Function Call Operator ()

- In this case, it should be a `const` member function because obtaining a substring should *not* modify the original String object.
- Suppose `string1` is a String object containing the string `"AEIOU"`.
- When the compiler encounters the expression `string1(2, 3)`, it generates the member-function call

```
string1.operator()( 2, 3 )
```

- which returns a `String` containing `"IOU"`.
- Another possible use of the function call operator is to enable an alternate `Array` subscripting notation.
- Instead of using C++'s double-square-bracket notation, such as in `chessBoard[row][column]`, you might prefer to overload the function call operator to enable the notation `chessBoard(row, column)`, where `chessBoard` is an object of a modified two-dimensional `Array` class.