C++ 17 Language Features

Contents

a) Core Language Features

1. Structured Bindings

- Structured bindings allow you to unpack multiple values from a `tuple`, `pair`, `array`, or user-defined `struct`/`class` into separate variables in a simple way.
- Before C++17, if you had a function returning multiple values (like `std::pair` or `std::tuple`), you needed `std::tie` or manual extraction. Structured bindings make this easier.

```cpp
// Function returning a pair of int and string
std::pair<int, std::string> getData() {
    return { 1, "Alice" };
}

int main() {
    int id1;
    std::string name1;
    // Before C++17 // Manual unpacking
    std::tie(id1, name1) = getData();

    std::cout << "ID: " << id1 <<
        ", Name: " << name1 << '\n';

    // C++17: Structured binding
    auto [id2, name2] = getData();
    std::cout << "ID: " << id2 <<
        ", Name: " << name2 << '\n';
}
```

Key Takeaways

- Introduced in C++17 to simplify variable unpacking.
- Works with `tuple`s, `pair`s, `array`s, and `struct`s.
- Avoids manual `std::tie` or accessing elements with `.first`/`.second`.
- Improves readability and reduces boilerplate code.

2. Selection Statements with Initializer

- In C++17, you can **declare and initialize a variable directly inside an `if` or `switch` statement**. This makes the code **more readable** and **reduces scope pollution**.

```
int getValue() { return 10; }
char getGrade() { return 'B'; }

int main() {

   // Declare & initialize x inside if
   if (int x = getValue(); x > 5) {
      cout << "x is greater than 5\n";
   }

   // Declare & initialize inside switch
   switch (char grade = getGrade(); grade) {
   case 'A': cout << "Excellent\n"; break;
   case 'B': cout << "Good\n"; break;
   case 'C': cout << "Average\n"; break;
   default: cout << "Invalid Grade\n";
   }
}
```

Key Takeaways

- Introduced in C++17 to allow variable initialization directly in `if` and `switch`.
- Reduces variable scope, preventing accidental misuse.
- Improves code readability and structure.
- The initialized variable exists only inside `if` or `switch`.

3. Inline Variables

What Problem Did C++17 `inline` Variables Solve?

- Before C++17, when you declared **global variables in a header file**, you could run into **multiple definition errors** if that header was included in multiple source (`.cpp`) files.
- Let's say we have a header file `<config.h>`

```
#ifndef CONFIG_H
#define CONFIG_H

// Global variable definition
int globalValue = 10;

#endif
```

- And we include this in two different source files:

File1.cpp	File2.cpp

`#include "config.h"`
`void func1() {`
`  globalValue += 5;`
`}`
	`#include "config.h"`
	`void func2() {`
	`  globalValue *= 2;`
	`}`

- Now, when we **compile** both files independently, it will compile. After compilation, the **linker will complain** about multiple definitions of `globalValue`, because each `.cpp` file gets its own copy from the header file.

Why Does the Linker Complain About Multiple Definitions Even with Include Guards?

a. Understanding Include Guards

- Include guards like:

```
#ifndef CONFIG_H
#define CONFIG_H
...
#endif
```

- Prevent multiple inclusions of the same header file within a **single translation unit** (i.e., one `.cpp` file).
- Important:
  - Include guards do NOT prevent the header from being included in multiple `.cpp` files. **They only stop multiple inclusions within the same file**.

b. Compilation vs Linking: Understanding the Process

- When you compile a C++ program, it typically goes through two main phases:
- Compilation Phase
  - Each .`cpp` file is compiled independently into an object file (`.o` or `.obj`).
  - The compiler does not check for multiple definitions at this stage.
- Linking Phase
  - The linker takes all object files (`file1.o`, `file2.o`, etc.) and combines them into a final executable (`.exe`, `a.out`, etc.).
  - If multiple object files contain the same global variable definition, the linker will throw an error.

Solution in C++17: `inline` Variables

- C++17 introduced the `inline` **specifier for variables**, allowing a **single definition across multiple translation units** (source files).

```
#ifndef CONFIG_H
#define CONFIG_H

// Now it's an inline variable
inline int globalValue = 10;

#endif
```

- How Does inline Work?
  - Without `inline` → Every `.cpp` file gets a separate copy, causing multiple definitions.
  - With `inline` → The compiler ensures there is only one definition across all translation units.

4. Nested Namespace Definitions

- Before C++17, if you wanted to define **nested** `namespaces`, you had to write them in a **verbose** way:

```
namespace A {
   namespace B {
      namespace C {
         void sayHello() {
            std::cout << "Hello from A::B::C!";
         }
      } // namespace C
   } // namespace B
} // namespace A
```

- C++17 introduced nested `namespace` definitions, allowing us to define nested `namespaces` in a more compact way:

```
namespace A::B::C {
   void sayHello() {
      std::cout << "Hello from A::B::C!";
   }
}
```

5. The `__has_include` Preprocessor

- The `__has_include` is a preprocessor directive that checks whether a header file exists before including it. This prevents compilation errors due to missing headers.
- Benefits of `__has_include`
  - **Prevents compilation errors** if a header is missing.
  - **Provides fallback options** for different compilers or environments.
  - **Useful for cross-platform development** where some headers might be unavailable.

```
#if __has_include(<filesystem>)
#include <filesystem>
namespace fs = std::filesystem;
#else
#include <experimental/filesystem>
namespace fs = std::experimental::filesystem;
#endif
```

6. Compile-time lambdas

- A lambda function can now be `constexpr`, meaning: It can be evaluated at compile time if possible.

Why Was constexpr Lambda Introduced?

- Before C++17, lambdas could not be `constexpr`.
- Only normal functions could be `constexpr`.
- Example:

```
// Error: Lambda cannot be constexpr in C++14
constexpr auto square = [](int x) { return x * x; };
// Not allowed before C++17
constexpr int result = square(4);
```

In C++17, lambdas can be `constexpr` if:

- Their body **can be evaluated at compile time**.
- They **do not use runtime-only features**.

- Example:

```
int main() {
   // Define a constexpr lambda
   constexpr auto square = [](int x) { return x * x; };
   // Use it at compile-time
   constexpr int result = square(5);  // Computed at compile-time!
   std::cout << "Square of 5: " << result; // Output: 25
}
```

```
constexpr auto printMessage = []() {
   std::cout << "Hello World";
   };
//ERROR: std::cout is not a constexpr
constexpr void callMessage() {
   printMessage();
}
int main() {
   callMessage();  // Compile-time error
}
```

- `constexpr` **does not mean "always evaluated at compile-time"**, it just means "**can be evaluated at compile-time if used in a constant expression**".
- If you try to **evaluate `printMessage()` at compile-time**, the compiler **will** give an error.
- In the above example, we are forcing the **`printMessage()`** to be evaluated at compile time by calling it inside the `constexpr` **`callMessage()`** function. This `constexpr` **`callMessage()`** function will be evaluated at compile-time and hence the **`printMessage()`** function is forced to be evaluated at compile-time. Hence compiler gives an error message.

Summary Table

Feature	Before C++17	C++17 (`constexpr` Lambdas)
`constexpr` allowed?	❌ No, lambdas cannot be `constexpr`	✅ Yes, lambdas can be `constexpr`
Compile-time execution?	❌ Only with `constexpr` functions	✅ Possible with lambdas
Can mix runtime & compile-time?	❌ No	✅ Yes

7. Non-type template parameters with `auto`

What are Template Parameters?

- Before diving into **non-type template parameters**, let's recap what template parameters are in general. Templates are a powerful feature in C++ that allow you to write code that can work with different types without having to write separate versions for each type. Think of it as a blueprint for creating code.
- Template parameters are the "placeholders" in the template that you fill in with specific **types** or **values** when you instantiate the template.

Types of Template Parameters:

- There are two main kinds of template parameters:
  i. **Type Template Parameters**:
    - These are the most common. They represent **types**. You use the `typename` or `class` keyword to declare them.

```
template <typename T>  // T is a type template parameter
class MyContainer {
   T data; // Data of type T
};

MyContainer<int> int_container;   // Instantiate with int
MyContainer<double> double_container; // Instantiate with double
```

  ii. **Non-Type Template Parameters**:
    - These represent **values** or **constant expressions** rather than types. They allow you to make your templates even more flexible by parameterizing them with specific values.
    - Non-type template parameters can be integers, pointers, references, or even other templates. They are specified ***without*** `typename` or `class`.

```
// N is a non-type template parameter (int)
template <int N>
class MyArray {
   int data[N]; // Array of size N
public:
   constexpr int getSize() const {
      return N;
   }
};
```

```
int main() {
    MyArray<10> array10;  // Array of 10 ints
    MyArray<20> array20;  // Array of 20 ints

    static_assert(array10.getSize() == 10);
}
```

C++17 Feature: `auto` as a Template Parameter

- With C++17, we can now **use `auto` instead of a specific type** for non-type template parameters. This means the compiler will **deduce the type automatically**.

```
template <auto N>
struct Array {
    static constexpr auto value = N;
};

int main() {
    Array<10> a1;         // N is deduced as int
    Array<'A'> a2;        // N is deduced as char
    Array<3.14> a3;       // N is deduced as double

    cout << a1.value << endl;  // 10
    cout << a2.value << endl;  // A
    cout << a3.value << endl;  // 3.14
}
```

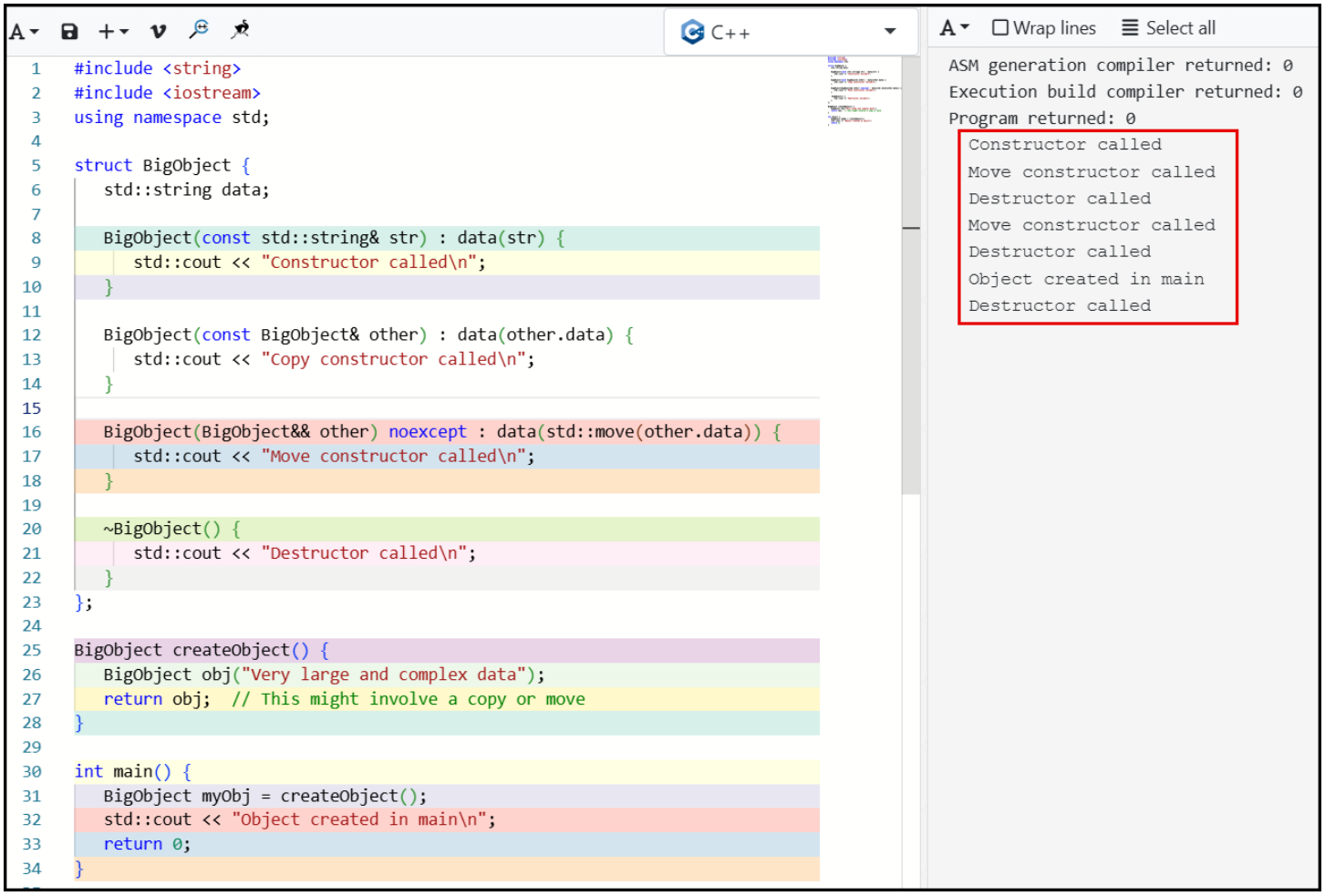
8. Guaranteed Copy Elision

What is Copy Elision?

- Copy elision is an **optimization technique** that allows the compiler to **eliminate unnecessary copies or moves** when objects are returned from functions or passed around. This improves performance and avoids unnecessary overhead.
- Before C++17, compilers **could** perform copy elision, **but it was not mandatory**. Some copies or moves could still happen depending on the compiler's behaviour.
- Imagine you have a function that creates a complex object and returns it. A naive implementation might involve creating a temporary copy of that object when returning it. This copying can be expensive, especially for large objects. Consider this simplified example:

```cpp
#include <string>
#include <iostream>
using namespace std;

struct BigObject {
   std::string data;

   BigObject(const std::string& str) : data(str) {
      std::cout << "Constructor called\n";
   }

   BigObject(const BigObject& other) : data(other.data) {
      std::cout << "Copy constructor called\n";
   }

   BigObject(BigObject&& other) noexcept : data(std::move(other.data)) {
      std::cout << "Move constructor called\n";
   }

   ~BigObject() {
      std::cout << "Destructor called\n";
   }
};

BigObject createObject() {
   BigObject obj("Very large and complex data");
   return obj;  // This might involve a copy or move
}

int main() {
   BigObject myObj = createObject();
   std::cout << "Object created in main\n";
   return 0;
}
```

- In older versions of C++, the `return obj;` line in `createObject()` could have triggered a copy or move constructor to create a temporary object, which was then used to initialize `myObj`.
- This is inefficient. Compilers were allowed to optimize this away (called copy elision or return value optimization (RVO)), **but it wasn't guaranteed**. You might see the copy/move constructor being called.



Summary of Object Lifecycle

Event	Object	Action
1	`obj` (inside `createObject()`)	**Constructor called**
2	Temporary object	**Move constructor called** (`obj` → `temporary`)
3	`obj` destroyed	**Destructor called**
4	`myObj` (inside `main()`)	**Move constructor called** (`temporary` → `myObj`)
5	Temporary object destroyed	**Destructor called**
6	Execution of `main()`	`"Object created in main"`
7	`myObj` destroyed	**Destructor called**

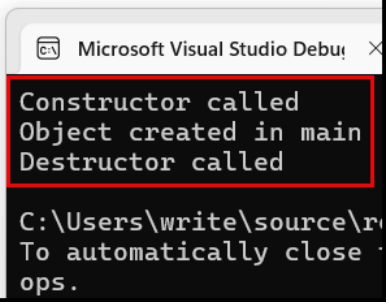
What Changed in C++17?

- In C++17, Guaranteed Copy Elision was introduced. This means that in certain situations, the compiler **must eliminate copies or moves**—even if the copy/move constructor has side effects (like printing something when an object is copied).

When is Copy Elision Guaranteed?

- Copy elision is mandatory in the following cases:
    1. Returning a temporary object from a function (above example) / Returning a prvalue

```
BigObject createObject() {
    BigObject obj("Very large and complex data");
    return obj;  // This might involve a copy or move
}

int main() {
    BigObject myObj = createObject();
    std::cout << "Object created in main\n";
    return 0;
}
```



    2. Initializing an object with a prvalue of the same type

```
struct Error {
    Error() { cout << "Ctor called\n"; }
    Error(const Error&) { cout << "Copy Ctor called\n"; }
};
void test() {
    throw Error();
}
int main() {
    try { test();}
    catch (Error e) {
    }
}
// Ctor called
// Copy Ctor called
```

- Throwing an exception by value (`throw Error();`), (compiler may create a copy).
- Catching an exception by value (`catch (Error e)`), Copy Elision Guaranteed (since C++17).

Why is Copy Elision Important?

- **Better Performance** – Eliminates unnecessary copies, making code more efficient.
- **More Predictable Behaviour** – You can rely on the fact that no copies/moves will happen in these cases.
- **No Need for Move Constructor** – In many cases, you don't need to define a move constructor, since the compiler can optimize object creation.

What If the Copy Constructor Has Side Effects?

- Before C++17, if you had a copy constructor like this:

```
Example(const Example&) {
   std::cout << "Copy Constructor called\n";
}
```

- It might get called in some cases. **After C++17, even if the copy constructor prints something, it will not be called in the guaranteed copy elision cases.**

9. New Attributes

- In C++17, few more **attributes** were introduced to provide additional information to the compiler and improve code quality.
- They do not change the program's logic but help the compiler enforce better practices or optimize the code.

Prevents Accidental `fallthrough` in Switch Statements

- Problem:
  - In a `switch` statement, if you forget to add a `break;`, execution will continue to the next `case`, which can lead to unintended behaviour.
- Solution:
  - Use `[[fallthrough]]` to explicitly indicate that falling through to the next `case` is intentional.
- **Without `[[fallthrough]]`**, the compiler might generate a warning for missing `break;`, thinking it's a mistake. The attribute makes it clear that we **intentionally** allow execution to continue.

```
void checkNumber(int num) {
    switch (num) {
    case 1:
        std::cout << "Case 1\n";
        // Intentional fallthrough to the next case
        [[fallthrough]];
    case 2:
        std::cout << "Case 2\n";
        break;
    case 3:
        std::cout << "Case 3\n";
        break;
    default:
        std::cout << "Default case\n";
    }
}

int main() {
    checkNumber(1);  // Output: Case 1, Case 2
    return 0;
}
```

Prevents Ignoring Return Values: `[[nodiscard]]`

- Problem:
    - Sometimes, a function `return`s a value that should not be ignored, such as an error code. If the programmer forgets to use the returned value, it can lead to unnoticed bugs.
- Solution:
    - Mark the function with `[[nodiscard]]` to warn the user if the `return` value is ignored.

```
[[nodiscard]] int computeSum(int a, int b) {
    return a + b;
}

int main() {
    // Warning: return value is ignored
    computeSum(5, 10);
    return 0;
}
```

Prevents Unused Variable Warnings: `[[maybe_unused]]`

- Problem:
  - Sometimes, we declare a variable but don't use it immediately (e.g., debugging, future use, conditional compilation). The compiler may give a warning about the unused variable.
- Solution:
  - Use `[[maybe_unused]]` to tell the compiler that a variable may be unused intentionally.

```
void testFunction() {
   // No warning even if unused
   [[maybe_unused]] int debugVar = 42;

   cout << "This function does something else.\n";
}

int main() {
   testFunction();
   return 0;
}
```

- **Without** `[[maybe_unused]]`, the compiler may give a warning about `debugVar` not being used. This attribute is useful when writing flexible code that might conditionally use variables.

Summary Table

Attribute	Purpose	Example Usage
`[[fallthrough]]`	Avoids compiler warnings for intentional `switch` fallthrough	Inside a `case` in `switch` statements
`[[nodiscard]]`	Warns when `return` value is ignored	On function `return` values that must be used
`[[maybe_unused]]`	Suppresses unused variable warnings	For debugging or conditionally used variables

10. UTF-8 String Literals

- **UTF-8 string literals** (using the `u8` prefix) were introduced in **C++11**, but C++17 made changes related to UTF-8 support.

What is UTF-8?

- UTF-8 is a way to encode characters from any language (e.g., English, Chinese, Arabic, emojis) using 1 to 4 bytes. It is the most widely used character encoding on the internet.

What are UTF-8 String Literals?

- UTF-8 string literals allow you to represent Unicode text in a portable way. In C++17, you can write a UTF-8 string using the `u8` prefix:

```
const char* utf8_text = u8"Hello, 世界! 😊";
```

  - This ensures that the string is encoded in **UTF-8**.

11. Hexadecimal Floating-Point Literals

What's the Goal?

- Sometimes, it's useful to represent floating-point numbers in hexadecimal (base-16) notation.
- This can be helpful for **low-level programming**, **working with hardware**, or when you need very **precise control** over the binary representation of a floating-point value.
- **Hexadecimal Numbers**: Hexadecimal numbers use `16` digits: `0–9` and `A–F` (or `a–f`). Each hexadecimal digit represents 4 bits.
- **Floating-Point Numbers**: Floating-point numbers are numbers with a fractional part (e.g., `3.14`, `2.718`, `1.0e–5`). They are stored in a specific format (usually IEEE 754) that involves a mantissa (the significant digits) and an exponent.

Hexadecimal Floating-Point Literals

- They allow floating-point numbers to be expressed in base-16 (hexadecimal) notation with an explicit binary exponent (base 2).

Syntax

- A hexadecimal floating-point literal has the following format:

  `0x<hex_digits>.<hex_digits>p<exponent>`

  - The literal must start with `0x` or `0X` (indicating hexadecimal).
  - The exponent part uses `p` or `P` (instead of `e` for decimal floating-point literals).
  - The exponent is written in decimal but represents a **power of 2** (not `10`).
  - If there is no fractional part, the decimal point can be omitted.

```
int main() {
   double a = 0x1.8p2;  // 1.8 (hex) × 2^2  = 1.5 × 4 = 6.0
   double b = 0x1p4;    // 1.0 (hex) × 2^4  = 16.0
   double c = 0x0.4p2;  // 0.4 (hex) × 2^2  = 0.25 × 4 = 1.0
   double d = 0xAp-1;   // A (hex) × 2^-1   = 10 × 0.5 = 5.0

   cout << "a: " << a << "\n"; // 6.0
   cout << "b: " << b << "\n"; // 16.0
   cout << "c: " << c << "\n"; // 1.0
   cout << "d: " << d << "\n"; // 5.0

   return 0;
}
```

12. Simplified `static_assert`

- What Changed?
  - Prior to C++17, static_assert required two arguments:

```
static_assert(condition, "Error message");  // C++11, C++14
```

- In C++17, the second argument (`message`) became optional:

```
static_assert(condition);  // C++17+
```

- Why This Change?
  - The message was often redundant because the failed assertion already provides **sufficient information**.
  - It simplifies the syntax when an explanatory message isn't necessary.

13. Fold Expressions

- **Fold Expressions** were introduced in **C++17** to simplify the expansion of **variadic templates** when applying **binary operators** to a **pack of template arguments**.

Variadic Templates

- A variadic template is a template that can take a variable (unknown) number of arguments. This is useful when you don't know how many arguments you will need in advance.
- Example of a Normal Template
  - A regular function template might look like this:

```
template <typename T>
void print(T value) {
   cout << value << endl;
}
```

  - This works fine if you want to print one value, but what if you want to print multiple values?

Parameter Pack

- A **parameter pack** is used in variadic templates to **represent multiple template parameters**.

```
// Base Case
void print() { cout << endl; }

template <typename First, typename... Rest>
void print(First first, Rest... rest) {
   cout << first << " ";
      print(rest...); // Expands the parameter pack

}
int main() {
   print(1, 2, 3, 4);
}
```

- Here:
  - `First` represents the first argument.
  - `Rest...` (parameter pack) represents the remaining arguments.
  - `print(rest...);` expands the parameter pack by calling `print()` recursively with the remaining arguments.

How It Works

- Initial Function Call:

```
print(1, 2, 3, 4);
```

Summary

Function Call	First Type	first Value	Rest... Values	typename... for Rest...
print(1, 2, 3, 4);	int	1	{2, 3, 4}	{int, int, int}
print(2, 3, 4);	int	2	{3, 4}	{int, int}
print(3, 4);	int	3	{4}	{int}
print(4);	int	4	{} (empty)	{} (empty pack)
print();	(Base Case)	(None)	(None)	(None)

Fold Expressions (C++17)

- Writing recursive variadic templates can be cumbersome. **C++17 introduced Fold Expressions**, which simplify operations on **parameter packs**.

What is a Fold Expression?

- A **fold expression** applies a **binary operator** (like +, *, &&, ||, ,) to all elements of a parameter pack **without recursion**.
- Example:

```
template <typename... Args>
int sum(Args... args) {
   return (args + ...);
}

int main() {
   cout << sum(1, 2, 3, 4) << "\n";
}// Output: 10
```

- Expands to:

```
((1 + 2) + 3) + 4
```

Types of Fold Expressions

1. Unary Right Fold: (`... op pack`)

```
template <typename... Args>
auto sum1(Args... args) {
   // Unary right fold
   return (... + args);
}
```

- Expansion:

Function Call	Expansion
`sum1(1, 2, 3, 4)`	`(... + 1, 2, 3, 4)`
Expands to	`(1 + (2 + (3 + 4)))`

- There are **no separate method calls** like in a recursive variadic function. Instead, the **compiler directly evaluates the expression** following operator precedence from the innermost bracket first.

2. Unary Left Fold: (`pack op ...`)

```
template <typename... Args>
auto sum2(Args... args) {
   // Unary left fold
   return (args + ...);
}
```

- Expansion:

Function Call	Expansion
`Sum2(1, 2, 3, 4)`	`(1, 2, 3, 4 + ...)`
Expands to	`(((1 + 2) + 3) + 4)`

3. Binary Right Fold: (`pack op ... op init`)

```
template <typename... Args>
auto sum3(Args... args) {
   // Binary right fold with identity element 0
   return (args + ... + 0);
}
```

- Expansion:

Function Call	Expansion
`Sum3(1, 2, 3, 4)`	`(1, 2, 3, 4 + ... + 0)`
Expands to	`1 + (2 + (3 + (4 + 0)))`

4. Binary Left Fold: (`init op ... op pack`)

```
template <typename... Args>
auto sum4(Args... args) {
   // Binary left fold with identity element 0
   return (0 + ... + args);
}
```

- Expansion:

Function Call	Expansion
`Sum4(1, 2, 3, 4)`	`(0 + ... + 1, 2, 3, 4)`
Expands to	`(((0 + 1) + 2) + 3) + 4)`

- Example:

```
template <typename... Args>
auto sum1(Args... args) {
   // Unary right fold
   return (... + args);
}

template <typename... Args>
auto sum2(Args... args) {
   // Unary left fold
   return (args + ...);
}

template <typename... Args>
auto sum3(Args... args) {
   // Binary right fold with identity element 0
   return (args + ... + 0);
}

template <typename... Args>
auto sum4(Args... args) {
   // Binary left fold with identity element 0
   return (0 + ... + args);
```

```
}

template <typename... Args>
void print(Args... args) {
   (cout << ... << args) << endl;
}
int main() {
   cout << "sum1: " << sum1(1, 2, 3, 4) << "\n";
   cout << "sum2: " << sum2(1, 2, 3, 4) << "\n";
   cout << "sum3: " << sum3(1, 2, 3, 4) << "\n";
   cout << "sum4: " << sum4(1, 2, 3, 4) << "\n";
   print("Hello ", 1, " World ", 3.14);
}
/*
sum1: 10
sum2: 10
sum3: 10
sum4: 10
Hello 1 World 3.14
*/
```

14. Template Argument Deduction for Class Templates

- Before C++17, when using `class templates`, you always had to specify the **template parameters explicitly**.
- **C++17 allows the compiler to deduce the template parameters automatically** in certain cases, just like it does for function templates.

```
template <typename T1, typename T2>
class Pair {
public:
   T1 first;
   T2 second;
   Pair(T1 a, T2 b) : first(a), second(b) {}
};
int main()
{
   // Before C++17, you must specify the type in the template
   Pair<int, double> p1(10, 5.5);
   Pair<string, int> p2("Hello", 5);

   // C++17 allows you to omit the type in the template
   Pair p3(42, 3.14);                // Deduces Pair<int, double>
   Pair p4(string("Hello"), 100);    // Deduces Pair<string, int>
   return 0;
}
```

15. New rules for `auto` deduction from `std::initializer_list`

- Before C++17, **when you used `auto` with `{}` (braced-init-list), the behavior was confusing**. It sometimes led to `std::initializer_list`, even when you expected something else. C++17 **changed the rules to make it more intuitive**.

Before C++17 (`auto` preferred `std::initializer_list`)

- Let's look at an example:

```
auto x = { 1, 2, 3 };  // What is x?
```

- In C++11 and C++14, `x` was not an `std::vector<int>` or `std::array<int, 3>`.
- Instead, `x` was deduced as: `std::initializer_list<int>`.

C++17 Changed the Rule!

- Starting in C++17, `auto` now follows direct list initialization rules, meaning it does not deduce `std::initializer_list` unless explicitly specified.
- Example 1: Single Value in `{}`
  - `auto a = {10}; // What is 'a' in C++17?`
  - Before C++17: `std::initializer_list<int>`
  - Since C++17: `std::initializer_list<int>`
- Example 2: Using auto with Direct Assignment
  - `auto b{10}; // What is 'b'?`
  - Before C++17: `std::initializer_list<int>`
  - Since C++17: `int` (because `{10}` is treated as direct initialization)
- Example 3: Explicit `std::initializer_list`
  - `auto c = {10, 20, 30}; // What is 'c'?`
  - Still `std::initializer_list<int>` in C++17, because multiple elements inside `{}` cannot be a single `int`.

16. Lambda capture *this by value

- A **lambda function** can capture variables from its surrounding scope.
- Before C++17, capturing `this` only worked **by reference**, meaning any changes to the object after the lambda was created would be reflected inside the lambda.

Before C++17: Only `this` Capture (by Reference)

- Before C++17, **capturing `this`** in a lambda captured the **pointer to the object**, not a copy.
- Capturing `*this` used to give compilation error.

```
class Person {
public:
   std::string name;
   Person(std::string n) : name(n) {}
   auto introduce() {
      // return [*this]() { -> Error before C++17
      return [this]() {  // Capturing 'this' (pointer)
         std::cout << "Hello, I am " << name << endl;
         };
   }
};
int main() {
   auto p = Person("Alice");
   auto greet = p.introduce(); // Capture lambda
   p.name = "Bob";  // Modify original object
   greet();  // Output: "Hello, I am Bob"
}
```

C++17: Capture `*this` by Value

- In C++17, we can capture the object by value using `[*this]`.
- This makes the lambda store a copy of the object, so changes to the original don't affect the lambda.

```
class Person {
public:
   std::string name;
   Person(std::string n) : name(n) {}
   auto introduce() {
      return [*this]() {  // Capturing '*this' (copy)
         cout << "Hello, I am " << name << endl;
         };
   }
};
int main() {
   auto p = Person("Alice");
   auto greet = p.introduce(); // Capture a copy
   p.name = "Bob";  // Modify original object
   greet();  // Output: "Hello, I am Alice"
}
```

17. The `constexpr if`

- Before C++17, writing compile-time conditional code in C++ required complex **template specialization** or **SFINAE (Substitution Failure Is Not An Error)** tricks.
- **C++17 introduced `constexpr if` to simplify compile-time decisions.**

What is constexpr if?

- `constexpr if` allows you to conditionally compile only the relevant branch of an `if` statement based on a compile-time condition.

Why is constexpr if Useful?

- Avoids unnecessary template specializations.
- Doesn't compile the unused branch.
- Makes compile-time decisions clearer and simpler.

Before C++17: The Old Way (SFINAE)

- Before `constexpr if`, you had to use **SFINAE with** `enable_if`, which was more complex:

```
template <typename T, enable_if_t<is_integral<T>::value, int> = 0>
void printType(T value) {
   cout << value << " is an integer\n";
}

template <typename T, enable_if_t<!is_integral<T>::value, int> = 0>
void printType(T value) {
   cout << value << " is NOT an integer\n";
}

int main() {
   printType(42);
   printType(3.14);
}
```

- What is **SFINAE**?
    - **SFINAE** is a rule in C++ template metaprogramming that allows the compiler to discard invalid template instantiations during overload resolution. In simpler terms, if the compiler tries to create a specific version of a template and that version doesn't make sense (it results in an error), the compiler

doesn't stop the whole compilation process. Instead, it just ignores that version of the template and looks for other, valid versions.

- How is **SFINAE** Used in above Code?
    - Code uses `std::enable_if_t` along with template parameters to implement **SFINAE**. Let's analyse the first `printType` function:

```
template <typename T, enable_if_t<is_integral<T>::value, int> = 0>
void printType(T value) {
   cout << value << " is an integer\n";
}
```

    - **Template Parameters**: The function has two template parameters:
        - `T` (the type of the value)
        - Defaulted template parameter using `std::enable_if_t`.
    - The SFINAE Trick: Let's say we call `printType(42);`. `T` is `int`.
    - `is_integral<T>::value` is `true`. Therefore `enable_if_t <true, int>` becomes `int`. So, the template parameter has a valid type.
    - Now, let's say you call `printType(3.14)`. `T` is `double`. `std::is_integral<double>::value` is `false`. Therefore, `std::enable_if_t<false, int>` becomes `void`.
    - If we make the second template parameter as a non-deduced context (like making it a function argument), the compiler will try to create a version of the template with the `void` type. Since `void` is not a valid type for a non-type `template` parameter in this context, the compiler doesn't generate an error. Instead, it says, "This version of the template doesn't work," and discards it. It does not generate an error!

New Way

- Unused branches are completely discarded.

```
template <typename T>
void printType(T value) {
   // If T is an integer type
   if constexpr (std::is_integral<T>::value) {
      std::cout << value << " is an integer\n";
   }
   else {
      std::cout << value << " is NOT an integer\n";
```

```
    }
}

int main() {
    printType(42);        // Output: 42 is an integer
    printType(3.14);      // Output: 3.14 is NOT an integer
}
```

18. Direct-List Initialization of Enums

- Before C++17, initializing strongly-typed `enum`s (`enum class`) required explicit casting.
- C++17 introduced direct-list initialization `{}` for `enum`s, making initialization safer and more convenient.

Summary Table (C++17 Only)

Initialization	Scoped Enum (`enum class`)	Unscoped Enum (`enum`)
`Color c1 = Color::Red;`	✅ Allowed	✅ Allowed
`Color c2 = static_cast<Color>(1);`	✅ Allowed	✅ Allowed
`Color c3{Color::Red};`	✅ Allowed	✅ Allowed
`Color c4{1};`	❌ Error in C++17	✅ **NEW in C++17**
`Color c5 = 1;`	❌ Error in C++17	✅ Allowed (implicit)

19. Standardization of `std::uncaught_exceptions`

What is `std::uncaught_exception` (Before C++17)?

- Before C++17, there was a function called `std::uncaught_exception()`. It returned `true` if an `exception` was currently being handled (i.e., it had been `thrown`, but no `catch` block had yet handled it).
- This function was useful for cleanup operations in destructors or other places where you needed to know if an `exception` was in flight.

The Problem (Before C++17):

- The problem was that `std::uncaught_exception()` could only tell you if any `exception` was uncaught. It couldn't tell you how many uncaught exceptions there were. This was important in situations involving nested exceptions or exceptions thrown during stack unwinding.

```
struct Test {
   ~Test() {
      // Only tells if *an* exception is active
      if (uncaught_exception()) {
         cout << "Destructor called during exception handling!\n";
      }
   }
};

void func() {
   Test t;
   throw runtime_error("Error!"); // Exception thrown
}

int main() {
   try {
      func();
   }
   catch (...) {
      cout << "Exception caught!\n";
   }
   return 0;
}

// Destructor called during exception handling!
// Exception caught!
```

- **Problem:**
  - This approach **only tells if an exception is active**, but doesn't tell how many exceptions are in progress.

C++17 Code Using `std::uncaught_exceptions()`

- Tells exactly how many exceptions are active.
- Useful in RAII objects (like destructors) that behave differently during exception handling.

- Helps in nested exception handling scenarios.

```
struct Test {
   ~Test() {
      std::cout << "Uncaught exceptions count: "
         << std::uncaught_exceptions() << "\n";
   }
};

void func() {
   Test t;
   throw std::runtime_error("Error!"); // Exception thrown
}

int main() {
   try {
      func();
   }
   catch (...) {
      std::cout << "Exception caught!\n";
   }
   return 0;
}
/*
Uncaught exceptions count: 1
Exception caught!
*/
```

20. Aggregate initialization with inheritance

What are Aggregates?

- An aggregate is a simple data structure. In C++, it's one of the following:
  - An array.
  - A struct or class with:
    - No user-declared constructors (except for `defaulted` or `deleted` constructors).
    - No `private` or `protected` non-static data members.
    - No `virtual` functions.
    - No `virtual`, `private`, or `protected` base `classes`.

What is Aggregate Initialization?

- Aggregate initialization is a way to initialize aggregates using curly braces `{}`.
- We provide a list of values within the braces, and these values are used to initialize the members of the aggregate **in the order they are declared**.

```
struct Point {
   int x;
   int y;
};

Point p{ 10, 20 }; // Aggregate initialization
```

- Here, `p` is initialized with `x = 10` and `y = 20`.

What Changed in C++17?

- **Before** C++17, aggregate **initialization did not work** with `class`es that had base `class`es (inheritance).
- In C++17, this was changed to allow aggregate initialization to be used with `class`es that inherit from other `class`es.

```
struct Base {
   int baseValue;
};

struct Derived : Base {
   int derivedValue;
};

int main() {
   Derived d = { 10, 20 };  // Aggregate initialization
   cout << "Base value: "
      << d.baseValue << "\n";    // Output: Base value: 10
   cout << "Derived value: "
      << d.derivedValue << "\n"; // Output: Derived value: 20

   return 0;
}
```

21.Inherited constructors

Delegating Constructors

- Introduced in C++11
- Allows a constructor to call another constructor within the same class.
- Avoids code duplication when multiple constructors share initialization logic.

```
class Example {
   int x, y;
public:
   Example(int a, int b) : x(a), y(b) { cout << "Main constructor\n"; }
   Example(int a) : Example(a, 0) { cout << "Delegating constructor\n"; }
};

int main() {
   Example e(10); // Calls Example(int) → Example(int, int)
}
```

Inherited Constructors

- Introduced in C++11 (Basic version)
- Improved in C++14 (Fixed default argument inheritance)
- Enhanced in C++17 (Handled overload resolution and deleted constructors properly)

C++11 Version (Basic Constructor Inheritance)

```
class Base {
public:
   Base(int x) { cout << "Base constructor: "
      << x << endl; }
};

class Derived : public Base {
public:
   // Inherits constructors from Base
   using Base::Base;
};

int main() {
   Derived d(10); // Calls Base(int)
}
```

- **Problem in C++11:** Default arguments from base class constructors were **not inherited**.

C++14 Fix (Inherited Default Arguments)

```
class Base {
public:
   Base(int x = 42) { cout << "Base constructor: "
      << x << endl; }
};

class Derived : public Base {
public:
   // Now inherits default argument
   using Base::Base;
};

int main() {
   Derived d; // Calls Base(42), works in C++14
}
```

C++17 Fix (Overload Resolution and Deleted Constructors)

```
class Base {
public:
   Base(int) { cout << "Base(int) called" << endl; }
   Base(double) = delete;  // Deleted constructor
};

class Derived : public Base {
public:
   using Base::Base;
};

int main() {
   Derived d(10);    // Works
   // Derived d2(5.5); // ERROR in C++17: Base(double) is deleted
}
```

22. Non-Static Data Member Initialization of Aggregates

- We saw what are Aggregates here.
- Before **C++17**, aggregate members **could not** have default initializers.
- **C++17** now allows **non-static data members** to be initialized directly in the `class`.

```
struct Point {
    int x = 10;  // Default value
    int y = 20;  // Default value
};

int main() {
    Point p1;         // Uses defaults: x = 10, y = 20
    Point p2 = { 30 }; // Overrides x, but y = 20
    Point p3 = { 5, 15 }; // Overrides both x and y
}
```

- If no value is provided, the default member initializer is used.
- If a value is provided during initialization, it overrides the default.

b) Library Features

23.The `std::variant`

- `std::variant` is a **type-safe alternative to** `union` introduced in C++17.
- It allows a variable to hold **one value at a time from multiple possible types**, similar to how a `union` works but with **type safety** and **more features**.

Using `union` (Unsafe)

```
union Data {
   int i;
   double d;
};

int main() {
   Data data;
   data.i = 42;
   std::cout << data.i << "\n"; // OK

   data.d = 3.14;
   // Undefined Behaviour(data.i is now overwritten)
   std::cout << data.i << "\n";
}
```

- **Problem**: `union` does not track **which type is currently stored**, so accessing the wrong type leads to **undefined behaviour**.

Using `std::variant`

```
#include <variant> // C++17 feature

int main() {
   // Can store either int or double
   variant<int, double> data;

   data = 42;
   cout << get<int>(data) << "\n"; // 42

   data = 3.14;
   // Safe, automatically tracks type
   cout << get<double>(data) << "\n"; // 3.14
}
```

Accessing Values

```cpp
#include <variant> // C++17 feature

int main() {
   // Can store either int or double
   variant<int, double> data;

   data = 42;
   // Throws std::bad_variant_access if wrong type
   std::cout << std::get<int>(data) << "\n";

   if (auto p = std::get_if<int>(&data)) {
      std::cout << "Integer: " << *p << "\n";
   }

   data = 3.14;
   if (auto p = std::get_if<double>(&data)) {
      std::cout << "Double: " << *p << "\n";
   }
   std::visit([](auto&& val) {
      std::cout << "Value: " << val << "\n";
      }, data);

   std::cout << "Current index: " << data.index() << "\n";
}

/*
42
Integer: 42
Double: 3.14
Value: 3.14
Current index: 1
*/
```

24. The `std::optional`

- The `std::optional` is a **wrapper type** introduced in C++17 that represents **an optional (or nullable) value**.
- It helps handle cases where a function might return **either a valid value or nothing** but, in a **type-safe** way.

- Example 1:

```cpp
#include <optional> // C++17 feature

std::optional<int> findNumber(bool found) {
   return found ?
      std::optional<int>(42)
      : std::nullopt; // Safe alternative to nullptr
}

int main() {
   std::optional<int> result = findNumber(false);
   if (result) {
      std::cout << *result << "\n";
   }
   else {
      std::cout << "Not found\n";
   }
} // Not Found
```

- Example 2:

```cpp
#include <optional> // C++17 feature

optional<string> getUsername(bool loggedIn) {
   return loggedIn ?
      optional<string>("Alice") :
      nullopt;
}

int main() {
   auto username = getUsername(false);
   cout << "User: "
      << username.value_or("Guest") << "\n";
}
// Output: User: Guest
```

Summary

- `std::optional<T>` represents an optional value (can be empty or hold a value).
- Safer than raw pointers (no null dereferencing issues).
- Use `value_or()` to provide default values.
- No heap allocation (unlike `std::unique_ptr`).
- Best for return values where a result may be missing.

25. The `std::string_view`

The Problem: Passing Strings Efficiently

- In C++, strings are often represented by `std::string`. When you pass a `std::string` to a function, it often involves copying the entire `string`, which can be expensive, especially for long `string`s. Sometimes, you just need to look at the string; you don't need to modify it. Copying it in that case is wasteful.

```
// Takes a copy (expensive)
void printString(std::string s) {
   std::cout << s << "\n";
}
// Prone to dangling pointers
void printString(const char* s) {
   std::cout << s << "\n";
}

int main() {
   std::string name = "Alice";
   // Copies the entire string
   printString(name);
   // No copy, but prone to dangling pointers
   printString(name.c_str());
}
```

Solution: `std::string_view`

- The `std::string_view` is a lightweight, non-owning "view/reference" into a `string`.
- Think of it as a way to look at a `string` without making a copy. It's like borrowing a book instead of buying it – you can read the book (view the `string`), but you don't own it (you don't copy the `string`'s data).

```
#include <string_view>
// No copying!
void printString(std::string_view s) {
   std::cout << s << "\n";
}

int main() {
   std::string name = "Alice";
   printString(name); // OK
   printString("Bob");// OK
}
```

Benefits of `std::string_view`

Feature	`std::string`	`const char*`	`std::string_view`
Stores actual string	✅ Yes	❌ No	❌ No
Avoids copy overhead	❌ No	✅ Yes	✅ Yes
Works with string literals	❌ No	✅ Yes	✅ Yes
Works with `std::string`	✅ Yes	❌ No	✅ Yes
Safer than raw pointers	✅ Yes	❌ No	✅ Yes

26. The `std::filesystem`

The Problem: Working with Files and Directories

- Before C++17, working with files and directories in C++ was a bit of a mess. You had to use platform-specific functions (like those from the operating system) which made your code less portable (it wouldn't work easily on different operating systems like Windows, macOS, and Linux). It was also often cumbersome and error-prone.

The Solution: `std::filesystem`

- C++17 introduced the `<filesystem>` header, providing a standardized and portable way to work with files and directories.

Key Concepts:

- **Paths**: A `std::filesystem::path` object represents a file or directory path. It's like a string, but it's designed specifically for file system paths and handles them intelligently (e.g., dealing with different path separators like `/` and `\`).
- **Operations**: `std::filesystem` provides functions for common file system operations, such as:
    - Creating directories (`std::filesystem::create_directory`)
    - Checking if a file or directory exists (`std::filesystem::exists`)
    - Getting the size of a file (`std::filesystem::file_size`)
    - Renaming files or directories (`std::filesystem::rename`)
    - Copying files (`std::filesystem::copy_file`)

- Removing files or directories (`std::filesystem::remove`, `std::filesystem::remove_all`)
- Iterating through directories (`std::filesystem::directory_iterator`)

27.Parallel Algorithms

The Problem: Doing Things Faster

- Imagine you have a large list of numbers, and you want to do something with each number, like doubling it. A simple way is to go through the list one by one and double each number. But if the list is huge, this can take a long time.

The Solution: Parallel Algorithms (C++17)

- Modern computers have multiple cores (like having multiple brains). Parallel algorithms allow you to split the task into smaller pieces and do them at the same time on different cores, making the whole process much faster. It's like having multiple workers doubling numbers simultaneously.
- C++17 introduced parallel versions of many standard algorithms (functions that do common tasks), like `std::for_each`, `std::sort`, `std::transform`, and many others.

How They Work:

- These parallel algorithms automatically divide the work and distribute it across multiple threads (lightweight units of execution) to run on different cores. You don't have to manage the threads yourself; the library handles it for you.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution> // Parallel execution policies

int main() {
   std::vector<int> numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   // Sequential (normal) way:
   // std::for_each(numbers.begin(),
   //    numbers.end(), [](int& n){ n *= 2; });
```

```
    // Parallel way:
    std::for_each(std::execution::par, // Use parallel execution policy
        numbers.begin(), numbers.end(), [](int& n) { n *= 2; });

    for (int n : numbers) {
        std::cout << n << " "; // Output: 2 4 6 8 10 12 14 16 18 20
    }
    std::cout << std::endl;

    return 0;
}
```

C++17 introduces **execution policies** that control how algorithms run:

Execution Policy	Behaviour
`std::execution::seq`	Sequential execution (default, single-threaded)
`std::execution::par`	Parallel execution (multi-threaded)
`std::execution::par_unseq`	Parallel + vectorized execution (allows SIMD optimizations)

Frequently Used in Specific Scenarios

28. The `std::any`

What Problem Does `std::any` Solve?

- Imagine you want to store a variable, but you don't know what type of variable it will be until runtime. Maybe it could be an integer, a string, a floating-point number, or even a custom object.
- Before C++17, handling this kind of situation was tricky and often involved using `void*` (which is dangerous and type-unsafe) or `variant` types (which require you to specify all possible types in advance).
- `std::any` provides a type-safe way to store and retrieve values of any type without needing to know the type at compile time.

```cpp
#include <iostream>
#include <any>
using namespace std;
int main() {

    std::any myVariable;
    myVariable = 10; // Now it holds an integer
    cout << "Integer value: "
       << any_cast<int>(myVariable) << endl;

    myVariable = 3.14; // Now it holds a double
    cout << "Double value: "
       << any_cast<double>(myVariable) << endl;

    // Now it holds a string
    myVariable = string("Hello, world!");
    // We can check the type of the variable before casting
    if (myVariable.type() == typeid(string)) {
       string str = any_cast<string>(myVariable);
       cout << "It's a string: " << str << endl;
    }
    return 0;
}
/*
Integer value: 10
Double value: 3.14
It's a string: Hello, world!
*/
```

29. The `std::clamp`

What Problem Does `std::clamp` Solve?

- Imagine you have a value, and you want to make sure it stays within a specific range.
- For example, you might have a player's health in a game, and you want to ensure it never goes below `0` or above `100`. Before C++17, you'd probably write code like this:

```cpp
int health = 150;
if (health < 0) { health = 0; }
else if (health > 100) { health = 100; }
cout << "Clamped health: "
   << health << endl; // Output: Clamped health: 100
```

- `std::clamp` provides a much cleaner and more concise way to achieve the same result.

- The `std::clamp` takes three arguments:
    - The value you want to clamp.
    - The lower bound of the range.
    - The upper bound of the range.

```
#include <iostream>
#include <algorithm> // Required for clamp
using namespace std;
int main() {
   int health = 150;

   health = std::clamp(health, 0, 100); // health = 100
   cout << "Clamped Health: " << health << endl;
}
/*
Clamped Health: 100
*/
```

30. The `std::invoke`

What Problem Does `std::invoke` Solve?

- Imagine you have something that you want to call – it could be a regular function, a member function of a class, a function object (like a lambda), or even a pointer to a member.
- Before C++17, calling these different "**callables**" could sometimes be a bit awkward and **require slightly different syntax**.

What is a Callable Object?

- A callable object is anything that you can call using parentheses (), such as:
    - Normal functions
    - Lambda functions
    - Function pointers
    - Member functions
    - Function objects (functors)
- `std::invoke` provides a unified and consistent way to call anything that's callable.

- Syntax

```
#include <functional>  // std::invoke is in <functional>
std::invoke(callable, args...);
```

  - `callable` → The function, function pointer, lambda, or member function to call.
  - `args...` → The arguments to pass to the function.

```
// 1. Regular function
void myFunction(int x, double y) {
   cout << "Regular function: "
      << x << ", " << y << endl;
}
// 2. Lambda function (function object)
auto myLambda = [](const string& str) {
   cout << "Lambda: " << str << endl;
   };
// 3. Member function
struct MyClass {
   void myMethod(int z) {
      cout << "Member function: " << z << endl;
   }
};

int main() {
   // Call regular function
   invoke(myFunction, 10, 3.14);

   // Call lambda
   invoke(myLambda, "Hello from lambda!");

   MyClass obj;
   // Call member function on an object
   invoke(&MyClass::myMethod, obj, 42);

   // Calling a member function through a pointer to member:
   auto memberFuncPtr = &MyClass::myMethod;
   // Call member function through pointer
   invoke(memberFuncPtr, obj, 99);
}
/*
Regular function: 10, 3.14
Lambda: Hello from lambda!
Member function: 42
Member function: 99
*/
```

Why is `std::invoke` Useful?

- **Genericity**: It's extremely useful when writing generic code (templates) where you might need to call different types of callables. You don't need to write special cases for each type.
- **Code Clarity**: It makes your code cleaner and more expressive, especially when dealing with member functions or function pointers.
- **Metaprogramming**: `std::invoke` is often used in more advanced metaprogramming techniques.

31. The `std::apply`

What Problem Does `std::apply` Solve?

- Imagine you have a function that takes a specific number of arguments, and you have those arguments stored in a `tuple` (or a `std::pair` which can be seen as a tuple of size 2). How do you pass the tuple's elements as individual arguments to the function?
- Before C++17, this was a bit cumbersome. `std::apply` provides a clean and elegant solution.
- The `std::apply` takes two arguments:
    - The function you want to call.
    - The `tuple` containing the arguments.

```
#include <iostream>
#include <tuple>
#include <functional> // Required for apply
using namespace std;

// A function that takes three arguments
void myFunction(int x, double y, const string& z) {
   cout << "x: " << x << ", y: "
      << y << ", z: " << z << endl;
}

int main() {
   // Create a tuple containing the arguments
   auto myTuple = make_tuple(10, 3.14, "Hello from tuple!");
```

```
    // Call myFunction with the tuple's elements using apply
    // Output: x: 10, y: 3.14, z: Hello from tuple!
    std::apply(myFunction, myTuple);

    // Example with a lambda
    auto myLambda = [](int a, char b, bool c) {
        cout << "a: " << a << ", b: " << b
             << ", c: " << c << endl;
        };
    auto anotherTuple = make_tuple(25, 'Z', true);
    std::apply(myLambda, anotherTuple);

    return 0;
}
```

32. The `std::gcd`, `std::lcm`

What Problem Do `std::gcd` and `std::lcm` Solve?

- These functions solve the common mathematical problems of finding the Greatest Common Divisor (GCD) and the Least Common Multiple (LCM) of two integers.
    - GCD: The greatest common divisor of two integers is the largest positive integer that divides both of them without leaving a remainder. 1 For example, the GCD of 12 and 18 is 6.
    - LCM: The least common multiple of two integers is the smallest positive integer that is divisible by both of them. For example, the LCM of 12 and 18 is 36.
- Before C++17, you'd have to write your own functions to calculate these. Now, C++ provides these in the standard library, making your life easier.

```
#include <iostream>
#include <numeric> // Required for gcd and lcm
using namespace std;
int main() {
    int a = 12;
    int b = 18;

    int gcd_result = gcd(a, b);
    int lcm_result = lcm(a, b);

    cout << "GCD of " << a <<
        " and " << b << " is: " << gcd_result << endl;
```

```
    cout << "LCM of " << a <<
       " and " << b << " is: " << lcm_result << endl;

    return 0;
}
// GCD of 12 and 18 is: 6
// LCM of 12 and 18 is : 36
```

Occasionally Used

33. The `std::as_const`

- `std::as_const` provides a simple and clear way to convert an object to a `const&` (a constant reference), guaranteeing that it won't be modified in that specific context.

```
#include <utility> // Required for std::as_const
int myVariable = 42;
// Use std::as_const to get a const reference
const int& constRef = std::as_const(myVariable);
```

34. The `std::reduce`

- Imagine you have a container (like a `std::vector`) of numbers, and you want to calculate their sum.
- `std::reduce` takes a range (beginning and end iterators), an initial value, and an operation (by default, it's addition). It applies the operation cumulatively to the elements in the range, starting with the initial value. Crucially, it can do this in parallel by splitting the range into chunks and processing them independently.

```
#include <iostream>
#include <vector>
#include <numeric> // For std::reduce

int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
    // Calculate the sum
    int sum = std::reduce(numbers.begin(),
        numbers.end(), 0); // 0 is the initial value

    std::cout << "Sum: " << sum << std::endl; // Output: 15
```

```
    // You can also use a different operation (e.g., multiplication):
    int product = std::reduce(numbers.begin(), numbers.end(),
        1, std::multiplies<int>()); // 1 is the initial value for multiplication
    std::cout << "Product: " << product << std::endl; // Output: 120

    return 0;
}
```

35. Prefix Sums: `std::exclusive_scan` and `std::inclusive_scan`

- They involve calculating a running total (or some other cumulative operation) of the elements in a sequence.
- **`std::exclusive_scan`**: For each element, it calculates the sum of all the elements before it. The "exclusive" part means the current element is excluded from the sum.
- **`std::inclusive_scan`**: For each element, it calculates the sum of all the elements up to and including it*. The "inclusive" part means the current element is included in the sum.

```
std::vector<int> numbers = { 1, 2, 3, 4, 5 };
std::vector<int> exclusive_results(numbers.size());
std::vector<int> inclusive_results(numbers.size());

std::exclusive_scan(numbers.begin(), numbers.end(),
    exclusive_results.begin(), 0); // 0 is the initial value
std::inclusive_scan(numbers.begin(), numbers.end(),
    inclusive_results.begin(), 0); // 0 is the initial value
```

36. The `std::sample`

What Problem Does `std::sample` Solve?

- Imagine you have a large collection of items (like a vector of numbers, a set of names, etc.), and you want to randomly select a smaller subset of these items. This is called sampling. **`std::sample`** provides a convenient and efficient way to do this in C++.

```
#include <iostream>
#include <vector>
#include <random>   // For random number generation
#include <algorithm> // For std::sample
```

```cpp
int main() {
    std::vector<int> population = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    std::vector<int> sample; // To store the sampled elements
    int sampleSize = 3;      // Number of elements to sample

    // 1. Create a random number engine (you need this for random selection)
    std::random_device rd;  // Obtain a random seed from the OS
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()

    // 2. Use std::sample to select the sample
    std::sample(population.begin(), population.end(),
        std::back_inserter(sample), // Where to store the sample
        sampleSize,                 // How many elements to sample
        gen);                       // The random number engine

    std::cout << "Original population: ";
    for (int num : population) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    std::cout << "Sample: ";
    for (int num : sample) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

37. Occasionally Used

- `std::not_fn` → Inverts predicate logic, useful in functional programming.
- `std::byte` → Type-safe representation of raw memory, useful in low-level programming.
- Splicing for `std::map` and `std::set` → Transfers nodes efficiently, useful in advanced data structure manipulations.
- `std::owner_less` → Compares `std::shared_ptr` and `std::weak_ptr` without affecting ownership.
- `try_emplace`, `insert_or_assign` → Optimized insertions for `std::map` and `std::unordered_map`.
- String conversion (`std::to_chars`, `std::from_chars`) → Efficient string to number conversion without heap allocation.

38. Rarely Used / Niche Features

- Rounding functions for `std::chrono` → Helps round durations but not commonly needed.
- `std::bool_constant`, `std::void_t`, `std::conjunction`, `std::disjunction`, `std::negation` → Useful for template metaprogramming but niche.
- `std::is_contiguous` → Checks if a container stores elements contiguously (mostly theoretical).
- `std::size`, `std::empty`, `std::data` (Non-member versions) → Syntactic sugar but not groundbreaking.
- `noexcept` in the type system → Important for exception safety but mostly affects library writers.
- Emplace family returns reference → Minor optimization in container operations.
- constexpr iterators → Helps with compile-time computations, but rare in general usage.

39. Very Rare / Special Use Cases

- Hexadecimal floating-point literals (Already discussed Above) → Almost never needed outside of specific numerical applications.
- Dynamic memory allocation for over-aligned data (`std::aligned_alloc`) → Useful for SIMD or hardware optimizations.
- Fixed order of evaluation of expressions → Improves code predictability but isn't a library feature per se.
- C++17 library alignment with C11 → Mostly for compiler implementers, not regular developers.