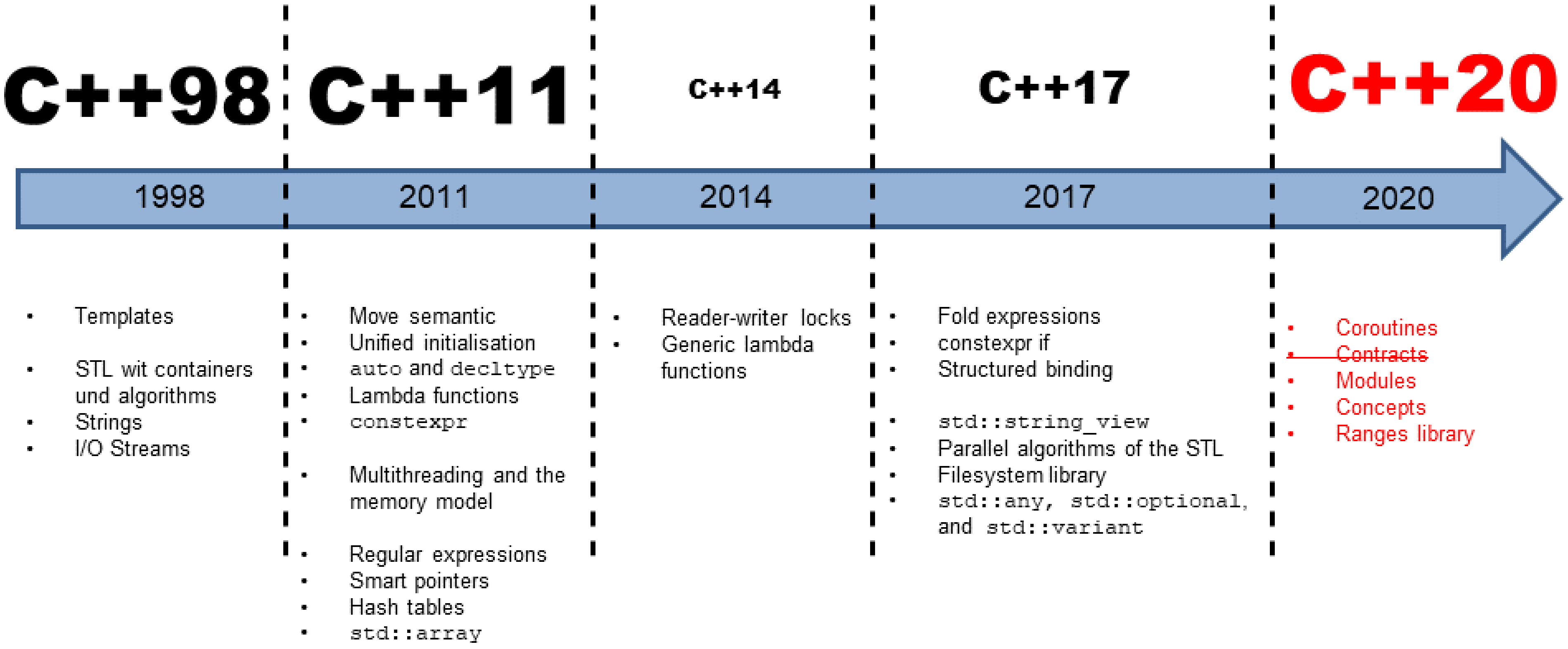
C++ 20

Ir. Johan Decorte

Brief history of C++

1979: developed by Bjarne Stroustrup



History of C++

- Created by Bjarne Stroustrup in 1979
- Extension of the C language
- It has gone through many versions, each adding new
    - Features
    - Libraries
    - Enhancements

  to improve
    - Performance
    - Usability
    - Maintainability
    - Express power of the code (=readability)

C++98 (ISO/IEC 14882:1998)

- Released: 1998
- Major Features:
  - First standardized version by ISO.
  - Templates: Introduced template functions and classes to enable generic programming.
  - STL (Standard Template Library): Added containers (like vector, map, list), iterators, and algorithms.
  - Namespaces: To avoid name conflicts in large programs.
  - Exceptions: A system for handling errors in a structured way.
  - New Casting Operators: dynamic_cast, static_cast, reinterpret_cast, and const_cast.
  - Type bool: Introduction of a bool type with true and false values.
  - Mutable Keyword: To allow modification of members in const objects.

C++03 (ISO/IEC 14882:2003)

- Released: 2003
- Major Features:
  - A bug-fix release and performance improvements over C++98.
  - Fixed various issues in the language, but no major new features.
  - Added better support for the Standard Template Library (STL).

C++11 (ISO/IEC 14882:2011)

- Released: 2011
- Major Features:
  - Move Semantics: Introduced rvalue references (&&) for efficient memory handling in objects.
  - Smart Pointers: Introduced std::unique_ptr and std::shared_ptr for memory management and avoiding memory leaks.
  - Auto Keyword: Type inference for variables, improving code readability.
  - Lambda Expressions: Allowed inline, anonymous functions for functional-style programming.
  - Range-based for loops: A simplified loop for iterating over collections.
  - Concurrency Support: New libraries like <thread>, <mutex>, and <future> to support multithreading.
  - nullptr: A type-safe null pointer replacement for NULL.
  - Override and Final Keywords: Control over virtual function overrides.
  - Enum Classes: Strongly typed and scoped enumerations.
  - Uniform Initialization: Braced initialization syntax for initializing containers and objects.

C++14 (ISO/IEC 14882:2014)

- Released: 2014
- Major Features:
  - Generic Lambdas: Allowed lambdas to deduce types automatically.
  - Return Type Deduction: Allowed functions to infer return types automatically with the auto keyword.
  - Relaxed constexpr: Expanded the use of constexpr functions, allowing more complex computations at compile time.
  - Binary Literals: Added support for binary literals (0b prefix).
  - Digit Separators: Allowed single quotes in numeric literals for readability (1'000'000).
  - Standardized make_unique: For constructing std::unique_ptr more safely.

C++17 (ISO/IEC 14882:2017)

- Released: 2017
- Major Features:
    - Structured Bindings: A shorthand for deconstructing objects and tuples.
    - If and Switch with Initializers: Enhanced the control flow by allowing initialization within if and switch statements.
    - Fold Expressions: Simplified variadic templates with fold expressions.
    - std::optional: A wrapper to indicate the presence or absence of a value.
    - std::variant: A type-safe union for managing multiple types.
    - std::any: A type-safe container for single values of any type.
    - Filesystem Library: Introduced utilities for interacting with the file system (<filesystem>).
    - Parallel Algorithms: Added parallel execution policies to many algorithms in the STL.
    - constexpr if: Conditional compilation of template code.

C++20 (ISO/IEC 14882:2020)

- Released: 2020
- Major Features:
  - **Ranges**: A new library for working with ranges of elements in a more composable way.
  - **Modules**: A modular compilation system for better code organization and faster compilation.
  - **Concepts**: Constraints on template parameters for more readable and debuggable generic code.
  - **Coroutines**: Native support for asynchronous programming with coroutines (co_await, co_yield, co_return).
  - Three-way Comparison (<=>): Simplifies operator overloading with the spaceship operator.
  - Calendar and Time Zone Library: Added <chrono> support for calendars and time zones.
  - constexpr Expansion: More operations, including dynamic memory allocation, allowed in
  - …

C++23 (ISO/IEC 14882:2023)

- Released: 2023
- Major Features:
  - Pattern Matching: Introduces new syntax to improve conditional branching and matching of data structures.
  - Deduction Guides: Simplifies template instantiation by deducing types from constructor arguments.
  - std::expected: A new type for handling return values that may represent an error, similar to std::optional.
  - Range Improvements: Further enhancements to the ranges library introduced in C++20.
  - Static operator[]: Allows arrays of fixed sizes within classes to behave like a single object when accessed.
  - constexpr for Virtual Functions: Support for virtual functions in constexpr contexts.
  - std::move_only_function: For functions that can be moved but not copied.
  - Reflection: Added preliminary support for metaprogramming by reflecting on types.

Future: C++26

- Expected: 2026
- Potential Features (speculative):
  - Improved compile-time reflection.
  - More improvements in metaprogramming, async programming, and parallelism.
  - Advances in language safety and simplicity.

Modules

Modules

- Advantages
  - No header files
  - Separation into interface files and implementation files is possible but not needed
  - Modules explicitly state what should be exported (e.g. classes, functions, …)
  - No need for include guards
  - Modules are processed only once → faster build times
  - Preprocessor macros have no effect on modules
  - Order of module imports is not important

Modules

- Create a module:

```cpp
// mymodule.cpp
export module MYMODULE;

namespace MYMODULE {
  auto GetWelcomeHelper() { return "Welcome to MYMODULE!"; }
  export auto GetWelcome() { return GetWelcomeHelper(); }
}
```

- Consume a module:

```cpp
// main.cpp
import MYMODULE;

int main() {
  std::cout << MYMODULE::GetWelcome();
}
```

Modules

- C++20 doesn't specify if and how to modularize the Standard Library
- Visual Studio makes it available as follows:
  - `std.regex` → <regex>
  - `std.filesystem` → <filesystem>
  - `std.memory` → <memory>
  - `std.threading` → <atomic>, <condition_variable>, <future>, <mutex>, <shared_mutex>, and <thread>
  - `std.core` → everything else in the C++ Standard Library

Modules

- You can “import” header files, e.g.:
  - `import <iostream>`
  - Implicitly turns the `iostream` header into a module
  - Improves build throughput, as `iostream` will then be processed only once
  - Comparable to precompiled header files (PCH)

1 Introduction

- 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
  - Encapsulate implementation details
- Complete, working code examples
  - Will point out current compiler differences as we go

1 Introduction

- C++ creator Bjarne Stroustrup
  - “Modules offer a historic opportunity to improve code hygiene and compile times for C++ (bringing C++ into the 21st century).”
- Immediate benefits in every program
- `import` standard library headers
  - Eliminates repeated processing of `#includes`
  - Modules are compiled once, then reused where `imported`

1 Introduction

Compiler Support for Modules (January 2023)

- Each compiler requires different commands
  - Provided in a text file for copy/paste
- Tracking compiler status
  - https://github.com/royjacobson/modules-report
- I’ll post updates at https://deitel.com

2 Compilation and Linking Before C++20

- Since the 1970s, C++ has always had a modular architecture for managing code
  - headers and source-code files
- Preprocessor performs text substitutions and other text manipulations on each source-code file
- **Translation unit –** preprocessed source-code file
- Compiler converts translation units into object code
- Linker combines app object code with library object code to create an executable

2 Compilation and Linking Before C++20

Problems with Header-File/Source-Code-File Model

- Order of `#includes` can cause subtle errors
- Compiler & linker don't always report a C++ entity that has different declarations in multiple translation units
- Reprocessing `#included` content is slow
  - One header can be included dozens or hundreds of times in large systems
- Eliminate reprocessing by `importing` as **header units** headers
  - Can significantly improve compilation times in large codebases

2 Compilation and Linking Before C++20

Problems with Header-File/Source-Code-File Model

- Other preprocessor problems
  - Definitions in headers can violate **One Definition Rule (ODR)**
    - “No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, template, default argument for a parameter (for a function in a given scope), or default template argument.”
  - No encapsulation — everything available where `#included`
  - Accidental cyclic dependencies
  - Compiler cannot check macros

3 Advantages and Goals of Modules

- Better organization and componentization of large codebases
- Smaller translation unit sizes
- Reduced compilation times
- Eliminate repetitive `#include` processing
  - Compiled module is not reprocessed for every file using it
- Eliminate `#include` ordering issues
- Eliminate preprocessor directives that can introduce subtle errors
- Eliminating One Definition Rule (ODR) violations

3 Advantages and Goals of Modules

Cons of Modules

- Incomplete support (January 2023)
- Existing codebases need to be modified to benefit from modules
- Do not solve packaging and distribution problems
  - Package managers in several other popular languages make this easier
- Compiled modules are compiler-specific
  - Not currently portable so still need to distribute modules as source code
- Uptake slow
  - Developers and organizations reviewing capabilities, deciding how to structure new codebases, potentially modifying existing ones ...
- Few recommendations and guidelines
  - **C++ Core Guidelines not yet updated for modules (November 2024)**

4 Example: Transitioning to Modules—Header Units

- **Goal: Eliminate the preprocessor**
- Preexisting libraries are provided as
  - header-only libraries
  - headers and source-code files
  - headers and platform-specific object-code files
- Some libraries might never be modularized
- **Transitional step: `import` (most) existing headers**

4 Example: Transitioning to Modules—Header Units

How Header Units Differ from Header Files

- Compiler produces information to treat header as a module
- **Compiled once**
  - Improves compilation performance in large-scale systems
- **`import` order is irrelevant**
- Header units implicitly "export" their contents
- `import` does not add code to a translation unit
- `#include`/`#define` don't affect subsequent `import`

4 Example: Transitioning to Modules—Header Units

Import all headers as header units (if possible)

- Not all headers can be imported
- Use `#include` if `import` produces errors
  - For example, header depends on `#define`d preprocessor macros—referred to as preprocessor state

4 Example: Transitioning to Modules—Header Units

Compiling with Header Units in Microsoft Visual Studio

- Right-click project name in **Solution Explorer** and select **Properties**
- In **Property Pages** dialog, select **All Configurations** from the **Configuration** drop-down
- Under **Configuration Properties > C/C++ > Language**, set **C++ Language Standard** to **ISO C++20 Standard (/std:c++20)**
- Under **Configuration Properties > C/C++ > All Options**, set **Scan Sources for Module Dependencies** option to **Yes**
- Click **Apply**, then click **OK**
- Add code to your project and build/run

4 Example: Transitioning to Modules—Header Units

Compiling with Header Units in g++

- Compile each header you'll `import` as a header unit
  - `g++ -fmodules-ts -x c++-system-header iostream`
    - `-fmodules-ts` currently required rather than `-std=c++20`
    - `-x c++-system-header` indicates that we are compiling a C++ standard library header as a header unit
- Compile source-code
  - `g++ -fmodules-ts fig16_01.cpp -o fig16_01`
- Run the executable
  - `./fig16_01`

4 Example: Transitioning to Modules—Header Units

Compiling with Header Units in clang++

- Compile each header you'll `import` as a header unit
  - `clang++-16 -std=c++20 -xc++-system-header --precompile iostream -o iostream.pcm`
    - `-xc++-system-header` indicates that we are compiling a C++ standard library header as a header unit
    - `--precompile` tells the compiler to compile the header as a header unit
  - Creates a **precompiled module (`.pcm`) file**
- Compile source-code
  - `clang++-16 -std=c++20 -fmodule-file=iostream.pcm fig16_01.cpp -o fig16_01`
- Run the executable
  - `./fig16_01`

5 Modules Can Reduce Translation Unit Sizes and Compilation Times

- Eliminating repeated preprocessing of the same header files across many translation units
  - Reduces compilation times
- Simple program with fewer than 80 characters

```
#include <iostream>

int main() {
   std::cout << "Welcome to C++20 Modules!\n";
}
```

5 Modules Can Reduce Translation Unit Sizes and Compilation Times

- Compilers have a flag to see preprocessor results
  - `-E` in `g++` and `clang++`
  - `/P` in Visual C++
- Translation unit sizes on our system were
  - 1,023,010 bytes in `g++`
  - 1,883,270 bytes in `clang++`
  - 1,497,116 bytes in Visual C++
- 11,000 to 21,000 times the size of the original source file
  - Large projects might have thousands of translation units

6 Example: Creating and Using a Module

- Module interface specifies module members available for use in other translation units
- `export` declaration
  - `export` a declaration or definition
  - `export` a group of declarations in braces
  - `export` a namespace
  - `export` a namespace member
    - Also exports the namespace’s name

6.1 `module` Declaration for a Module Interface Unit

- Module unit
  - A translation unit that is part of a module
  - A module unit composed of one translation unit is commonly referred to simply as a module

6.1 module Declaration for a Module Interface Unit

Module Declaration and Module Naming

- Module names
  - Convention: lowercase identifiers separated by dots (`.`)
    - `deitel.time` or `deitel.math`
  - Dots do not have special meaning
    - `deitel.time` and `deitel.math` not “submodules” of `deitel`
- Declarations from the `module` declaration to the end of the translation unit are part of the **module purview**
  - As are declarations from other units that make up the module
- `export module` introduces **primary module interface unit**
  - Specifies module members client code can access
  - One such unit per module

6.1 module Declaration for a Module Interface Unit

Module Interface File Extensions

- Microsoft Visual C++ uses `.ixx` filename extension for module interface units
- To add a module interface unit to your Visual C++ project
  - Right-click **Source Files** folder and select **Add > Module...**
  - In **Add New Item** dialog, specify a filename/location and click **Add**
- `.ixx` not required
  - For other extensions, right-click the file, select **Properties** and set **Item Type** to **C/C++ compiler**

- **g++** and **clang++** do not require special filename extensions for module interface units

6.1 module Declaration for a Module Interface Unit

Common Module Filename Extensions

- `.ixx`—Microsoft Visual C++ filename extension for primary module interface unit
- `.ifc`—Microsoft Visual C++ filename extension for compiled primary module interface unit
- `.cpp`—Filename extension for C++ source code, including module units
- `.cppm`—A recommended **clang++** filename extension for module units (recognized by Visual C++)

- `.pcm`—**clang++** compiled primary module interface unit

6.2 Exporting a Declaration

- `export` a declaration to make it available outside the module
- `export`ed functions are part of the module's interface
- `export`ed declarations must appear after a `module` declaration
  - at file scope (known as global namespace scope) or
  - in a named namespace's scope
- `export`ed declarations **must not have internal linkage**
  - `static` variables and functions at global namespace scope in a translation unit
  - `const` or `constexpr` global variables in a translation unit
  - identifiers declared in "unnamed namespaces"
- Preprocessor macros are for use only in that module and cannot be exported
- `export` complete definition of templates, `constexpr`/`inline` functions
  - Compiler needs access wherever module is imported

6.3 Exporting a Group of Declarations

- Can `export` a group of declarations in braces
- `Exports` every declaration in the braces
- These braces do not define a scope

6.4 Exporting a namespace

- Programs may define identifiers in different scopes
- Sometimes a variable of one scope will collide with a variable of the same name in a different scope
- Naming conflicts result in errors
- C++ solves this problem with `namespaces`
  - Each `namespace` defines a scope for identifiers
  - Helps ensure that these identifiers in other `namespaces`

6.4 Exporting a namespace

Defining and exporting namespaces

- `namespace` body delimited by braces (`{ }`) containing constants, data, classes and functions
- `namespace`s must be at global namespace scope or nested in other namespaces
    - Members may be defined in several identically named `namespace` blocks
- `export`ing a given `namespace` block, does not **`export`** identifiers in other identically named `namespace` blocks
- To use member, qualify name with `namespace` name and `::`
    - Or provide a `using` declaration or `using` directive

6.5 Exporting a `namespace` Member

- Can `export` specific `namespace` members
- `namespace` name is also `export`ed
- Does not `export` `namespace`’s other members

6.6 Importing a Module to Use Its Exported Declarations

- `Import` module to use its `exported` declarations
  - Available from `import` to the end of the translation unit
- Does not insert code into translation unit

6.6 Importing a Module to Use Its Exported Declarations

Compiling in VC++

- Add **fig16_03.cpp** to **Source Files** folder
- Run your project to compile the module and the main application

6.6 Importing a Module to Use Its Exported Declarations

Compiling in g++

- Compile `<string>` and `<iostream>` as header units
  - `g++ -fmodules-ts -x c++-system-header string`
  - `g++ -fmodules-ts -x c++-system-header iostream`
- Compile module interface unit to produce the file `welcome.o`
  - `g++ -fmodules-ts -c -x c++ welcome.ixx`
  - `-c` says to compile `welcome.ixx`, but not link it
  - `-x c++` option indicates that `welcome.ixx` is a C++ file
    - If we name `welcome.ixx` as `welcome.cpp`, then `-x c++` is not required
- Compile the main application and link it with `welcome.o`
  - `g++ -fmodules-ts fig16_03.cpp welcome.o -o fig16_03`

6.6 Importing a Module to Use Its Exported Declarations

Compiling This Example in `clang++`

- Compile `<string>` and `<iostream>` as header units
  - `clang++ -std=c++20 -xc++-system-header --precompile string -o string.pcm`
  - `clang++ -std=c++20 -xc++-system-header --precompile iostream -o iostream.pcm`
- Compile **module interface unit**
  - `clang++ -std=c++20 -fmodule-file=string.pcm -x c++-module welcome.ixx --precompile -o welcome.pcm`
- Compile main application and link with `welcome.pcm`
  - `clang++-16 -std=c++20 -fmodule-file=iostream.pcm fig16_03.cpp -fprebuilt-module-path=. welcome.pcm -o fig16_03`

6.7 Example: Attempting to Access Non-Exported Module Contents

- Strong encapsulation
  - Modules do not implicitly `export` declarations
  - Precise control over the declarations you `export`
- In our example, `cube` is not exported
  - Other translation units cannot call it
- Key difference from headers
  - Header's contents usable wherever it's `#included`

6.7 Example: Attempting to Access Non-Exported Module Contents

- Good practice
  - Put `exported` identifiers in namespaces to avoid name collisions if multiple modules `export` the same identifier
- Namespace names typically mimic their module names
  - `deitel.math` module contains namespace `deitel::math`

6.7 Example: Attempting to Access Non-Exported Module Contents

- g++ Error Messages
  - `g++ -fmodules-ts -x c++-system-header iostream`
  - `g++ -fmodules-ts -c -x c++ deitel.math.ixx`
  - `g++ -fmodules-ts fig16_05.cpp deitel.math.o`

```
fig16_05.cpp: In function 'int main()':
fig16_05.cpp:12:49: error: 'cube' is not a member of 'deitel::math'
   12 |    std::cout << "cube(e) = " << deitel::math::cube(3) << '\n';
      |                                               ^~~~
```

6.7 Example: Attempting to Access Non-Exported Module Contents

- `clang++` Error Messages
  - `clang++-16 -std=c++20 -xc++-system-header --precompile iostream -o iostream.pcm`
  - `clang++-16 -std=c++20 -x c++-module deitel.math.ixx --precompile -o deitel.math.pcm`
  - `clang++-16 -std=c++20 -fmodule-file=deitel.math.pcm -fmodule-file=iostream.pcm fig16_05.cpp -o fig16_05`

```
fig16_05.cpp:12:47: error: declaration of 'cube' must be imported
from module 'deitel.math' before it is required
   std::cout << "cube(3) = " << deitel::math::cube(3) << '\n’;
                                              ^
/usr/src/lesson16/fig16_04-05/deitel.math.ixx:12:8: note:
declaration here is not visible
   int cube(int x) {
       ^
1 error generated.
```

7 Global Module Fragment

- Some headers cannot be compiled as header units
  - Might require **preprocessor state**, such as macros defined in your translation unit or other headers
- `#include` in the **global module fragment**
  - `module;`
  - Place first in module unit, before `module` declaration
- **May contain only preprocessor directives**

7 Global Module Fragment

- Module interface unit can `export` a declaration `#included` in the global module fragment
  - `importing` implementation units can use that declaration
- **global module** contains
  - All module units' global module fragments
  - All non-modularized code in non-module translation units, such as the one containing `main`

8 Separating Interface from Implementation

- Can define interface and implementation in
  - separate source files
  - one source file
- We’ll demonstrate examples of both

8.1 Example: Module Implementation Units

- Can split a module definition into multiple source files for smaller, more manageable pieces
  - E.g., for a team of developers working on different aspects of the same module
- Example
  - **Primary module interface unit** for a module's **interface**
  - Separate source file for module's **implementation details**

8.1 Example: Module Implementation Units

Primary Module Interface Unit

- `deitel.math` module's primary module interface unit `exports` the `deitel::math` namespace
  - Function prototype for the function `average`

8.1 Example: Module Implementation Units

Module Implementation Unit

- File containing `module` declaration without `export`
- Implicitly imports its module's interface
- Compiler combines primary module interface unit and its corresponding module implementation unit(s) into one named module

8.1 Example: Module Implementation Units

Compiling This Example in Visual C++

- Ensure project includes in its **Source Files** folder
  - `deitel.math.ixx`—primary module interface unit,
  - `deitel.math-impl.cpp`—module implementation unit
  - `fig16_08.cpp`—`main` application
- Run your project to compile the module and the main application

8.1 Example: Module Implementation Units

Compiling This Example in g++

- Compile `<algorithm>`, `<iostream>`, `<iterator>`, `<numeric>` and `<vector>` as header units
- Compile primary module interface unit
  - `g++ -fmodules-ts -c -x c++ deitel.math.ixx`
- Compile module implementation unit
  - `g++ -fmodules-ts -c deitel.math-impl.cpp`
- Compile the main application and link it with `deitel.math.o` and `deitel.math-impl.o`
  - `g++ -fmodules-ts fig16_08.cpp deitel.math.o deitel.math-impl.o -o fig16_08`

8.1 Example: Module Implementation Units

- **Compiling This Example in clang++**
- Compile <algorithm>, <iostream>, <iterator>, <numeric> and <vector> as header units
- Compile primary module interface unit into precompiled module (.pcm) file:
  - ```
    clang++-16 -std=c++20 -fmodule-file=vector.pcm
       -x c++-module deitel.math.ixx --precompile
       -o deitel.math.pcm
    ```
- Compile the module implementation unit into an object file:
  - ```
    clang++-16 -std=c++20 -fmodule-file=deitel.math.pcm
       -fmodule-file=vector.pcm -fmodule-file=numeric.pcm
       -c deitel.math-impl.cpp -o deitel.math-impl.o
    ```
- Compile app and link with deitel.math-impl.o/deitel.math.pcm:
  - ```
    clang++-16 -std=c++20 -fmodule-file=algorithm.pcm
       -fmodule-file=iostream.pcm -fmodule-file=iterator.pcm
       -fmodule-file=vector.pcm fig16_08.cpp deitel.math-impl.o
       -fprebuilt-module-path=. deitel.math.pcm -o fig16_08
    ```
    ```

8.2 Example: Modularizing a Class

- Simplified version of Lesson 9’s `Time` class
- Interface in a **primary module interface unit**
- Implementation in a **module implementation unit**
- Module `deitel.time`
- Class in the `deitel::time` namespace

8.2 Example: Modularizing a Class

- **`deitel.time` Primary Module Interface Unit**
  - `export`s namespace `deitel::time` containing the `Time` class definition
- **`deitel.time` Module Implementation Unit**
  - `using namespace deitel::time;`
    - Enables module implementation unit to access namespace's contents
    - Translation units importing `deitel.time` do not see this

8.2 Example: Modularizing a Class

Compiling This Example in Visual C++

- Ensure that your project includes
  - `deitel.time.ixx`—primary module interface unit
  - `deitel.time-impl.cpp`—module implementation unit
  - `fig16_11.cpp`—main application file
- Run your project to compile the module and the main application

8.2 Example: Modularizing a Class

Compiling This Example in g++

- Compile `<iostream>`, `<string>` and `<stdexcept>` as header units
- Compile the primary module interface unit
  - `g++ -fmodules-ts -c -x c++ deitel.time.ixx`
- Compile the module implementation unit
  - `g++ -fmodules-ts -c deitel.time-impl.cpp`
- Compile main application and link with `deitel.time.o` and `deitel.time-impl.o`
  - `g++ -fmodules-ts fig16_11.cpp deitel.time.o deitel.time-impl.o  -o fig16_11`

8.2 Example: Modularizing a Class

Compiling This Example in clang++

- Use previous commands to compile the standard library headers `<iostream>`, `<string>` and `<stdexcept>` as header units
- Compile the primary module interface unit into a precompiled module (`.pcm`) file:
  - ```
    clang++-16 -std=c++20 -fmodule-file=string.pcm
       -x c++-module deitel.time.ixx --precompile -o deitel.time.pcm
    ```
- Compile the module implementation unit into an object file:
  - ```
    clang++-16 -std=c++20 -fmodule-file=deitel.time.pcm
       -fmodule-file=string.pcm -fmodule-file=stdexcept.pcm
       -c deitel.time-impl.cpp -o deitel.time-impl.o
    ```
  - `-fmodule-file=deitel.math.pcm` specifies primary module interface unit name
- Compile the main application and link it with `deitel.math-impl.o` and `deitel.math.pcm`
  - ```
    clang++-16 -std=c++20 fig16_11.cpp deitel.time-impl.o
       -fmodule-file=iostream.pcm -fmodule-file=string.pcm
       -fmodule-file=stdexcept.pcm -fprebuilt-module-path=.
       deitel.time.pcm -o fig16_11
    ```
    ```

8.3 `:private` Module Fragment

- Enables separating interface from implementation in one translation unit

```
export module name;

// code for primary module interface

module :private; // implementation details below this

// implementation details
```

- Primary module interface unit must be module’s only unit
- Changes to the implementation details do not affect module’s interface, nor other translation units that import this module

8.3 `:private` Module Fragment

- Cameron DaCamara from Microsoft's Visual C++ Team
  - Use when you want to have all your compiled code and interface code together in the same translation unit
  - "The way I think of the `:private` module fragment is that it is essentially a module implementation unit after `module :private;`, but I don't need to compile a separate `.cpp` file in order to implement details of the interface."
  - **Major benefit: "having all the code in a single interface can help guide your toolset's optimization decisions (perhaps making better ones) without fancy linker-based technology."**

9 Partitions

- Can divide a module's interface and/or implementation into smaller pieces
- Helps organize a module's components into smaller, more manageable translation units
- Can reduce compilation times in large systems
    - Only translation units that have changed and translation units that depend on those changes need to be recompiled
- Compiler aggregates a module's partitions into a single named module for import into other translation units

9.1 Example: Module Interface Partition Units

- `deitel.math` module that exports four functions in its primary module interface unit—`square`, `cube`, `squareRoot` and `cubeRoot`.
- Split into two module interface partition units (`powers` and `roots`) to show partition syntax
- Aggregate `export`ed declarations into a single primary module interface partition

9.1 Example: Module Interface Partition Units

- `deitel.math:powers`
  - `export module deitel.math:powers;`
  - module interface partition unit
  - partition name is "`powers`"
  - partition is part of the module `deitel.math`
- Module partitions are not visible outside their module, so they cannot be imported into translation units that are not part of the same module

9.1 Example: Module Interface Partition Units

`deitel.math:roots` Module Interface Partition Unit

- `export module deitel.math:roots;`
    - Module interface partition unit with the partition name "`roots`”
    - Partition is part of module `deitel.math`

9.1 Example: Module Interface Partition Units

- Rules for partitions
  - Module interface partitions with the **same module name** are part of the **same module**
  - Partitions are not implicitly known to one another
  - They do not implicitly import the module’s interface.
  - Partitions may be imported only into other module units that belong to the same module
  - One module interface partition unit can import another from the same module to use the other partition’s features

9.1 Example: Module Interface Partition Units

`deitel.math` Primary Module Interface Unit

- `deitel.math.ixx` primary module interface unit
- Every module must have a primary module interface unit containing `export module` and no partition name
- `import` and `export` the module interface partition units
  - Each `import` is followed by a colon (`:`) and the name of a module interface partition unit (in this case, `powers` or `roots`).
- `export` before `import` indicates each module interface partition unit's exported members also should be part of `deitel.math`'s primary module interface

- Module users cannot see its partitions

9.1 Example: Module Interface Partition Units

- Can `export import` primary module interface units.
- Assume we have modules named `A` and `B`
- In `A`
  - `export import B;`
- Translation unit that imports `A` also imports `B` and can use its exported declarations
- If you `export import` a **header unit**, its preprocessor macros are available for use only in the importing translation unit
  - To use a macro, explicitly import the header

9.1 Example: Module Interface Partition Units

Compiling This Example in Visual C++

- Add the files deitel.math-powers.ixx deitel.math-roots.ixx and deitel.math.ixx to your Visual C++ project using the steps from Section 6.1, then add the file fig16_15.cpp to the project's Source Files folder. Run your project to compile the module and the main application.

9.1 Example: Module Interface Partition Units

Compiling This Example in g++

- Compile `<cmath>` and `<iostream>` as header units
- Compile each module interface partition unit:
  - `g++ -fmodules-ts -c -x c++ deitel.math-powers.ixx`
  - `g++ -fmodules-ts -c -x c++ deitel.math-roots.ixx`
- Then, compile the primary module interface unit:
  - `g++ -fmodules-ts -c -x c++ deitel.math.ixx`
- Compile main application and link with `deitel.math-powers.o`, `deitel.math-roots.o` and `deitel.math.o`
  - `g++ -fmodules-ts fig16_15.cpp deitel.math-powers.o deitel.math-roots.o deitel.math.o -o fig16_15`

9.1 Example: Module Interface Partition Units

Compiling This Example in `clang++`

- Must build the partitions before the primary module interface unit
- At the time of this writing, clang++ will not compile `<cmath>` as a header unit
  - Change the `import` statement to `#include <cmath>`
- Compile `<iostream>` as a header unit
- Compile each module interface partition unit into a precompiled module (`.pcm`) file:
  - `clang++ -std=c++20 -x c++-module deitel.math-powers.ixx --precompile -o deitel.math-powers.pcm`
  - `clang++ -std=c++20 -x c++-module deitel.math-roots.ixx --precompile -o deitel.math-roots.pcm`
- Compile the primary module interface unit into a precompiled module (.pcm) file:
  - `clang++ -std=c++20 -x c++-module deitel.math.ixx -fprebuilt-module-path=. --precompile -o deitel.math.pcm`
- Compile main application and link with `deitel.math-powers.pcm`, `deitel.math-roots.pcm` and `deitel.math.pcm`:
  - `clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_15.cpp -fprebuilt-module-path=. deitel.math.pcm deitel.math-powers.pcm deitel.math-roots.pcm -o fig16_15`

9.2 Module Implementation Partition Units

- **At the time of this writing, none of our preferred compilers support module implementation partitions**
- Can divide module implementations into module implementation partition units to define a module’s implementation details across multiple source-code files
- Again, can help you organize a module’s components into smaller, more manageable translation units and possibly reduce compilation times in large systems
- `module` declaration must not contain `export`
  - `module` *`ModuleName`*`:`*`PartitionName`*`;`
- Module implementation partition units do not implicitly import the primary module interface

9.3 Example: “Submodules” vs. Partitions

- Some libraries are quite large
- Might want the flexibility to import only portions of a larger library
- Library vendor can divide into logical “submodules,” each with its own primary module interface unit
- Can also provide a primary module interface unit that aggregates the “submodules” by importing and re-exporting their interfaces

9.3 Example: “Submodules” vs. Partitions

`deitel.math.powers` Primary Module Interface Unit

- Rename `deitel.math-powers.ixx` as
  - “`-name`” indicate th`deitel.math.powers.ixx` at the `powers` partition was part of module `deitel.math`
  - Now `deitel.math.powers` is a primary module interface unit
- Declare a primary module interface unit with a dot-separated name
  - `export module deitel.math.powers;`

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in Visual C++

- Add `deitel.math.powers.ixx` and `fig16_17.cpp` to your Visual C++ project
- Run the project

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in g++

- Compile `<iostream>` as a header unit
- Compile the primary module interface unit:
  - ```
    g++ -fmodules-ts -c
        -x c++ deitel.math.powers.ixx
    ```
- Compile main application and link with `deitel.math.powers.o`
  - ```
    g++ -fmodules-ts fig16_17.cpp
        deitel.math.powers.o -o fig16_17
    ```

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in clang++

- Compile `<iostream>` as a header unit
- Compile the primary module interface unit into a precompiled module (`.pcm`) file:
  - `clang++ -std=c++20 -x c++-module deitel.math.powers.ixx --precompile -o deitel.math.powers.pcm`
- Compile the main application and link it with `deitel.math.powers.pcm`:
  - `clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_17.cpp -fprebuilt-module-path=. deitel.math.powers.pcm -o fig16_17`

9.3 Example: “Submodules” vs. Partitions

`deitel.math.roots` Primary Module Interface Unit

- Rename `deitel.math-roots.ixx` as `deitel.math.roots.ixx`
  - “`-name`” indicate that the `roots` partition was part of module `deitel.math`
  - Now `deitel.math.roots` is a primary module interface unit
- Declare a primary module interface unit with a dot-separated name
  - `export module deitel.math.roots;`

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in Visual C++

- Add `deitel.math.roots.ixx` and `fig16_19.cpp` to your Visual C++ project
- Run the project

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in g++

- Compile <iostream> and <cmath> as header units
- Compile the primary module interface unit:
  - `g++ -fmodules-ts -c -x c++ deitel.math.roots.ixx`
- Compile main application and link with `deitel.math.roots.o`
  - `g++ -fmodules-ts fig16_19.cpp deitel.math.roots.o -o fig16_19`

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in clang++

- At the time of this writing, clang++ would not compile `<cmath>` header as a header unit
  - Remove `import` statement in line 5 of Fig. 18
  - Place the following lines before the module declaration
    - ```
      module;
      #include <cmath>
      ```
- Compile the standard library header `<iostream>` as a header unit
- Compile primary module interface unit into a precompiled module (`.pcm`) file:
  - ```
    clang++ -std=c++20 -x c++-module deitel.math.roots.ixx
        --precompile -o deitel.math.roots.pcm
    ```
- Compile the main application and link it with `deitel.math.powers.pcm`:
  - ```
    clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_19.cpp
        -fprebuilt-module-path=. deitel.math.roots.pcm -o fig16_19
    ```
    ```

9.3 Example: “Submodules” vs. Partitions

deitel.math Primary Module Interface Unit

- `deitel.math.powers` and `deitel.math.roots` are separate modules but imply a logical relationship
- For convenience, we can aggregate these in a primary module interface unit that `export imports` both “submodules
- With “submodules” developers now have the flexibility to
  - `import deitel.math.powers` to use `square` and `cube`
  - `import deitel.math.roots` to use `squareRoot` and `cubeRoot`
  - `import deitel.math` to use all four functions

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in Visual C++

- Add the Fig. 16, 18 and 20 .ixx files and fig16_21.cpp to your Visual C++ project, as you did in Section 6.1. Run the project to compile the code and run the application.

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in g++

- Compile the primary module interface unit
  - `g++ -fmodules-ts -c -x c++ deitel.math.ixx`
- Compile the main application and link it with `deitel.math.powers.o`, `deitel.math.roots.o` and `deitel.math.o`:
  - `g++ -fmodules-ts fig16_21.cpp deitel.math.powers.o deitel.math.roots.o deitel.math.o -o fig16_21`

9.3 Example: “Submodules” vs. Partitions

Compiling This Example in clang++

- Compile the primary module interface unit into a precompiled module (`.pcm`) file:
  - `clang++ -std=c++20 -x c++-module deitel.math.ixx -fprebuilt-module-path=. --precompile -o deitel.math.pcm`
- Compile main application and link it with `deitel.math.powers.pcm`, `deitel.math.roots.pcm` and `deitel.math.pcm`:
  - `clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_21.cpp -fprebuilt-module-path=. deitel.math.pcm deitel.math.powers.pcm deitel.math.roots.pcm -o fig16_21`

10 Additional Modules Examples

- Importing the modularized Microsoft and clang++ standard libraries
- Module restrictions and the compilation errors you’ll receive if you violate those restrictions
- Difference between module members that other translation units can use by name vs. module members that other translation units can use indirectly

10.1 Example: Importing the C++ Standard Library as Modules

- C++ standard does not currently require compilers to provide a modularized standard library
- Microsoft provides one for Visual C++, which is split into several modules

10.1 Example: Importing the C++ Standard Library as Modules

- `std.core`—Contains most of the standard library, except for the following items
- `std.filesystem`—Contains <filesystem> header's capabilities
- `std.memory`—Contains <memory> header's capabilities
- `std.regex`—Contains <regex> header's capabilities
- `std.threading`—Contains capabilities of all the concurrency-related headers: <atomic>, <condition_variable>, <future>, <mutex>, <shared_mutex> and <thread>
- Cannot `import` and also `#include` standard library headers
- C++23: Modules named `std` and `std.compat`

10.2 Example: Cyclic Dependencies Are Not Allowed

- A module cannot have a dependency on itself
  - Cannot import itself directly or indirectly
- Cannot compile this example in g++ or clang++ because each requires a primary module interface unit to be compiled before you can import it

10.3 Example: imports Are Not Transitive

- Modules have strong encapsulation and do not export declarations implicitly
- Thus, import statements are not transitive

10.4 Example: Visibility vs. Reachability

- A declaration is **visible** in a translation unit if you can use its name
  - As in all examples that have exported items so far
- Some declarations are **reachable but not visible**
  - Cannot explicitly mention the declaration’s name in another translation unit, but the declaration is **indirectly accessible**
- **Anything visible is reachable, but not vice versa**

10.4 Example: Visibility vs. Reachability

- Fig. 29 imports `deitel.time` module, calls the module's exported `getTime` function to get a `Time` object
- We infer variable `t`'s type
  - If you replace `auto` with `deitel::time::Time`, you'd get an error
    - (Visual C++): `'Time': is not a member of 'deitel::time'`
  - Error occurs because `Time` is **not visible** in this translation unit.
  - `Time`'s definition is **reachable** because `getTime` returns a `Time` object—the compiler knows this, so it can infer variable's `t`'s type.
  - When a class definition is reachable, the class's members become visible

11 Migrating Code to Modules

- Frequently referred to the C++ Core Guidelines for advice and recommendations on the proper ways to use various language elements
- Modules technology is still new, the popular compilers' modules implementations are not complete, and the C++ Core Guidelines have not yet been updated with modules recommendations
- Few articles and videos discuss experiences with migrating existing software systems to modules

11 Migrating Code to Modules

- Cameron DaCamara, “Moving a Project to C++ Named Modules,” August 10, 2021. Accessed January 25, 2023.
  - **https://devblogs.microsoft.com/cppblog/moving-a-project-to-cpp-named-modules/**
- Steve Downey, “Writing a C++20 Module,” July 5, 2021. Accessed January 25, 2023.
  - https://www.youtube.com/watch?v=AO4piAqV9mg
- Daniela Engert, “Modules: The Beginner’s Guide,” May 2, 2020.Accessed January 25, 2023.
  - https://www.youtube.com/watch?v=Kqo-jIq4V3I
- Yuka Takahashi, Oksana Shadura and Vassil Vassilev, “Migrating Large Codebases to C++ Modules,” August 22, 2019. Accessed January 25, 2023.
  - https://arxiv.org/abs/1906.05092
- Nathan Sidwell, “Converting to C++20 Modules,” October 4, 2019. Accessed January 25, 2023
  - https://www.youtube.com/watch?v=KVsWIEw3TTw

12 Future of Modules and Modules Tooling

- C++23 modular standard library
- Tooling to help you use modules is under development and will continue to evolve over several years

12 Future of Modules and Modules Tooling

- Module-aware build tools that manage compiling software systems (Visual C++ already has this)
- Tools to produce cross-platform module interfaces so developers can distribute a module interface description and object code, rather than source code
- Dependency-checking tools to ensure that required modules are installed
- Module discovery tools to determine which modules and versions are installed
- Tools that visualize module dependencies, showing you the relationships among modules in software systems
- Module packaging and distribution tools to help developers install modules and their dependencies conveniently across platforms

12 Future of Modules and Modules Tooling

References

- Daniel Ruoso, “Requirements for Usage of C++ Modules at Bloomberg,” July 12, 2021. Accessed January 25, 2023. https://isocpp.org/files/papers/P2409R0.pdf
- Nathan Sidwell, “P1184: A Module Mapper,” July 10, 2020. Accessed January 25, 2023. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1184r2.pdf
- Rob Irving, Jason Turner and Gabriel Dos Reis, “Modules Present and Future,” June 18, 2020. Accessed January 25, 2023. https://cppcast.com/modules-gaby-dos-reis/
- Cameron DaCamara, “Practical C++20 Modules and the Future of Tooling Around C++ Modules,” May 4, 2020. Accessed January 25, 2023. https://www.youtube.com/watch?v=ow2zV0Udd9M
- Nathan Sidwell, “C++ Modules and Tooling,” October 4, 2018. Accessed January 25, 2023. https://www.youtube.com/watch?v=4yOZ8Zp_Zfk
- Gabriel Dos Reis, “Modules Are a Tooling Opportunity,” October 16, 2017. Accessed January 25, 2023. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0822r0.pdf

Ranges

Ranges

- What’s a range?
  - An object referring to a sequence/range of elements
  - Similar to a begin/end iterator pair, but not replace them
- Why ranges?
  - Provide nicer and easier to read syntax:

```
vector<int> data{ 11, 22, 33 };
sort(begin(data), end(data)); // before C++20
sort(data);                   // C++20
```

  - Eliminate mismatching begin/end iterators
  - Allows “range adaptors” to lazily transform/filter underlying sequences of elements

Ranges

- Based on two core components:
  - **Views**: range adaptors: lazily evaluated, non-owning, non-mutating
  - **Algorithms**: all Standard Library algorithms accepting ranges instead of iterator pairs
  - Views can be chained using pipes → |

Ranges

- Example of chaining views:

```
vector<int> data{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto result = data | views::remove_if([](int i) { return i % 2 == 1; })
                   | views::transform([](int i) { return to_string(i); });
// result == {"2","4","6","8","10"};
```

- **Note**: all lazily executed: nothing is done until you iterate over `result`

Ranges

- Example of a filtering and transforming chain of range adaptors:

```
int total = accumulate(
    view::ints(1) |
    view::transform([](int i) {return i * i; }) |
    view::take(10),0);
```

  - `view::ints(1)` lazily generates an infinite sequence of integers
  - this is lazily squared
  - And finally we only take the first 10 elements of the infinite sequence and accumulate these

Ranges: algorithms (e.g. for_each)

```cpp
#include <iostream>
#include <ranges>
#include <vector>
#include <algorithm>

int main()
{
    // VIEWS
    using std::views::filter,
          std::views::transform,
          std::views::reverse;

    // Some data for us to work on
    std::vector<int> numbers = { 6, 5, 4, 3, 2, 1 };

    // Lambda function that will provide filtering
    auto is_even = [](int n) { return n % 2 == 0; };

    // Process our dataset
    auto results = numbers | filter(is_even)
                           | transform([](int n) { return ++n; })
                           | reverse;

    // Use lazy evaluation to print out the results
    auto print = [](int n) { std::cout << n << " "; };

    // C++17
    std::for_each(results.begin(), results.end(), print);
    std::cout << std::endl;

    // C++20
    std::ranges::for_each(results, print);
    std::cout << std::endl;
}
```

coroutines

Coroutines

- What's a coroutine?
  - A function,
  - with one of the following:
    - **`co_await`:** suspends evaluation of a coroutine while waiting for a computation to finish
    - **`co_return`:** returns from a coroutine (just return is not allowed)
    - **`co_yield`:** returns a value from a coroutine back to the caller, and suspends the coroutine, subsequently calling the coroutine again continues its execution
    - a range-based **`for co_await`** loop:
      `for co_await (for-range-declaration : expression) statement`

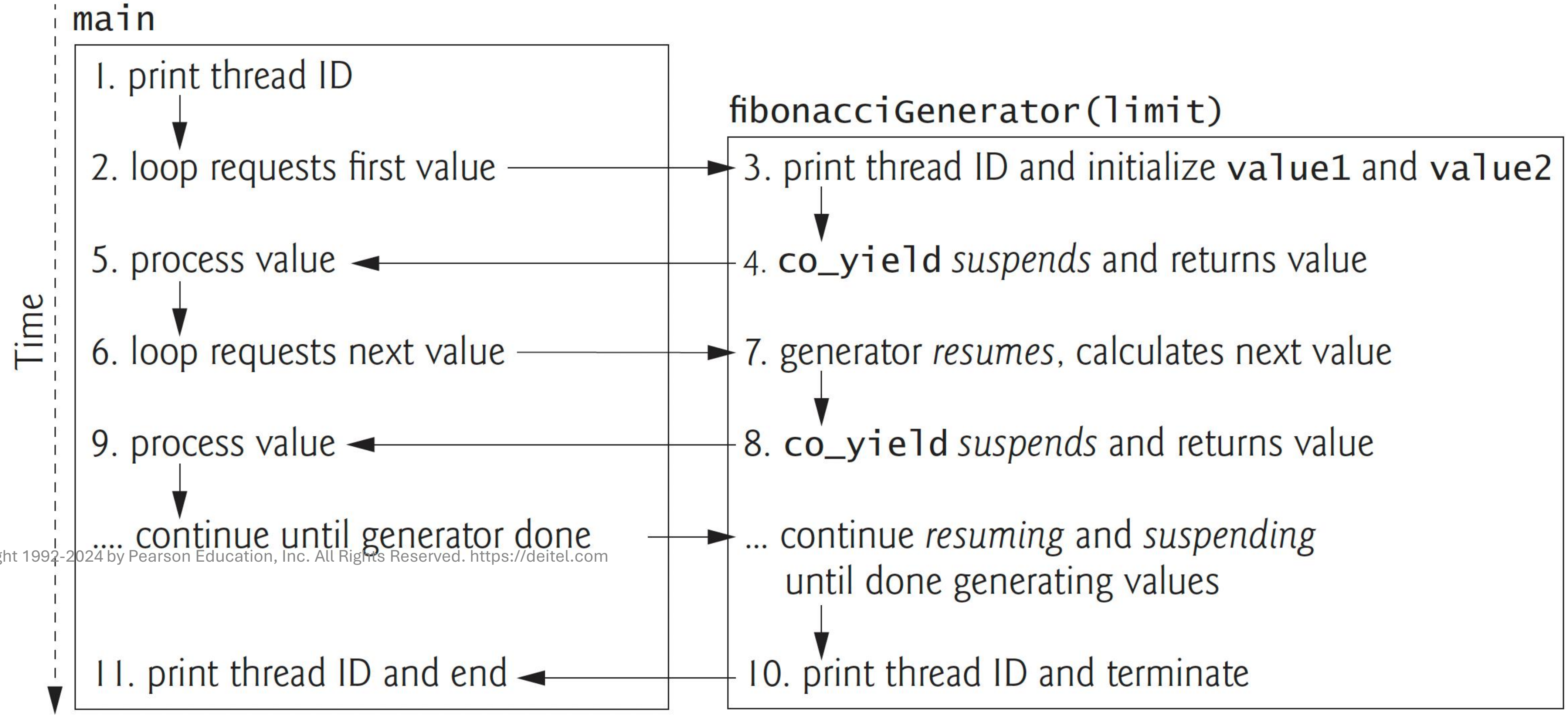
Coroutines

- What are coroutines used for?
    - They simplify implementing:
        - Generators
        - Asynchronous I/O
        - Lazy computations
        - Event driven applications

Coroutines

- C++20 contains language additions to support coroutines
- Standard Library does not yet include helper classes such as generators
- Visual C++ includes experimental helper classes, for example:
  - `std::experimental::generator<T>`

Creating a Generator Coroutine with `co_yield` and the `generator` Library—Diagram Showing the Flow of Control for a Generator Coroutine

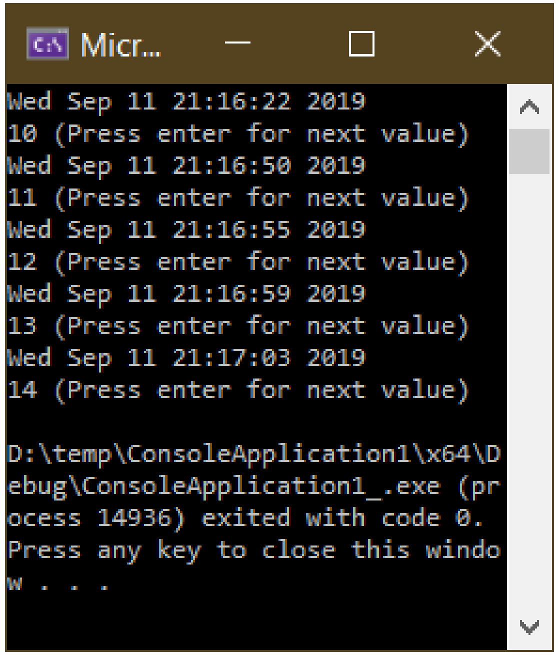


Coroutines

- Example (VC++):

```
experimental::generator<int> GetSequenceGenerator(
  int startValue, size_t numberOfValues)
{
  for (int i = startValue; i < startValue + numberOfValues; ++i) {
    time_t t = system_clock::to_time_t(system_clock::now());
    cout << std::ctime(&t);
    co_yield i;
  }
}

int main()
{
  auto gen = GetSequenceGenerator(10, 5);
  for (const auto& value : gen) {
    cout << value << " (Press enter for next value)" << endl;
    cin.ignore();
  }
}
```



Concepts

15.4 C++20 Concepts: A First Look

- Simplify generic programming
- Stroustrup:
  - “Concepts complete C++ templates as originally envisioned”
  - “dramatically improve your generic programming and make the current workarounds and low-level techniques feel like error-prone and tedious assembly programming.”

15.4 C++20 Concepts: A First Look

- 74 predefined concepts and can define your own
- Type’s requirements or relationships between types
- Test attributes of types
- Test whether types support various operations
- Can be applied to any parameter of any template and to any use of `auto`

15.4 C++20 Concepts: A First Look

- Traditionally, template requirements were implicit
- `printContainer` function template
  - ```
    template <typename T>
    void printContainer(const T& items) {
       for (const auto& item : items) {
          std::cout << item << " ";
       }
    }
    ```
- Argument must be iterable with range-based `for`
- Element type must support the `<<` operator
- Requirements typically would be documented in comments, but compiler cannot enforce comments
    ```

15.4 C++20 Concepts: A First Look

- **Concepts specify requirements explicitly in code**
- Compiler can determine that a type is not compatible with a template before instantiating it
  - Fewer, more precise error messages
  - Potential compile-time performance improvements
- Overload function templates with the **same signature**

15.4.2 Constrained Function Template with a C++20 Concepts `requires` Clause

- Each C++20 concept is a compile-time predicate expression that evaluates to `true`/`false`
- C++ Core Guidelines recommend
  - specify concepts for every template parameter
  - using standard's predefined concepts if possible
- `requires` clause + constraint expression
  - constrain `multiply`'s parameters to integer or floating-point type

15.4.2 Constrained Function Template with a C++20 Concepts `requires` Clause

Disjunctions and Conjunctions

- Logical OR (||) operator forms a disjunction
  - Either or both operands must be `true` for the compiler to instantiate the template
  - If both `false`, ignores the template as a potential match
- Logical AND (&&) operator forms a conjunction
  - both operands must be `true` for the compiler to instantiate the template

15.5 Type Traits

- C++11 introduced `<type_traits>`
    - test at compile-time whether types have various traits
    - generate template code based on those traits
- For example, test whether a type is
    - a fundamental type like `int` (`std::is_fundamental`)
    - a class type (`std::is_class`)
- Check whether type arguments satisfy a template's requirements
- Generate template code based on test results
- Performed at compile-time **during template instantiation**
    - Often leading to many cryptic error messages

15.5 Type Traits

C++20 Predefined Concepts Often Use Type Traits

- `std::integral` implemented using type trait `std::is_integral`
- `std::floating_point` implemented using type trait `std::is_floating_point`

15.6 C++20 Concepts: A Deeper Look—Creating a Custom Concept

- Concepts often aggregate multiple constraints, including other predefined concepts and type traits
- ```
  template<typename T>
  concept Numeric = std::integral<T> || std::floating_point<T>;
  ```
- Type parameter represents type to test
- Concepts with multiple type parameters can test relationships between types
  - E.g., `std::same_as` tests whether two type parameters have the same type
  ```

15.6 C++20 Concepts: A Deeper Look—Using a Concept

- Any concept can be placed in a `requires` clause following the template header
- Updated `multiply` function template
-
```
template<typename T>
   requires Numeric<T>
T multiply(T first, T second) {return first * second;}
```

15.6 C++20 Concepts: A Deeper Look—Using a Concept

`requires` clause function template's signature

- ```
  template<typename T>
  T multiply(T first, T second) requires Numeric<T> {
     return first * second;
  }
  ```
- Required
  - Member function defined in a class template's body does not have a `template` header
  - Need to use a function template's parameter names in a constraint, you must use a trailing requires clause, so the parameter names are in scope before the compiler evaluates the requires clause.
  ```

Many More New Features…

Designated Initializers

- Designated initialization

```
struct Data {
  int anInt = 0;
  std::string aString;
};

Data d{ .aString = "Hello" };
```

Spaceship Operator <=>

- Official name: ***three-way comparison operator***
- Three-way: comparing 2 objects and then comparing result with 0
  - (a <=> b) < 0 // true if a < b
  - (a <=> b) > 0 // true if a > b
  - (a <=> b) == 0 // true if a is equal/equivalent to b
- A bit like C `strcmp()` returning -1, 0, or 1

Spaceship Operator <=>

- **Common case**: automatically write all comparison operators to compare X with Y (memberwise):
  - `auto X::operator<=>(const Y&) = default;`

`partial_ordering`: allows incomparable values,
i.e. x < y, x > y, and x == y could all be false.

Spaceship Operator <=>

```
class Point {
  int x; int y;
public:
  friend bool operator==(const Point& a, const Point& b){ return a.x==b.x && a.y==b.y; }
  friend bool operator< (const Point& a, const Point& b){ return a.x < b.x ||
                                                          (a.x == b.x && a.y < b.y); }
  friend bool operator!=(const Point& a, const Point& b) { return !(a==b); }
  friend bool operator<=(const Point& a, const Point& b) { return !(b<a); }
  friend bool operator> (const Point& a, const Point& b) { return b<a; }
  friend bool operator>=(const Point& a, const Point& b) { return !(a<b); }
  // ... non-comparison functions ...
};
```

C++17

C++20

```
#include <compare>
class Point {
  int x; int y;
public:
  auto operator<=>(const Point&) const = default;
  // ... non-comparison functions ...
};
```

Spaceship Operator <=>

- Standard Library types include support for <=>
  - `vector`, `string`, `map`, `set`, `sub_match`, …
- Example:

```
namespace std {
  // [vector], class template vector
  template<class T, class Allocator = allocator<T>> class vector;

  template<class T, class Allocator>
    bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
- template<class T, class Allocator>
-   bool operator!=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
- template<class T, class Allocator>
-   bool operator< (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
- template<class T, class Allocator>
-   bool operator> (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
- template<class T, class Allocator>
-   bool operator<=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
- template<class T, class Allocator>
-   bool operator>=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
+ template<class T, class Allocator>
+   synth-three-way-result<T> operator<=>(const vector<T, Allocator>& x, const vector<T, Allocator>& y);

  [...]
}
```

Calendars & Timezones

- `<chrono>` is extended to support calendars and timezones
- Only Gregorian calendar is supported
    - Other calendars are easily added and can easily interoperate with `<chrono>`

Calendars & Timezones

- Creating a year:
  - `auto y1 = year{ 2019 };`
  - `auto y2 = 2019y;`
- Creating a month:
  - `auto m1 = month{ 9 };`
  - `auto m2 = September;`
- Creating a day:
  - `auto d1 = day{ 18 };`
  - `auto d2 = 18d;`

Calendars & Timezones

- Creating a full date:
  - `year_month_day fulldate1{ 2019y, September, 18d };`
  - `auto fulldate2 = 2019y / September / 18d;`
  - `year_month_day fulldate3{ Monday[3]/September/2019 };`

Calendars & Timezones

- New duration type aliases (similar to seconds, minutes, …)
  - `using days   = duration<`*`signed integer type of at least 25 bits`*`, ratio_multiply<ratio<24>, hours::period>>;`
  - `using weeks  = …;`
  - `using months = …;`
  - `using years  = …;`
- Example:

```
weeks w{ 1 }; // 1 week
days d{ w };  // Convert 1 week into days
```

Calendars & Timezones

- New clocks (besides system_clock, steady_clock, high_resolution_clock):
  - `utc_clock`: represents Coordinated Universal Time (UTC), measures time since 00:00:00 UTC, Thursday, 1 January 1970, including leap seconds
  - `tai_clock`: represents International Atomic Time (TAI), measures time since 00:00:00, 1 January 1958, and was offseted 10 seconds ahead of UTC at that date, it does not include leap seconds
  - `gps_clock`: represents Global Positioning System (GPS) time, measures time since 00:00:00, 6 January 1980 UTC, it does not include leap seconds
  - `file_clock`: alias for the clock used for `std::filesystem::file_time_type`, epoch is unspecified

Calendars & Timezones

- New `system_clock`-related type aliases
  - `template<class Duration>`
    `using sys_time = std::chrono::time_point<std::chrono::system_clock, Duration>;`
  - `using sys_seconds = sys_time<std::chrono::seconds>;`
  - `using sys_days = sys_time<std::chrono::days>;`
- Example:

```
system_clock::time_point t =
    sys_days{ 2019y / September / 18d }; // date -> time_point
auto yearmonthday =
    year_month_day{ floor<days>(t) };    // time_point -> date
```

Calendars & Timezones

- Date + Time:

```
auto t = sys_days{2019y/September/18d} + 9h + 35min + 10s; // 2019-09-18 09:35:10 UTC
```

- Timezone conversion:
  - Convert UTC to Denver time:

```
zoned_time denver = { "America/Denver", t };
```

  - Construct a local time in Denver:

```
auto t = zoned_time{ "America/Denver",
    local_days{Wednesday[3] / September / 2019} + 9h };
```

  - Get current local time:

```
auto t = zoned_time{ current_zone(), system_clock::now() };
```

- Output:

```
cout << t << endl;  // 2016-05-29 07:30:06.153
```

Text Formatting (std::format)

- Currently, two ways to format text in C++:
  - I/O streams
    - Recommended way, because of safety and extensibility
  - `printf()`
    - Not safe
    - Not extensible
    - Easier to read because no series of << insertion operators
    - Separation of the formatting string and the arguments

Text Formatting (std::format)

- New in C++20: `std::format()`
  - Safe
  - Extensible
  - Easy to read because no series of << insertion operators
  - Separation of the formatting string and the arguments
- Example:

```
std::string s = std::format("Hello CPP {} Team!", 2020);
```

Text Formatting (std::format)

- Goals:
  - Mini language focused on formatting (not type information)
  - Extensible (custom format strings for user-defined types)
  - Positional arguments
  - Locale-specific and locale-independent formatting
  - Better alignment control
  - …

Text Formatting (std::format)

- `printf()` can be translated almost automatically to `std::format()`

printf	new
-	<
+	+
space	*space*
#	#
0	0
hh	unused
h	unused
l	unused
ll	unused
j	unused
z	unused
t	unused
L	unused
c	c (optional)
s	s (optional)
d	d (optional)
i	d (optional)
o	o
x	x
X	X
u	d (optional)
f	f
F	F
e	e
E	E
a	a
A	A
g	g (optional)
G	G
n	unused
p	p (optional)

Text Formatting (std::format)

- `std::format()` supports the following alignments
  - Left: <
  - Centered: ^
  - Right: >
- Example

```
format("{:=^30}", “Hello C++ 2020"); // ========== Hello C++ 2020 =========
```

Text Formatting (std::format)

- Extensible for user-defined types
- User-provided functions for parsing and formatting
- Need to provide a specialization of `std::formatter<>` for your type and implement:
  - `formatter<>::parse()`
  - `formatter<>::format()`

Text Formatting (std::format)

- Positional arguments, useful for translated format strings
- Example:

```
format("String '{}' has {} characters.", str, str.length());
format("{1} karakters lang is de tekst '{0}'.", str, str.length());
```

Math Constants

- `<numbers>`
- Following mathematical constants are defined:
  - `e`, `log2e`, `log10e`
  - `pi`, `inv_pi`, `inv_sqrtpi`
  - `ln2`, `ln10`
  - `sqrt2`, `sqrt3`, `inv_sqrt3`
  - `egamma`
  - `phi`
- In `std::numbers`

std::source_location

- <source_location>
- Represents information about a specific location in a source code
  - line, column, file_name, function_name
- Construct one using `source_location::current()`
- Example:

```
void LogInfo(string_view info,
             const source_location& loc = source_location::current()) {
  cout << loc.file_name() << ":" << loc.line() << ": " << info << endl;
}

int main() {
  LogInfo("Welcome to BeCPP 2019!");
}
```