[New title]

# Modern C++ Full Throttle: Intro to C++20 & the Standard Library

A Presentation-Only Intro to Fundamentals, Arrays, Vectors, Pointers, OOP, Ranges, Views, Functional Programming; Brief Intro to Concepts, Modules & Coroutines

Presented by
Paul Deitel, CEO
Deitel & Associates, Inc.
paul@deitel.com
https://deitel.com









## Paul Deitel

- CEO, Deitel & Associates, Inc.
- MIT grad with 44 years in computing
- One of the world's most experienced programming-languages trainers
  - Have taught professional courses to industry, military and academic software developers worldwide since 1992
- Among the world's best-selling programming-language textbook/professional book/video authors

## Keeping In Touch

- paul@deitel.com
- https://deitel.com/contact-us (goes to email above)
- Facebook: https://facebook.com/DeitelFan
- Instagram Threads: @DeitelFan
- LinkedIn: https://linkedin.com/company/deitel-&-associates
- Twitter: https://twitter.com/deitel (@deitel)
- Mastodon: https://mastodon.social/@deitel
- YouTube: https://youtube.com/DeitelTV



3

## My Upcoming O'Reilly Live Training Courses https://deitel.com/LearnWithDeitel

- Oct 15 —Python Data Science Full Throttle: Intro AI, Big Data and Cloud Case Studies (new segment on GenAI API programming)
- Nov 5 — Java Full Throttle (with updates through Java 22 & 23)
- Nov 12 — Python Full Throttle (with updates through Python 3.12)
- Dec 3 — Python Full Throttle (with updates through Python 3.12)
- Dec 10 —Python Data Science Full Throttle: Intro AI, Big Data and Cloud Case Studies (new segment on GenAI API programming)



4

## Today's Presentation Is Based on Our C++20 Products

- **C++20 Fundamentals LiveLessons** (53+ hours)
  - https://learning.oreilly.com/videos/-/9780136875185/
- **C++ How to Program: An Objects Natural Approach, 11/e**
  - College textbook
  - https://deitel.com/cpphtp11
  - Amazon Kindle Reader app: https://amzn.to/3qpIxHs
  - Buy/Rent @ VitalSource: https://deitel.com/CPPHTP11onVitalSource
- **C++20 for Programmers: An Objects Natural Approach**
  - Assumes you are a programmer
  - https://deitel.com/cpp20fp
  - https://learning.oreilly.com/library/view/-/9780136905776/
  - Available in print and various e-book formats





5

## C++20 "Big Four"

- Intros today: **Ranges**, **Concepts**, **Modules** and a **Coroutine**
- Will be covered extensively in **C++20 "Big Four" Full Throttle: Ranges, Concepts, Modules, Coroutines & More**
  - In-Depth, Presentation-Only Treatment of the Big Four, Containers, Iterators, Algorithms, Views, Functional Programming, Templates, Metaprogramming, Concurrency
- Video in **C++20 Fundamentals LiveLessons**
  - https://learning.oreilly.com/videos/-/9780136875185/
- C++20 for Programmers
  - https://learning.oreilly.com/library/view/-/9780136905776/





6

## Rhythm of the Course

- Whirlwind tour of Modern C++ using C++20
  - with an intro to several C++23 updates
- Lecture only, source-code focused presentation
- Six lecture segments and three breaks
  - Two 7-minute breaks in first three hours
  - 45-minute meal break
  - Two 7-minute breaks in last three hours
- See C++20 for Programmers index to determine video lesson to view in C++20 Fundamentals
- Questions after the course? paul@deitel.com





## C++20 Compilers We Use

- Windows: Microsoft Visual Studio Community (Visual C++)
  - https://visualstudio.microsoft.com/downloads/
- Windows/macOS/Linux: GNU Compiler Collection (**g++**)
  - https://gcc.gnu.org
- Windows/macOS/Linux: LLVM/Clang (**clang++**)
  - https://llvm.org/
- Before You Begin video lesson
  - https://learning.oreilly.com/videos/c-20-fundamentals/9780136875185/
- Before You Begin PDF
  - https://deitel.com/cpphtp11BYB

## C++20 Compilers We Use

- Popular compilers mostly up-to-date now
- Xcode's `clang++` version is generally further behind
  - Version 15.3+ supports C++20 text formatting
  - Still missing some C++20 features
- Older Xcode versions
  - Most examples will run using `{fmt}` library (https://github.com/fmtlib/fmt)
    - Change `<format>` to `<fmt/format.h>`
    - Change calls to `format()` or `std::format()` to `fmt::format()`
    - Point compiler at the `{fmt}` library's `include` folder







## Getting the Source Code

- GitHub users: Clone the repository at
  - https://github.com/pdeitel/CPlusPlusFullThrottlePart1
  - Not a GitHub user? Click the green **Code** button and select **Download ZIP**
- We assume the examples are in your user account's **Documents** folder in a subfolder named **examples**
  - You may want to **move the folder** and **rename** it

## Docker

- Tool for packaging and executing software
- **Docker Desktop** for Windows or macOS
  - https://www.docker.com/get-started
- Sign up for a **Docker Hub** account
  - https://hub.docker.com
- GNU Compiler Collection (version 14.2)
  - `docker pull gcc:latest`
- LLVM/Clang (version 19)
  - `docker pull teeks99/clang-ubuntu:19`







## Getting Your Questions Answered

- C++ documentation: https://cppreference.com/
- Search https://stackoverflow.com with **[tag] c++**
- Live C++ discussions
  - https://cpplang-inviter.cppalliance.org
  - https://www.includecpp.org/discord/

## Note About Slides

- Today's presentation corresponds to **30+ hours of video** in my **C++20 Fundamentals LiveLessons**
  - https://learning.oreilly.com/videos/-/9780136875185/
- The slides are for your reference after today
  - I will not cover many of them directly, but will touch on many of their points in the code presentations
  - For full details, see my videos or our book
    - https://learning.oreilly.com/videos/-/9780136875185/ (most up-to-date)
    - https://learning.oreilly.com/library/view/-/9780136905776/



13



14

# Part 1: Compiling and executing C++ programs

- Microsoft Visual C++
- GNU g++
  - **Docker** and **g++** instructions on page 3 of script PDF in the training interface's **Resources** tab
- LLVM clang++
  - **Docker** and **clang++** instructions on page 3 of script PDF in the training interface's **Resources** tab





## Part 2: Intro to C++ Programming

- Write simple C++ applications
- `cout`/`cin` for command-line output and input
- Fundamental types
- Declare variables
- Arithmetic, equality and relational operators





## Demo: fig02_04.cpp

- Addition program

# Common Escape Sequences

| Escape sequence | Description |
|---|---|
| \n | Newline. Positions the screen cursor to the beginning of the next line. |
| \t | Horizontal tab. Moves the screen cursor to the next tab stop. |
| \\ | Backslash. Includes a backslash character in a string. |
| \" | Double quote. Includes a double-quote character in a string. |





# Arithmetic Operators

| Operation | Arithmetic operator | C++ expression |
|---|---|---|
| Addition | + | f + 7 |
| Subtraction | - | p - c |
| Multiplication | * | b * m |
| Division | / | x / y |
| Remainder | % | r % s |

# Equality and Relational Operators

| Algebraic operator | C++ operator | Sample condition | Meaning |
|---|---|---|---|
| Relational operators | | | |
| > | > | x > y | x is greater than y |
| < | < | x < y | x is less than y |
| ≥ | >= | x >= y | x is greater than or equal to y |
| ≤ | <= | x <= y | x is less than or equal to y |
| Equality operators | | | |
| = | == | x == y | x is equal to y |
| ≠ | != | x != y | x is not equal to y |



21



22

## Part 3: Control Statements, Part 1

- `if` and `if`…`else` selection statements
- `while` iteration statement
- Compound assignment operators
- Increment and decrement operators
- Why fundamental data types are not portable
- Objects Natural Case Study: Super-Sized Integers with Boost Multiprecision





## Keywords

- https://en.cppreference.com/w/cpp/keyword

## Demo: `control_statement_snippets.cpp`

- `if`
- `if…else`
- conditional operator (`?:`)
- `while`





## `bool` Data Type

- Condition
  - Any expression that evaluates to zero or nonzero
  - Zero is false, nonzero is true
- Data type **`bool`**
  - Values **`true`** and **`false`**—each is a C++ keyword
- For compatibility with C
  - any nonzero numeric value is **`true`**
  - **`0`** is **`false`**

## Demo: narrowing.cpp

- Preventing narrowing conversions
- Uniform initialization syntax







## Compound Assignment Operators

- c = 3, d = 5, e = 4, f = 6, g = 12

| OPERATOR | SAMPLE EXPRESSION | EXPLANATION | ASSIGNS |
|---|---|---|---|
| += | `c += 7` | `c = c + 7` | 10 to c |
| -= | `d -= 4` | `d = d - 4` | 1 to d |
| *= | `e *= 5` | `e = e * 5` | 20 to e |
| /= | `f /= 3` | `f = f / 3` | 2 to f |
| %= | `g %= 9` | `g = g % 9` | 3 to g |

# Increment and Decrement Operators

| Operator | Name | Example | Explanation |
|---|---|---|---|
| `++` | prefix increment | `++number` | Increment `number` by 1, then use the new value of `number` in the expression. |
| `++` | postfix increment | `number++` | Use the current value of `number` in the expression, then increment `number` by 1. |
| `--` | prefix decrement | `--number` | Decrement `number` by 1, then use the new value of `number` in the expression. |
| `--` | postfix decrement | `number--` | Use the current value of `number` in the expression, then decrement `number` by 1. |





# Fundamental Types Are Not Portable

- `int` might be
  - 16 bits (2 bytes)
  - 32 bits (4 bytes)
  - 64 bits (8 bytes)
- Sometimes must write multiple versions of programs to use different integer types on different platforms

## Fundamental Types Are Not Portable

- `int` at least 16 bits
- `long` at least 32 bits
- `long long` at least 64 bits
- Only requirement: `int` <= `long` <= `long long`
- Integer type names with sizes specified
  - https://en.cppreference.com/w/cpp/types/integer







## Demo: fig03_05.cpp

- Objects Natural Case Study: Super-Sized Integers
- Some Apps Integers Beyond `long long` Range
- Example: RSA Public-Key Cryptography uses prime numbers with hundreds of digits
- `long long` type can store only
  - –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  - Maximum of 19 decimal – integers in the quintillions

## Demo: fig03_05.cpp (cont.)

- Boost Multiprecision Open-Source Library
- https://github.com/boostorg/multiprecision/
- Most Boost libraries now individually installable
- `cpp_int` class for huge integers
- Boost provides 168 open-source C++ libraries
  - "Breeding ground" for new capabilities that often are incorporated into the C++ standard libraries





## Part 3: Control Statements, Part 2

- `for` and `do`…`while` iteration statements
- C++20 text formatting
- `switch` multiple-selection statement
- `break` and `continue` statements
- Logical operators





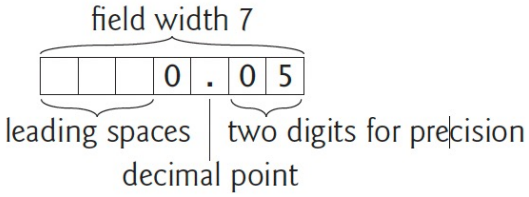
## Demo: fig04_04.cpp

- Compound-interest calculations
- C++20 text formatting (string interpolation)

# [FYI] C++20 Text Formatting

C++20 text formatting

- Placeholder `{:>7.2f}`
  - Colon (`:`) introduces a format specifier
  - `>7` – right-align (`>`) in a field width of 7 character positions
  - `.2f` – format a floating-point number (`f`) with two digits of precision (`.2`) to the right of the decimal point










# [FYI] C++20 Text Formatting

- Format string `"\n{}{:>20}\n"`
  - `"Year"` simply placed at first placeholder's (`{}`) position
  - `{:>20}` indicates that its argument, the 17-character string `"Amount on Deposit"`, should be right-aligned (`>`) in a field of `20` characters
  - Inserts three leading spaces to right-align the 17-character string in the 20-character field
  - Strings are left-aligned by default, so the `>` is required here to force right alignment

## [FYI] Compound-Interest Calculations; Introducing C++20 Text Formatting

- Format string `"{:>4d}{:>20.2f}\n"`
  - `{:>4d}` formats `year`'s value as an integer (`d` means decimal integer) right-aligned (`>`) in a field of width `4`—right-aligns all the `year` values under the four-character `"Year"` column
  - `{:>20.2f}` formats `amount`'s value right-aligned (`>`) in a field
  - width of `20` as a floating-point number (`f`) with two digits (`.2`) to the right of the decimal point—aligns decimal points vertically, as is typical with monetary amounts
  - Field width of 20 right-aligns amounts under the column heading `"Amount on Deposit"`





## [FYI] Compound-Interest Calculations; Introducing C++20 Text Formatting

- `float` – single-precision floating-point number
  - Typically 4 bytes and approximately seven significant digit
- `double` – double-precision floating-point number
  - Most of today's systems store these in eight bytes of memory with approximately 15 significant digits—approximately double the precision of `float`s.
  - Most programmers use type `double`
  - Floating-point literals are `double`s
- `double` provides at least as much precision as `float`
- `long double` provides at least as much precision as `double`
- Floating-point types suffer **representational error**
  - 10 / 3 yields 3.3333333…

## Demo: fig04_05.cpp

- `do`…`while` Iteration Statement
- Output the numbers 1–10







## Demo: fig04_06.cpp

- `switch` multiple-selection statement
- Count letter grades based on numeric grade values
- `switch` chooses among many different actions based on the possible values of a variable or expression that evaluates to an integral constant

## [FYI] **`break`** and **`continue`** Statements

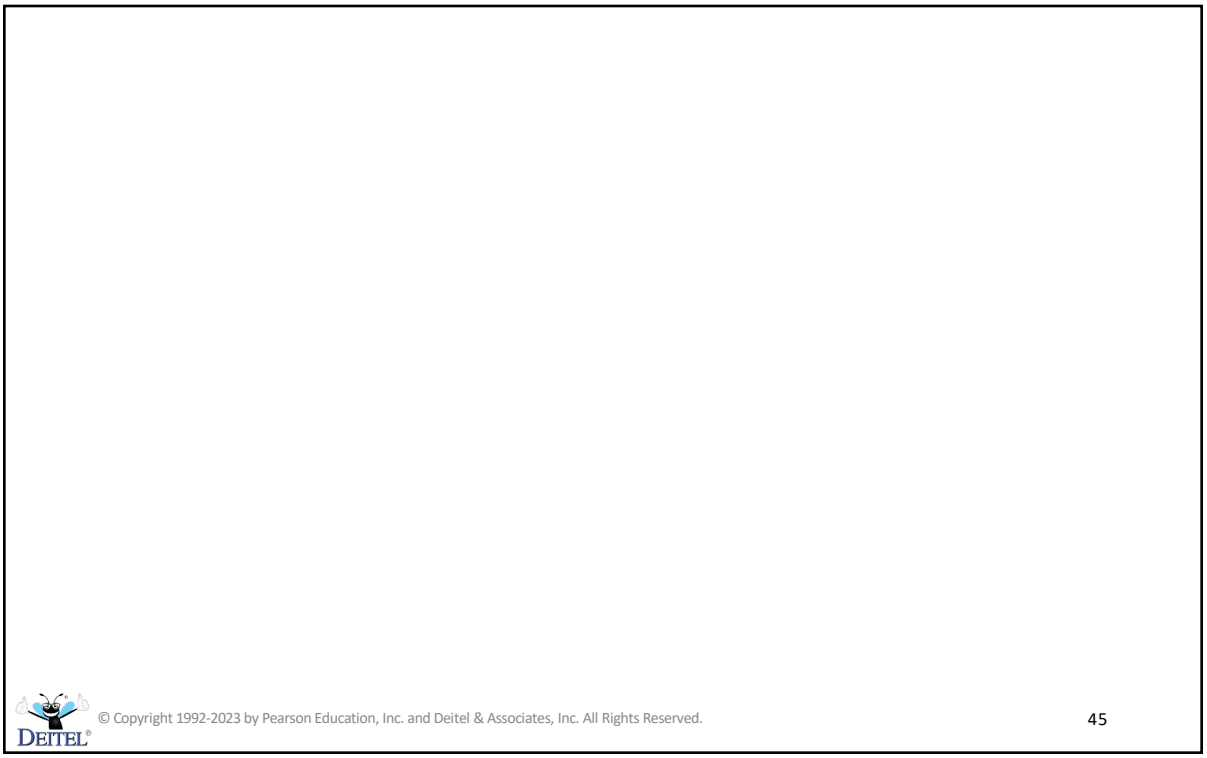
- Alter the flow of control
- Executing **`break`** in a **`while`**, **`for`**, **`do`**…**`while`** or **`switch`** causes immediate exit from that statement
- Executing **`continue`** in a **`while`**, **`for`** or **`do`**…**`while`** skips the remaining statements in the loop body and proceeds to the next loop iteration





## Logical Operators

- Logical AND (**&&**)
  - True only if both left and right operands are true
  - Short circuit: Right side evaluates only if left side is true
- Logical OR (**||**)
  - True if either operand or both are true
  - Short circuit: Right side evaluates only if left side is false
- Logical negation (**!**)
  - Reverses its operands true/false value




45

## Part 4: Functions and an Intro to Function Templates

- Custom function definitions/function prototypes
- Key C++ standard library headers
- Random-number generation for simulation
- Scoped `enum`s
- Selection statements with initializers
- C++20's `using enum` declarations
- References and pass-by-reference
- Overloaded functions
- Function templates for generating overloaded functions




46

# Demo: fig05_01.cpp

Function Definitions and Function Prototypes

- Create a custom function definition
- Declare the function with a function prototype
  - Describes the function's interface
  - If a function is defined first, the definition serves as the prototype
- Function name and parameter types form the function signature
- Return type is not part of function signature
  - Signatures cannot differ only by return type
- Functions in same scope must have unique signatures







# [FYI] Order of Evaluation of a Function's Arguments

- Argument evaluation order is not specified
- If arguments are expressions, order of evaluation could affect the values of one or more of the arguments
- Could cause subtle logic errors
- Assign arguments to variables before a call to force order

# Argument Coercion

- Forcing arguments to the appropriate types
  - E.g., can call a function with an `int` argument, even if prototype specifies a `double` parameter
- Error if the arguments cannot be implicitly converted to the expected parameter types







# Argument-Promotion Rules and Implicit Conversions

| Data types | |
|---|---|
| `long double` | |
| `double` | |
| `float` | |
| `unsigned long long int` | (synonymous with `unsigned long long`) |
| `long long int` | (synonymous with `long long`) |
| `unsigned long int` | (synonymous with `unsigned long`) |
| `long int` | (synonymous with `long`) |
| `unsigned int` | (synonymous with `unsigned`) |
| `int` | |
| `unsigned short int` | (synonymous with `unsigned short`) |
| `short int` | (synonymous with `short`) |
| `unsigned char` | |
| `char and signed char` | |
| `bool` | |

## C++ Standard Library Headers

- Standard library
  - https://en.cppreference.com/w/cpp/standard_library
- Headers with brief synopsis
  - https://en.cppreference.com/w/cpp/header
- Package Managers for third-party libraries
  - https://vcpkg.io/en/
    - 2400+ libraries (https://devblogs.microsoft.com/cppblog/whats-new-in-vcpkg-march-2024)
  - https://conan.io/
    - 1500 libraries
  - Others: https://moderncppdevops.com/pkg-mngr-roundup/







## Case Study: Random-Number Generation

- `<random>` header
- Replaces deprecated `rand` function (from C)
  - Predictable and poor statistical properties
- Can produce **nondeterministic random numbers** that can't be predicted
- Important for simulations and security scenarios where predictability is undesirable

## Case Study: Random-Number Generation

- Many classes representing random-number generation engines and distributions
  - **engine** – implements a random-number generation algorithm that produces pseudorandom numbers
  - **distribution** – controls the range of values produced by an **engine**, the types of those values and the statistical properties of the values
- `uniform_int_distribution`
  - Distributes pseudorandom integers evenly over a range





## Seeding the Random-Number Generator with `random_device`

- `random_device` typically used to seed a random-number generator
  - Produces evenly spread random integers, which cannot be predicted—nondeterministic
  - Slow—typically used only to seed engines
- ***Check docs***: **`random_device`** might be predictable on some platforms

# Demo: fig05_05.cpp

Game of Chance; Introducing Scoped **enum**s

- Roll two six-sided dice and calculate sum
- 7 or 11 on the first roll, you win
- 2, 3 or 12 on the first roll (called "craps"), you lose (i.e., the "house" wins)
- 4, 5, 6, 8, 9 or 10 on the first roll—that sum becomes your "point"
  - To win, keep rolling the dice until you "make your point"
  - The player loses by rolling a 7 before making the point
- Selection statements with initializers







# C++20: `using enum` Declaration

- For cases in which context is obvious
- Use constants without the type name and `::`
- `using enum Status;`
  - Use `keepRolling`, `won` and `lost`
- `using Status::keepRolling;`
  - Use just `keepRolling` without `Status::`
- Best to place inside block that uses the constants

# Demo: fig05_08.cpp

References and Reference Parameters

- Pass-by-value
  - Passes copy of argument's value
- Pass-by-reference
  - Called function can access caller's original variable directly
- Reference parameter
  - Alias for corresponding argument in a function call
- Pass-by-reference can be good for performance
  - Eliminates overhead of copying large amounts of data
  - Can use `const` reference parameters to prevent modification







# Demo: fig05_10.cpp

Function Overloading

- Functions of same name but different signatures
- Compiler selects proper function by examining the number, types and order of the arguments in the call
- Typically for functions that perform similar tasks but on data of different types or with different numbers of arguments
- Many math library functions are overloaded
- Complete overload resolution details
  - https://en.cppreference.com/w/cpp/language/overload_resolution

# Demo: maximum.h & fig05_13.cpp

## Function Templates

- Use if overloaded functions' logic and operations are identical
- Write one function template definition to define a **family of overloaded functions**
- C++ generates **function template specializations** (also called **template instantiations**) to handle calls for the provided argument types
- Known as generic programming







# Part 5: `array`s and `vector`s

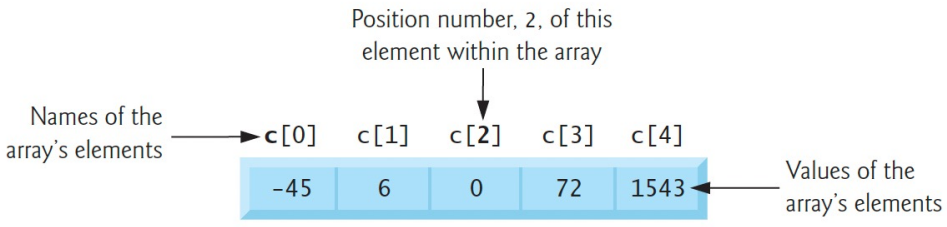
- C++ standard library class template `array`
- Declare, initialize and refer to `array` elements
- Range-based `for` statement
- Pass `array`s to functions
- Sorting and searching `array` data with C++ standard library functions
- Objects-natural case study: C++ standard library class template `vector`






# `array`s

- Elements (data items) are arranged contiguously in memory
- Refer to an element with the `array` name followed by the element's position number in square brackets (`[]`).




# Demo: fig06_01.cpp

## Initializing `array` Elements in a Loop

- `array` elements are not implicitly initialized unless the array is declared `static`
- `size_t` unsigned integral type that represents an object's size—commonly used for array indices
- No automatic `array` bounds checking with `[]`
  - Program can "walk off" either end of an `array` without warning
- `at` member function throws exception if index out of bounds
  - Sample error message from g++ on Linux

```
terminate called after throwing an instance of
  what(): array::at: __n (which is 10) >= _Nm (which is 5)
Aborted
```







# Demo: fig06_02.cpp

## Initializing an `array` with an Initializer List

- Follow `array` name with a brace-delimited comma-separated list of initializers
- Enables **CTAD (class-template argument deduction)**
  - Compiler determines element type and number of elements
- If fewer initializers than elements, remaining elements are **value initialized**
  - Fundamental numeric types are set to `0`, `bool`s are set to `false`, objects get their default initialization
- More initializers than elements is a compilation error

# Demo: fig06_03.cpp

## Range-Based `for` Statement

- Common to process all elements of an `array`
- Range-based `for` does this without using a counter
  - Ensures your code does not "step outside" `array`'s bounds
- Range-based `for`'s header declares a range variable to left of colon (`:`) and `array`'s name to right
  - Declaring a range variable as a `const` reference can improve performance – avoids copying large objects

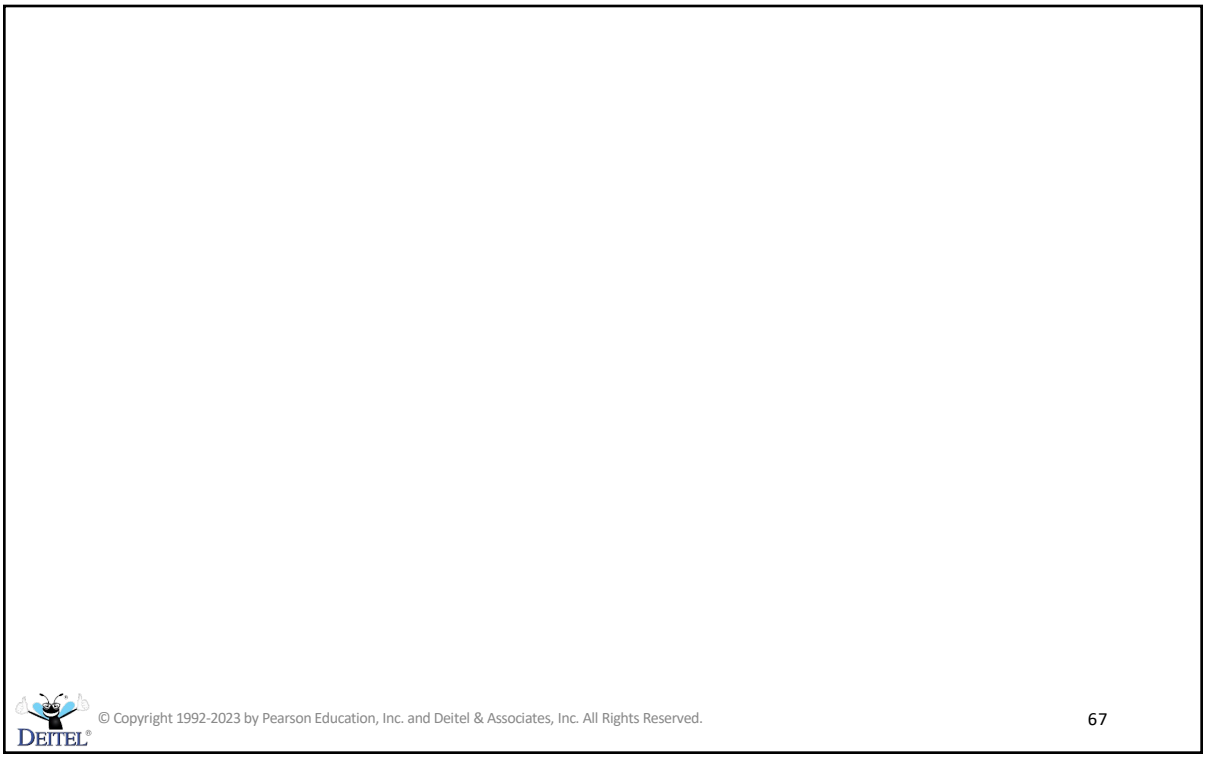




# Demo: fig06_14.cpp

## Objects-Natural Case Study: C++ Standard Library Class Template `vector`

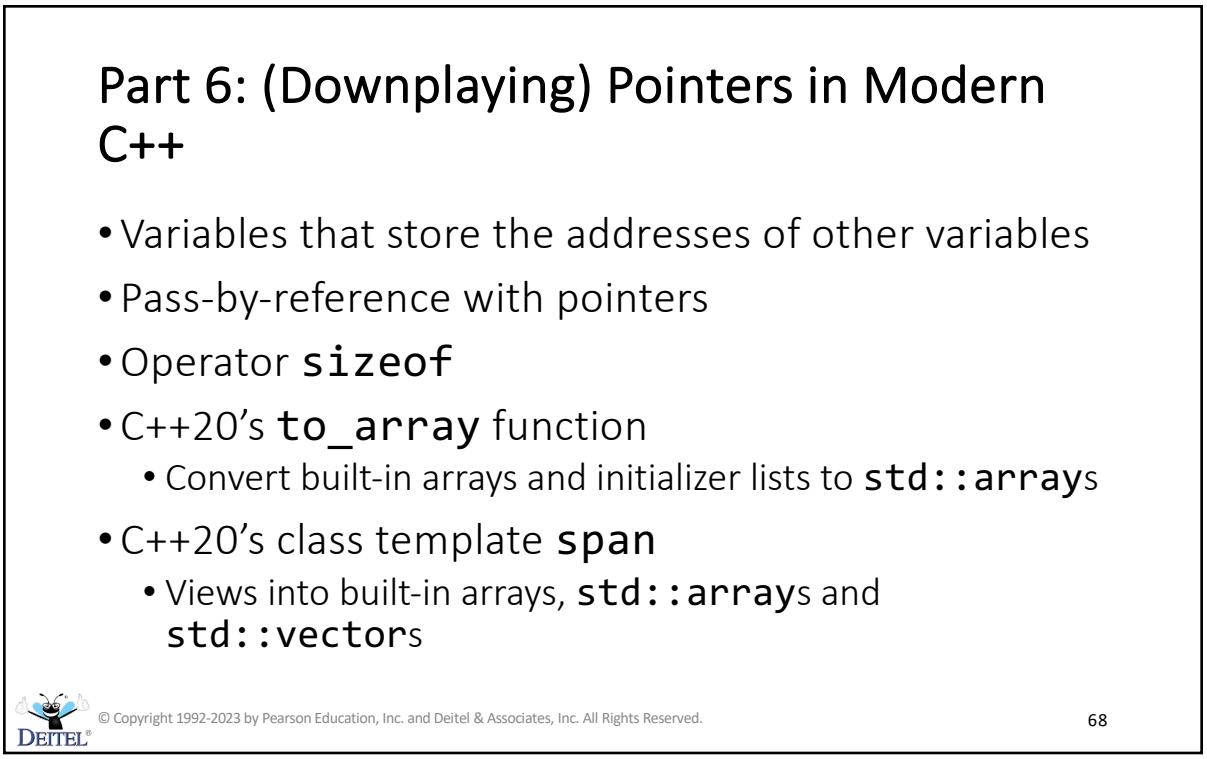
- Similar to `array` but supports dynamic resizing
- Many of the same member functions
- Both have member function `at` for bounds-checking
- Intro to C++'s exception-handling mechanism for detecting and handling exceptions




67

## Part 6: (Downplaying) Pointers in Modern C++

- Variables that store the addresses of other variables
- Pass-by-reference with pointers
- Operator `sizeof`
- C++20's `to_array` function
  - Convert built-in arrays and initializer lists to `std::array`s
- C++20's class template `span`
  - Views into built-in arrays, `std::array`s and `std::vector`s




68

## Downplaying Pointers

- Powerful, difficult, error-prone
- Various Modern C++ features eliminate need for most pointers
- New software-development projects:
  - Use references over pointers
  - Use `std::array` and `std::vector` over built-in pointer-based arrays
  - Use `std::string` objects over pointer-based C-strings





## Sometimes Pointers Are Still Required

- Legacy code
- Custom dynamic data structures
  - Prefer C++ standard library's existing dynamic containers
- Command-line arguments
- Pass arguments by reference with pointers if there's a possibility of a `nullptr`
  - C++ references must refer to something

## Two C++20 Features for Avoiding Pointers

- `to_array()` converts built-in array to `std::array`
- `span` – safer way to pass a built-in array to a function
  - Iterable—can use them with range-based `for`
  - Can use them with standard library container-processing algorithms
- Avoid using pointers, pointer-based arrays and pointer-based strings whenever possible

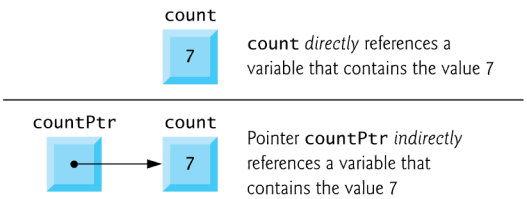






## Pointer Variable Declarations and Initialization

- `int count{7};`
- `int* countPtr{&count};`




# Pointer Variable Declarations and Initialization

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







# Demo: fig07_03.cpp

- Pass-by-reference with pointers

## Built-In Arrays

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





## Declaring and Accessing a Built-In Array

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

## Initializing Built-In Arrays

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





## Passing Built-In Arrays to Functions

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

## Declaring Built-In Array Parameters

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





## Standard Library Functions **begin** and **end**

- `// sort contents of colors`
  `std::sort(std::begin(colors), std::end(colors));`
- Can be applied to built-in arrays
  - `// sort contents of built-in array n`
    `std::sort(std::begin(n), std::end(n));`
  - works only in the scope that originally defines the array

## Built-In Array Limitations

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





## Demo: fig07_06.cpp

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

- C++ Core Guidelines
  - Prefer **`std::array`** and **`std::vector`** to built-in array
  - Safer and do not decay to pointers when you pass them to functions
- C++20's **`std::to_array`** function (header **`<array>`**) conveniently creates a **`std::array`** from built-in array or initializer list

# Demo: `fig07_12.cpp`

## Objects-Natural Case Study: C++20 **span**s—Views of Contiguous Container Elements

- `std::span` (header `<span>`)
  - View contiguous elements of a container
  - Contain pointer to first element and number of elements
  - "Sees" container's elements — does not have its own copy
- Built-in arrays passed as arguments decay to pointers
  - Parameter loses array size information
- C++ Core Guidelines: Pass built-in arrays as **span**s







## Part 7: `string`s, `string_view`s, Text Files, CSV Files and Regex

- `string_view`s—views of contiguous characters
- Regular expressions
  - Search strings for patterns
  - Validate data
  - Replace substrings





## Demo: fig08_08.cpp

- Header **`<string_view>`**
- Read-only views of C-strings or **`std::string`** objects
- Contains
  - a pointer to the first character in a contiguous sequence
  - a count of the number of characters
- Many **`std::string`**-style operations on C-strings without the overhead of creating and initializing **`std::string`**s
- C++ Core Guidelines
  - Prefer **`std::string`** if you need to "own character sequences"

## Raw String Literals

- \ in string introduces an escape sequence (e.g., \n)
- To include a backslash in a string, must use \\
- Makes some strings difficult to read
- Windows path
  - `string winPath{"C:\\MyFolder\\MySubFolder\\MyFile.txt"}`
- Raw string Windows Path
  - `string winPath{R"(C:\MyFolder\MySubFolder\MyFile.txt)"}`





## Raw String Literals

- Can include optional delimiters up to 16 characters
  - `R"MYDELIMITER(`*stringContents*`)MYDELIMITER"`
- May also include line breaks for multline strings
  - Handy for hard coding HTML, XML, JSON documents
  - `R"(multiple lines`<br>`of text)"`
  - In memory: `"multiple lines\nof text"`

# Objects Natural Case Study: Introduction to Regular Expressions

- Recognize patterns in text
  - Phone numbers
  - e-mail addresses
  - ZIP Codes
  - URLs
- Extract data from unstructured text
- Clean and transform data





# Objects Natural Case Study: Introduction to Regular Expressions

- Validate data
  - U.S. ZIP Code – five digits (e.g., 02215) or five digits followed by a hyphen and four more digits (e.g., 02215-4775)
  - Last name – letters, spaces, apostrophes and hyphens
  - E-mail – allowed characters in the allowed order
  - U.S. Social Security number – three digits, a hyphen, two digits, a hyphen and four digits, and adheres to other rules about specific numbers that can be used

# Regex Repositories

- https://regex101.com
- https://regexr.com/
- http://www.regexlib.com
- https://www.regular-expressions.info





# Character Classes

| Character class | Matches |
|---|---|
| `\d` | Any digit (0–9). |
| `\D` | Any character that is not a digit. |
| `\s` | Any whitespace character (such as spaces, tabs and newlines). |
| `\S` | Any character that is not a whitespace character. |
| `\w` | Any **word character** (also called an **alphanumeric character**)—that is, any uppercase or lowercase letter, any digit or an underscore |
| `\W` | Any character that is not a word character. |

## Custom Character Classes

- Custom Character Classes Are Defined in `[]`
  - `[aeiou]` matches a lowercase vowel
  - `[A-Z]` matches an uppercase letter
  - `[a-z]` matches a lowercase letter
  - `[a-zA-Z]` matches any lowercase or uppercase letter.





## Quantifiers

- * quantifier matches **zero or more occurrences** of the subexpression to its left
- **+** quantifier is like * but matches **at least one occurrence** of a subexpression.
- **?** quantifier matches zero or one occurrences of a subexpression.
- **{n,}** quantifier matches at least **n** occurrences of a subexpression.
- **{n,m}** quantifier matches between **n** and **m** (inclusive) occurrences of a subexpression.

## Regular Expressions

- Demo: fig08_15.cpp
  Matching entire strings to regular expressions
- Demo: fig08_16.cpp
  Regular expression replacements
- Demo: fig08_17.cpp
  Matching patterns throughout a string



95



96

# Part 8: Custom Classes

- Define a custom class and use it to create objects
- Member functions and data members
- Constructors for initializing objects
- Separate a class's interface and implementation
- Destructors for "termination housekeeping"
- `struct`s for aggregate types
- C++20 designated initializers for aggregates





# Demo: fig09_10-12 (Time.h, Time.cpp and fig09_12.cpp)

Separating interface from implementation

- Constructor with default arguments
- *set*/*get* member functions

## Aggregates in C++20

- **Aggregate type**–built-in array, `array` or object of a class
  - No user-declared constructors
  - No `private` or `protected` non-`static` data members
  - No `virtual` functions
  - No `virtual`, `private` or `protected` base classes
- **C++20 change:** No user-declared constructors
  - Prevents a case in which initializing an aggregate object could circumvent calling a user-declared constructor
- Can use a `class` with all `public` data
- `struct` is a class with `public` members by default





## Aggregates in C++20

```
• struct Record {
      int account;
      string first;
      string last;
      double balance;
  };
```

## Initializing an Aggregate

- Initialize an object of aggregate type Record
  - `Record record{100, "Brian", "Blue", 123.45};`
- Aggregates may contain in-class initializers:
  - ```
    struct Record {
        int account;
        string first;
        string last;
        double balance{0.0};
    };
    ```





## Initializing an Aggregate

- Can initialize an aggregate-type object with fewer initializers than data member
  - `Record record{0, "Brian", "Blue"};`
- Remaining data members are initialized as follows:
  - Data members with in-class initializers use those values
  - Data members without in-class initializers are initialized with empty braces (`{}`)
    - Empty-brace initialization sets fundamental-type variables to `0`, sets `bool`s to `false` and **value initializes** objects—that is, they're zero initialized, then the default constructor is called for each object


    ```

## C++20: Designated Initializers

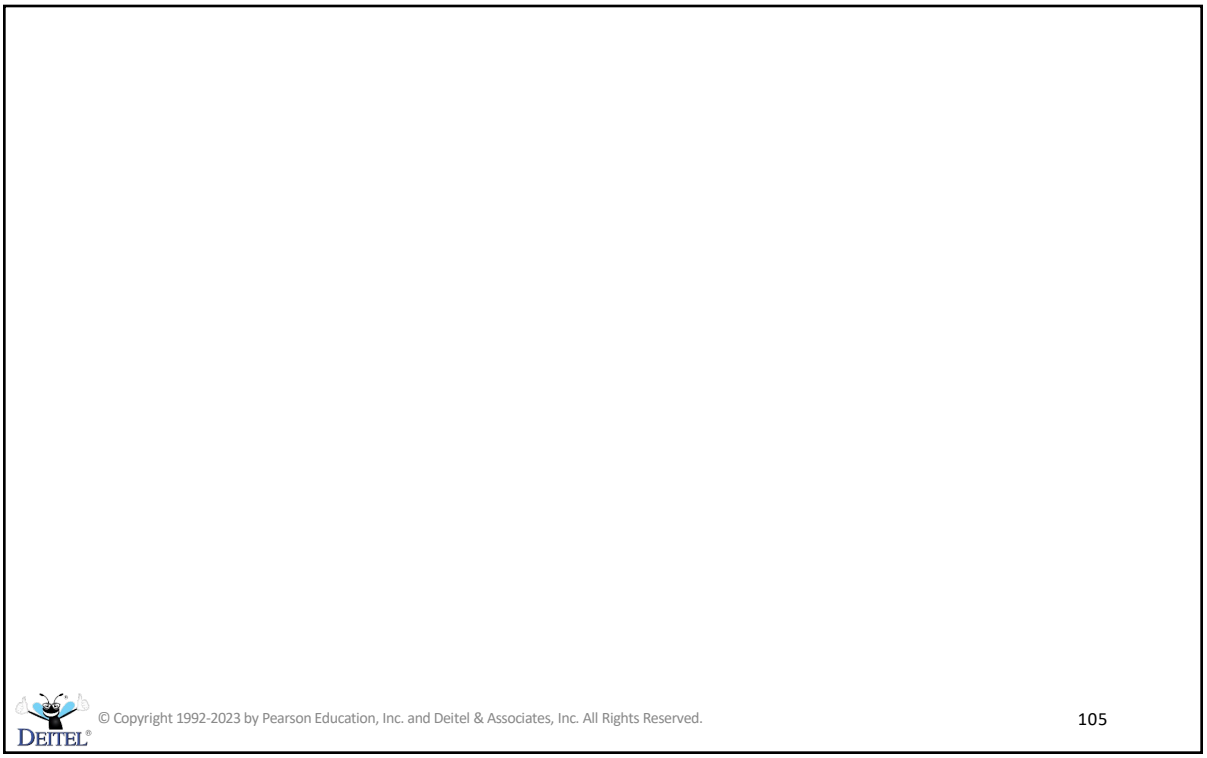
- C++20 aggregates support **designated initializers**
- Specify which data members to initialize by name
  - `Record record{.first{"Sue"}, .last{"Green"}};`
- Identifiers that you specify must be listed in the same order as they're declared in the aggregate type
- Remaining data members get default initializer values:
  - `account` is set to 0 and
  - `balance` is set to its default value in the type definition—in this case, 0.0.





## Other Benefits of Designated Initializers

- Adding new data members to an aggregate type will not break existing statements that use designated initializers
- New data members that are not explicitly initialized receive their default initialization
- Improved compatibility with C




105

# Part 9—OOP: Inheritance and Runtime Polymorphism

- Base classes and derived classes
- Runtime polymorphism
- `override` keyword
- `final` functions and classes
- Abstract and concrete classes
- Interfaces




106

## Demo: fig10_10

### Classes **`SalariedEmployee`** & **`SalariedCommissionEmployee`**

- Employee types in a payroll application
  - Base-class salaried employees are paid a fixed weekly salary
  - Derived-class salaried commission employees receive a weekly salary plus a percentage of their sales
- Demonstrating **`virtual`** functions and polymorphism





## Do Not Call **`virtual`** Functions from Constructors and Destructors

- In a constructor or destructor, the type used to determine which function to call is the class of that constructor or destructor
- Invokes the base-class version, even if the base-class constructor or destructor is called while creating or destroying a derived-class object

## virtual Destructors

- Include a `virtual` destructor in every class that contains `virtual` functions
- Prevents subtle errors when a derived class has a custom destructor
- In a class that does not have a destructor, the compiler generates a non-`virtual` one
- Most classes with `virtual` functions use
  - `virtual ~SalariedEmployee() = default;`





## Abstract Classes and Pure `virtual` Functions

- Class that typically defines a common public interface for classes derived from it
- Other classes are concrete
- Abstract class has one or more pure `virtual` functions
  - `virtual void draw() const = 0;`
  - "`= 0`" is a pure specifier
- Class with all pure `virtual` functions
  - **pure abstract class** or **interface**
- Can declare abstract base class pointers/references

# Employee Payroll Application Case Study

Implementation inheritance version

- Use to define closely related classes with many of the same data members and member function implementations
- https://learning.oreilly.com/course/c-20-fundamentals/9780136875185/







# Rethinking the **`Employee`** Hierarchy Using Composition and Dependency Injection

- Implementation inheritance creates tightly coupled classes
  - Base class's data members and member functions are inherited into derived classes
  - Changes to a base class directly affect all derived classes
  - Makes modifying class hierarchies difficult
- https://learning.oreilly.com/videos/c-20-fundamentals/9780136875185/9780136875185-CP20_Lesson10_22/

# Interface Inheritance Is Best for Flexibility

- `CompensationModel`s will use interface inheritance
  - Base class contains only pure `virtual` functions
  - Called an interface or a pure abstract class
  - C++ Core Guidelines recommend inheriting from interface rather than classes with implementation details
- Interfaces typically do not have data members
- Interface inheritance may require more work
  - Concrete classes must define everything, even if data and member-function implementations are similar or identical among classes
- But gives you additional flexibility
  - Eliminates tight coupling between classes







# Part 10: Operator Overloading, Copy/Move Semantics and Smart Pointers

- Special member functions
- *rvalue* references, move semantics, and moving vs. copying
- Manage dynamic memory with smart pointers
- `MyArray` class
  - Special member functions for copy and move semantics
  - Overloads many unary and binary operators
- C++20's three-way comparison operator (`<=>`)





# Operator Overloading Fundamentals

- C++ allows most existing operators to be overloaded by defining `operator` functions
  - Non-`static` member functions operate on an object of the class
  - Non-member functions receive objects as arguments
- Operators that work by default for all objects
  - **Memberwise Assignment (=)**
    - Can be dangerous for classes with pointer members—generally requires custom special member functions
  - **Address (&)**—returns a pointer to an object

## [FYI] Operator Overloading Fundamentals: Rules and Restrictions on Operator Overloading

- Cannot change an operator's precedence
  - Can use parentheses to *force* evaluation order
- Grouping cannot be changed
  - If it groups left-to-right, so does its overloads
- "Arity" (the number of operands) cannot be changed
  - Unary operators remain unary, binary operators remain binary
  - Operators **&**, *****, **+** and **-** each have unary and binary versions
- Cannot create new operator symbols
- Cannot change how an operator works on only fundamental-type values
  - For example, cannot make + subtract two `int`s





## (Downplaying) Dynamic Memory Management with **`new`** and **`delete`**

- Can allocate and deallocate memory
  - Objects or arrays of any built-in or user-defined type
  - For decades, performed with operators **`new`**/**`delete`**
    - **C++ Core Guidelines recommend against using `new`/`delete`**
    - You'll likely see **`new`**/**`delete`** in legacy C++ code
  - Section 11.4 in C++20 for Programmers discusses the old way and its problems
    - https://learning.oreilly.com/library/view/c-20-for-programmers/9780136905776/ch11.xhtml#sec11_4

# Demo: fig11_02.cpp

## Modern C++ Dynamic Memory Management—RAII and Smart Pointers

- Common design pattern
  - Allocate dynamic memory
  - Assign the address of that memory to a pointer
  - Use the pointer to manipulate the memory
  - Deallocate the memory when it's no longer needed
- If an exception occurs before deallocation → **memory leak**





# Demo: fig11_02.cpp (cont.)

- C++ Core Guidelines recommend **RAII—Resource Acquisition Is Initialization**
  - Create local object and **acquire the resource during construction**
  - Use the object
  - When the object goes out of scope, **destructor called automatically** to release the resource

# Smart Pointers

- Smart pointers
  - Use RAII to manage dynamically allocated memory for you
- Header `<memory>`
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`





# `unique_ptr` Ownership

- Only one `unique_ptr` at a time can own a dynamically allocated object
- Assigning one `unique_ptr` to another **transfers ownership**
  - Also when passing one `unique_ptr` to another's constructor
- Move assignment operator and move constructor
- **Last `unique_ptr` that owns the dynamic memory will delete it**
- Ideal mechanism for returning ownership of dynamically allocated objects to client code
  - When it goes out of scope in the client code, the destructor deletes the dynamically allocated object

## [FYI] `unique_ptr` to a Built-In Array

- `auto ptr{make_unique<int[]>(10)};`
- Dynamically allocates a built-in array with the number of elements specified by its argument (10)
  - Elements are value initialized
- `unique_ptr` to an array provides an **overloaded subscript operator (`[]`)** for element access
  - `ptr[2] = 7;`
  - `cout << ptr[2] << "\n";`







## `MyArray` Case Study: Crafting a Valuable Class with Operator Overloading

- Full case study at
  - https://learning.oreilly.com/videos/c-20-fundamentals/9780136875185/9780136875185-CP20_Lesson11_09/
- Class development goal—crafting valuable classes
- Problems with built-in arrays
  - No bounds checking
  - Can "walk off" either end—often a fatal runtime error
  - Must be indexed from 0
  - Cannot input/out entire array with >> or <<
  - Cannot be meaningfully compared
    - Array names are simply pointers to where the arrays begin in memory
  - Don't know their own sizes—**span**s help solve this
  - Cannot assign one array to another

# Demo: MyArray.h, MyArray.cpp, fig11_03.cpp

**`MyArray` Case Study**

- **Crafting a valuable class with operator overloading**
- Can implement more robust classes, like `array`/`vector`
- Class `MyArray`
  - `unique_ptr` to a built-in array of `int`s
  - Range checking subscript (`[]`) operator
  - Input/output with `>>` and `<<`
  - Comparable with `==` and `!=`
  - Know their own size—easier to pass to functions
  - Can be assigned to one another via `=`
  - Can be converted to `bool` `false` or `true` values to check if empty
  - `++` operators that add 1 to every element
  - `+=` to adds a specified value to every element







# Demo: MyArray.h, MyArray.cpp, fig11_03.cpp (cont.)

- Demonstrate the **five special member functions**
- **RAII (Resource Acquisition Is Initialization)** to **prevent memory leaks**
- Not meant to replace standard library class templates `array` and `vector`, nor is it meant to mimic their capabilities
- Demonstrate key C++ language and library features you'll find useful when you build your own classes

## Rule of Zero

- C++ Core Guidelines – design classes so compiler can autogenerate special member functions
  - Copy constructor, copy assignment operator, move constructor, move assignment operator, destructor
- To accomplish, composing each class's data using
  - Fundamental-type members
  - Objects of classes that do not require custom resource processing or that do it for you using RAII





## Rule of Five

- Classes that manage their own resources should define the five special member functions
- C++ Core Guidelines—if a class requires one special member function, it should define them all
- Known as the **Rule of Five**

## Rule of Five Defaults

- Even for classes with the compiler-generated special member functions, some experts recommend declaring them explicitly with `= default`

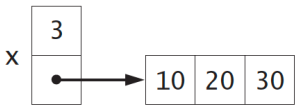




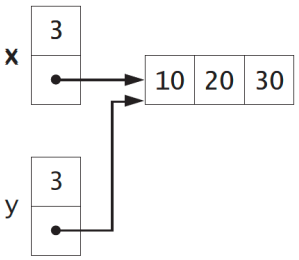


## Shallow Copy

Before x is shallow copied into y

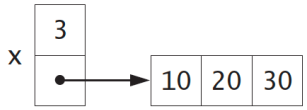



After x is shallow copied into y

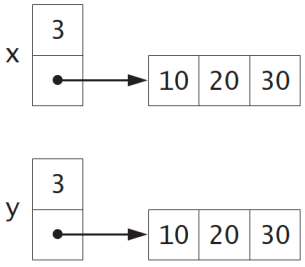



# Deep Copy

Before x is deep copied into y




After x is deep copied into y



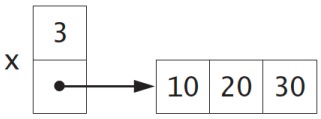




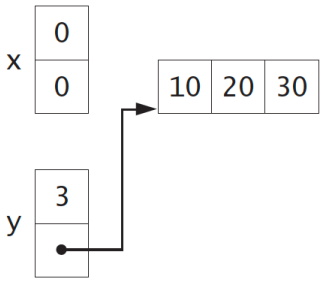



# Moving Does Not Move Anything

Before x is moved into y




After x is moved into y



# [FYI] Choosing Member vs. Non-Member Functions

- Functions in general, can be
  - member functions (can access internals)
  - `friend` functions (can access internals)
  - non-`friend` free functions
- C++ Core Guidelines
  - A function should be a member only if it needs direct access to the class's internal implementation details
- Another reason—commutative operators
  - Class `HugeInt` for arbitrary-sized integers
    - `bigInt + 7`
      `7 + bigInt`
  - To support these, typically use non-member `friend` functions
    - `friend HugeInt operator+(const HugeInt& left, int right);`
      `friend HugeInt operator+(int left, const HugeInt& right);`





# Demo: fig11_06.cpp

C++20 Three-Way Comparison (`<=>`)

- Often compare objects—e.g., sorting
- Can overload `==`, `!=`, `<`, `<=`, `>`, `>=`
- Define `<` and `==`
- Define `!=`, `<=`, `>`, `>=` in terms of `<` and `==`
  - `bool operator<=(const Time& right) const`
    `{return !(right < *this);}`
- Only difference in `!=`, `<=`, `>` and `>=` among classes is the argument type

# Demo: fig11_06.cpp (cont.)

- Most types can use the **`default`** C++20 **three-way comparison operator `<=>`**
  - Requires **`<compare>`**
  - Also called the spaceship operator
- **`default`** version works for any class in which all data members support comparison
- For built-in array data members, compiler applies **`operator<=>`** element-by-element
- https://en.cppreference.com/w/cpp/language/operator_comparison







# Part 11: Exceptions

- `try`, `catch` and `throw`
- What happens to uncaught exceptions





# Introduction—Exceptions and Exception Handling

- **exception** — a problem that occurs during a program's execution
- **exception handling** — helps you write robust, **fault-tolerant programs** that catch infrequent problems
  - deal with them and continue executing
  - perform appropriate cleanup for exceptions that cannot or should not be handled and terminate gracefully
  - terminate abruptly in the case of unanticipated exceptions — called failing fast

# Demo: fig12_04.cpp

Stack Unwinding and Uncaught Exceptions

- When exception not caught, next outer `try` block in an **enclosing scope** can attempt to catch
- Stack unwinding
  - Function terminates & local variables go out of scope
  - Control returns to invoking statement
  - If in a `try`, block terminates and attempt is made to catch the exception; otherwise, stack unwinding continues
- Uncaught exception calls `terminate`
  - Calls `abort` to terminate the program







# Constructors, Destructors and Exception Handling

- Subtle issues for exceptions in constructors and destructors
  - why constructors should throw exceptions when they encounter errors
  - how to catch exceptions that occur in a constructor's member initializers
  - why destructors should not throw exceptions

# Throwing Exceptions from Constructors

- What happens when an error is detected in a constructor?
- **Constructor cannot return a value to indicate an error**
  - Must indicate that the object was not constructed properly
- Option: Return the improperly constructed object
  - Client code must check for inconsistent state
- Option: Set a global variable outside the constructor
  - Poor software engineering
- Preferred alternative: Throw an exception
  - **Do not throw exceptions from the constructor of a global object**
  - Cannot be caught because they're constructed *before* main executes





# Throwing Exceptions from Constructors

- Before throwing, release raw explicitly managed resources, like memory, to prevent memory leaks
- Destructor for the object being constructed will not be called to release resources
- Destructors for data members already constructed will be called
- **Always use smart pointers to manage dynamically allocated memory**

# Demo: fig12_05.cpp

Catching Exceptions in Constructors via Function **`try`** Blocks

- Member initializers execute *before* constructor body and could throw exceptions
- Cannot simply wrap body statements in a **`try`** block
- Must use a function **`try`** block
- For a constructor
  - Place the **`try`** keyword *after* the parameter list and *before* the colon (**`:`**) that introduces the member-initializer list
  - Define the member initializers
  - Define the body
  - Follow the body with **`catch`** blocks



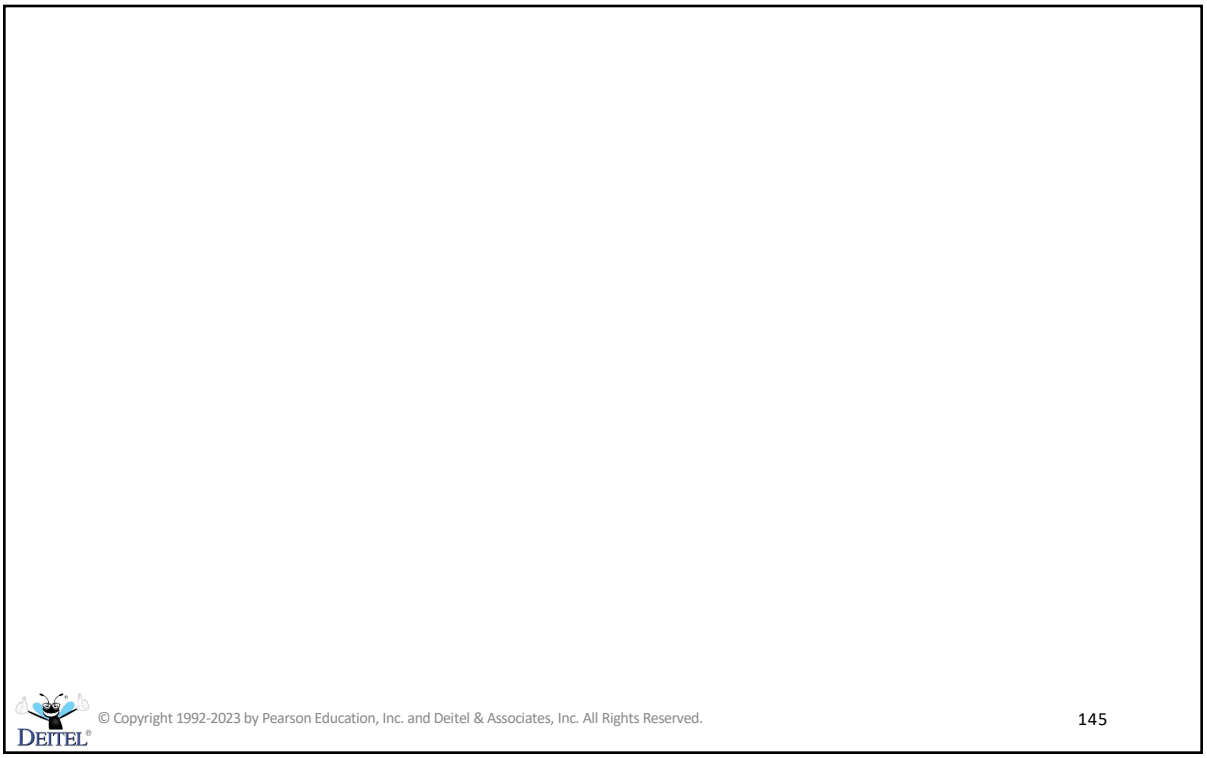




# Demo: fig12_05.cpp (cont.)

- Enables initial exception processing
  - E.g., logging or throwing a more appropriate exception
- Object cannot be fully constructed
  - Each **`catch`** following a function **`try`** block must throw a new exception or rethrow the existing one
- May be used with other functions with the following syntax

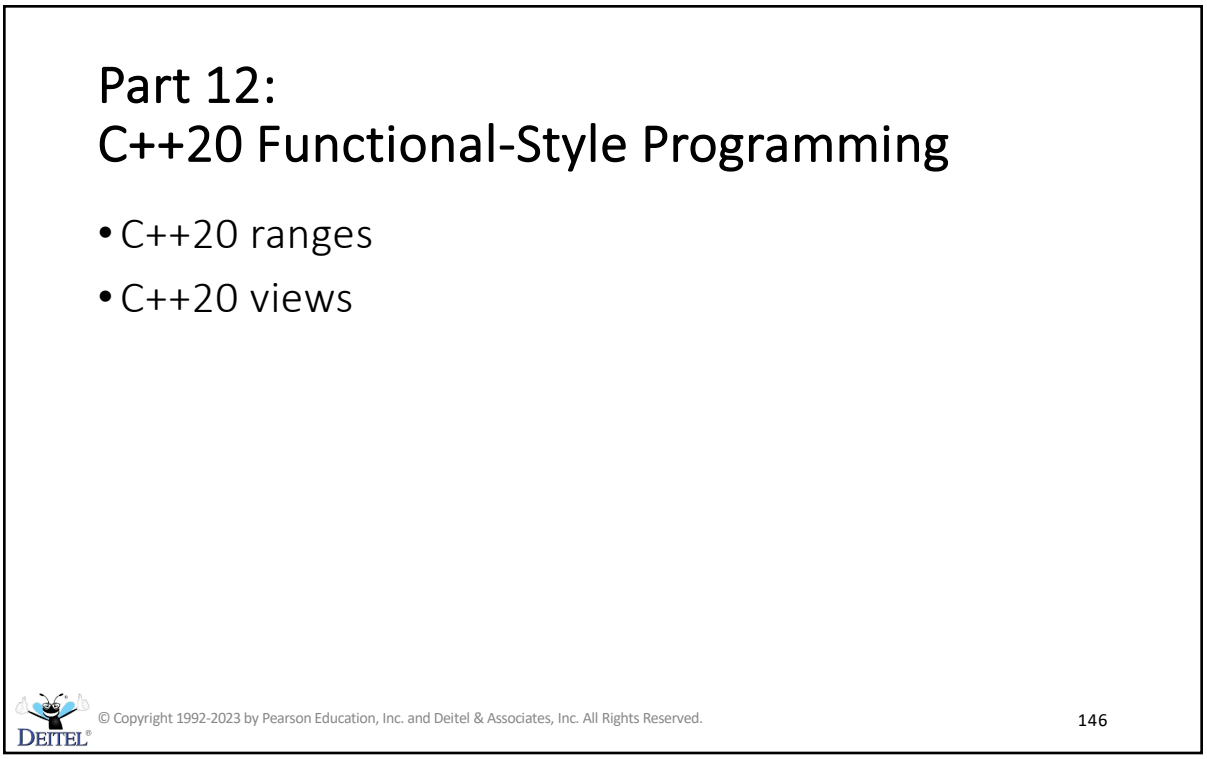
```
void myFunction() try {
   // do something
}
catch (const ExceptionType& ex) {
   // exception processing
}
```




145

# Part 12:
# C++20 Functional-Style Programming

- C++20 ranges
- C++20 views




146

## Intro to Functional-Style Programming

- C++ "Functional-style" features
  - More concise code
  - Easier to read, debug and modify
  - Often fewer errors
- Unlike Java, cannot yet be parallelized





## What vs. How

- As a program's tasks get more complicated
  - Code can become harder to read, debug and modify, and more likely to contain errors
  - Specifying how code works can become complex
- Functional-style programming
  - Specify ***what*** you want to do
  - Library code figures out ***how*** to do it
  - Eliminates many errors
- Emphasis on immutability
  - Avoids operations that modify variables' values
  - How? Check out the code for a functional-style reduction

## Demo: fig06_11.cpp

- Using `accumulate` to perform a reduction
- Summing the elements of an `array`





## Passing Functions as Arguments to Other Functions

- Higher-order functions
  - Many standard library functions allow you to customize how they work by passing other functions as arguments
  - Commonly used in functional programming
- `accumulate` totals elements by default
  - An overload receives as its fourth argument a function that defines *how* to perform the reduction
  - Rather than simply totaling the values, Fig. 6.12 calculates the product of the value.

## Demo: fig06_12.cpp

- Computing the product of an `array`'s elements using `accumulate`





## Filter, Map and Reduce: Intro to C++20's Ranges Library

- C++ standard library has enabled functional-style programming for many years
- C++20's ranges library (header `<ranges>`) makes it more convenient
- Two key aspects of this library
  - A range is a collection of elements that you can iterate over
  - A view enables you to specify an operation that manipulates a range
- Views are composable
  - Can chain them together to apply multiple operations

# Demo: fig06_13.cpp

- Demonstrates several functional-style operations using C++20 ranges





# Filter, Map and Reduce: Intro to C++20's Ranges Library

## Generating a Sequential Range of Integers with `views::iota`

- Typical example so far: Create an `array`, populate, process
- Can also generate values on-demand (lazy)
- **Lazy evaluation**
  - Reduces your program's memory consumption
  - Improves performance when all the values are not needed at once
  - **Values are not generated until you iterate over the results**
- `views::iota` generates a **half-open range** of integers
  - Does not include last value of specified range

# Filter, Map and Reduce: Intro to C++20's Ranges Library

## Filtering Items with `views::filter`

- Select only the range elements that match a condition
  - Often produces fewer elements than the original range
- Could use a loop + `if` statement – error-prone
- `views::filter` focuses on *what* we want to accomplish, not the iteration details
- Use | operator to connect multiple operations
  - Forms a pipeline of operations
  - Pipeline begins with a range (the data source), followed by an arbitrary number of operations connected by |





# Filter, Map and Reduce: Intro to C++20's Ranges Library

## Filtering Items with `views::filter`

- Argument is a function that receives one value to process and returns a `bool` indicating whether to keep the value
- Pipeline in lines 27–28 concisely represents *what* we want to do but not *how* to do it
  - `views::iota` knows how to generate integers
  - `views::filter` knows how to use its argument to determine whether to keep each value
- No results are produced until you iterate over values2

# Filter, Map and Reduce: Intro to C++20's Ranges Library

## Mapping Items with `views::transform`

- Produces a result with the same number of elements as the original range being mapped
- `views::transform` argument is a function that receives a value to process and returns the mapped value, possibly of a different type
- Pipeline in lines 32–33 adds another operation to the pipeline from lines 27–28
  - `views::iota` produces a value
  - `views::filter` keeps only even values
  - `views::transform` calculates the square of the even values





# Filter, Map and Reduce: Intro to C++20's Ranges Library

## Combining Filtering and Mapping Operations into a Pipeline

- Pipelines may contain arbitrary number of operations separated by | operators
- Lines 37–39 combine the preceding operations into a single pipeline, and line 40 displays the results

# Filter, Map and Reduce: Intro to C++20's Ranges Library

## Reducing a Range Pipeline with `accumulate`

- Standard library functions like `accumulate` also work with lazy range pipelines
- Line 44 performs a reduction that sums the squares of the even integers produced by the pipeline in lines 37–39

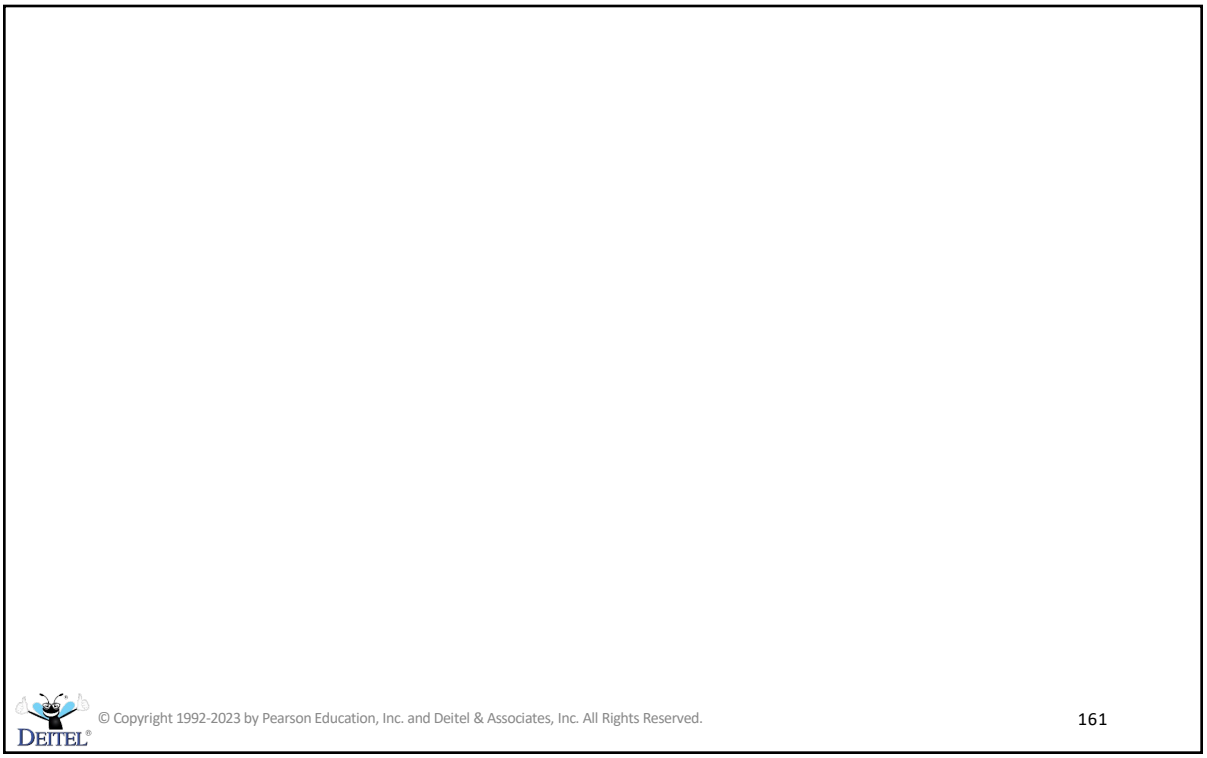




# Filter, Map and Reduce: Intro to C++20's Ranges Library

## Filtering and Mapping an Existing Container's Elements

- Range pipeline data source can be a C++ container
- Line 47 creates an `array` containing 1–10, then uses it in a pipeline that calculates the squares of the even integers in the `array`




161

## Part 13: Intros to Concepts, Modules, and Coroutines – Three of C++20's "Big Four" Features

- Ranges, Concepts, Modules, Coroutines
  - Focus of my second C++20 course coming in April
- Just introduced C++20 Ranges/Views
- Now a quick intro to
  - C++20 Concepts
  - C++20 Modules
  - C++20 Coroutines




162

## Concepts Introduction

- Part of the templates mechanism
- Constrain template parameters and overload function templates based on **type requirements**
- Compiler checks before template instantiation
  - Yields many fewer and more precise error messages
- Demo: `fig15_04.cpp` — Unconstrained function template
- Demo: `fig15_04.cpp` — Concept constrained template





## Concepts Introduction—Other C++20 Template Features

- C++20 abbreviated function templates
  - Define a function template without `template` header by using `auto` as the parameter type
  - Each `auto` type is determined by the compiler independently of all others – unless constrained by concepts
- C++20 templated lambdas
  - Lambda expressions with template parameters
  - Useful if you need multiple parameters to use exactly the same type

# Modules Introduction

- C++ creator Bjarne Stroustrup on modules

  "historic opportunity to improve code hygiene and compile times for C++ (bringing C++ into the 21st century)."
- Module
  - New way to organize code
  - Uniquely named, reusable group of related declarations and definitions with a well-defined interface
  - Control which declarations are visible outside a module
  - True encapsulation of implementation details


165

# Modules Introduction

- Immediate benefits in every program
  - `import` standard library headers
    - Eliminates repeated processing of `#include`s
    - Compiled once, then reused where `import`ed
  - C++23: `import` the entire standard library
    - `import std; // now: VC++, clang++ (partial)`
- Reduces translation unit sizes
- In big apps, significantly improved compile times
- Demo: `fig16_01.cpp` — importing a standard library header


166

## Coroutines Introduction

- Functions that can suspend execution and remember where they left off
- Can write concurrent code with sequential style
- Demo: `fig18_01.cpp` — Generator function







## 20.15 Some Key C++23 Features

- In our book, we mention C++23 features and some possible future C++ features including
  - the `std::mdarray` container (Section 6.13),
  - contracts (Section 12.13)
  - ranges enhancements (Section 14.10)
  - the modularized standard library (Section 16.12)
  - concurrent data structures (Section 17.15)
  - parallel ranges algorithms (Section 17.15)
  - executors (Sections 18.5 and 18.8)


169

## 20.15 Some Key C++23 Features

- `cppreference.com` table showing compiler support for C++23 lists over 120 items
  - https://en.cppreference.com/w/cpp/compiler_support/23
  - 25 categorized as "defect report" improvements (labeled as DR in the compiler-support table)
  - Others are new features or enhancements to existing features, such as new ranges and views capabilities that provide additional functional-style programming capabilities
- Many summarized at https://en.wikipedia.org/wiki/C%2B%2B23


170

## 20.15 Some Key C++23 Features—C++23 Formatted Output

- `<print>` header provides functions **`std::print`** and **`std::println`** ("print line")
  - Output formatted strings directly to standard output
  - Overloads for writing formatted text to other streams
- `std::println` automatically outputs a newline after outputting its arguments.


171

## 20.15 Some Key C++23 Features—C++23 Formatted Output

- Assume `int` variable `total` exists and contains `385`
- `std::cout << std::format("Sum is {}\n", total);`
  - can be written more concisely
- `std::print("Sum is {}\n", total);`
- `std::println("Sum is {}", total);`
- More info: https://en.cppreference.com/w/cpp/header/print


172

## 20.15 Some Key C++23 Features—C++23 Ranges Enhancements

- 11 new `std::ranges` algorithms and 14 new `std::views` functional-style programming capabilities
- `ranges::to` converts range into a container of elements
  - Convenient to store results of functional-style operations
- `results` is a `vector<int>` containing `0`, `2`, `4`, `6` and `8`

```
auto isEven{[](int x) {return x % 2 == 0;}};
auto results{
   std::views::iota(0)
      | std::views::filter(isEven)
      | std::views::take(5)
      | std::ranges::to<std::vector>()
}
```


173

## 20.15 Some Key C++23 Features—C++23 Ranges Enhancements

- Algorithms for searching for items in ranges
  - `ranges::contains`, `ranges::contains_subrange`, `ranges::find_last`, `ranges::find_last_if`, `ranges::find_last_if_not`, `ranges::starts_with`, `ranges::ends_with`


174

## 20.15 Some Key C++23 Features—C++23 Ranges Enhancements

- View operations `views::zip` and `views::zip_transform` for processing parallel ranges of elements
- Use `views::zip` to produce `tuple`s containing corresponding elements from `studentNames` and `gradePointAverages`, then display

```
using namespace std::string_literals;
std::vector names{"Meriem"s, "Pierre"s, "Sierra"s};
std::vector averages{3.9, 3.5, 4.0};

for (const auto& [name, gpa] : std::views::zip(names, averages)) {
   std::println("{}: {}", name, gpa);
}
```

- Output:

```
Meriem: 3.9
Pierre: 3.5
Sierra: 4.0
```


175

## 20.15 Some Key C++23 Features—C++23 Ranges Enhancements

- `views::enumerate` provides a convenient mechanism to access an element's index and value in a range
- Iterate through `vector colors`, displaying each element's index and value without the need for a mutable counter variable:
  - 

```
using namespace std::string_literals;
std::vector colors{"red"s, "orange"s, "yellow"s};

for (const auto& [i, color] : std::views::enumerate(colors)) {
   std::println("{}: {}", i, color);
}
```

  - Output:

```
0: red
1: orange
2: yellow
```

- More information on C++23 ranges and views enhancements: https://en.cppreference.com/w/cpp/header/ranges


176

## 20.15 Some Key C++23 Features—C++23 Generator Coroutines

- Section 18.4 created a generator coroutine using Sy Brand's **`tl::generator`** library
- C++23 provides **`std::generator`**
  - Should be a one-for-one replacement for `tl::generator`
- More info:
  - https://wg21.link/p2502
  - https://en.cppreference.com/w/cpp/header/generator


177

## 20.15 Some Key C++23 Features—C++23 Multidimensional Subscript Operator

- Section 6.13 introduced multidimensional arrays
- Access element using `[]` for each dimension:
  - `values[row][column]`
- Preceding syntax does not work with custom classes
- C++23 enables overloading the `[]` operator for multiple dimensions by allowing an arbitrary number of comma-separated parameters
  - `values[row, column]`


178

## 20.15 Some Key C++23 Features—C++23 std::mdspan

- Section 7.10 introduced **`std::span`** for creating views of contiguous elements in containers
- C++23 provides **`std::mdspan`** for multidimensional views of contiguous data
- Also supports selecting **subviews**, known as **slices**
- **`mdspan`** implements an overloaded multidimensional **`[]`** operator for accessing elements in an **`mdspan`**'s view


179

## 20.15 Some Key C++23 Features—C++23 std::mdspan

- View a one-dimensional, six-element array of **`int`**s as 2-by-3 data and display the elements by row:
  - ```
    std::array values{2, 3, 5, 7, 11, 13};
    auto values2D{std::mdspan(values.data(), 2, 3)};

    for (size_t row{0}; row != values2D.extent(0); ++row) {
       for (size_t column{0}; column != values2D.extent(1); ++column) {
          std::print("{:>2d}  ", values2D[row, column]);
       }

       std::println("");
    }
    ```
  - Output:
    ```
     2   3   5
     7  11  13
    ```
- More info:
  - https://en.cppreference.com/w/cpp/container/mdspan


180
    ```

## 20.15 Some Key C++23 Features—C++23 Container Enhancements

- Can initialize stack and queue container adapters from iterator pairs representing ranges of elements in other containers
  - `std::vector values{2, 3, 5, 7, 11, 13};`
    `std::queue myQueue{`
    `values.begin(), values.end()};`
- More info: https://wg21.link/p1425


181

## 20.15 Some Key C++23 Features—C++23 Container Enhancements

- Associative containers (Section 13.11) enable heterogeneous lookup
  - Can search for keys using values of compatible types without first converting them to container's key type
  - E.g., if keys are `std::string`s, you can pass search functions C-style strings or `std::string_view`s
- Now supported for associative container member functions `erase` and `extract`
- `https://wg21.link/p2077`


182

## 20.15 Some Key C++23 Features—C++23 Container Enhancements

- `<flat_map>` and `<flat_set>` headers provide container adaptors for manipulating sorted sequence containers (`vector`s by default) as maps and sets
- Provide better performance characteristics than the sorted associative containers.
- For more information, see
  - https://www.sandordargo.com/blog/2022/10/05/cpp23-flat_map
  - https://en.cppreference.com/w/cpp/header/flat_map
  - https://en.cppreference.com/w/cpp/header/flat_set


183

## 20.15 Some Key C++23 Features—C++23 Attribute `[[assume]]`

- Indicate assumptions compiler should make about your source code so it can optimize better
- ```
  int quotient(int numerator, int denominator) {
     [[assume(denominator != 0)]];
     return numerator / denominator;
  }
  ```
- https://en.cppreference.com/w/cpp/language/attributes/assume


184
  ```

## 20.15 Some Key C++23 Features—C++23 <stacktrace> Header

- C++23 `<stacktrace>` Header
  - Various programming languages, such as Java, C# and Python, enable programmatic access to information about the functions on the function-call stack when an exception occurs
  - `<stacktrace>` header adds these capabilities to C++
  - https://en.cppreference.com/w/cpp/header/stacktrace



185



186

186

# Wrap-Up

- Thanks for attending!
- Please fill out the course survey



187

# My Upcoming O'Reilly Live Training Courses https://deitel.com/LearnWithDeitel

- Oct 15 — Python Data Science Full Throttle: Intro AI, Big Data and Cloud Case Studies (new segment on GenAI API programming)
- Nov 5 — Java Full Throttle (with updates through Java 22 & 23)
- Nov 12 — Python Full Throttle (with updates through Python 3.12)
- Dec 3 — Python Full Throttle (with updates through Python 3.12)
- Dec 10 — Python Data Science Full Throttle: Intro AI, Big Data and Cloud Case Studies (new segment on GenAI API programming)



188

# Our C++20 Products

- **C++20 Fundamentals LiveLessons** (53+ hours)
  - https://learning.oreilly.com/videos/-/9780136875185/
- **C++20 for Programmers: An Objects Natural Approach**
  - Assumes you are a programmer
  - https://deitel.com/cpp20fp
  - https://learning.oreilly.com/library/view/-/9780136905776/
  - Available in print and various e-book formats
- **C++ How to Program: An Objects Natural Approach, 11/e**
  - College textbook
  - https://deitel.com/cpphtp11
  - Now available in various e-book formats
    - Amazon Kindle Reader app: https://amzn.to/3qplxHs
    - Buy/Rent @ VitalSource: https://deitel.com/CPPHTP11onVitalSource







# Keeping In Touch

- paul@deitel.com
- https://deitel.com/contact-us (goes to email above)
- Facebook: https://facebook.com/DeitelFan
- Instagram Threads: @DeitelFan
- LinkedIn: https://linkedin.com/company/deitel-&-associates
- Twitter: https://twitter.com/deitel (@deitel)
- Mastodon: https://mastodon.social/@deitel
- YouTube: https://youtube.com/DeitelTV