

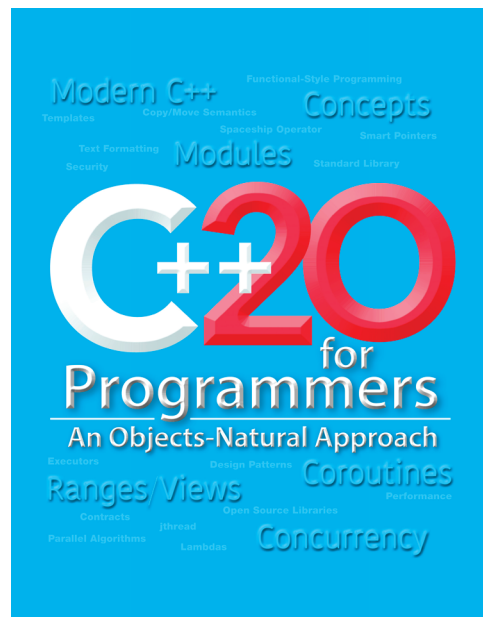
# 20

## Other Topics and a Look Toward the Future of C++

### Objectives

In this chapter, you'll:

- Use `const_cast` to temporarily treat a `const` object as a non-`const` object.
- Understand storage classes and storage duration.
- Use namespaces to ensure that identifiers are unique.
- Use `mutable` members in `const` objects.
- Use operator keywords in place of corresponding operator symbols.
- Use class-member pointer operators `.*` and `->*`.
- Determine an object's type with runtime type information (RTTI), `dynamic_cast`, `typeid` and `type_info`.
- Use the C++17 and C++20 `[[nodiscard]]` attribute to indicate that a function's return value should not be ignored.
- Using smart pointers to manage dynamic memory for objects shared throughout a program.
- Determine types at compile time with `decltype`.
- Receive braced initializers as function arguments.
- Use C++20 attributes `[[likely]]` and `[[unlikely]]` to specify selection-statement paths that are likely or unlikely to execute, so the compiler can optimize accordingly.
- Get an overview of new features being considered for C++23.



<b>20.1</b>	Introduction	<b>20.7</b>	Case Study: Payroll System Using Polymorphism and Runtime Type Information — Downcasting, <code>dynamic_cast</code> , <code>typeid</code> and <code>type_info</code>
<b>20.2</b>	<code>const_cast</code> Operator	<b>20.8</b>	Inheriting Base-Class Constructors
<b>20.3</b>	Storage Classes and Storage Duration	<b>20.9</b>	C++17 and C++20: <code>[[nodiscard]]</code> Attribute
20.3.1	Storage Duration	<b>20.10</b>	<code>shared_ptr</code> and <code>weak_ptr</code> Smart Pointers
20.3.2	Local Variables and Automatic Storage Duration	20.10.1	Reference Counted <code>shared_ptr</code>
20.3.3	Static Storage Duration	20.10.2	<code>weak_ptr</code> : <code>shared_ptr</code> Observer
20.3.4	<code>mutable</code> Class Members	<b>20.11</b>	Trailing Return Types for Functions
20.3.5	Mechanical Demonstration of a <code>mutable</code> Data Member	<b>20.12</b>	<code>decltype</code>
<b>20.4</b>	namespaces	<b>20.13</b>	<code>initializer_list</code> Class Template
20.4.1	Defining namespaces	<b>20.14</b>	C++20: <code>[[likely]]</code> and <code>[[unlikely]]</code> Attributes
20.4.2	Accessing namespace Members with Qualified Names	<b>20.15</b>	A Look Toward C++23
20.4.3	<code>using</code> Directives Should Not Be Placed in Headers	<b>20.16</b>	Wrap-Up
20.4.4	Aliases for namespace Names		
20.4.5	C++17 Nested Namespaces		
<b>20.5</b>	Operator Keywords		
<b>20.6</b>	Pointers to Class Members ( <code>.*</code> and <code>-&gt;*</code> )		


## 20.1 Introduction

20 This chapter presents miscellaneous C++ features. We'll occasionally update it online as  
23 we write about additional C++20 features and new features being developed for C++23  
and beyond. The chapter discusses:

- The `const_cast` operator, which allows you to remove the `const` qualification of a variable.
- Storage classes and storage duration, which determine an object's lifetime.
- namespaces, which help ensure that every identifier in a program has a unique name, helping avoid name conflicts (for example, between your code and library code).
- Operator keywords for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`.
- Operators `.*` and `->*` that you can use with pointers to class members to access a data member or member function.
- Navigating a class hierarchy with downcasting.
- Runtime type information (RTTI), which enables you to dynamically discover an object's type.
- Inheriting base-class constructors—useful when a derived class's constructors have the same parameters and simply pass the arguments to the base class's constructors.
- 17 • C++17's `[[nodiscard]]` attribute, which indicates that a function's return value should not be ignored so compilers can warn you when the return value is not used in your program.

- C++20's `[[nodiscard]]` enhancement, which allows you to specify a string that the compiler will display as the reason the function's caller should not ignore the return value. 20
- `shared_ptr` and `weak_ptr` smart pointers for managing shared dynamically allocated objects. 11
- Trailing return types for functions, which are typically used in complex function templates.
- `decltype`, which can be used at compile time to determine types—typically used in template metaprogramming. 11
- Functions that receive `initializer_lists`, enabling functions to process any number of arguments.
- Attributes `[[likely]]` and `[[unlikely]]`, which specify branches of `if...else` or `switch` statements that are more or less likely to execute, so the compiler can optimize the code accordingly. 20
- Features being considered for C++23. 23

## 20.2 const\_cast Operator

C++ provides the `const_cast` operator<sup>1</sup> for casting away `const`. In general, it's dangerous to use the `const_cast` operator because it allows a program to modify a variable that was declared `const`. But, there are cases where it's desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions that have non-`const` parameters and that do not modify their parameters—if you wish to pass `const` data to such a function, you'd need to cast away the data's `const`-ness; otherwise, the compiler would report errors. Similarly, you could pass **non-const data** to a function that treats it as if it were constant then returns that data as a constant. In such cases, you might need to cast away the `const`-ness of the returned data, as we demonstrate in Fig. 20.1. 

---

```

1 // fig20_01.cpp
2 // Demonstrating const_cast.
3 #include <cctype> // contains prototype for function toupper
4 #include <iostream>
5 #include <string>
6
7 // returns the larger of two strings
8 const std::string& maximum(
9     const std::string& first, const std::string& second) {
10     return (first > second ? first : second);
11 }
12

```

---

**Fig. 20.1** | Demonstrating operator `const_cast`. (Part 1 of 2.)

1. “`const_cast` conversion.” Accessed March 11, 2022. [https://en.cppreference.com/w/cpp/language/const\\_cast](https://en.cppreference.com/w/cpp/language/const_cast).

```

13 int main() {
14     std::string s1{"hello"}; // non-const string
15     std::string s2{"goodbye"}; // non-const string
16
17     // const_cast required to allow the const reference returned by
18     // maximum to be assigned to the std::string reference max
19     std::string &max{const_cast<std::string&>(maximum(s1, s2))};
20
21     std::cout << "The larger string is: " << max << "\n";
22
23     for (char& character : max) {
24         character = std::toupper(character);
25     }
26
27     std::cout << "The larger string capitalized is: " << max << "\n";
28 }

```

```

The larger string is: hello
The larger string capitalized is: HELLO

```

**Fig. 20.1** | Demonstrating operator `const_cast`. (Part 2 of 2.)

In this program, function `maximum` (lines 8–11) receives two strings as `const std::string&` parameters and returns a `const std::string&` that refers to the larger string. Function `main` declares the two `std::string`s as non-const objects (lines 14–15); thus, **these strings are modifiable**. In `main`, we wish to output the larger of the two strings, then modify that string by converting it to uppercase letters.



Err

Function `maximum`'s two parameters are of type `const std::string&`. If the function were to return `std::string&`, the compiler would issue an error message indicating that a `const std::string&` cannot be converted to a `std::string&`. If allowed, that would be dangerous because it attempts to treat data that the function believes to be `const` as if it were non-const data.


Though `maximum` believes the data to be constant, the original strings in `main` are not constant. Therefore, `main` should be able to modify the contents of those strings as necessary. We know these strings are modifiable, so we use `const_cast` (line 19) to demonstrate the mechanics of casting away the `const`-ness of the reference `maximum` returns. Lines 23–25 then use that reference to convert the contents of the larger string to uppercase letters. Without the `const_cast` in line 19, this program will not compile, because you are not allowed to assign a `const std::string&` to a `std::string&`. On the other hand, you can always assign a `std::string&` to a `const std::string&` so you can treat a non-const string as a constant.



CG

**The C++ Core Guidelines say not to cast away `const`-ness.**<sup>2</sup> For cases in which you need `const_cast` in custom classes, such as sharing code between `const` and non-const member functions, **core guideline ES.50** provides sample code showing how to encapsulate this in the class definition. In general, a `const_cast` should be used only if you know that the original data is not constant—such as in a non-const member function of a class,

2. C++ Core Guidelines, “ES.50: Don’t cast away `const`.” Accessed March 10, 2022. <https://iso-cpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const>.

which can be called only on a non-const object of the class.<sup>3</sup> Otherwise, unexpected results may occur. 

## 20.3 Storage Classes and Storage Duration

As you know, programs use identifiers for variable names, functions and class names. The attributes of variables include **name**, **type**, **size** and **value**. Each identifier in a program also has other attributes, including scope, **linkage** and **storage duration**.

As we discussed in Section 5.10, an identifier’s scope is where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program. An identifier’s linkage determines whether it’s known only in the source file where it’s declared or across multiple files that are compiled, then linked together. C++20 modules (Chapter 16) introduce **module linkage** for identifiers known only in the module that defines them. An identifier’s storage-class specifier helps determine its storage duration and linkage. 20

### Storage Class Specifiers

C++ provides several **storage-class specifiers** that determine a variable’s storage duration: **extern**, **mutable**, **static** and **thread\_local**. Storage-class specifier **mutable** is used exclusively with classes, and **thread\_local** is used in multithreaded applications.

#### 20.3.1 Storage Duration

An identifier’s **storage duration** determines the period during which that identifier’s storage exists in memory.<sup>4</sup> Some exist briefly, some are repeatedly created and destroyed, and others exist for a program’s entire execution.

The storage-class specifiers can be split into four storage durations: **automatic**, **static**, **dynamic** and **thread**. In Chapter 11, you learned that you can dynamically allocate additional memory at execution time. Variables allocated dynamically have **dynamic storage duration**. The rest of this section focuses on automatic and static storage duration, and **mutable** data members.

#### 20.3.2 Local Variables and Automatic Storage Duration

Variables with **automatic storage duration** include:

- local variables declared in functions
- function parameters

Such automatic variables are created when program execution enters the block in which they’re defined. They exist while the block is active, and they’re destroyed when the program exits the block. An automatic variable exists only from where it’s defined to the closing brace of the block in which the definition appears, or for the entire function body in the case of a function parameter. Local variables are of automatic storage duration by default.

3. C++ Core Guidelines, “ES.50: Don’t cast away const.” Accessed March 10, 2022. <https://iso-cpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const>.

4. “Storage class specifiers.” Accessed March 11, 2022. [https://en.cppreference.com/w/cpp/language/storage\\_duration](https://en.cppreference.com/w/cpp/language/storage_duration).



Automatic storage is a means of conserving memory because automatic storage duration variables exist in memory only when the block in which they're defined is executing. Declare variables as close as possible to where they're first used.

### 20.3.3 Static Storage Duration

Keywords `extern` and `static` declare identifiers for variables with **static storage duration** and functions. Variables with static storage duration exist in memory from the point at which the program begins execution until the program terminates. Such a variable is initialized once when its declaration is encountered. A function name exists when the program begins execution. Even though function names and static-storage-duration variables exist from the start of program execution, their scope determines where they can be used in the program.

#### Identifiers with Static Storage Duration

There are two types of identifiers with static storage duration—external identifiers (such as global variables) and local variables declared with the storage-class specifier `static`. **Global variables** are created by placing variable declarations outside any class or function definition. Global variables retain their values throughout a program's execution. Global variables and global functions can be referenced by any function that follows their declarations or definitions in the source file. If a global variable is declared `static`, it is known only in that translation unit. If a C++20 module (Chapter 16) contains a global `static` variable, that variable may not be exported by the module. In general, you should avoid global variables.

#### static Local Variables

Local `static` variables are known only in the function that declares them but retain their values when the function returns to its caller. The next time the function is called, each `static` local variable contains the value it had when the function last completed execution. The following statement declares local `static` variable `count` and initializes it to 1:

```
static int count{1};
```

The initialization occurs only the first time this statement is encountered. All numeric variables of `static` storage duration are **initialized to zero by default**. It's nevertheless a good practice to explicitly initialize all variables, so each variable's initial value is clear.

Storage-class specifiers `extern` and `static` determine an identifier's **linkage** when they're applied explicitly to external identifiers such as global variables and global function names:<sup>5</sup>

- `extern` indicates that the identifier is visible to other translation units.
- `static` indicates that the identifier is visible only in the current translation unit.

20 C++20 modules (Chapter 16) must explicitly export identifiers to make them visible to other translation units.

---

5. "Storage class specifiers." Accessed March 11, 2022. [https://en.cppreference.com/w/cpp/language/storage\\_duration](https://en.cppreference.com/w/cpp/language/storage_duration).

### 20.3.4 mutable Class Members

Section 20.2 introduced the `const_cast` operator for removing the “const-ness” of a type. A `const_cast` operation can also be applied to a data member of a `const` object from the body of a `const` member function of that object’s class. This enables the `const` member function to modify the data member, even though the member function considers the object to be `const`.

For example, consider a linked list that maintains its contents in sorted order. Searching through the linked list does not require modifications to the data of the linked list, so the search function could be a `const` member function of the linked-list class. However, it’s conceivable that a linked-list object—to make subsequent searches more efficient—might keep track of the location of the last successful match. If the next search operation attempts to locate an item that appears later in the list, the search could begin from the location of the last successful match rather than from the beginning of the list. To do this, the `const` member function that performs the search must be able to modify the data member that keeps track of the last successful search.

For a class with a “secret” implementation detail that should always be modifiable—such as our linked-list example—the C++ Core Guidelines recommend using the storage-class specifier `mutable` to designate that a data member is always modifiable, even in a `const` member function or `const` object.<sup>6,7</sup> Though `mutable` is a storage-class specifier, it does not affect a variable’s storage duration or linkage.



### 20.3.5 Mechanical Demonstration of a mutable Data Member

Figure 20.2 presents a mechanical example of a `mutable` member to prove that such a member is always modifiable. Class `TestMutable` (lines 6–15) contains

- a constructor (line 8),
- function `getValue` (lines 10–12), which is a `const` member function that returns a copy of `value`, and
- a private data member `value` that’s declared `mutable` (line 14).

Function `getValue` increments the `mutable` data member `value` in the return statement (line 11). Typically, a `const` member function cannot modify the object on which it’s called. Because the data member `value` is `mutable`, this `const` function can modify the data.

---

```

1 // fig20_02.cpp
2 // Demonstrating storage-class specifier mutable.
3 #include <iostream>
4

```

---

**Fig. 20.2** | Demonstrating a `mutable` data member. (Part 1 of 2.)

---

6. C++ Core Guidelines, “ES.50: Don’t cast away `const`.” Accessed March 10, 2022. <https://iso-cpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const>.

7. “cv (const and volatile) type qualifiers—mutable specifier.” Accessed March 11, 2022. <https://en.cppreference.com/w/cpp/language/cv>.

```

5 // class TestMutable definition
6 class TestMutable {
7 public:
8     TestMutable(int v = 0) : value{v} { }
9
10    int getValue() const {
11        return ++value; // increments value
12    }
13 private:
14    mutable int value; // mutable member
15 };
16
17 int main() {
18     const TestMutable test{99};
19
20     std::cout << "Initial value: " << test.getValue()
21         << "\nModified value: " << test.getValue() << "\n";
22 }

```

```

Initial value: 100
Modified value: 101

```

**Fig. 20.2** | Demonstrating a mutable data member. (Part 2 of 2.)

Line 18 declares `const TestMutable` object `test` and initializes it to 99. Line 20 calls the `const` member function `getValue`, which adds one to `value` and returns its new value. The compiler allows the `getValue` call on the object `test` because it's a `const` object and `getValue` is a `const` member function. However, `getValue` modifies variable `value`. Thus, when line 20 invokes `getValue`, the new `value` (100) is output to prove that the mutable data member was modified. Line 21 demonstrates this again, displaying 101.

## 20.4 namespaces

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will collide with a variable of the same name in a different scope, possibly creating a naming conflict. Such overlapping can occur at many levels. Identifier overlapping frequently occurs in libraries that use the same names for global identifiers (such as functions). This can cause compilation errors for multiple definitions of the same identifiers.

C++ solves this problem with **namespaces**.<sup>8</sup> Each namespace defines a scope containing identifiers. To use a **namespace member**, the member's name must be qualified with the namespace name and the **scope resolution operator** (`::`), as in

*MyNameSpace::member*

or a `using` directive must appear before the name is used in the program. Typically, such `using` statements are placed at the beginning of the file that uses the namespace members. For example, placing the following `using` directive at the beginning of a source-code file

```
using namespace MyNameSpace;
```

8. "Namespaces." Accessed March 11, 2022. <https://en.cppreference.com/w/cpp/language/namespace>.



specifies that the code in the file can use *MyNameSpace* members without preceding each member with

```
MyNameSpace::
```

A using declaration of the form

```
using std::cout;
```

brings one name into the scope where the declaration appears. A using directive of the following form brings all the names from the specified namespace (std) into the scope where the directive appears:

```
using namespace std;
```

Generally, you should precede a member with its namespace name and the scope resolution operator (::) to prevent naming conflicts. Not all namespaces are guaranteed to be unique. Two third-party vendors might inadvertently use the same identifiers for their namespace names. Figure 20.3 demonstrates namespaces. Namespaces are commonly used in Modern C++, and now in C++20 modules. See Chapter 16 for more information. 20




---

```

1 // fig20_03.cpp
2 // Demonstrating namespaces.
3 #include <fmt/format.h>
4 #include <iostream>
5
6 int integer1{98}; // global variable
7
8 // create namespace Example
9 namespace Example {
10 // declare two constants and one variable
11     const double pi{3.14159};
12     const double e{2.71828};
13     int integer1{8};
14
15     void printValues(); // prototype
16
17     // nested namespace
18     namespace Inner {
19         // define enumeration
20         enum Years {fiscal1 = 2020, fiscal2, fiscal3};
21     }
22 }
23
24 // create unnamed namespace
25 namespace {
26     double doubleInUnnamed{88.22}; // declare variable
27 }
28
29 int main() {
30     // output value doubleInUnnamed of unnamed namespace
31     std::cout << fmt::format("doubleInUnnamed = {}\n", doubleInUnnamed);
32

```

---

**Fig. 20.3** | Demonstrating the use of namespaces. (Part 1 of 2.)

```

33 // output global variable
34 std::cout << fmt::format("(global) integer1 = {}\n", integer1);
35
36 // output values of Example namespace
37 std::cout << fmt::format(
38     "pi = {}\ne = {}\ninteger1 = {}\nfiscal3 = {}\n", Example::pi,
39     Example::e, Example::integer1, Example::Inner::fiscal3);
40
41 Example::printValues(); // invoke printValues function
42 }
43
44 // display variable and constant values
45 void Example::printValues() {
46     std::cout << "\nIn printValues:\n";
47     std::cout << fmt::format(
48         "integer1 = {}\npi = {}\ne = {}\n", pi, e, integer1);
49     std::cout << fmt::format(
50         "doubleInUnnamed = {}\n(global) integer1 = {}\nfiscal3 = {}\n",
51         doubleInUnnamed, ::integer1, Inner::fiscal3);
52 }

```

```

doubleInUnnamed = 88.22
(global) integer1 = 98
pi = 3.14159
e = 2.71828
integer1 = 8
fiscal3 = 2022

In printValues:
integer1 = 8
pi = 3.14159
e = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
fiscal3 = 2022

```

**Fig. 20.3** | Demonstrating the use of namespaces. (Part 2 of 2.)

### 20.4.1 Defining namespaces

Err 

Lines 9–22 use the keyword `namespace` to define namespace `Example`, which like a class, has a body delimited by braces (`{}`). `Example`'s members consist of two constants (`pi` and `e` in lines 11–12), an `int` (`integer1` in line 13), a function (`printValues` in line 15) and a **nested namespace** (`Inner` in lines 18–21). `Example`'s `integer1` member has the same name as global variable `integer1` (line 6). Variables with the same name must have different scopes; otherwise, compilation errors occur. A namespace can contain constants, data, classes, nested namespaces, functions, etc. Definitions of namespaces must occupy the global scope or be nested within other namespaces. Unlike classes, different namespace members can be defined in separate namespace blocks. For example, each standard library header has a namespace block placing its contents in namespace `std`.

Lines 25–27 create an **unnamed namespace** containing `doubleInUnnamed`. Variables, classes and functions in an unnamed namespace are accessible only in the current transla-

tion unit. However, unlike variables, classes or functions with `static` linkage, those in the unnamed namespace may be used as template arguments. The unnamed namespace has an implicit `using` directive, so its members appear to occupy the **global namespace**. They are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace and are accessible in all scopes following the declaration in the file. Each separate translation unit has its own unique unnamed namespace.

### 20.4.2 Accessing namespace Members with Qualified Names

Line 31 outputs the value of variable `doubleInUnnamed`, which is directly accessible as part of the unnamed namespace. Line 34 outputs the value of global variable `integer1`. For both of these variables, the compiler first attempts to find a local declaration of the variables in `main`. There are no local declarations, so the compiler assumes those variables are in the global namespace.

Lines 37–39 output the values of `pi`, `e`, `integer1` and `fiscal3` from the `Example` namespace. Each must be qualified with `Example::` because the program does not provide any `using` directive or `using` declarations indicating that it will use `Example`'s members. In addition, member `integer1` must be qualified because a global variable has the same name. Otherwise, the global variable's value is output. `fiscal3` is a member of nested namespace `Inner`, so it must be qualified with `Example::Inner::`.

Function `printValues` (defined in lines 45–52) is in the `Example` namespace, so it can access other `Example` members directly without using a namespace qualifier. Line 47–51 output `integer1`, `pi`, `e`, `doubleInUnnamed`, global variable `integer1` and `fiscal3`. Notice that `pi` and `e` are not qualified with `Example`. Variable `doubleInUnnamed` is still accessible because it's in the unnamed namespace, and the variable name does not conflict with any other `Example` members. The global version of `integer1` must be qualified with the scope resolution operator (`::`) because its name conflicts with an `Example` member. Also, `fiscal3` must be qualified with `Inner::`. When accessing members of a nested namespace, the members must be qualified with the namespace name unless the member is used within the nested namespace. Placing `main` in a namespace is a compilation error.



### 20.4.3 using Directives Should Not Be Placed in Headers

namespaces are particularly useful in large-scale applications that use many class libraries. In such cases, there's a higher likelihood of naming conflicts. For such projects, there should never be a `using` directive in a header—this brings the corresponding names into any file that includes the header. This could result in name collisions and subtle, hard-to-find errors. Instead, use only fully qualified names in headers (for example, `std::cout` or `std::string`).

### 20.4.4 Aliases for namespace Names

namespaces can be **aliased**.<sup>9</sup> This might be useful when dealing with long namespace identifiers or nested namespaces. For example, assuming we have the namespace identifier `CPlusPlus20forProgrammers`, the statement

```
namespace CPP20FP = CPlusPlus20forProgrammers;
```

9. "Namespace aliases." Accessed March 11, 2022. [https://en.cppreference.com/w/cpp/language/namespace\\_alias](https://en.cppreference.com/w/cpp/language/namespace_alias).

## 20.12 Chapter 20 Other Topics and a Look Toward the Future of C++

creates the shorter **namespace alias** `CPP20FP` for `CP1usP1us20forProgrammers`. Similarly, you could define an alias for the nested namespace `Inner` in Fig. 20.3 as follows:

```
namespace Innermost = Example::Inner;
```

### 17 20.4.5 C++17 Nested Namespaces

As of C++17, a nested namespace, such as `Inner` in lines 18–21, may be defined separately from its enclosing namespace. For example, we could remove lines 18–21 and define the nested namespace `Inner` as follows:

```
namespace Example::Inner {  
    // define enumeration  
    enum Years {fiscal1 = 2020, fiscal2, fiscal3};  
}
```

The notation `Example::Inner` indicates that `Example` is the enclosing namespace and `Inner` is a nested namespace of `Example`.

## 20.5 Operator Keywords

The following table shows the **operator keywords** you can use in place of various C++ operators. You can use operator keywords if you have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.

Operator	Operator keyword	Description
<i>Logical operator keywords</i>		
<code>&amp;&amp;</code>	<code>and</code>	logical AND
<code>  </code>	<code>or</code>	logical OR
<code>!</code>	<code>not</code>	logical NOT
<i>Inequality operator keyword</i>		
<code>!=</code>	<code>not_eq</code>	inequality
<i>Bitwise operator keywords</i>		
<code>&amp;</code>	<code>bitand</code>	bitwise AND
<code> </code>	<code>bitor</code>	bitwise inclusive OR
<code>^</code>	<code>xor</code>	bitwise exclusive OR
<code>~</code>	<code>compl</code>	bitwise complement
<i>Bitwise assignment operator keywords</i>		
<code>&amp;=</code>	<code>and_eq</code>	bitwise AND assignment
<code> =</code>	<code>or_eq</code>	bitwise inclusive OR assignment
<code>^=</code>	<code>xor_eq</code>	bitwise exclusive OR assignment

Figure 20.4 demonstrates several of the operator keywords. The program declares and initializes two `bool` variables (lines 7–8). Lines 13–19 use the logical operator keywords to perform various logical operations with `bool` variables `a` and `b`.

```

1 // fig20_04.cpp
2 // Demonstrating operator keywords.
3 #include <fmt/format.h>
4 #include <iostream>
5
6 int main() {
7     bool a{true};
8     bool b{false};
9
10    std::cout << fmt::format("a = {}; b = {}\n\n", a, b);
11
12    std::cout << "Logical operator keywords:\n"
13              << fmt::format("  a and a: {}\n", a and a)
14              << fmt::format("  a and b: {}\n", a and b)
15              << fmt::format("  a or a: {}\n", a or a)
16              << fmt::format("  a or b: {}\n", a or b)
17              << fmt::format("  not a: {}\n", not a)
18              << fmt::format("  not b: {}\n", not b)
19              << fmt::format("a not_eq b: {}\n", a not_eq b);
20 }

```

```
a = true; b = false
```

```
Logical operator keywords:
```

```

  a and a: true
  a and b: false
  a or a: true
  a or b: true
  not a: false
  not b: true
a not_eq b: true

```

**Fig. 20.4** | Demonstrating operator keywords.

## 20.6 Pointers to Class Members (. \* and ->\*)

C++ provides the `.*` and `->*` operators for accessing class members via pointers. This capability is primarily for advanced C++ programmers. We provide only a mechanical example of using pointers to class members here. Figure 20.5 demonstrates the pointer-to-class-member operators. For answers to frequently asked questions about the `.*` and `->*` operators, see <https://isocpp.org/wiki/faq/pointers-to-members>.

```

1 // fig20_05.cpp
2 // Demonstrating operators .* and ->*.
3 #include <iostream>

```

**Fig. 20.5** | Demonstrating operators `.*` and `->*`. (Part 1 of 2.)

```

4
5 // class Test definition
6 class Test {
7 public:
8     Test(int x) : value{x} {};
9
10    void testFunction() {
11        std::cout << "testFunction called\n";
12    }
13
14    int value; // public data member
15 };
16
17 void arrowStar(Test* ptr); // prototype
18 void dotStar(Test* ptr); // prototype
19
20 int main() {
21     Test test{8};
22     arrowStar(&test); // pass address to arrowStar
23     dotStar(&test); // pass address to dotStar
24 }
25
26 // access member function of Test object using ->*
27 void arrowStar(Test* ptr) {
28     auto functionPtr{&Test::testFunction}; // pointer to a member function
29     (ptr->*functionPtr)(); // invoke function indirectly
30 }
31
32 // access members of Test object data member using .*
33 void dotStar(Test* ptr) {
34     auto valuePtr{&Test::value}; // pointer to a data member
35     std::cout << "value is " << (*ptr).*valuePtr << "\n"; // access value
36 }

```

```

testFunction called
value is 8

```

**Fig. 20.5** | Demonstrating operators .\* and ->\*. (Part 2 of 2.)

### Class Test



The program declares class `Test` (lines 6–15) containing the public member function `testFunction` and the public data member `value`. **The client code can create pointers to class members for only those class members that are accessible to the client code.** In this example, both member function `test` and data member `value` are publicly accessible.

### main Function

Lines 17–18 provide prototypes for the functions `arrowStar` (defined in lines 27–30) and `dotStar` (defined in lines 33–36), which demonstrate the `->*` and `.*` operators, respectively. Line 21 creates a `Test` object named `test` and initializes its `value` member to 8. Lines 22–23 call functions `arrowStar` and `dotStar` with a pointer to the `test` object (i.e., `&test`).

### arrowStar Function

Line 28 in function `arrowStar` declares and initializes variable `functionPtr` as a **pointer to a member function**. We declared the variable `auto` so the compiler can infer its type, which in this case is a pointer to a member function of class `Test` that takes no arguments and returns `void`. The type for a **pointer to a function** includes as part of its type both the function's **return type** and its **parameter list**. We initialize `functionPtr` with `&Test::testFunction`—that is, the address of class `Test`'s `testFunction` member. Note that the initializer uses the **address operator** (`&`) to get the member function's address. Line 29 invokes the member function stored in `functionPtr` using the `->*` operator. Because `functionPtr` is a pointer to a member of a class, the `->*` operator must be used rather than the `->` operator to invoke the function.

### dotStar Function

Line 34 in `dotStar` declares and initializes `valuePtr`. Again, we declared the variable `auto` so the compiler can infer its type—in this case, a pointer to an `int` data member (`value`) of class `Test`. Line 35 dereferences the pointer `ptr` then uses the `.*` operator to access the member to which `valuePtr` points.

## 20.7 Case Study: Payroll System Using Polymorphism and Runtime Type Information — Downcasting, `dynamic_cast`, `typeid` and `type_info`

Section 10.9 presented a simple payroll system using runtime polymorphism. We defined an abstract base class `Employee` with concrete derived classes `SalariedEmployee` and `CommissionEmployee`. When processing `Employee` objects polymorphically in that example, we did not need to worry about the “specifics.” We simply called each employee's `toString` and `earnings` member functions, and the correct functions were called based on the employee object's type.

For the current pay period, our fictitious company has decided to give a 10 percent base-salary increase to each `SalariedEmployee`. To adjust a `SalariedEmployee`'s base salary, we'll determine each `Employee`'s specific type at execution time, then act appropriately. Figure 20.6 uses **runtime type information (RTTI)** and **dynamic casting** to enable a program to determine an object's type at execution time. Before using this capability in your own programs, **be sure to read the C++ Core Guidelines' cautions about using RTTI at the end of this section.**




---

```

1 // fig20_06.cpp
2 // Demonstrating downcasting and runtime type information (RTTI).
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <typeinfo>
6 #include <vector>
7 #include "Employee.h"
8 #include "SalariedEmployee.h"
9 #include "CommissionEmployee.h"
10
```

---

**Fig. 20.6** | Demonstrating downcasting and runtime type information. (Part 1 of 2.)

```

11 int main() {
12     // create derived-class objects
13     SalariedEmployee salaried{"John Smith", 800.0};
14     CommissionEmployee commission{"Sue Jones", 10000.0, .06};
15
16     // create and initialize vector of base-class pointers
17     std::vector<Employee*> employees{&salaried, &commission};
18
19     // polymorphically process each element in vector employees
20     for (Employee* employeePtr : employees) {
21         std::cout << fmt::format("{}\n", employeePtr->toString());
22
23         // determine whether employeePtr points to a SalariedEmployee;
24         // if not, dynamic_cast returns nullptr which evaluates to false
25         if (auto ptr{dynamic_cast<SalariedEmployee*>(employeePtr)}) {
26             double oldBaseSalary = ptr->getSalary();
27             std::cout << fmt::format("old salary: {:.2f}\n", oldBaseSalary);
28             ptr->setSalary(1.10 * oldBaseSalary);
29             std::cout << fmt::format(
30                 "new salary with 10% increase is: {:.2f}\n",
31                 ptr->getSalary());
32         }
33
34         std::cout << fmt::format("earned {:.2f}\n\n",
35             employeePtr->earnings());
36     }
37
38     // display each object's type
39     for (const Employee* employeePtr : employees) {
40         std::cout << fmt::format("{}\n", typeid(*employeePtr).name());
41     }
42 }

```

```

name: John Smith
salary: $800.00
old salary: $800.00
new salary with 10% increase is: $880.00
earned $880.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

16SalariedEmployee
18CommissionEmployee

```


**Fig. 20.6** | Demonstrating downcasting and runtime type information. (Part 2 of 2.)


This program reuses classes `Employee`, `SalariedEmployee` and `CommissionEmployee` from Section 10.9. Lines 13–14 create one `SalariedEmployee` and one `CommissionEmployee` object. Line 17 uses the objects' addresses to initialize the vector `employees`. Lines 20–36 iterate through `employees` and display each employee's information by polymorphically invoking member function `toString` (line 21).



### Determining an Object's Type with `dynamic_cast`

As lines 20–36 encounter a `SalariedEmployee` object, we wish to increase its base salary by 10 percent. When processing `Employees` polymorphically, we cannot know in advance whether `employeePtr` will point to a `SalariedEmployee` or a `CommissionEmployee` at any given time. So how do we identify `SalariedEmployees` to increase their base salaries?

To accomplish this, we use operator `dynamic_cast` to determine whether the current `Employee`'s type is `SalariedEmployee`.<sup>10</sup> This operation is known as **downcasting**. The compiler will allow access to derived-class-only members from a base-class pointer aimed at a derived-class object if we explicitly cast the base-class pointer down to a derived-class pointer. Line 25 dynamically downcasts `employeePtr` from type `Employee*` to type `SalariedEmployee*`. If `employeePtr` points to an object that *is a SalariedEmployee* object, then `ptr` is initialized with that object's address; otherwise, `ptr` is initialized with the value `nullptr`. You also may use `dynamic_cast` with references. If the referenced object does not have the *is-a* relationship with the `dynamic_cast`'s specified type, the operator throws a **bad\_cast** exception. Using `auto` in line 25 enables the compiler to infer `ptr`'s type (`SalariedEmployee*`) from the `dynamic_cast` expression. 

We use `dynamic_cast` here to check the underlying object's type. A `static_cast` would simply convert the `Employee*` to a `SalariedEmployee*`, ignoring the object's actual type. If we were to do that, the program would attempt to increase every `Employee`'s salary, resulting in undefined behavior for each object that's not a `SalariedEmployee`. 

If the value returned by the `dynamic_cast` operator is not `nullptr`, the object is the correct type, and the `if` statement (lines 25–32) performs the special processing required for the `SalariedEmployee` object. Lines 26, 28 and 31 invoke `SalariedEmployee` functions `getSalary` and `setSalary` to retrieve and update the employee's base salary.

### Calculating the Current Employee's Earnings

Lines 34–35 invoke `earnings` on the current employee being processed. Recall that `earnings` is declared `virtual` in the base class, so the program polymorphically invokes each employee's `earnings` member function.

### Displaying an Employee's Type

Lines 39–41 display each employee's object type. Operator `typeid`<sup>11</sup> (line 40)—which requires the header `<typeinfo>` (line 5)—returns a reference to a `type_info`<sup>12</sup> object containing information about its operand's type, including the type's name. The `type_info` member function `name` returns an implementation-depending pointer-based string containing the `typeid` argument's type name. The sample output was produced with `g++`, which precedes each class's name with the number of characters in the class name (e.g., "16SalariedEmployee")—`clang++` produces the same output. Visual C++ produces:

```
class SalariedEmployee
class CommissionEmployee
```

10. "dynamic\_cast conversion." Accessed March 12, 2022. [https://en.cppreference.com/w/cpp/language/dynamic\\_cast](https://en.cppreference.com/w/cpp/language/dynamic_cast).

11. "typeid operator." Accessed March 12, 2022. <https://en.cppreference.com/w/cpp/language/typeid>.

12. "type\_info operator." Accessed March 12, 2022. [https://en.cppreference.com/w/cpp/types/type\\_info](https://en.cppreference.com/w/cpp/types/type_info).

### Compilation Errors That We Avoided By Using `dynamic_cast`

We avoid several compilation errors in this example by downcasting the `Employee` pointer to a `SalaryedEmployee` pointer (line 25). If lines 26, 28 and 31 used a base-class pointer from the current element of `employees` rather than a derived-class pointer to invoke derived-class-only functions `getSalary` and `setSalary`, we'd receive a compilation error at each of these lines. As you learned in Section 10.6.3, attempting to invoke derived-class-only functions through a base-class pointer is not allowed.

### RTTI Cautions

The C++ Core Guidelines indicate that `dynamic_cast` is overused and should be used only “where class hierarchy navigation is unavoidable.”<sup>13</sup> For more information on the primary uses of RTTI and its alternatives, see core guidelines C.146–C.148:

- C.146—“Use `dynamic_cast` where class hierarchy navigation is unavoidable”: [https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-dynamic\\_cast](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-dynamic_cast)
- C.147—“Use `dynamic_cast` to a reference type when failure to find the required class is considered an error”: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-ref-cast>
- C.148—“Use `dynamic_cast` to a pointer type when failure to find the required class is considered a valid alternative”: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-ptr-cast>

## 11 20.8 Inheriting Base-Class Constructors

Base-class constructors, destructors and overloaded assignment operators are not inherited by derived classes. However, derived-class constructors, destructors and overloaded assignment operators can call their base-class versions.

Sometimes a derived class's constructors simply pass the constructor arguments to a base-class constructor with the same parameters. In such cases, you can specify that a derived class should inherit its base class's constructors. To do so, include a `using` declaration of the following form anywhere in the derived-class definition:

```
using BaseClass::BaseClass;
```

We suggest placing this where you'd typically place the constructor's prototype. In the preceding declaration, `BaseClass` is the base class's name. The compiler generates a corresponding derived-class constructor for each base-class constructor that calls that base-class one. The generated constructors have the derived class's name and perform only default initialization for the derived class's additional data members, if any.

---

13. C++ Core Guidelines, “C.146: Use `dynamic_cast` where class hierarchy navigation is unavoidable.” Accessed March 12, 2022. [https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-dynamic\\_cast](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-dynamic_cast).

When you inherit constructors:<sup>14,15</sup>

- Each generated constructor has the same access specifier (`public`, `protected` or `private`) as its corresponding base-class constructor.
- If a constructor is deleted in the base class by placing `= delete` in its prototype, the corresponding derived-class constructor also is deleted.
- If the derived class does not explicitly define constructors, the compiler still generates a default constructor in the derived class.
- A given base-class constructor is not accessible if a constructor that you explicitly define in the derived class has the same parameter list.
- A base-class constructor’s default arguments are not inherited. Instead, the compiler generates overloaded constructors in the derived class. For example, if the base class declares the constructor

```
BaseClass(int x = 0, double y = 0.0);
```

the compiler generates the following derived-class constructors without default arguments

```
DerivedClass();
DerivedClass(int x);
DerivedClass(int x, double y);
```

These each call the *BaseClass* constructor that specifies the default arguments.

## 20.9 C++17 and C++20: `[[nodiscard]]` Attribute

17  
20

Some functions return values that you should not ignore. For example, Section 2.7 introduced the `string` member function `empty`. When you want to know whether a `string` is empty, you must not only call `empty` but also check its return value in a condition, such as:

```
if (s.empty()) {
    // do something because the string s is empty
}
```

In C++20, `string`’s `empty` function is declared with the `[[nodiscard]]` attribute,<sup>16</sup> so the compiler issues a warning if the caller does not use the return value. Since C++17, many library functions have been enhanced with `[[nodiscard]]` to help you write correct code.

20  
SE

### C++20’s `[[nodiscard("with reason")]]` Attribute

20

One problem with C++17’s `[[nodiscard]]` attribute is that it did not specify why you should not ignore a given function’s return value. So, in C++20, you can include a descriptive message that the compiler will display as part of the warning message, as in:

```
[[nodiscard("Insight goes here")]]
```

14. “Using-declaration.” Accessed March 10, 2022. [https://en.cppreference.com/w/cpp/language/using\\_declaration](https://en.cppreference.com/w/cpp/language/using_declaration)

15. C++ Standard, “9.9 The `using` declaration.” Accessed March 10, 2022. <https://timsong-cpp.github.io/cppwp/n4861/namespace.udcl>.

16. Section 9.12.8 of the “Working Draft, Standard for Programming Language C++.” Accessed May 10, 2020. <http://wg21.link/n4861>.

20 You may also use this attribute on your own function definitions. Figure 20.7 shows a cube function declared with C++20's `[[nodiscard]]` version. You place the attribute before the return type—typically on a line by itself for readability (line 4). Line 10 calls cube but does not use the returned value. The three outputs show the compiler warnings from Visual C++, clang++ and g++, respectively. These are just warnings, so the program still compiles and runs. The primary purpose of this example is to demonstrate the compiler warning messages produced by `[[nodiscard]]`. This program does not produce any output when it runs because it ignores function cube's return value when it should not.

```

1 // fig20_07.cpp
2 // [[nodiscard]] attribute.
3
4 [[nodiscard("Do not ignore! Otherwise, you won't know the cube of x")]]
5 int cube(int x) {
6     return x * x * x;
7 }
8
9 int main() {
10     cube(10); // generates a compiler warning
11 }
```

```
C:\Users\pauldeitel\examples\ch20\fig20_07.cpp(10,8): warning C4858: discarding return value: Do not ignore! Otherwise, you won't know the cube of x.
```

```
fig20_07.cpp:10:4: warning: ignoring return value of function declared with 'nodiscard' attribute: Do not ignore! Otherwise, you won't know the cube of x. [-Wunused-result]
  cube(10); // generates a compiler warning
  ^~~~~ ~
1 warning generated.
```

```
fig20_07.cpp: In function 'int main()':
fig20_07.cpp:10:8: warning: ignoring return value of 'int cube(int)', declared with attribute 'nodiscard': 'Do not ignore! Otherwise, you won't know the cube of x.' [-Wunused-result]
  10 |     cube(10); // generates a compiler warning
      |         ~~~~~^~~~~
fig20_07.cpp:5:5: note: declared here
   5 | int cube(int x) {
      |     ^~~~~
```

**Fig. 20.7** | `[[nodiscard]]` attribute.

## 20.10 shared\_ptr and weak\_ptr Smart Pointers

Sections 11.4–11.5 introduced dynamic memory allocation and smart pointers. You learned that many common bugs in C and C++ code are related to pointers and dynamically allocated memory and that smart pointers help you avoid such errors by providing additional functionality that strengthens memory allocation and deallocation. Smart pointers also help you write **exception-safe code** (introduced in Section 12.3). If a pro-

gram throws an exception before `delete` has been called on a pointer to dynamically allocated memory, a memory leak occurs. On the other hand, **smart pointers use resource acquisition is initialization (RAII)** (Section 11.5). A smart pointer is an object, so **when a smart pointer goes out of scope for any reason, including an exception, the smart pointer’s destructor will still be called, helping avoid memory leaks.** In addition to class template `unique_ptr` (discussed in Section 11.5), C++11 introduced other smart pointer options with additional functionality. ✖ Err 11

### 20.10.1 Reference Counted `shared_ptr`

C++11 `shared_ptr`s (header `<memory>`) hold an internal pointer to a resource (e.g., a dynamically allocated object) that may be shared throughout a program.<sup>17</sup> You can have many `shared_ptr`s to the same resource. As the type name implies, **`shared_ptr`s share a resource**—if you change the resource with one `shared_ptr`, the changes also will be “seen” by the other `shared_ptr`s. The internal pointer is deleted when the last `shared_ptr` to the resource is destroyed. Internally, `shared_ptr`s use **reference counting** to determine how many `shared_ptr`s point to the resource. Each time you create a new `shared_ptr` to a resource, the **reference count** increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted and the memory is released. 11

#### `shared_ptr`s vs. `unique_ptr`s

The C++ Core Guidelines indicate that you should prefer a `unique_ptr` over a `shared_ptr` if there will be only one owner of a given resource at a time.<sup>18</sup> ● CG

#### Customizing How Shared Resources Are Destroyed

`shared_ptr`s also allow you to determine how the resource will be destroyed. For most dynamically allocated objects, `delete` is used. However, some resources require more complex cleanup. In that case, you can supply a custom **deleter** function (a function pointer, lambda or function object) to the `shared_ptr` constructor. The deleter specifies how to destroy the resource. When the reference count reaches zero, and the resource is ready to be destroyed, the `shared_ptr` calls the custom deleter function.

#### Example Using `shared_ptr`

Fig. 20.8 defines a simple `Book` class (lines 12–17) with a `string` representing the `Book`’s title. `Book`’s destructor (line 15) displays a message indicating which `Book` object is being destroyed. We use this class to demonstrate the basic functionality of `shared_ptr`.

---

```

1 // fig20_08.cpp
2 // Demonstrate shared_ptr.
3 #include <algorithm>
```

---

**Fig. 20.8** | `shared_ptr` example program. (Part 1 of 3.)

17. “`std::shared_ptr`.” Accessed March 12, 2022. [https://en.cppreference.com/w/cpp/memory/shared\\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr).

18. C++ Core Guidelines, “F.27: Use a `shared_ptr<T>` to share ownership.” Accessed March 18, 2022. [https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-shared\\_ptr](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-shared_ptr).

```
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <memory>
7 #include <string>
8 #include <string_view>
9 #include <vector>
10
11 // class Book
12 class Book {
13 public:
14     explicit Book(std::string_view bookTitle) : title{bookTitle} {}
15     ~Book() {std::cout << fmt::format("Destroying Book: {}\n", title);}
16     std::string title; // title of the Book
17 };
18
19 // a custom delete function for a pointer to a Book
20 void deleteBook(Book* book) {
21     std::cout << "Custom deleter for a Book, ";
22     delete book; // delete the Book pointer
23 }
24
25 // compare the titles of two Books for sorting
26 bool compareTitles(
27     std::shared_ptr<Book> ptr1, std::shared_ptr<Book> ptr2) {
28     return (ptr1->title < ptr2->title);
29 }
30
31 int main() {
32     // create a shared_ptr to a Book and display the reference count
33     std::shared_ptr<Book> bookPtr{
34         std::make_shared<Book>("C++ How to Program")};
35     std::cout << fmt::format("Reference count for Book {} is: {}\n",
36         bookPtr->title, bookPtr.use_count());
37
38     // create another shared_ptr to the Book and display reference count
39     std::shared_ptr<Book> bookPtr2{bookPtr};
40     std::cout << fmt::format("Reference count for Book {} is: {}\n",
41         bookPtr->title, bookPtr.use_count());
42
43     // change the Book's title and access it from both pointers
44     bookPtr2->title = "Java How to Program";
45     std::cout << fmt::format(
46         "Updated Book title:\nbookPtr: {}\nbookPtr2: {}\n",
47         bookPtr->title, bookPtr2->title);
48
49     // create a std::vector of shared_ptrs to Books (BookPtrs)
50     std::vector<std::shared_ptr<Book>> books{
51         std::make_shared<Book>("C How to Program"),
52         std::make_shared<Book>("Intro to Python"),
53         std::make_shared<Book>("C# How to Program"),
54         std::make_shared<Book>("C++ How to Program")};
```

**Fig. 20.8** | shared\_ptr example program. (Part 2 of 3.)

```

55
56 // print the Books in the vector
57 std::cout << "\nBooks before sorting:\n";
58 for (auto book : books) {
59     std::cout << book->title << "\n";
60 }
61
62 // sort the vector by Book title and print the sorted vector
63 std::sort(books.begin(), books.end(), compareTitles);
64 std::cout << "\nBooks after sorting:\n";
65 for (auto book : books) {
66     std::cout << book->title << "\n";
67 }
68
69 // create a shared_ptr with a custom deleter
70 std::cout << "\nshared_ptr with a custom deleter.\n";
71 std::shared_ptr<Book> bookPtr3{
72     new Book("Android How to Program"), deleteBook};
73 bookPtr3.reset(); // release the Book this shared_ptr manages
74
75 // shared_ptrs are going out of scope
76 std::cout << "\nEnd of main: shared_ptr objects going out of scope.\n";
77 }

```

```

Reference count for Book C++ How to Program is: 1
Reference count for Book C++ How to Program is: 2
Updated Book title:
bookPtr: Java How to Program
bookPtr2: Java How to Program

Books before sorting:
C How to Program
Intro to Python
C# How to Program
C++ How to Program

Books after sorting:
C How to Program
C# How to Program
C++ How to Program
Intro to Python

shared_ptr with a custom deleter.
Custom deleter for a Book, Destroying Book: Android How to Program

End of main: shared_ptr objects going out of scope.
Destroying Book: C How to Program
Destroying Book: C# How to Program
Destroying Book: C++ How to Program
Destroying Book: Intro to Python
Destroying Book: Java How to Program

```

**Fig. 20.8** | shared\_ptr example program. (Part 3 of 3.)

### Creating shared\_ptrs

- 11 The program uses C++11 `shared_ptr`s to manage several `Book` instances. Lines 33–34 use standard library function `make_shared` to create a `shared_ptr` to a dynamically allocated `Book` object with the title "C++ How to Program".<sup>19</sup> This is the recommended way to create a `shared_ptr`.<sup>20</sup> You can further simplify this statement by declaring `bookPtr` with `auto`. The `shared_ptr` manages the `Book` object and initially sets its reference count to 1—that is, one `shared_ptr` currently refers to the dynamically allocated object. Class `shared_ptr` provides various constructors—if a new `shared_ptr` is initialized with an existing one, they both share ownership of the resource and the reference count increases by 1. If you make multiple `shared_ptr`s with the same pointer, the `shared_ptr`s won't acknowledge each other, and the reference count will be wrong. When the `shared_ptr`s are destroyed, they decrease the reference count by one. The `shared_ptr` destructor call that decrements the shared resource's reference count to 0 also deletes the resource.

### Manipulating shared\_ptrs

Lines 35–36 display the `Book`'s `title` and the number of `shared_ptr`s referencing that object—that is, the current reference count. We use the `->` operator to access the `Book`'s data member `title`—`shared_ptr`s overload the pointer operators `->` and `*` so you can use them like raw pointers. The `shared_ptr` member function `use_count` returns the reference count.

Line 39 creates another `shared_ptr` to the same `Book` using the `shared_ptr` constructor that receives a `shared_ptr` argument—this increments the `Book`'s reference count by one. You also can use `shared_ptr`'s assignment operator (`=`) to create a `shared_ptr` to the same resource. Lines 40–41 display the `Book`'s `title` and reference count again to show that the reference count increased by one when we created the second `shared_ptr`.

As mentioned earlier, changes made to the resource of a `shared_ptr` are “seen” by all `shared_ptr`s to that resource. Line 44 uses `bookPtr2` to change the `Book`'s `title`, then lines 45–47 display the `Book`'s `title` using both `bookPtr` and `bookPtr2` to show that `bookPtr` “sees” the change.

### Manipulating shared\_ptrs in an STL Container

Next, we demonstrate using `shared_ptr`s in an STL container. Lines 50–54 create a vector of `shared_ptr<Book>` objects and add four elements, using `make_shared` to dynamically allocate each `Book` and return a `shared_ptr` to it. Lines 58–60 display the vector's initial contents. Then, line 63 sorts the `Books` by title, using the function `compareTitles` (lines 26–29) to customize how the sort algorithm compares `Book` objects—in this case, by comparing the `Books`' `titles`. Lines 65–67 display the vector's sorted contents.

19. “`std::make_shared`, `std::make_shared_for_overwrite`.” Accessed March 12, 2022. [https://en.cppreference.com/w/cpp/memory/shared\\_ptr/make\\_shared](https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared).

20. C++ Core Guidelines, “R.22: Use `make_shared()` to make `shared_ptr`s.” Accessed March 12, 2022. [https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-make\\_shared](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-make_shared).



### `shared_ptr` Custom Deleter

Lines 71–72 create a `shared_ptr` with a custom deleter function `deleteBook` (lines 20–23), which is passed to the `shared_ptr` constructor as the second argument. A custom deleter function must take one argument of the `shared_ptr`'s internal pointer type. When the last `shared_ptr` for a given `Book` object is destroyed, the `shared_ptr` passes its internal `Book*` to the custom deleter, which is responsible for deleting the managed resource. In this example, `deleteBook` displays a message showing that the custom deleter was called, then deletes the dynamically allocated `Book` object.

One use for custom deleters is when using third-party C libraries. Rather than providing a class with a constructor and destructor as a C++ library would, C libraries frequently provide one function that returns a pointer to a `struct` representing a resource and another that does the necessary cleanup when the resource is no longer needed. Using a custom deleter allows you to use a `shared_ptr` to keep track of the resource and still ensure that it's freed correctly.

### Resetting a `shared_ptr`

Line 73 calls the `shared_ptr` member function `reset`, which sets the `shared_ptr` to `nullptr`, so it no longer points to a shared resource. Since `bookPtr3` was the only `shared_ptr` to the `Book` object created in lines 71–72, the `shared_ptr` calls its custom deleter to release the resource. Other versions of `reset` allow you to pass a new resource to manage.

### `shared_ptr`s Are Destroyed When They Go Out of Scope

When `main` terminates, all the `shared_ptr`s and the `vector` in this example go out of scope and are destroyed. Before the `vector` is destroyed, its `shared_ptr` elements are destroyed. The output shows that each `Book` object is destroyed automatically by the `shared_ptr`s.

## 20.10.2 `weak_ptr`: `shared_ptr` Observer

11

A `weak_ptr` points to the resource managed by a `shared_ptr` without assuming any responsibility for it.<sup>21</sup> A `shared_ptr`'s reference count does not increase when a `weak_ptr` references the shared resource. Thus, a **`shared_ptr`'s resource can be deleted even if there are still `weak_ptr`s pointing to it. When the last `shared_ptr` to a resource is destroyed, any remaining `weak_ptr`s are set to `nullptr`.** The C++ Core Guidelines say to use `weak_ptr`s “to break cycles of `shared_ptr`s.”<sup>22</sup> As we'll demonstrate later in this section, this helps avoid memory leaks caused by circular references.



A `weak_ptr` cannot directly access the resource it points to—you must create a `shared_ptr` from the `weak_ptr` to access the resource. This enables a program to determine whether the resource still exists. There are two ways to create a `shared_ptr` from a `weak_ptr`:

21. “`std::weak_ptr`.” Accessed March 12, 2022. [https://en.cppreference.com/w/cpp/memory/weak\\_ptr](https://en.cppreference.com/w/cpp/memory/weak_ptr).

22. C++ Core Guidelines, “R.24: Use `std::weak_ptr` to break cycles of `shared_ptr`s.” Accessed March 12, 2022. [https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-weak\\_ptr](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-weak_ptr).

Err 

- You can pass the `weak_ptr` to the `shared_ptr` constructor, which creates a `shared_ptr` to the resource being pointed to by the `weak_ptr` and properly increases the reference count. If the resource has already been deleted, the `shared_ptr` constructor will throw a `bad_weak_ptr` exception.
- You can also call the `weak_ptr` member function `lock`, which returns a `shared_ptr` to the `weak_ptr`'s resource.<sup>23</sup> If the `weak_ptr` does not currently point to a resource, `lock` returns an empty `shared_ptr` (i.e., a `shared_ptr` with its internal pointer set to `nullptr`). You should use `lock` if an empty `shared_ptr` isn't considered an error in the context of your application.

Err 

One typical use-case of `weak_ptr`s is in **circularly referential data**—a situation in which two objects refer to each other internally, as the example in Figs. 20.9–20.12 demonstrates. Another use-case is avoiding **dangling pointers**<sup>24</sup>—that is, pointers to dynamically allocated objects that are subsequently deleted. Using such a pointer is undefined behavior that often crashes a program. **Because a `weak_ptr` cannot directly access the resource it points to, the resource can be deleted.** If the program subsequently attempts to use the `weak_ptr`'s `lock` function to get a `shared_ptr`, `lock` returns `nullptr` to indicate that the resource no longer exists.

### Class Definitions for Classes `Author` and `Book`

We'll demonstrate `weak_ptr`s using classes `Author` and `Book` (Figs. 20.9–20.12). Each class contains pointers to an object of the other class. We also use `public` data members here only to simplify the code. This creates a **circular type reference** between the class definitions:

- class `Author` defines a `weak_ptr` and a `shared_ptr` to a `Book` (lines 20–21 in Fig. 20.9), and
- class `Book` defines a `weak_ptr` and a `shared_ptr` to an `Author` (lines 20–21 in Fig. 20.10).

To declare these pointers, each class needs to know that the other exists, which you usually accomplish by including the corresponding class's header. However, we cannot do that here because class `Author` depends on class `Book`, which, in turn, depends on class `Author`. Including `Book.h` in `Author.h` and including `Author.h` in `Book.h` results in compilation errors. Interestingly, declaring a pointer does not require a complete type definition—it simply requires knowing that the type exists. This is the purpose of the **forward class declarations** in line 8 of Figs. 20.9 and 20.10. Once you need to use a pointer to interact with an object, then you need the full class definition, as you'll see in the `.cpp` files for classes `Author` (Fig. 20.11) and `Book` (Fig. 20.12).

23. “`std::weak_ptr::lock`.” Accessed March 12, 2022. [https://en.cppreference.com/w/cpp/memory/weak\\_ptr/lock](https://en.cppreference.com/w/cpp/memory/weak_ptr/lock).

24. “When is `std::weak_ptr` useful?” Accessed March 17, 2022. <https://stackoverflow.com/questions/12030650/when-is-stdweak-ptr-useful>.

---

```
1 // Fig. 20.9: Author.h
2 // Author class definition.
3 #pragma once
4 #include <memory>
5 #include <string>
6 #include <string_view>
7
8 class Book; // forward declaration of class Book
9
10 // Author class definition
11 class Author {
12 public:
13     explicit Author(std::string_view authorName);
14     ~Author();
15
16     // print the title of the Book this Author wrote
17     void printBookTitle();
18
19     std::string name; // name of the Author
20     std::weak_ptr<Book> weakBookPtr; // Book the Author wrote
21     std::shared_ptr<Book> sharedBookPtr; // Book the Author wrote
22 };
```

---

**Fig. 20.9** | Author class definition.


---

```
1 // Fig. 20.10: Book.h
2 // Book class definition.
3 #pragma once
4 #include <memory>
5 #include <string>
6 #include <string_view>
7
8 class Author; // forward declaration of class Author
9
10 // Book class definition
11 class Book {
12 public:
13     explicit Book(std::string_view bookTitle);
14     ~Book();
15
16     // print the name of this Book's Author
17     void printAuthorName();
18
19     std::string title; // title of the Book
20     std::weak_ptr<Author> weakAuthorPtr; // Author of the Book
21     std::shared_ptr<Author> sharedAuthorPtr; // Author of the Book
22 };
```

---

**Fig. 20.10** | Book class definition.

Using classes Author and Book, we'll show that setting an Author object's shared\_ptr to point to a Book object and setting that Book object's shared\_ptr to point back to the

Err  Author object creates a memory leak. Then, we'll show how we can use the `weak_ptrs` to fix this problem.

### Member Function Definitions for Classes Author and Book

Figures 20.11 and 20.12 define Author's and Book's member functions, respectively. For an Author object's member functions to manipulate a corresponding Book object through a pointer, class Author must know class Book's complete definition. So line 9 in Fig. 20.11 includes `Book.h`. Similarly, for a Book object's member functions to manipulate a corresponding Author object through a pointer, class Book must know class Author's complete definition. So line 8 in Fig. 20.12 includes `Author.h`.

---

```

1 // Fig. 20.11: Author.cpp
2 // Author member-function definitions.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <memory>
6 #include <string>
7 #include <string_view>
8 #include "Author.h"
9 #include "Book.h"
10
11 Author::Author(std::string_view authorName) : name(authorName) {}
12
13 Author::~Author() {
14     std::cout << fmt::format("Destroying Author: {}\n", name);
15 }
16
17 // print the title of the Book this Author wrote
18 void Author::printBookTitle() {
19     // if weakBookPtr.lock() returns a non-empty shared_ptr
20     if (std::shared_ptr<Book> bookPtr{weakBookPtr.lock()}) {
21         // show the reference count increase and print the Book's title
22         std::cout << fmt::format("Reference count for Book {} is {}\n",
23             bookPtr->title, bookPtr.use_count());
24         std::cout << fmt::format("Author {} wrote the book {}\n",
25             name, bookPtr->title);
26     }
27     else { // weakBookPtr points to NULL
28         std::cout << "This Author has no Books.\n";
29     }
30 }

```

---

**Fig. 20.11** | Author member-function definitions.

---

```

1 // Fig. 20.12: Book.cpp
2 // Book member-function definitions.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <memory>

```

---

**Fig. 20.12** | Book member-function definitions. (Part I of 2.)

---

```

6 #include <string>
7 #include <string_view>
8 #include "Author.h"
9 #include "Book.h"
10
11 Book::Book(std::string_view bookTitle) : title(bookTitle) {}
12
13 Book::~Book() {
14     std::cout << fmt::format("Destroying Book: {}\n", title);
15 }
16
17 // print the name of this Book's Author
18 void Book::printAuthorName() {
19     // if weakAuthorPtr.lock() returns a non-empty shared_ptr
20     if (std::shared_ptr<Author> authorPtr{weakAuthorPtr.lock()}) {
21         // show the reference count increase and print the Author's name
22         std::cout << fmt::format("Reference count for Author {} is {}\n",
23             authorPtr->name, authorPtr.use_count());
24         std::cout << fmt::format("The book {} was written by {}\n",
25             title, authorPtr->name);
26     }
27     else { // weakAuthorPtr points to NULL
28         std::cout << "This Book has no Author.\n";
29     }
30 }

```


---

**Fig. 20.12** | Book member-function definitions. (Part 2 of 2.)

Both classes define destructors that display a message to indicate when an object of either class is destroyed (Figs. 20.11 and 20.12, lines 13–15). Class `Author`'s `printBookTitle` function (Fig. 20.11, lines 18–30) displays its `Book`'s `title` and reference count (lines 22–23), then shows that the `Author` wrote that `Book` (lines 24–25). Similarly, class `Book`'s `printAuthorName` function (Fig. 20.12, lines 18–30) displays its `Author`'s name and reference count (lines 22–23), then shows that the `Book` was written by that `Author` (lines 24–25).

Recall that you can't access a resource directly through a `weak_ptr`. For `printBookTitle` and `printAuthorName` to perform the tasks in lines 22–25 of each, these functions must first create a `shared_ptr` from the `weak_ptr` data member by calling the `weak_ptr`'s `lock` function (line 20 in each figure). If the `weak_ptr` does not reference a resource, the `lock` function returns a `shared_ptr` containing `nullptr`, which evaluates to `false` in line 20. Otherwise, the new `shared_ptr` contains a valid pointer to the `weak_ptr`'s resource, the condition in line 20 evaluates to `true`, and lines 22–25 of each figure can access the corresponding resource. Lines 22–23 of each function display the `shared_ptr`'s reference count to show that creating the new `shared_ptr` **increased the reference count for the corresponding `Book` (Fig. 20.11) or `Author` (Fig. 20.12)**. The new `shared_ptr` created in line 20 of each figure is a local object, so it's destroyed when its enclosing block ends. At that point, the corresponding reference count decreases by one.

**main Function**

Err  Figure 20.13 demonstrates the memory leak caused by the circular reference between Author and Book. Lines 11–14 create shared\_ptrs to an object of each class. Line 17 sets the Books’s weak\_ptr data member to point to the Author, and line 18 sets the Author’s weak\_ptr data member to point to the Book. Similarly, lines 21–22 set the shared\_ptr data members of each object. The Author and Book objects now reference each other.

---

```

1 // fig20_13.cpp
2 // Demonstrate use of weak_ptr.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <memory>
6 #include "Author.h"
7 #include "Book.h"
8
9 int main() {
10 // create a Book and an Author
11 std::shared_ptr<Book> bookPtr(
12     std::make_shared<Book>("C++ How to Program"));
13 std::shared_ptr<Author> authorPtr(
14     std::make_shared<Author>("Deitel & Deitel"));
15
16 // reference the Book and Author to each other
17 bookPtr->weakAuthorPtr = authorPtr;
18 authorPtr->weakBookPtr = bookPtr;
19
20 // set the shared_ptr data members to create the memory leak
21 bookPtr->sharedAuthorPtr = authorPtr;
22 authorPtr->sharedBookPtr = bookPtr;
23
24 // reference count for bookPtr and authorPtr is two
25 std::cout << fmt::format("Reference count for Book {} is {}\n",
26     bookPtr->title, bookPtr.use_count());
27 std::cout << fmt::format("Reference count for Author {} is {}\n\n",
28     authorPtr->name, authorPtr.use_count());
29
30 // access the cross references to print the data they point to
31 std::cout << "Access Author name and Book title via weak_ptrs:\n";
32 bookPtr->printAuthorName();
33 std::cout << "\n";
34 authorPtr->printBookTitle();
35
36 // reference count for each shared_ptr is two
37 std::cout << fmt::format("Reference count for Book {} is {}\n",
38     bookPtr->title, bookPtr.use_count());
39 std::cout << fmt::format("Reference count for Author {} is {}\n\n",
40     authorPtr->name, authorPtr.use_count());
41
42 // the shared_ptrs go out of scope, the Book and Author are destroyed
43 std::cout << "End of main. shared_ptrs going out of scope.\n";
44 }

```

---

**Fig. 20.13** | shared\_ptrs cause a memory leak in circularly referential data. (Part 1 of 2.)

```

Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

Access Author name and Book title via weak_ptrs:
Reference count for Author Deitel & Deitel is 3
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 3
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

End of main. shared_ptrs going out of scope.

```

**Fig. 20.13** | `shared_ptrs` cause a memory leak in circularly referential data. (Part 2 of 2.)

Next, lines 25–28 display the reference counts for the `shared_ptrs` to show that each object is referenced by two `shared_ptrs`:

- the `Book` object is referenced by the `shared_ptr` in lines 11–12 and by the one in the `Author` object (line 22), and
- the `Author` object is referenced by the `shared_ptr` in lines 13–14 and by the one in the `Book` object (line 21).

**Note that the `weak_ptrs` don't affect the reference counts.** Line 32 calls the `Book` object's `printAuthorName` function to display the information stored in the `Book`'s `weak_ptr` data member. Line 34 calls the `Author` object's `printBookTitle` function to display the information stored in the `Author`'s `weak_ptr` data member. Each of these functions also displays the fact that another `shared_ptr` was created during the function call increasing the `Book`'s and `Author`'s reference counts to 3. Finally, lines 37–40 display the `Book`'s and `Author`'s reference counts again to show that the additional `shared_ptrs` created in the `printAuthorName` and `printBookTitle` were destroyed when the functions finished executing, decreasing each object's reference count to 2. Then, `main` terminates.

### Memory Leak

At the end of `main`, the `shared_ptrs` to the instances of `Author` and `Book` we created go out of scope and are destroyed, but **notice that the output doesn't show the destructors for classes `Author` and `Book`**. This program has a **memory leak**—the `Author` and `Book` objects aren't destroyed because of their `shared_ptr` data members. When `bookPtr` is destroyed at the end of the `main` function, the reference count for the object of class `Book` becomes one—the object of `Author` still has a `shared_ptr` to the `Book` object, so it isn't deleted. When `authorPtr` goes out of scope and is destroyed, the reference count for the object of class `Author` also becomes one—the `Book` object still has a `shared_ptr` to the `Author` object. Neither object is deleted because the reference count for each is still one.



### Fixing the Memory Leak

Now, comment out lines 21–22 in `main` by placing `//` at the beginning of each line. This prevents the code from setting the `shared_ptr` data members for classes `Author` and `Book`. Recompile the code and run the program again to produce the following output. We

bolded the last two lines showing the Author and Book objects were properly destroyed.

```
Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

Access Author name and Book title via weak_ptrs:
Reference count for Author Deitel & Deitel is 2
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 2
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

End of main. shared_ptrs going out of scope.
Destroying Author: Deitel & Deitel
Destroying Book: C++ How to Program
```

Notice that the initial reference count for each object is now 1 instead of 2 because we don't set the `shared_ptr` data members. The last two lines of the output show that the Author and Book objects were destroyed at the end of `main`. We eliminated the memory leak by using only the `weak_ptr` data members, which don't affect the reference count but still allow us to access the resource when we need it by creating a temporary `shared_ptr` to the resource. When the `shared_ptrs` we created in `main` are destroyed, **their reference counts become 0**, and the instances of classes Author and Book are deleted properly.

## 11 20.11 Trailing Return Types for Functions

Section 14.3 introduced lambda expressions. As you saw, lambdas that return values are declared with trailing return types in which the return type is specified to the right of the parameter list. Trailing-return-type syntax also can be used with functions and member functions. To specify a trailing return type, place the keyword `auto` before the function name, then follow the function's parameter list with `->` and the return type. For example, to specify a trailing return type for a function template `maximum` that finds the largest of three values of the same type, `T`, you'd write

```
template <typename T>
auto maximum(T x, T y, T z) -> T {
    // statements
}
```

As you build more complex function templates, there are cases for which only trailing return types are allowed. Such complex function templates are beyond this book's scope. For more information on trailing return types, see the C++ Standard Section 9.3.3.5, "Functions" at

<https://timsong-cpp.github.io/cppwp/n4861/dcl.fct>



## 20.12 decltype

C++11's `decltype` operator enables the compiler to determine an expression's type at compile time.<sup>25</sup> If the expression is a function call, `decltype` determines the function's return type, which for a function template often changes based on the type(s) used to specialize the template. 11

The `decltype` operator is particularly useful when working with complex template types for which it's often difficult to provide, or even determine, the proper type declaration. Rather than writing a complex type declaration, for example, representing the return type of a function, you can place in the parentheses of `decltype` an expression that returns the complex type and let the compiler "figure it out." Determining types at compile time with `decltype` is used frequently in C++ standard library class templates and in template metaprogramming.

The format of a `decltype` expression is

```
decltype(expression)
```

The *expression* is not evaluated. `decltype` is commonly used with trailing return types in function definitions that return complex types. C++14 added `decltype(auto)` to infer a function's return type as in: 14

```
decltype(auto) functionName(parameters)
```

For more information on `decltype`, see the C++ Standard Section 9.2.8.4, "Decltype specifiers" at

<https://timsong-cpp.github.io/cppwp/n4861/decl.type.decltype>

## 20.13 initializer\_list Class Template

Previously, you learned how to use C++11's braced initialization capabilities to initialize variables and containers. You can also define functions and constructors that receive braced initializers as arguments. To do so, you specify an `initializer_list` parameter. Figure 20.14 defines a function template `sum` (lines 9–19) that receives an `initializer_list` (line 10) and sums its elements. An initializer list can be used with the range-based for statement (lines 14–16) to iterate through all items in the `initializer_list`. In `main`, lines 24, 28 and 34 demonstrate passing a braced initializer to `sum`'s `initializer_list` parameter.

---

```

1 // fig20_14.cpp
2 // Summing the elements of a braced initializer
3 #include <fmt/format.h>
4 #include <initializer_list>
5 #include <iostream>
6 #include <string>
7
```

---

**Fig. 20.14** | Summing the elements of a braced initializer.

25. "decltype specifier." Accessed March 12, 2022. <https://en.cppreference.com/w/cpp/language/decltype>.

```

8 // sum the elements of an initializer_list
9 template <typename T>
10 T sum(std::initializer_list<T> list) {
11     T total{}; // value initialize total based on type T
12
13     // sum all the elements in list; requires += operator for type T
14     for (auto item : list) {
15         total += item;
16     }
17
18     return total;
19 }
20
21 int main() {
22     // display the sum of four ints contained in a braced initializer
23     std::cout << fmt::format("The sum of {} is: {}\n",
24         "{1, 2, 3, 4}", sum({1, 2, 3, 4}));
25
26     // display the sum of three doubles contained in a braced initializer
27     std::cout << fmt::format("The sum of {} is: {}\n",
28         "{1.1, 2.2, 3.3}", sum({1.1, 2.2, 3.3}));
29
30     // display the sum of two strings contained in a braced initializer
31     std::string s1{"Happy "};
32     std::string s2{"birthday!"};
33     std::cout << fmt::format("The sum of {} is: {}\n",
34         "{\n\"Happy \", \"birthday!\n\"", sum({s1, s2}));
35 }

```


```

The sum of {1, 2, 3, 4} is: 10
The sum of {1.1, 2.2, 3.3} is: 6.6
The sum of {"Happy ", "birthday!"} is: Happy birthday!

```

**Fig. 20.14** | Summing the elements of a braced initializer.

## 20 20.14 C++20: `[[likely]]` and `[[unlikely]]` Attributes

**Perf**  Today's compilers use sophisticated optimization techniques<sup>26</sup> to tune your code's performance. C++20 introduces the attributes `[[likely]]` and `[[unlikely]]` that enable you to provide additional hints to help compilers optimize `if`, `if...else` and `switch` statements code for better performance.<sup>27</sup> These attributes indicate paths of execution that are likely or unlikely to be taken. Many of today's compilers already provide mechanisms like this, so `[[likely]]` and `[[unlikely]]` standardize these features across compilers.<sup>28</sup>

26. "Optimizing Compiler." Accessed February 6, 2021. [https://en.wikipedia.org/wiki/Optimizing\\_compiler#Specific\\_techniques](https://en.wikipedia.org/wiki/Optimizing_compiler#Specific_techniques).

27. We searched for insights as to why and how you'd use this feature to produce better optimized code. There is little information at this point other than the proposal document "Attributes for Likely and Unlikely Statements (Revision 2)" (<https://wg21.link/p0479r2>). Section 3, "Motivation and Scope," suggests who should use these features and what they should be used for.

To use these, place `[[likely]]` or `[[unlikely]]` before an `if`'s or `else`'s body:

```
if (condition) [[likely]] {
    // statements
}
else {
    // statements
}
```

or in a `switch`'s case label:

```
switch (controllingExpression) {
    case 7:
        // statements
        break;
    case 11: [[likely]]
        // statements
        break;
    default:
        // statements
        break;
}
```

There are subtle issues when using these attributes. Using too many `[[likely]]` and `[[unlikely]]` attributes in your code could actually reduce performance.<sup>29</sup> The document that proposed adding these to the language says for each, “This attribute is intended for specialized optimizations which are implementation-specific. General usage of this attribute is discouraged.”<sup>30</sup> For other subtleties, see the proposal document at:



<https://wg21.link/p0479r2>

If you're working on systems with strict performance requirements, you may want to investigate these attributes further.

## 20.15 A Look Toward C++23

23

We presented examples of and, in some cases, simply mentioned several possible features for C++23 and beyond throughout this book, including the `std::mdarray` container (Section 6.13), contracts (Section 12.13), ranges enhancements (Section 14.10), a modularized standard library (Section 16.12), concurrent data structures (Section 17.15), parallel ranges algorithms (Section 17.15) and executors (Sections 18.5 and 18.8).

28. Herb Sutter. “Trip Report: Winter ISO C Standards Meeting (Jacksonville).” Sutter's Mill, April 3, 2018. <https://herbsutter.com/2018/04/>. Herb Sutter is the Convener of the ISO C++ committee and a Software Architect at Microsoft.

29. Section 9.12.6, “Working Draft, Standard for Programming Language C++,” ISO/IEC, April 3, 2020, <https://github.com/cplusplus/draft/releases/download/n4861/n4861.pdf>.

30. “Attributes for Likely and Unlikely Statements (Revision 2),” <https://wg21.link/p0479r2>, Section VIII, “Technical Specifications.”

In this section, we overview other features that are being considered for C++23.<sup>31,32</sup> The links we provide to C++ Standards Committee papers in this section are forwarding links that will bring you to the most up-to-date version of each paper. The following `deitel.com` link will forward you to the C++ Standard Committee GitHub site’s list of C++23 features currently in progress:

<https://deitel.com/CP1usP1us230penIssues>

You can track the latest C++23 standard draft efforts at:

<https://eel.is/c++draft/>

and

<https://github.com/cplusplus/draft/>

Also, check out the C++23 Wikipedia page, which lists many smaller tweaks to the language and libraries:

<https://en.wikipedia.org/wiki/C%2B%2B23>

## 23 Features Likely to Be Included in C++23

- **Asynchronous execution capabilities:** Since *C++20 for Programmers* was sent to publication, the C++ Standards Committee has released an extensive updated proposal for asynchronous-execution capabilities—“P2300R4: `std::execution`” (January 18, 2022). The proposed new features include three kinds of components—**schedulers**, **senders**, and **receivers**—and **customizable asynchronous algorithms**. The concept of an executor has been replaced by **schedulers** and **senders**. For the current version of this paper, visit <https://wg21.link/P2300>. P2300R4 was the most recent version of the paper at the time of this writing.
- **C++23 Ranges:** Since *C++20 for Programmers* was sent to publication, an updated version of the paper “P2214: A Plan for C++23 Ranges” was released (February 18, 2022). For the current version of this paper, visit <https://wg21.link/P2214>. P2214R2 was the most recent version of the paper at the time of this writing.
- **`std::generator` and `std::lazy`:** In Section 18.4, we created a generator coroutine using Sy Brand’s **generator** library. The papers “`std::generator: Synchronous Coroutine Generator for Ranges`” and “Add lazy coroutine (coroutine task) type” propose C++ standard library capabilities for generator functions and generalized lazy-evaluation capabilities. For the current versions of these papers, visit <https://wg21.link/P2168> and <https://wg21.link/p1056>, respectively.
- **Formatted output:** We used C++20 text formatting throughout the book, mostly via the open-source library `{fmt}` because our preferred compilers did not yet

20

31. Bryce Adelstein Lelbach, Fabio Fracassi, Ben Craig, Billy Baker, Nevin Liber, Ville Voutilainen and Inbal Levi, “Library Evolution Plan for Completing C++23,” November 9, 2021. Accessed March 17, 2022. <https://wg21.link/p2489>. [Note: This link will forward to the most recent version of this paper when a newer version is published.]

32. “C++23,” Wikipedia. Wikimedia Foundation. Accessed March 17, 2022. <https://en.wikipedia.org/wiki/C%2B%2B23>.

support the standard `<format>` header and the `std::format` function. Chapter 19 presented many more C++20 text formatting features using the standard capabilities, which were added to Visual C++ as we sent *C++20 for Programmers* to publication. The paper “Formatted output” proposes adding standard I/O and better Unicode support. For the current version of this paper, visit <https://wg21.link/p2093>.

- **MDSPAN:** The paper “MDSPAN” proposes a `std::mdspan` for creating views into multidimensional arrays and for slicing `mdspans`—that is, creating sub-views. For the current version of this paper, visit <https://wg21.link/p0009>.
- **Extended floating-point types:** The paper “P1467R8: Extended floating-point types and standard names” proposes allowing C++ implementations to provide additional floating-point types for other well-known floating-point layouts and defines how such types should interact with both the existing floating-point types and the additional ones defined by the implementation. For the current version of this paper, visit <https://wg21.link/p1467>.

### Other Features Being Considered for C++23

23

- **Associative container enhancements:** The paper “P2363: Extending associative containers with the remaining heterogeneous overloads” proposes additional member function overloads to improve the performance of certain existing operations. For the current version of this paper, visit <https://wg21.link/P2363>.
- **Comparing smart pointers to raw pointers:** The paper “P2249: Mixed comparisons for smart pointers” proposes allowing smart and raw pointers to be compared for equality to determine whether they point to the same object. For the current version of this paper, visit <https://wg21.link/P2249>.
- **Additional `constexpr` enhancements in the standard library:** The paper “P2283: `constexpr` for specialized memory algorithms” proposes adding `constexpr` to various standard library algorithms in header `<memory>`. For the current version of this paper, visit <https://wg21.link/P2283>.
- **Passing list initializers to standard algorithms:** In Section 20.13, we demonstrated creating a function that could receive a list initializer as an argument. The paper “P2248: Enabling List-Initialization For Algorithms” proposes new standard library algorithm overloads that accept initializer lists as arguments. For the current version of this paper, visit <https://wg21.link/P2248>.
- **Memory allocator enhancements:** The papers “Allocator-aware library wrappers for dynamic allocation” and “`allocate_unique` and `allocator_delete`” propose enhancements that enable developers to customize allocators with more aspects of the standard library—currently, you can do this only for container elements and `shared_ptr`s. For the current version of these papers, visit <https://wg21.link/P0211> and <https://wg21.link/P0316>.
- **`std::span` enhancements:** The paper “P2447: `std::span` And The Missing Constructor” proposes fixes for various scenarios in which you might expect to be able to use a `std::span`, but which currently produce compilation errors. For the current version of this paper, visit <https://wg21.link/P2447>.“

- **std::function\_ref enhancements:** The paper "P2472: Make function\_ref More Functional" proposes enhancements that make references to functions easier to use. For the current version of this paper, visit <https://wg21.link/P2472>.

## 20.16 Wrap-Up

In this chapter, we overviewed miscellaneous C++ topics beyond what we were able to include in the print book. We used the `const_cast` operator to “cast away” the `const`-ness of an object. We discussed storage classes, storage duration and objects lifetimes. Then, we used the `mutable` storage-class specifier to indicate that a data member should always be modifiable, even when it appears in an object that’s currently being treated as a `const`.

We showed how to use namespaces to avoid naming conflicts. You saw that some operator symbols can be represented as operator keywords. We also showed the mechanics of the `->*` and `.*` pointer-to-member operators.

Next, we showed how to use `dynamic_cast` to safely downcast a base-class pointer to a derived-class pointer so you could access derived-class-specific capabilities when polymorphically processing objects related by a class hierarchy. In that example, we also demonstrated runtime type information (RTTI) for dynamically discovering an object’s type, and we highlighted the C++ Core Guidelines’ cautions regarding hierarchy navigation with `dynamic_cast`.

We showed how to inherit base-class constructors for cases in which a derived class’s constructors have the same parameters as the base class’s constructor and simply forward their arguments to the corresponding base-class constructors.

17 We presented C++17’s `[[nodiscard]]` attribute for indicating that a function’s  
20 return value should not be ignored and used C++20’s `[[nodiscard]]` enhancement to  
specify a reason why the return value should not be ignored. The compiler displays the  
reason in a warning message if you do not use a `[[nodiscard]]` function’s return value.

11 We showed how to use the C++11 smart pointer standard library class templates  
`shared_ptr` and `weak_ptr` for managing shared resources. We demonstrated how to use  
custom deleter functions to allow `shared_ptr`s to manage resources that require special  
destruction procedures. We also explained how `weak_ptr`s can be used to prevent memory  
leaks in circularly referential data.

11 We discussed trailing return types for functions and showed that C++11’s `decltype`  
operator enables the compiler to determine an expression’s type at compile time. Both of  
these capabilities are used extensively in template metaprogramming. Next, you learned  
11 how to use the C++11 class template `initializer_list` to create a function that can be  
called with a braced initializer.

20 We presented C++20 attributes `[[likely]]` and `[[unlikely]]` for indicating which  
branches of `if...else` or `switch` statements are more or less likely to execute, so the com-  
piler can optimize the code accordingly. Finally, we presented features being considered  
23 for C++23.

## 20.17 Closing Notes

We will regularly update online Chapters 19 and 20—especially Chapter 20’s look ahead to C++23. We are currently writing *C++23 for Programmers* and hope to publish it as close as possible to when C++23 is approved by the ISO C++ Standards Committee. If you

subscribe to O'Reilly Online Learning (<https://learning.oreilly.com>), we'll be adding C++23 "Rough Cut" content to our *C++20 for Programmers* e-book and "Sneak Peek" video content to our *C++20 Fundamentals LiveLessons*. We'd be grateful for your comments, criticisms and suggestions for improving this evolving new C++23 material!