C++ 11 Language Features

Contents

a) Core Language Features

Type Deduction and Inference

1. Keyword `auto`

- Automatic type deduction for variables.

```
auto i = 42; // i has type int
double f();
auto d = f(); // d has type double
```

- The type of a variable declared with `auto` is deduced from its initializer. Thus, an initialization is required.

```
auto i; // ERROR: can't deduce the type of i
```

2. Type Traits

- **C++11 introduced Type Traits**, which are a set of template-based utilities that allow **compile-time type introspection and transformation**.
- They are defined in the `<type_traits>` header and are useful for **metaprogramming**, **SFINAE**, and **optimizing template code**.
- Key Features of Type Traits
    - Type Properties – Check if a type satisfies certain characteristics.
    - Type Relationships – Compare types.
    - Type Modifications – Transform types at compile time.

i. Type Properties

Trait	Meaning
`std::is_integral<T>::value`	`true` if `T` is an integral type (`int`, `char`, etc.)
`std::is_floating_point<T>::value`	`true` if `T` is a floating-point type (`float`, `double`, etc.)
`std::is_pointer<T>::value`	`true` if `T` is a pointer (`T*`)
`std::is_array<T>::value`	`true` if `T` is an array type (`int[]`, etc.)
`std::is_class<T>::value`	`true` if `T` is a `class`/`struct`
`std::is_enum<T>::value`	`true` if `T` is an enumeration

```cpp
#include <iostream>
#include <type_traits>

template <typename T>
void checkType()
{
   if (std::is_integral<T>::value)
      std::cout << "T is an integral type\n";
   else
      std::cout << "T is NOT an integral type\n";
}

int main()
{
   checkType<int>();      // Output: T is an integral type
   checkType<double>();   // Output: T is NOT an integral type
}
```

ii. Type Relationships

Trait	Meaning
`std::is_same<T1, T2>::value`	`true` if `T1` and `T2` are the same type
`std::is_base_of<Base, Derived>::value`	`true` if Base is a base `class` of Derived
`std::is_convertible<From, To>::value`	`true` if `From` can be converted to `To`

```cpp
#include <iostream>
#include <type_traits>

int main() {
   std::cout << std::boolalpha;
   std::cout << "int and int: " <<
      std::is_same<int, int>::value << '\n';       // true
   std::cout << "int and double: " <<
      std::is_same<int, double>::value << '\n';   // false
}
/*
int and int: true
int and double: false
*/
```

iii. Type Modifications

Trait	Meaning
`std::remove_pointer<T>::type`	Removes pointer (`T*` → `T`)
`std::remove_reference<T>::type`	Removes reference (`T&` → `T`)
`std::add_const<T>::type`	Adds `const` (`T` → `const T`)
`std::decay<T>::type`	Converts an array/function to a pointer
`std::make_signed<T>::type`	Converts an `unsigned` type to `signed`

```
#include <iostream>
#include <type_traits>

int main() {
   using T = int*;
   using NoPtr = std::remove_pointer<T>::type;
   using ConstNoPtr = std::add_const<NoPtr>::type;
   std::cout << std::boolalpha;
   std::cout << // Output: 1 (true)
      std::is_same<ConstNoPtr, const int>::value << '\n';
}
```

Iteration and Loops

3. Range-based `for` loops

- C++11 introduces a new form of for loop, which iterates over all elements of a given range, array, or collection.
- The general syntax is as follows:

```
for (auto& elem : container) { /* ... */ }
```

- Example

```
int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (auto y : x)
   cout << y << " ";
```

Functions and Lambdas

4. Lambda Expressions

- A lambda expression, sometimes also referred to as a lambda function or as a lambda, is a simplified notation for defining and using an **anonymous** (Lacking a name or identity) function object.

What are Function Objects?

- Function objects are often used as callbacks, providing a more powerful and type-safe alternative to function pointers (especially in C++).
- A typical use of a function object is in writing callback functions. A callback in procedural languages, such as C, may be performed by using function pointers.

```
#include <stdlib.h>
#include <iostream>
using namespace std;

// Callback function.
int compareInts(const void* a, const void* b)
{
   return *(const int*)a - *(const int*)b;
}
int main(void)
{
   int items[] = { 4, 3, 1, 2 };

   qsort(items, sizeof(items) / sizeof(items[0]),
         sizeof(items[0]), compareInts);

      for (int i = 0; i < 4; i++)
         cout << items[i] << endl;

   return 0;
}
```

- A function object (or functor) is an object of a `class` that can be called like a function using the function call `operator ()`. This is achieved by overloading the `operator()` in the `class` definition. While the invocation syntax is the same as a regular function (e.g., `my_function_object(arguments)`), function objects are distinct from functions.

- Unlike function pointers, function objects can maintain state (member variables) and have their own type, enabling them to be used in generic programming (e.g., with STL algorithms) and improving type safety. They are also the underlying mechanism for lambdas in C++.

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

// Comparator predicate: returns true if a < b, false otherwise
struct IntComparator {
   bool operator()(const int& a, const int& b) const
   {
      return a < b;
   }
};
int main()
{
   IntComparator cpm;
   int a = 10;
   int b = 55;
   if (cpm(a, b))
   {
      cout << "a is Less Than b" << endl;
   }
   else
      cout << "a is greater than or equal to b" << endl;
   return 0;
}
```

- Instead of defining a named class with an `operator()`, later making an object of that `class`, and finally invoking it, we can use a shorthand which is **lambda**.

What is a Lambda?

- A **lambda expression** (often just called a "lambda") is a concise way to create an *anonymous* function object (or functor) *inline* where it's needed. It defines a function-like entity without a formal name, typically for short, simple operations.
- Syntax:

```
auto func = [capture_list](parameter_list) -> return type{
   function_body
};
```

- A lambda expression consists of a sequence of parts:
  - A possibly empty **`capture list`**, specifying what names from the definition environment can be used in the lambda expression's body, and whether those are copied or accessed by reference. The capture list is delimited by **`[]`**.
  - An optional **`parameter list`**, specifying what arguments the lambda expression requires. The parameter list is delimited by **`()`**.
  - An optional **`mutable`** specifier, indicating that the lambda expression's body may modify the state of the lambda.
  - An optional **`noexcept`** specifier.
  - An optional **`return`** type declaration of the form **`->`** type
  - A body, specifying the code to be executed. The body is delimited by **`{}`**.
- Example:

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

int main(void)
{
    // Lambda That Prints Hello World.
    [] {cout << "Hello World" << endl; } ();

    // Lambda That Does Nothing. It's a Valid Lambda.
    [] {}();

    // Lambda That Returns A Double
    auto l1 = []() -> double { return 50; };
    cout << l1() << endl;

    return 0;
}
```

5. The `noexcept` Specifier

- **C++11 introduced the `noexcept` specifier**, which is used to indicate that a function does not **`throw`** exceptions.
- This helps the compiler optimize code, enables better exception safety guarantees, and allows **`move`** operations to be more efficient.

- Key Features of noexcept
  - **Function Optimization**: Functions marked `noexcept` enable the compiler to generate better code.
  - **Move Semantics**: Containers like `std::vector` prefer `noexcept move` constructors for better efficiency.
  - **Exception Safety**: It helps signal to users that a function is guaranteed not to `throw`.

```cpp
#include <vector>
#include <iostream>
struct A {
   A() {}
   A(const A&) { std::cout << "Copy constructor called\n"; }
   // noexcept move constructor
   A(A&&) noexcept { std::cout << "Move constructor called\n"; }
};
struct B {
   B() {}
   B(const B&) { std::cout << "Copy constructor called\n"; }
   // Move constructor is NOT noexcept
   B(B&&) { std::cout << "Move constructor called\n"; }
};
int main() {
   std::vector<A> vecA;
   vecA.reserve(2);  // Reserve space for 2 elements
   vecA.emplace_back();  // Default construct A
   vecA.emplace_back();  // Default construct A
   std::cout << "--- Resizing vecA ---\n";
   vecA.resize(4);  // Move constructor is called (because noexcept is present)
   std::vector<B> vecB;
   vecB.reserve(2);  // Reserve space for 2 elements
   vecB.emplace_back();  // Default construct B
   vecB.emplace_back();  // Default construct B
   std::cout << "--- Resizing vecB ---\n";
   vecB.resize(4);  // Copy constructor is called (because move is NOT noexcept)
}
/*
--- Resizing vecA ---
Move constructor called
Move constructor called
--- Resizing vecB ---
Copy constructor called
Copy constructor called
*/
```

- When Will the Copy Constructor Be Called?
    - If the `move` constructor is not declared `noexcept`, `std::vector` may prefer the copy constructor instead of the `move` constructor.
    - This happens when `std::vector` needs to grow (i.e., reallocate and move elements to a new location).
    - STL containers like `std::vector` assume that if moving an object might `throw` an exception, then copying is a safer alternative.
- Summary

Feature	Explanation
`void func() noexcept;`	Declares `func` as not throwing exceptions
`noexcept(expression)`	Checks if an expression is `noexcept` at compile time
`std::vector` prefers `noexcept` move constructors	Improves performance
If `noexcept` function `throw`s	Calls `std::terminate()`
`noexcept` in templates	Enables conditionally marking functions as `noexcept`

Pointers and Memory Management

6. Null pointer literal

- **Type Safety:** `nullptr` is of type `std::nullptr_t`, defined in `<cstddef>`. This prevents accidental assignments of integer values (like `0` or `NULL`) to pointers, improving type safety. This is its biggest advantage.
- **Clear Intent:** Using `nullptr` explicitly signals that you intend a pointer to be null. This makes your code more readable and less prone to errors compared to using `0` or `NULL`, which could be misinterpreted as integers in some contexts.
- **No Implicit Conversion to Integer:** `nullptr` does *not* implicitly convert to an integer type. This prevents the common mistake of using a null pointer in an integer context.
- `std::nullptr_t`**:** The type `std::nullptr_t` is specifically designed for null pointers. **It's not an integral type**.

- **Prvalue:** `nullptr` is a *prvalue* (pure rvalue). This means you cannot take its address using the **&** operator. This is consistent with the idea that it represents a null pointer, not a memory location.
- Conversions:
    - An integral null pointer constant (like `0` or a `NULL` macro that expands to `0`) can be implicitly converted to `std::nullptr_t`. This is for backward compatibility.
    - The reverse conversion (from `std::nullptr_t` to an integral type) is not allowed implicitly. This further enforces type safety.

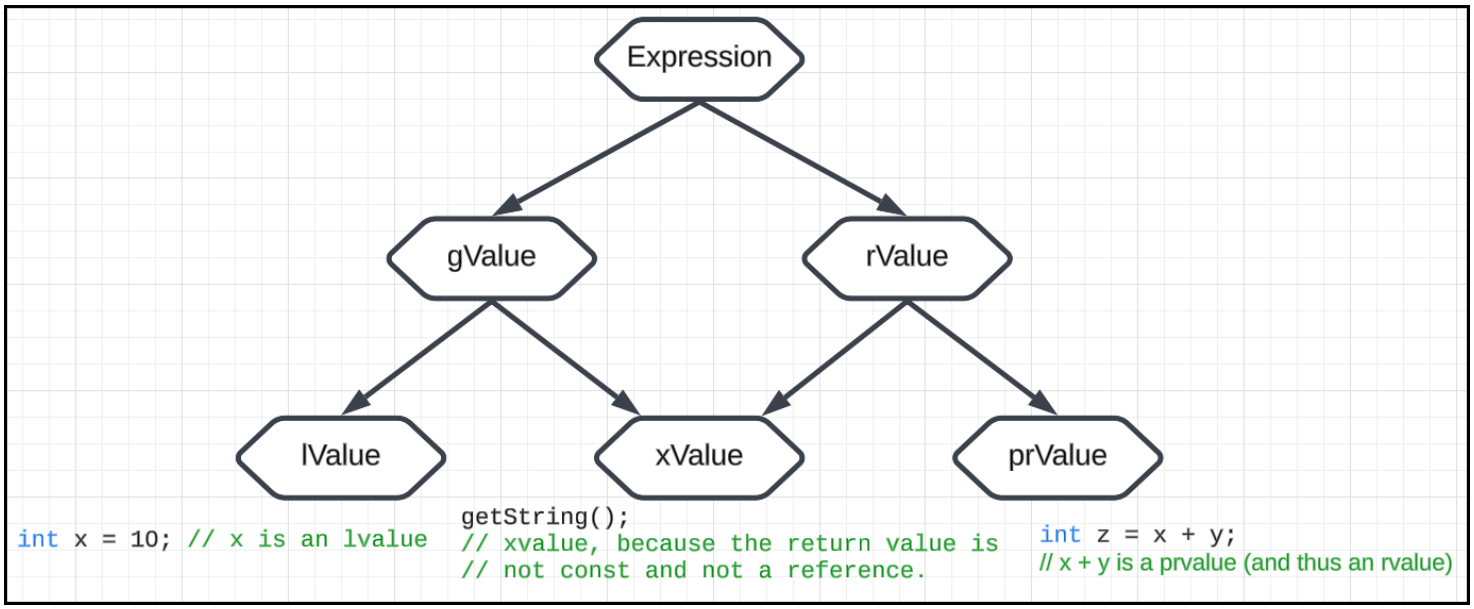
```
#include <iostream>
using namespace std;

void function(int x)
{
   cout << "Value of X: " << x << endl;
}

void function(char* ptr)
{
   if (ptr == nullptr)
   {
      cout << "Pointer is null. Allocating memory" << endl;
      // Do some stuff here...
   }
   else
   {
      cout << "Value contained in the pointer: " << *ptr << endl;
   }
}

int main(void)
{
   function(NULL);    // Ambiguous call; the integer version will be called.
   function(nullptr); // Pointer version will be called.

   char ch = 'A';
   char* ptr1 = &ch;
   function(ptr1);    // Pointer version will be called.
   return 0;
}
```

7. Rvalue References and Move Semantics

a. Lvalues and Rvalues

- What is a value?
    - Value is an expression which cannot be evaluated any further.
- What are Lvalues?
    - Lvalues (locator value) have storage addresses, means they are variables. When we use an object as an lvalue, we use the object's identity (its location in memory).
- What are Rvalues?
    - Rvalues are the **value of an expression**. Rvalue is just a term only used to distinguish from Lvalue. When we use an object as an rvalue, we use the object's value (its contents).
- Example:
    - Assume that we have written expression `5 + 6`. This expression evaluates to `11`, but in the program, we have not explicitly designated where in the computer this `11` is stored, the expression is an Rvalue.
    - Assume that we have declared a variable '`y`'. And we are assigning `y = 5 + 6`. Now '`y`' has the value `11`. So variable '`y`' is an Lvalue (Modifiable Lvalue).
- Note: The term "Modifiable Lvalue" is used because; `const` variables can't be modified even though it has storage.
- In C++11, an expression can be an:
    - **glvalue** (Generalized lvalue): An expression whose evaluation determines an identity (i.e., refers to a memory location). It can be an lvalue or an xvalue.
    - **prvalue** (Pure rvalue): An expression whose evaluation doesn't determine an identity. It's either a literal, the result of a function that returns by value, or a temporary object that isn't bound to a reference.
    - **xvalue** (eXpiring value): A glvalue that denotes an object whose resources can be "moved from." These are typically near the end of their lifetime.

- **lvalue**: A glvalue that designates a function or an object with a stable address (that you can take with &). It can appear on the left-hand side of an assignment.
- **rvalue**: A prvalue or an xvalue. It represents a value that is typically transient.




```cpp
#include <iostream>
#include <string>
#include <vector>

class MyString {
   std::string data;
public:
   MyString(const std::string& str) : data(str) {
   }
   MyString(const MyString& other) : data(other.data) {
      std::cout << "Copy Constructor\n";
   }
    MyString& operator=(const MyString& other) {
       data = other.data;
       std::cout << "Copy Assignment\n";
       return *this;
    }

   MyString(MyString&& other) noexcept : data(std::move(other.data)) {
      std::cout << "Move Constructor\n";
   }
    MyString& operator=(MyString&& other) noexcept {
       data = std::move(other.data);
       std::cout << "Move Assignment\n";
       return *this;
    }

   //// Copy-Swap Assignment Operator instead of both assignment operators
```

```
    //MyString& operator=(MyString other) {
    //    std::cout << "Copy Swap Operator\n";
    //    std::swap(data, other.data);
    //    return *this;
    //}
    ~MyString() {
    }
};

MyString getString() {         // Returns a prvalue (pure rvalue)
    return MyString("Hello");  // Temporary (prvalue)
}

int main() {
    MyString str1("World");       // str1 is an lvalue
    MyString str2 = std::move(getString()); // prvalue → Move Constructor called
    MyString str3 = str1;         // str1 is an lvalue → Copy Constructor called
    str3 = getString();           // Move Assignment

    return 0;
}
```

Templates and Generic Programming

8. Variadic Templates

- Variadic templates allow functions and classes to accept a **variable number of template parameters**.

How They Work:

- Variadic templates use a combination of parameter packs and recursion.
- **Parameter Packs**: A parameter pack is a template parameter that can accept zero or more arguments. There are two kinds:
    - **Type parameter pack**: `typename... Types`
    - **Non-type parameter pack**: `int... Values` (or other non-type template parameters)
- **Expansion**: You "expand" a parameter pack to access its individual elements. This is often done recursively.

- Syntax
    - A variadic template uses parameter packs, represented by `...` (ellipsis):

```
template <typename... Args>   // 'Args' is a parameter pack
void func(Args... args) {     // 'args' is a function parameter pack
   // Do something
}
```

        - `Args...` → A pack of types.
        - `args...` → A pack of values corresponding to those types.

```
#include <iostream>

// Base case: Function with a single argument
void print() {
   std::cout << "\nEnd of recursion\n";
}

// Variadic template function
template <typename T, typename... Args>
void print(T first, Args... rest) {
   std::cout << first << " ";
   print(rest...);  // Recursive call with remaining arguments
}

int main() {
   print(1, 2.5, "Hello", 'A');
   return 0;
}
```

- How It Works?
    - **`main()`** calls **`print()`**: The main function calls print with four arguments: **`1`**, **`2.5`**, **`"Hello"`**, **`'A'`**.
    - Variadic Template Instantiation: The compiler sees the call to **`print`** with these arguments and instantiates the variadic template function. It deduces the types:
        - **`T`** is int (the type of **`1`**).
        - **`Args...`** is a parameter pack containing the types **`double`**, **`const char*`**, and **`char`**.

- Inside the Variadic `print()`:
  - `first` is initialized with the value `1`.
  - `rest...` is a parameter pack containing the values `2.5, "Hello", 'A'`.
- Output: The code prints `1` (the value of `first`).
- Recursive Call: The crucial part is `print(rest...);`. This expands the parameter pack `rest...`. It effectively generates a new call to `print` like this:
  - `print(2.5, "Hello", 'A');// A new instantiation of print!`
- Repeated Instantiation and Calls: This process repeats. The compiler instantiates the `print` template again, this time with `T` being `double` and `Args...` being `const char*,` and `char`. It prints `2.5`, and then makes another recursive call: `print("Hello", 'A');`
- Base Case: Eventually, the parameter pack `Args...` becomes empty. The call `print()` with no arguments matches the base case function (the first print function you defined), which prints `"End of recursion"` and stops the recursion.

9. Static Assertions

- A **static assertion** (`static_assert`) is a **compile-time assertion** introduced in **C++11**. It allows checking conditions **at compile time**, ensuring that the program doesn't compile if the condition fails.
- Syntax:

```
static_assert(condition, "Error message");
```

  - `condition` → A `constexpr` (compile-time) `Boolean` expression.
  - `"Error message"` → A custom error message displayed if the assertion fails.
  - If condition evaluates to `false`, compilation fails with the error message.

```
#include <iostream>

static_assert(sizeof(int) == 4, "Integers must be 4 bytes!");
int main() {
   std::cout << "Compilation successful!\n";
   return 0;
}
```

- The `static_assert` statement checks if the size of an integer is `4` bytes. If it is not, the program will not compile and will display the error message.
- The `static_assert` statement is a compile - time check, and it is used to ensure that certain conditions are met during compilation. If the condition is not met, the program will not compile and will display an error message.

```
#include <iostream>
#include <type_traits>
template <typename T>
void process(T value)
{
   static_assert(std::is_integral<T>::value, "T must be an integral type!");
   std::cout << "Processing: " << value << "\n";
}
int main() {
   process(42);      // Works (int is integral)
   // process(3.14); // Compilation Fails
   return 0;
}
```

Literals and Constants

10. User-defined literals

- Literals are constant values like numbers (e.g., `42`), characters (e.g., `'a'`), and strings (e.g., `"hello"`).
- Different types of literals:
  - **Numeric literals**: Numbers, e.g., `42`, `3.14`.
  - **Character literals**: Single characters, e.g., `'a'`.
  - **String literals**: Sequences of characters, e.g., `"hello"`.
  - **Raw string literals**: Strings that can contain special characters without needing escape sequences, e.g., `R"(line 1\nline 2)"`.
- User-defined literals (UDLs):
  - Example of built in literal suffixes:
    - `u` or `U`: Unsigned integer (e.g., `10u, 12345U`).
    - `l` or `L`: Long integer (e.g., `100000L`).
    - `f` or `F`: Float (single-precision) (e.g., `3.14f`, `2.718F`).
    - `l` or `L`: Long double (e.g., `3.1415926535L`).

- `L`: Wide string literal (e.g., `L"hello"`).
- `u8`: UTF-8 string literal (e.g., `u8"hello"`).

- UDLs are custom literal suffixes that programmers can create. Literal suffixes are characters appended to a literal value (like a number, character, or string) that modify its type or interpretation.
- They always start with an underscore `_` (e.g., `10_km`, `25.5_degC`).
- A UDL is defined as a function (or function template) whose name starts with `operator""` followed by the suffix you want to use. The compiler treats the literal preceding the suffix as an argument to this function.
- Literal operators for floating-point numbers must take `long double`, `unsigned long long`, or `const char*` as parameters.
- Literal operators for integers must take `unsigned long long`, or `const char*` as parameters.

- Example:

```
#include <iostream>
using namespace std;
// User-defined literal for kilometers
long double operator""_km(long double km) {
   return km * 1000; // Convert to meters
}
// User-defined literal for meters
long double operator""_m(long double m) {
   return m/1000; // Convert to Kilometers
}
// User-defined literal for degrees Celsius
long double operator"" _degC(long double degC) {
   return degC + 273.15; // Convert to Kelvin
}
int main() {
   long double distance_meters = 5.0_km;
   std::cout << "5.0 km in meters: " << distance_meters << "m\n";

   long double distance_km = 5500.0_m;
   std::cout << "5500 m in km: " << distance_km << "km\n";

   long double tempK = 25.0_degC;
   std::cout << "25.0 degC in Kelvin: " << tempK << "K\n";
   return 0;
}
```

11. Generalized Constant Expressions

- Generalized Constant Expressions, commonly known as `constexpr`, were introduced in C++11.
- `constexpr` is a keyword that can be applied to variables and functions.
- It tells the compiler that it should evaluate the expression or function at compile time if possible. If the compiler can perform the evaluation at compile time, it will. If it can't (because the value depends on something not known at compile time), it will evaluate it at runtime.

`constexpr` Functions:

- A `constexpr` function is a function that can be evaluated at compile-time if its arguments are constant expressions. If the arguments are not constant expressions, it can still be evaluated at run-time.

`constexpr` Variables:

- A `constexpr` variable is guaranteed to have a constant value and can be used in contexts where a constant expression is required.

```
constexpr int add(int a, int b) {
    return a + b;
}

int main() {
                                // Evaluated at compile-time
    constexpr int result = add(3, 4);
    cout << "Compile Time Result: " << result << "\n";
    int x;
    cout << "Enter value for x: ";
    cin >> x;
                            // Evaluated at run-time
    int runtimeResult = add(x, 10);
    cout << "Runtime Result: " << runtimeResult;

    constexpr int size = 10;
    // Array size must be a constant expression
    int array[size];
    return 0;
}
```

Initialization

12. Initializer Lists

- C++11 introduced **uniform initialization** using **braced {} lists**, which provides a consistent syntax for initializing variables, arrays, structs, and objects.
- Before C++11, there were several ways to initialize objects, each with its own quirks:

```
int a(1);       // Variable definition
int b();        // Function declaration
int b(foo);          // Variable definition or function declaration

// Initialize array variable
string a[] = { "foo", " bar" };

// Error: initializer list for non-aggregate vector
vector<string> v = { "foo", " bar" }; // Error

void f(string a[]);
f( { "foo", " bar" } );   // Syntax error: block as argument
```

- Uniform initialization uses **{}** (braced lists), making the syntax more consistent across different data types.

```
int x1(5.3);    // OK, but x1 becomes 5
int x2 = 5.3;   // OK, but x2 becomes 5

// Uniform Initialization
int x3{ 5.3 };   // ERROR: narrowing
int x4 = { 5.3 };// ERROR: narrowing
char c1{ 7 };    // OK: even though 7 is an int, this is not narrowing
char c2{ 999 };  // ERROR: narrowing (if 999 doesn't fit into a char)

int arr1[] = { 1, 2, 3 }; // Old-style
int arr2[]{ 1, 2, 3 };    // Uniform initialization

std::vector<int> vec = { 10, 20, 30 };
std::vector<int> vec2{ 10, 20, 30 };
```

The `std::initializer_list<T>`

- C++11 introduced `std::initializer_list<T>` to handle uniform initialization in **user-defined `class`es**.

```
#include <initializer_list>
class MyClass {
public:
   MyClass(std::initializer_list<int> vals) {
      for (int val : vals) {
         std::cout << val << " ";
      }
      std::cout << "\n";
   }
};
int main() {
   MyClass obj{ 1, 2, 3, 4, 5 };
   return 0;
}
// Output: 1 2 3 4 5
```

Unicode and String Handling

13. Unicode Support

- C++11 introduced **Unicode support** by adding new **character types** and **string literals** for working with **UTF-8, UTF-16, and UTF-32** encoded text.
- C++11 introduces three new character types for Unicode:
  - `char16_t` → Represents UTF-16 characters (16-bit)
  - `char32_t` → Represents UTF-32 characters (32-bit)
  - `wchar_t` (existing) → Typically for wide characters (platform-dependent)
- C++11 allows you to define Unicode string literals using **prefixes**:

Encoding	Prefix	Example
UTF-8	`u8`	`u8"Hello"` (Stored as `char` array)
UTF-16	`u`	`u"Hello"` (Stored as `char16_t` array)
UTF-32	`U`	`U"Hello"` (Stored as `char32_t` array)
Wide String	`L`	`L"Hello"` (Stored as `wchar_t` array)

Note: `char8_t` → A dedicated type for UTF-8 characters was introduced in **C++20**.

14. Raw String Literals

- Raw string literals in C++11 provide a way to **include special characters**, like **backslashes** or **quotes**, in strings **without needing to escape them**.
- This can make the code more readable and less error-prone, especially when dealing with complex strings.
- They are defined using a special syntax:

  `R"delimiter(raw-string-content)delimiter"`

  - `R`: Indicates a raw string literal.
  - `"delimiter( ... )delimiter"`: The string is enclosed in parentheses, which are themselves enclosed between `delimiters`. The `delimiters` are optional. If you don't use a `delimiter`, it's as if you used an empty one. If you use a `delimiter`, it must be the same before and after the raw string content.

```
int main() {
   // Standard string with escaped characters
   std::string escaped = "C:\\path\\to\\file.txt";
   std::cout << "Escaped string:" << std::endl << escaped << std::endl;

   // Raw string literal
   std::string raw = R"**(C:\path\to\file.txt)**";
   std::cout << "Raw string literal:" << std::endl << raw << std::endl;
   return 0;
}
```

Concurrency and Threading

15. Thread-local Storage

- C++11 introduced thread-local storage using the `thread_local` keyword. This allows variables to have separate instances for each thread, avoiding race conditions without requiring explicit synchronization mechanisms like `std::mutex`.
- Declaring a variable with `thread_local` means each thread gets its own copy of the variable.
- The variable persists for the lifetime of the thread.
- It cannot be shared across threads but remains available as long as the thread exists.

```cpp
#include <iostream>
#include <thread>

// Each thread gets its own "counter"
thread_local int counter = 0;

void incrementCounter() {
   ++counter;
   printf("Thread ID: %d and Counter: %d\n",
      std::this_thread::get_id(), counter);
}

int main() {
   std::thread t1(incrementCounter);
   std::thread t2(incrementCounter);
   std::thread t3(incrementCounter);

   t1.join();
   t2.join();
   t3.join();

   return 0;
}
/*
Thread ID: 26744 and Counter: 1
Thread ID: 14060 and Counter: 1
Thread ID: 15788 and Counter: 1
*/
```

Miscellaneous

16. Long Long Type

- The `long long` type was introduced in C++11 as a standard integer type with at least 64 bits of storage.
- Key Features of `long long`
  - It is **guaranteed** to be at least **64 bits** (`sizeof(long long) >= 8`).
  - It allows representation of very large integer values.
  - The corresponding **unsigned version** is `unsigned long long`.
  - The **format specifier** for `long long` in `printf/scanf` is `%lld` for signed and `%llu` for unsigned.

```cpp
#include <iostream>
#include <limits>

int main() {
   // Maximum for 64-bit signed integer
   long long largeNumber = 9223372036854775807LL;
   // Maximum for 64-bit unsigned integer
   unsigned long long largeUnsigned = 18446744073709551615ULL;

   std::cout << "Signed long long: " << largeNumber << '\n';
   std::cout << "Unsigned long long: " << largeUnsigned << '\n';

   // Check limits
   std::cout << "Max long long: " <<
      std::numeric_limits<long long>::max() << '\n';
   std::cout << "Min long long: " <<
      std::numeric_limits<long long>::min() << '\n';

   return 0;
}
/*
Signed long long: 9223372036854775807
Unsigned long long: 18446744073709551615
Max long long: 9223372036854775807
Min long long: -9223372036854775808
*/
```

17. Attributes

- C++11 introduced *attributes*, a standardized way to provide additional information to the compiler about your code.
- Attributes are mainly used for:
    - Optimizations
    - Warnings
    - Code analysis
- Syntax:

```cpp
[[attribute_name]]                // General syntax
[[attribute_name(parameters)]]  // With parameters
```

- Example:

```cpp
#include <iostream>
#include <thread>

[[nodiscard]] int getValue() {
   return 42;
}

int main() {
   getValue();  // Compiler may warn: "Return value ignored"
   return 0;
}
```

Commonly Used C++11 Attributes

Attribute	Description
`[[noreturn]]`	Indicates that a function **never returns**.
`[[deprecated]]`	Marks a function/`class` as **deprecated** (discouraged for use).
`[[nodiscard]]`	Warns if a function's `return` value is ignored.
`[[maybe_unused]]`	Suppresses warnings for unused variables/functions.

18. Alignment Support

What is Alignment in C++?

- Alignment refers to how data is laid out in memory.
- Some CPU architectures require certain data types to be aligned for optimal performance.
- Misaligned accesses can cause performance penalties or even crashes on some architectures.

Why is Alignment Important?

- **Performance**: Unaligned memory accesses can be significantly slower on some architectures. The CPU might have to perform multiple memory accesses to fetch the data, or it might have to use special (and slower) instructions for unaligned accesses.
- **Portability**: Some architectures might not even support unaligned memory accesses at all, resulting in crashes or undefined behaviour.
- **Data Structures**: Proper alignment is crucial when working with data structures, especially when interfacing with hardware or other systems that have strict alignment requirements.
- C++11 introduced standardized alignment support using:
  - `alignas` (Specifier)
  - `alignof` (Operator)
  - `std::aligned_alloc` (Function)
- Example:

```cpp
#include <iostream>

struct alignas(16) MyStruct {
   int a;
   double b;
};

int main() {
   std::cout << "Alignment of int: "
      << alignof(int) << '\n';
   std::cout << "Alignment of double: "
      << alignof(double) << '\n';
   std::cout << "Alignment of MyStruct: "
      << alignof(MyStruct) << '\n';
}
```

- Summary:

Feature	Purpose
`alignas(N)`	Enforces **N-byte alignment** for variables & structures.
`alignof(T)`	Returns the **alignment requirement** of type `T`.
`std::aligned_alloc(N, size)`	Allocates **N-byte aligned** memory.
`std::free(ptr)`	Frees memory allocated with `std::aligned_alloc()`.

19. Keyword `decltype`

- `decltype` is a **compile-time type deduction feature** introduced in C++11.
- It **determines and returns the type** of an expression **without evaluating it**.
- Key Use Cases:
  - Infer **return types** of functions.
  - Deduce the type of **variables** and **expressions**.
  - Enable **generic programming** with **templates**.

```
#include <iostream>
#include <type_traits>
using namespace std;
int main() {
   int i = 4;
   const int j = 6;
   const int& k = i;
   int a[5];
   int* p;

   cout << boolalpha;

   cout<<"decltype(i) == int: "<< is_same<decltype(i), int>::value <<"\n";
   cout<<"decltype(1) == int: "<< is_same<decltype(1), int>::value <<"\n";
   cout<<"decltype(2 + 3) == int: "<< is_same<decltype(2 + 3), int>::value <<"\n";
   cout<<"decltype(i = 1) == int&: "<< is_same<decltype(i = 1), int&>::value <<"\n";
   cout<<"decltype((i)) == int&: "<< is_same<decltype((i)), int&>::value <<"\n";
   cout<<"decltype(j) == const int: "<< is_same<decltype(j), const int>::value <<"\n";
   cout<<"decltype(k) == const int&: "<< is_same<decltype(k), const int&>::value <<"\n";
   cout<<"decltype(a) == int[5]: "<< is_same<decltype(a), int[5]>::value <<"\n";
   cout<<"decltype(*p) == int&: "<< is_same<decltype(*p), int&>::value <<"\n";

   return 0;
}
```

20. Suffix/Trailing Return Type

- Sometimes, the `return` type of a function depends on an expression processed with the arguments.

```
template <typename T1, typename T2>
decltype(x + y) add(T1 x, T2 y);
```

- Unfortunately, the above code doesn't work. The compiler will parse codes from left to right, so it will issue an **error message** to indicate that variables `'x'` and `'y'` are used before their declarations.

- To solve this problem, C++11 introduced a feature called **trailing return types**. Using this feature, the previous program can be rewritten as follows:

```
template <typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x + y);
```

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

template<typename T1, typename T2>
auto sum(T1& t1, T2& t2) -> decltype(t1 + t2)
{
	return t1 + t2;
}

int main(void)
{
	auto i = 1;
	auto j = 1.3;
	cout << sum(i, j) << endl;
	return 0;
}
```

b) Class-Related Features

Constructors and Initialization

21. Delegating constructors

- **Delegating constructors** allow a constructor in a `class` to call another constructor **within the same** `class`.
- A delegating constructor calls another constructor in the same `class` using **member initializer list syntax**.

Why Use Delegating Constructors?

- Avoids **code duplication** when multiple constructors share initialization logic.
- Ensures a **single initialization path**, reducing the chance of inconsistencies.
- Improves **readability and maintainability**.

```cpp
#include <iostream>
#include <string>
using namespace std;
class Person {
   string name;
   int age;
public:
   // Primary Constructor
   Person(string n, int a) : name(n), age(a) {
      cout << "Primary Constructor: " << name << ", " << age << "\n";
   }

   // Delegating Constructor (calls primary constructor)
   Person(string n) : Person(n, 18) {
      cout << "Delegating Constructor\n";
   }
};

int main() {
   Person p1("Alice", 25);  // Calls primary constructor
   Person p2("Bob");        // Calls delegating constructor
   return 0;
}
```

22. Inherited Constructors

- **Inherited constructors** allow a derived `class` to inherit constructors from its base `class` **without explicitly redefining them**.
- This feature simplifies code and reduces duplication, especially in cases where the derived class **does not need additional initialization logic**.

The Problem Before C++11:

- Before C++11, if you wanted a derived `class` to have the same constructors as its base `class`, you had to manually define them in the derived `class`, even if they just called the base `class` constructors. This was tedious and error-prone.

```cpp
class Base {
public:
   Base(int x) : value(x) {}
   Base(int x, double y) : value(x), other_value(y) {}
   // ... other constructors ...

protected:
```

```
    int value;
    double other_value;
};

class Derived : public Base {
public:
    // Redundant, just calls Base(int x)
    Derived(int x) : Base(x) {}
    // Redundant, just calls Base(int x, double y)
    Derived(int x, double y) : Base(x, y) {}
    // ... other constructors ...
};
```

Inherited Constructors (C++11):

- C++11 provides a way to automatically inherit constructors using the `using` declaration:

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

class Base {
public:
    Base(int x) { cout << "Base Constructor: " << x << "\n"; }
};

class Derived : public Base {
    int y;
public:
    using Base::Base;

    // Explicitly initialize `y` since inherited constructors won't do it
    Derived(int x, int y) : Base(x), y(y) {
        cout << "Derived Constructor: " << y << "\n";
    }
};

int main() {
    Derived d1(42);      // Calls Base(int), does not initialize `y`
    Derived d2(10, 20);  // Calls Base(int) and Derived(int, int)
    return 0;
}
```

23. Default Member Initializers

- **Default member initializers** allow you to initialize **non-static** data members **directly in the `class` definition**, instead of using constructors.

Why Use Default Member Initializers?

- **Reduces redundancy** when multiple constructors need to initialize a member to the same default value.
- **Improves readability** by keeping default values next to member declarations.
- **Ensures consistency** since default values apply to all constructors unless explicitly overridden.
- Syntax:
    - You can initialize a member variable inside the class declaration using either:
        - **Brace {} initialization** (preferred, avoids narrowing conversions).
        - **Equal = initialization** (similar to assignment).

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Example {
   int x = 5;         // Default value
   double b{ 3.14 }; // Default value
   vector<int> numbers{ 1, 2, 3 };
public:
   Example() = default;           // Uses default values
   Example(int val, double d) : x(val), b(d) {}  // Overrides default

   void print() {
      cout << "x: " << x << "\n";
      cout << "b: " << b << "\n";
      for (int num : numbers)
         cout << num << " ";
      cout << "\n";
   }
};
int main() {
   Example e1;    // Uses default x = 5, b = 3.14
   Example e2(10, 2.2); // Uses x = 10, b = 2.2
   e1.print();
   e2.print();
}
```

Special Member Functions

24. Defaulted and Deleted Functions

- C++11 introduced **defaulted** and **deleted** functions to improve control over special member functions like constructors, destructors, and operators.

Defaulted Functions (`= default`)

- A **defaulted function** tells the compiler to generate the function's default implementation.
- Sometimes, if you add certain members (like a **custom destructor, Move Constructor/Assignment Operator**), the compiler **does not** generate special member functions (like the **copy constructor**). In such cases, you can explicitly ask the compiler to generate them using `= default`.
- These are the member functions that the compiler can automatically generate for you if you don't define them yourself:
  - Default constructor: `MyClass()`
  - Destructor: `~MyClass()`
  - Copy constructor: `MyClass(const MyClass& other)`
  - Copy assignment operator: `MyClass& operator=(const MyClass& other)`
  - Move constructor: `MyClass(MyClass&& other)`
  - Move assignment operator: `MyClass& operator=(MyClass&& other)`
- Why?
  - Avoids redundant boilerplate code.
  - Improves clarity and intent.
  - Ensures correct and efficient compiler-generated implementations.

```
class MyClass {
   // ... Other private members ...
public:
   // Explicitly defaulted default constructor
   MyClass() = default;
   // Explicitly defaulted copy constructor
   MyClass(const MyClass& other) = default;
   // Explicitly defaulted copy assignment operator
   MyClass& operator=(const MyClass& other) = default;
   // Explicitly defaulted move constructor
```

```
    MyClass(MyClass&& other) = default;
    // Explicitly defaulted move assignment operator
    MyClass& operator=(MyClass&& other) = default;
    // Explicitly defaulted destructor
    ~MyClass() = default;
};
```

- Example:

```
class Example {
private:
    int* ptr;  // Dynamic resource

public:
    // Constructor
    Example(int val) : ptr(new int(val)) {}
    // Custom destructor
    ~Example() { delete ptr; }

    // The compiler would not generate the copy constructor automatically
    // Explicitly tell the compiler to generate it
    Example(const Example&) = default;
};

int main() {
    Example e1(10);
    Example e2 = e1;  // Allowed because we used `= default`
    return 0;
}
```

Deleted Functions (`= delete`)

- A **deleted function** explicitly prevents a function from being used.
- The `= delete;` tells the compiler: "Do not generate this function. If someone tries to use it, produce a **compile-time error**."
- C++11 **deleted functions** also apply to built-in types.
- Why?
    - Prevents accidental object copying/movement.
    - Enforces design constraints.
    - Improves safety and maintainability.

```
void func(int) {
    cout << "Integer overload\n";
}

void func(double) = delete;  // Disable function for double types

class NonCopyable {
public:
    NonCopyable() = default;
    // Disable copying
    NonCopyable(const NonCopyable&) = delete;
    // Disable copy assignment
    NonCopyable& operator=(const NonCopyable&) = delete;

    void print() { cout << "Instance exists\n"; }
};

int main() {
    NonCopyable obj1;
    // NonCopyable obj2 = obj1;  // ERROR: Copying is deleted!
    obj1.print();
    func(10);       // Allowed
    // func(3.14);  // ERROR: This function is deleted!
}
```

Inheritance and Polymorphism

25. Override and Final

- C++11 introduced the `override` and `final` keywords to improve code clarity and prevent accidental mistakes when working with `virtual` **functions** in inheritance.

Prevent Mistakes When Overriding: `override`

- The `override` keyword is used to explicitly indicate that a member function in a derived `class` is intended to *`override`* a virtual function in its base `class`.
- If you accidentally misspell the function name or change the parameter list, the compiler will issue an error if you use `override`, preventing subtle bugs.

```
class Base {
public:
    virtual void show() {
        cout << "Base::show()\n";
    }
};
```

```
class Derived : public Base {
public:
   // Ensures we are overriding `Base::show()`
   void show() override {
      cout << "Derived::show()\n";
   }
};
```

Prevent Further Overriding: `final`

- The `final` keyword prevents derived classes from **overriding** a `virtual` function.

```
class Base {
public:
   // Cannot be overridden in derived classes
   virtual void display() final {
      cout << "Base::display()\n";
   }
};
class Derived : public Base {
public:
   // Error: Cannot override final function
   // void display() override {
   //     cout << "Derived::display()\n";
   // }
};
```

- The `final` can also be applied to **entire** `class`**es** to prevent inheritance.

```
// Cannot be inherited
class FinalClass final {
public:
   void hello() { cout << "Hello from FinalClass\n"; }
};
// Compilation Error: FinalClass is final
// class DerivedClass : public FinalClass {
// };
```

- Summary

Feature	Purpose
`override`	Ensures a function correctly overrides a base `class` `virtual` function.
`final` (function)	Prevents further overriding of a `virtual` function.
`final` (`class`)	Prevents a `class` from being inherited.

Enumerations

26. Strongly-typed Enumerations

- They offer improved type safety, scoping, and prevent implicit conversions that could lead to errors.
- **Scoped**: Unlike traditional `enum`s, strongly-typed `enum`s are scoped within their declaration. This means you access the enumerators using the `enum`'s name and the scope resolution operator `::`. This prevents name collisions with other `enum`s or variables in the same scope.
- **Type Safety**: Strongly-typed `enum`s do not implicitly convert to `int`eger types. This prevents accidental comparisons or assignments between unrelated `enum` values or between `enum` values and `int`egers. This significantly improves type safety and reduces the risk of errors.
- **Underlying Type**: You can explicitly specify the underlying `int`egral type for a strongly-typed `enum`. This gives you control over the size and range of values the `enum` can hold. If you don't specify it, the default underlying type is `int`.
- **No Implicit Conversions**: As mentioned, there are no implicit conversions to `int`egers or other `enum` types. This eliminates a common source of bugs with traditional `enum`s.

```cpp
#include <iostream>
using namespace std;
enum Color {
   Red,
   Green,
   Blue,
   Orange
};
enum Fruit {
   Apple,
   Banana,
   Grapes
};

int main() {
   // No Error
   if (Color::Blue == Fruit::Grapes) {
      cout << "Comparing Fruits and Color\n";
   }

   return 0;
}
```

```cpp
#include <iostream>
using namespace std;
enum class Color: int {
   Red,
   Green,
   Blue,
   Orange
};
enum class Fruit: char {
   Apple,
   Banana,
   Grapes
};

int main() {
   // Error
   if (Color::Blue == Fruit::Grapes) {
      cout << "Comparing Fruits and Color\n";
   }

   return 0;
}
```

Type Conversion

27. Explicit Conversion Operator

- C++11 introduced the `explicit` keyword which can be applied to **conversion operators** (and **constructors** with a single argument) to **prevent** unintended implicit conversions that could lead to subtle bugs or incorrect behaviour.
    - A **conversion operator** is a special member function that allows an object of **that** `class` to be converted to another type.
    - Example:

```
class MyNumber {
private:
   double value;

public:
   MyNumber(double val) : value(val) {}

   // Conversion operator to int
   operator int() const {
      // Truncate to integer
      return static_cast<int>(value);
   }
   // Conversion operator to string
   operator string() const {
      return to_string(value);
   }

   double getValue() const { return value; }
};
int main() {
   MyNumber num(3.14);

   // Implicit conversion to int using the operator int()
   int int_val = num;
   // Implicit conversion to string using the operator string()
   std::string str_val = num;

   std::cout << "Integer value: " << int_val << "\n";
   std::cout << "String value: " << str_val << "\n";
}
/*
* Integer value: 3
* String value: 3.140000
*/
```

- Using `explicit` to prevent implicit conversion.

```
class MyNumber {
private:
   double value;

public:
   MyNumber(double val) : value(val) {}

   // Conversion operator to int
   explicit operator int() const {
      // Truncate to integer
      return static_cast<int>(value);
   }

   // Conversion operator to string
   explicit operator string() const {
      return to_string(value);
   }

   double getValue() const { return value; }
};
int main() {
   MyNumber num(3.14);

   // Error: cannot convert from 'MyNumber' to 'int'
   // int int_val = num;
   // Error: cannot convert from 'MyNumber' to 'std::string'
   // std::string str_val = num;

   int int_val = static_cast<int>(num);
   std::string str_val = static_cast<std::string>(num);
   std::cout << "Integer value: " << int_val << "\n";
   std::cout << "String value: " << str_val << "\n";
}
/*
* Integer value: 3
* String value: 3.140000
*/
```

c) Standard Library Features

Smart Pointers

- C++11 introduced **smart pointers** to **manage dynamic memory safely and automatically**, reducing the risk of memory leaks and dangling pointers.

28. Exclusive Ownership: `std::unique_ptr`

- A `std::unique_ptr` **owns** a dynamically allocated object exclusively.
- **Cannot be copied** (ownership cannot be shared).
- **Can be moved** to transfer ownership.

```
#include <memory>  // For smart pointers

class Example {
public:
   Example() { std::cout << "Example Created\n"; }
   ~Example() { std::cout << "Example Destroyed\n"; }
   void sayHello() { std::cout << "Hello from Example\n"; }
};

int main() {
   std::unique_ptr<Example> ptr1 =
      std::make_unique<Example>();  // Allocates memory
   ptr1->sayHello();

   // Error: Cannot copy a unique_ptr
   // std::unique_ptr<Example> ptr2 = ptr1;

   // Ownership transferred
   std::unique_ptr<Example> ptr2 = std::move(ptr1);
   if (!ptr1) {
      std::cout << "ptr1 is now nullptr\n";
   }

   return 0;
}  // `ptr2` goes out of scope, memory is freed
```

29. Shared Ownership: `std::shared_ptr`

- A `std::shared_ptr` allows **multiple smart pointers** to share ownership of the same object.
    - Internally maintains a **reference count**.
    - When the last `shared_ptr` goes out of scope, the object is deleted.

```
#include <memory>

class Example {
public:
   Example() { std::cout << "Example Created\n"; }
   ~Example() { std::cout << "Example Destroyed\n"; }
};

int main() {
   // Creates object
   std::shared_ptr<Example> ptr1 = std::make_shared<Example>();
   std::shared_ptr<Example> ptr2 = ptr1;  // Shared ownership

   std::cout << "Reference Count: "
      << ptr1.use_count() << "\n";  // 2 references

   ptr1.reset();  // ptr1 releases ownership
   std::cout << "Reference Count: "
      << ptr2.use_count() << "\n";  // 1 reference left

   return 0;
}  // `ptr2` goes out of scope, object deleted
```

30. Prevent Circular References: `std::weak_ptr`

- A `std::weak_ptr` is a **non-owning** smart pointer that **references an object managed by** `std::shared_ptr`.
    - **Does NOT increase reference count**.
    - Used to **avoid cyclic dependencies**.

```cpp
#include <memory>

class B;  // Forward declaration

class A {
public:
    std::shared_ptr<B> b_ptr;
    ~A() { std::cout << "A Destroyed\n"; }
};

class B {
public:
    // Weak pointer avoids cyclic reference
    std::weak_ptr<A> a_ptr;
    ~B() { std::cout << "B Destroyed\n"; }
};

int main() {
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();

    a->b_ptr = b;
    b->a_ptr = a;  // Weak reference prevents memory leak

    return 0;  // Both objects are properly destroyed
}
```

Summary Table

Smart Pointer	Ownership	Copyable?	Use Case
`std::unique_ptr`	**Exclusive**	❌ No	Managing a single resource (e.g., file, socket, heap memory)
`std::shared_ptr`	**Shared**	✅ Yes	When multiple owners need access to the same object
`std::weak_ptr`	**Non-owning**	✅ Yes	Breaking cyclic references in `std::shared_ptr`

Containers

31. Hash Tables

- C++11 introduced the following hash-based containers, which are also known as unordered associative containers:
    - `std::unordered_map`: This container stores elements as key-value pairs, where each key is **unique**. It provides fast average-case lookup, insertion, and deletion of elements using a hash function.
    - `std::unordered_set`: This container stores **unique** elements, and like `std::unordered_map`, it uses a hash function to provide fast average-case access.
    - `std::unordered_multimap`: This is similar to `std::unordered_map`, but it allows **duplicate** keys.
    - `std::unordered_multiset`: This is similar to `std::unordered_set`, but it allows **duplicate** elements.
- These containers are defined in the `<unordered_map>` and `<unordered_set>` headers.
- They offer average-case **constant time complexity** for common operations like lookup, insertion, and deletion, making them very efficient for many applications.

32. Forward List

- C++11 introduced `std::forward_list`, a **singly linked list** available in the `<forward_list>` header.
- Unlike `std::list` (which is a **doubly linked list**), `std::forward_list` only maintains **a single pointer per node (to the next element)**, making it more **memory efficient** but also **less powerful** (no backward traversal).

```
std::forward_list<int> flist = { 10, 20, 30, 40 };
```

33. Array Container

- C++11 introduced `std::array`, a container that represents **fixed-size arrays** with some important features and benefits over the built-in C++ arrays.
- Unlike traditional C++ arrays (which are of fixed size and lack member functions), `std::array` provides a more modern and safer interface for handling arrays. It combines the **efficiency of C-style arrays** with the **power of STL containers**.

```cpp
#include <array>

int main() {
   // Create an array of 5 integers
   std::array<int, 5> arr = { 1, 2, 3, 4, 5 };

   // Access elements using []
   std::cout << "Element at index 2: " << arr[2] << std::endl;

   // Access elements using at() (with bounds checking)
   std::cout << "Element at index 3: " << arr.at(3) << std::endl;

   return 0;
}
```

For more on Containers, refer "STL Containers.docx"

For more on Algorithms, refer "STL Algorithms.docx"

For more on Concurrency, refer "Parallel and Concurrent Programming with C++.docx"

Time Utilities

34. The `std::chrono`

- C++11 introduced the `<chrono>` library, which provides a robust way to handle time-related operations, including **measuring time intervals**, working with different **time units**, and setting time points.

Core Components of `<chrono>`

- The `<chrono>` library has three primary components:
  A. `std::chrono::duration` → Represents a time duration.
  B. `std::chrono::time_point` → Represents a specific point in time.
  C. `std::chrono::clock` → Provides the current time.

A. Working with `std::chrono::duration`

- A **duration** represents a **span of time** and is **parameterized by**:
  - A representation type (e.g., `int`, `double`).
  - A time unit (e.g., seconds, milliseconds, nanoseconds).
- A `std::chrono::duration` represents a span of time, meaning a measurable interval or period between two points in time.
  - For example:
    - `5` seconds
    - `3.5` hours
  - All of these describe **durations**, or "how long" something takes.
- What does "**parameterized by**" mean?
  - The `std::chrono::duration` **template** is parameterized by two things:
    - A **representation type** → Specifies how to store the duration value (e.g., `int`, `double`).
    - A **time unit** → Specifies the unit of measurement (e.g., seconds, milliseconds, nanoseconds).
- Example:

```
std::chrono::duration<int, std::milli> five_hundred_milliseconds
   = std::chrono::milliseconds(500);
```

B. Working with `std::chrono::time_point`

- A `std::chrono::time_point` represents a specific point in time. Think of it like a timestamp.
- It's different from `std::chrono::duration`, which represents a span of time. A `time_point` is like a **coordinate on a timeline**, while a `duration` is like the **distance between two coordinates**.
- It is essentially a combination of a **clock** and a **duration**.

```
template <class Clock, class Duration = typename Clock::duration>
class time_point;
```

C. Working with `std::chrono::clock`

- `std::chrono` provides several clock types, each with its own characteristics.
- `std::chrono::system_clock`:
    - Represents the system's wall clock time. This is the time you'd typically see on your computer's clock.
    - Can be adjusted by the user or by network time synchronization (NTP). This means it's not guaranteed to be monotonic (it can jump forward or backward).
    - Useful for displaying the current time to the user or for logging timestamps.
- `std::chrono::steady_clock`:
    - Represents a clock that is guaranteed to be monotonic. It never goes backward.
    - The actual time it represents is not necessarily related to wall-clock time. It's primarily for measuring durations.
    - Ideal for benchmarking code or measuring time intervals. Use this for any timing-related tasks where monotonicity is essential.
- `std::chrono::high_resolution_clock`:
    - Represents the clock with the finest possible resolution available on the system.
    - It might be an alias for `system_clock` or `steady_clock`. The C++ standard doesn't specify which.

- Use this when you need the highest possible precision, but be aware that it might not always be the most accurate (due to underlying hardware limitations).

Support for regular expressions

35. Regular Expressions

- C++11 introduced **regular expressions (regex)** via the `<regex>` library, which allows you to search, match, and manipulate strings using **patterns**.
- In C++, regex support is provided via:
  - `std::regex` → Represents a regex pattern.
  - `std::smatch/std::cmatch` → Stores match results (for `string` / C-string).
  - `std::regex_search` → Searches for a match in a string.
  - `std::regex_match` → Checks if the entire string matches.
  - `std::regex_replace` → Replaces matched substrings.

```
#include <regex>

int main() {
   string email = "user@example.com";
   regex pattern(R"([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})");

   if (regex_match(email, pattern)) {
      cout << "Valid email\n";
   }
   else {
      cout << "Invalid email\n";
   }
}
```

Tuples

36. The `std::tuple`

- `std::tuple` is a **fixed-size** collection that can hold elements of **different types** (heterogeneous data). It is part of the `<tuple>` header.
- Why Use `std::tuple`?
    - Can store multiple values of different types.
    - Useful when returning multiple values from a function.
    - Provides type-safe access to elements.

```
int main() {
   std::tuple<int, double, std::string> myTuple(42, 3.14, "Hello");

   std::cout << "First: " << std::get<0>(myTuple) << "\n";
   std::cout << "Second: " << std::get<1>(myTuple) << "\n";
   std::cout << "Third: " << std::get<2>(myTuple) << "\n";
}
```

Random Number Generation

37. Random Number Generation

- C++11 introduced a **powerful and flexible** random number generation system in the `<random>` header.
- It replaces the old C-style functions like `rand()` and `srand()` with a modern, more robust approach.
- Random number generation in C++11 involves **three key components**:

Component	Purpose
Engine (`std::mt19937`)	Generates pseudo-random numbers
Distribution (`std::uniform_int_distribution`)	Defines how numbers are distributed
Seed (`std::random_device`)	Ensures randomness between runs

a. The Engine (e.g., `std::mt19937`):

- **What it does**: The engine is the core of the random number generator. It's an algorithm that produces a sequence of numbers that appear random. However, these

numbers are actually deterministic – if you start the engine with the same initial value (the **seed**), it will always produce the same sequence. This is why they are called pseudo random numbers. They are not truly random, but they are good enough for many practical applications.

- **Analogy**: Think of the engine as a complex mathematical formula that **spits out numbers**.
- `std::mt19937` (**Mersenne Twister**): This is a very common and powerful pseudo-random number engine. It's generally a good default choice. It's fast and produces high-quality pseudo-random numbers.

b. The Distribution (e.g., `std::uniform_int_distribution`):

- **What it does**: The distribution **takes the raw output from the engine** and shapes it to fit a specific probability distribution. For example, `std::uniform_int_distribution` produces numbers that are evenly distributed within a specified range (like rolling a fair die). Other distributions exist for generating numbers with different patterns (Gaussian, normal, etc.).
- **Analogy**: The distribution is like a **filter** or a Mold. It takes the raw numbers from the engine and transforms them into the desired form.
- `std::uniform_int_distribution`: This distribution generates integers within a given range (inclusive). For example, `std::uniform_int_distribution <int>(1, 6)` will produce integers between `1` and `6` (inclusive), each with equal probability.

c. The Seed (e.g., `std::random_device`):

- **What it does**: The seed is the **initial value** that you give to the **engine**. It's like the starting point for the engine's sequence of numbers. If you use the same seed every time you run your program, the engine will produce the same sequence of "random" numbers. This is often useful for debugging or testing, where you want predictable results. However, for true randomness between runs, you need a different seed each time.
- **Analogy**: The seed is like **setting the initial position of the gears** in your engine.

- `std::random_device`: This `class` is used to obtain a non-deterministic seed from the operating system (if available). It's the best way to get a truly random starting point for your engine.

Steps to Generate Random Numbers

- Step 1: Include the Required Headers

```
#include <iostream>    // For Input/Output
#include <random>      // For random number generation
```

- Step 2: Create a Random Device

```
std::random_device rd;  // Seed generator
```

- Step 3: Initialize the Engine with the Seed

```
// Mersenne Twister Engine is initialized with
// seed from random device
std::mt19937 engine(rd());
```

- Step 4: Define a Distribution

```
// Range: 1 to 100
std::uniform_int_distribution<int> dist(1, 100);
```

- Step 5: Generate Random Numbers

```
std::cout << "Random Number: " << dist(engine) << "\n";
```

I/O Facilities

- C++11 introduced **new standard functions** in `<string>` and `<sstream>` for **converting between strings and numbers** more easily and safely.

38. Converting Numbers to Strings

- Before C++11, we used `std::stringstream` for conversion, which was **verbose**. Now, `std::to_string()` simplifies it.

```
std::cout << "Integer as string: " << to_string(43)<< "\n";
std::cout << "Double as string: " << to_string(3.14) << "\n";
```

39. Converting Strings to Numbers

- C++11 provides functions to convert **strings to numbers**, replacing `std::stringstream` for this purpose.

Function	Converts to
`std::stoi(str)`	`int`
`std::stol(str)`	`long`
`std::stoll(str)`	`long long`
`std::stof(str)`	`float`
`std::stod(str)`	`double`
`std::stold(str)`	`long double`

- Example:

```
int i = std::stoi("123");
float f = std::stof("123.456");
double d = std::stod("123.456");
```

40. Compile-Time Rational Arithmetic

- C++11 introduced `std::ratio` in `<ratio>` to **perform fractional (rational) arithmetic at compile-time**.
- It is particularly useful for **unit conversions, time calculations (`std::chrono`), and precise mathematical computations** where integer division may cause precision loss.

What is `std::ratio`?

- A `std::ratio<N, D>` represents a **fraction** `N/D` (numerator/denominator), but it's evaluated **at compile time**.

```
#include <ratio>

int main() {
   using r = ratio<3, 4>;  // Represents 3/4

   cout << "Ratio: " << r::num <<
      "/" << r::den << "\n";  // Output: 3/4
```

```
    using r1 = ratio<1, 2>;
    using r2 = ratio<1, 3>;

    using sum = ratio_add<r1, r2>;  // (1/2 + 1/3) = 5/6
    using product = ratio_multiply<r1, r2>;  // (1/2 * 1/3) = 1/6

    cout << "Sum: " << sum::num <<
       "/" << sum::den << "\n";  // Output: 5/6
    cout << "Product: " << product::num
       << "/" << product::den << "\n";   // Output: 1/6
}
```