C++ 14 Language Features

Contents

[a) Core Language Features 2](#_Toc190727455)

[1. Digit Separators 2](#_Toc190727456)

[2. Binary Literals 2](#_Toc190727457)

[3. Generic Lambdas 3](#_Toc190727458)

[4. Generalized Lambda Captures 4](#_Toc190727459)

[5. Return Type Deduction for Functions 5](#_Toc190727460)

[6. The decltype(auto) 6](#_Toc190727461)

[7. Relaxed constexpr Restrictions 7](#_Toc190727462)

[8. Variable Templates 9](#_Toc190727463)

[9. Deprecated Attribute 9](#_Toc190727464)

[10. Sized Deallocation 10](#_Toc190727465)

[b) Class-Related Features 12](#_Toc190727466)

[11. Default Member Initializers for Bit-Fields 12](#_Toc190727467)

[12. Relaxed Requirements for constexpr Member Functions 12](#_Toc190727468)

[13. Inheriting Constructors 13](#_Toc190727469)

[14. Aggregate Initialization with Member Initializers 14](#_Toc190727470)

[c) Standard Library Features 16](#_Toc190727471)

[15. The std::make\_unique 16](#_Toc190727472)

[16. The std::exchange 17](#_Toc190727473)

[17. Heterogeneous Lookup in Associative Containers 18](#_Toc190727474)

[18. The std::quoted 19](#_Toc190727475)

# Core Language Features

### Digit Separators

* In **C++14**, **digit separators** (') were introduced to improve the readability of numeric literals.
* The **digit separator** is a **single quote (')** that you can place anywhere inside a numeric literal to make large numbers easier to read. It has **no effect on the value**; it is purely for formatting.
* Why are Digit Separators Useful?
  + When dealing with large numbers, it can be difficult to read and understand them at a glance. The digit separator allows programmers to group digits in a meaningful way, similar to how commas are used in everyday numbers.
  + For example:
    - 1000000 is harder to read.
    - 1'000'000 is easier to read.

|  |
| --- |
| using namespace std;  int main() {  int oneMillion = 1'000'000; // 1 million  int tenLakh = 10'00'000; // 1 million = 10 lakh  cout << "1 million: " << oneMillion << endl;  cout << "10 lakh: " << tenLakh << endl;  return 0;  } |

### Binary Literals

* A **binary literal** is a way to specify numbers in base-2 (binary) directly in C++ code using the 0b or 0B prefix.
* This allows programmers to define numbers using only 0s and 1s, which is useful for low-level programming, bitwise operations, and hardware programming.
* Example 1: A Simple Bitmask

const int FEATURE\_A = 0b0000'0001; // Bit 0

const int FEATURE\_B = 0b0000'0010; // Bit 1

const int FEATURE\_C = 0b0000'0100; // Bit 2

* Example 2:

|  |
| --- |
| using namespace std;  int main() {  int binaryNumber = 0b101010; // 42 in decimal  cout << "Binary 0b101010 is: " << binaryNumber << endl;  int anotherBinary = 0b1111; // 15 in decimal  cout << "Binary 0b1111 is: " << anotherBinary << endl;  int largeBinary = 0b1101'0101'1010'0110; // Grouped for better readability  cout << "Binary 0b1101'0101'1010'0110 is: " << largeBinary << endl;  } |

### Generic Lambdas

* A **lambda function** is an **anonymous function** (a function without a name) that you can define inline.
* In **C++14**, you can use auto as a lambda parameter type to make the lambda **type-generic** (like a function template).

|  |
| --- |
| int main() {  auto add = [](auto a, auto b) {  return a + b;  }; // Works with any type  cout << add(3, 4) << endl; // Works with int (3 + 4)  cout << add(3.5, 2.1) << endl; // Works with double (3.5 + 2.1)  cout << add(string("Hi "), "C++!") << endl; // Works with strings  } |

* Why Use Generic Lambdas?
  + **More Flexible**: Works with any data type (int, double, string, etc.).
  + **Avoids Code Duplication**: No need to write multiple versions of the same function for different types.
  + **Simplifies Code**: You don’t have to write explicit template functions.

#### **Comparison: C++11 vs. C++14**

| **Feature** | **C++11 (Fixed Type)** | **C++14 (Generic)** |
| --- | --- | --- |
| **Syntax** | [](int a, int b) { return a + b; } | [](auto a, auto b) { return a + b; } |
| **Works with** int**?** | ✅ | ✅ |
| **Works with** double**?** | ❌ (Would need another lambda) | ✅ |
| **Works with** string**?** | ❌ | ✅ |

### Generalized Lambda Captures

* What Are Lambdas?
  + Lambdas, or lambda expressions, are a way to define small, unnamed functions directly in your code. They can be very useful for quick, short-lived tasks, especially when working with algorithms and callbacks.
* What Are Lambda Captures?
  + Lambda captures allow lambdas to **capture and use variables from their enclosing scope** (the scope in which they are defined). This can be useful when you want the lambda to have access to local variables.
* Generalized Lambda Captures in C++14
  + Before **C++14**, lambdas could **only capture** variables **by value** ([=]) or **by reference** ([&]).
  + C++14 introduced **generalized lambda captures**, allowing us to **move objects** and **initialize new variables inside the capture list**.
* Example:
  + Before C++14, capturing **move-only** objects (like std::unique\_ptr) was not possible.
  + Before C++14, Capture expressions (e.g., x + 5) were not possible.

|  |
| --- |
| int main() {  auto ptr = std::make\_unique<int>(42);  // Move ptr into lambda using generalized capture  auto lambda1 = [uniquePtr = std::move(ptr)]() {  std::cout << \*uniquePtr << "\n";  };  lambda1(); // Output: 42  auto x = 10;  auto lambda2 = [y = x + 5]() { // Capturing an expression (x + 5)  std::cout << y << "\n";  };  lambda2();  } |

### Return Type Deduction for Functions

* In **C++14**, **Return Type Deduction for Functions** was introduced to make function definitions more flexible by allowing the compiler to deduce the return type automatically.
* What's Return Type Deduction?
  + In C++, every function has a return type – the type of value the function gives back when it's finished. Before C++14, you had to write this return type explicitly in the function's declaration.
  + Example:

|  |
| --- |
| // Explicit return type: int  int add(int x, int y) {  return x + y;  }  // Explicit return type: double  double divide(double x, double y) {  return x / y;  } |

* In **C++14**, you can **omit the return type**, and the compiler will automatically deduce it from the return statement.

|  |
| --- |
| // Compiler deduces return type  auto add(int a, int b) {  return a + b;  } |

**C++11 vs. C++14 Return Type Deduction**

| **Feature** | **C++11** | **C++14** |
| --- | --- | --- |
| **Function Return Type** | Must be explicitly declared | Can be deduced with auto |
| **Syntax** | int add(int a, int b){ return a + b; } | auto add(int a, int b) { return a + b; } |
| **Works with Iterators?** | Must specify complex type | Deduces iterator type automatically |
| **Template Functions?** | Requires explicit return type | Deduces return type automatically |

### The decltype(auto)

* In **C++14**, decltype(auto) was introduced to improve **return type deduction**, making it more flexible than just using auto.
* What is decltype?
  + decltype is an operator that lets you get the **type** of an expression.
  + Example:

int x = 10;

decltype(x) y = 20; // y is of type int (same as x)

* The Problem: Type Decay and References
  + Sometimes, decltype doesn't give you exactly the type you might expect, especially when dealing with functions that return references or when dealing with type decay.

|  |
| --- |
| int& get\_value() {  static int value = 42;  return value;  }  int main() {  // Type of ref1 is int  auto ref1 = get\_value();  ref1 = 50;  cout << "New Value: " << get\_value();  }  // Output: New Value: 42 |

* What is decltype(auto)?
  + decltype(auto) allows the compiler to **deduce** the exact type, just like decltype(expr) which evaluates the type of the expression.
  + It preserves references and const qualifiers, unlike auto, which may strip them away.
  + It is useful for return type deduction when you want to maintain the original type precisely.
* Example:

|  |
| --- |
| int& get\_value() {  static int value = 42;  return value;  }  int main() {  // Type of ref1 is int&  decltype(auto) ref1 = get\_value();  ref1 = 50;  cout << "New Value: " << get\_value();  }  // Output: New Value: 50 |

#### **Comparison Table**

| **Feature** | auto | decltype | decltype(auto) |
| --- | --- | --- | --- |
| **Deduces type?** | ✅ Yes | ❌ No (only evaluates type) | ✅ Yes |
| **Removes reference?** | ✅ Yes | ❌ No | ❌ No |
| **Removes** const**?** | ✅ Yes | ❌ No | ❌ No |
| **Used for function return type?** | ✅ Yes | ❌ No (used for type evaluation) | ✅ Yes |
| **Supports complex expressions?** | ✅ Yes | ✅ Yes | ✅ Yes |

### Relaxed constexpr Restrictions

* The constexpr is a keyword that you can use to **declare variables** or **functions** that can be **evaluated** at **compile time**.

#### The Problem: Restrictions in C++11

* In C++11, constexpr functions had very strict limitations.
  + They **could not have loops** (for, while, etc.).
  + They **could not use local variables**.
  + They **could not use** if **statements**.
  + Every constexpr function had to be one return statement.

#### The Solution: Relaxed Restrictions in C++14

* C++14 relaxed these rules, making constexpr functions more flexible.
* Now, constexpr functions can have:
  + **Loops** (for, while, do-while)
  + **Multiple statements** (instead of one return)
  + **Local variables** (with auto)
  + **Conditional statements** (if, switch, etc.)
* Example:

|  |
| --- |
| // Only single return allowed in C++11  constexpr int factorial(int n) {  return (n <= 1) ? 1 : (n \* factorial(n - 1));  }  constexpr int factorial(int n) {  int result = 1;  // Loops are allowed in C++14  for (int i = 2; i <= n; ++i) {  result \*= i;  }  return result;  } |

#### **Summary: What Changed in C++14?**

| **Feature** | **C++11** | **C++14** |
| --- | --- | --- |
| Multiple statements | ❌ No | ✅ Yes |
| Loops (for, while) | ❌ No | ✅ Yes |
| Local variables | ❌ No | ✅ Yes |
| Conditional statements (if, switch) | ❌ No | ✅ Yes |
| Modifying variables | ❌ No | ✅ Yes |

### Variable Templates

* Templates allow us to write generic code that works for multiple types.
* Before **C++14**, templates were used only for **functions** and **classes**.
* In **C++14**, we can now create **variable** **templates**.

#### What is a Variable Template?

* A variable template allows defining a **variable with a generic type**, just like function templates.
* Example:

|  |
| --- |
| // Define a variable template  template<typename T>  constexpr T pi = T(3.1415926535897932385);  int main() {  cout << pi<int> << "\n";  cout << fixed << setprecision(7);  cout << pi<float> << "\n";  cout << fixed << setprecision(15);  cout << pi<double> << "\n";  }  /\*  3  3.1415927  3.141592653589793  \*/ |

* The same pi variable works for different types (int, float, double).
* The constexpr ensures compile-time computation.
* No need to create multiple constants for each type.

### Deprecated Attribute

* The [[deprecated]] **attribute** was introduced in **C++14** to mark **functions, classes, variables, or enumerations** as outdated or obsolete. This helps warn developers when they use old code that might be removed in the future.

#### Why Do We Need [[deprecated]]?

* Imagine you're working on a large C++ project, and some functions or classes are no longer recommended because:
  + They are inefficient (e.g., replaced with better alternatives).
  + They have security issues.
  + They will be removed in future versions.
* Instead of removing the function immediately, you mark it as deprecated so that existing code still works, but compilers show a warning when someone tries to use it.

|  |
| --- |
| // Marking a function as deprecated with a message  [[deprecated("Use newFunction() instead.")]]  void oldFunction() {  cout << "This function is deprecated!\n";  }  // New recommended function  void newFunction() {  cout << "This is the new function!\n";  }  struct [[deprecated("Use NewClass instead.")]] OldClass {  int value;  };  struct NewClass {  int value;  };  int main() {  OldClass obj; // Warning: "Use NewClass instead."  oldFunction(); // Warning with message: "Use newFunction() instead."  newFunction(); // No warning  } |

### Sized Deallocation

* In **C++14**, the memory deallocation functions (operator delete and operator delete[]) can optionally receive the **size of the object** being deleted. This is known as **sized deallocation**.

#### Why is Sized Deallocation Useful?

* Optimization for Custom Allocators:
  + Some custom memory allocators need to know the exact size of the object when freeing memory.
  + Before C++14, operator delete had to figure out the size by other means, which could be inefficient.
* Potential Performance Improvements:
  + If the deallocation function knows the size upfront, it can free memory faster.
  + This helps in memory pools or custom allocators that manage different object sizes.

#### How Does It Work?

* In C++14, the compiler can pass the **size** of the object to operator delete, allowing optimized memory deallocation.
* C++11 and Before:

void operator delete(void\* ptr) noexcept;

void operator delete[](void\* ptr) noexcept;

* C++14 (Sized Deallocation Added):

void operator delete(void\* ptr, std::size\_t size) noexcept;

void operator delete[](void\* ptr, std::size\_t size) noexcept;

# Class-Related Features

### Default Member Initializers for Bit-Fields

* In **C++14**, **bit-fields** (integer members with a specified number of bits) can now have **default values** directly in the class or struct definition.

|  |
| --- |
| struct Flags {  unsigned int a : 1 {1}; // Default: 1  unsigned int b : 2 {0}; // Default: 0  unsigned int c : 3 {5}; // Default: 5  };  int main() {  Flags f; // Uses default values  std::cout << f.a << " " << f.b  << " " << f.c << "\n"; // Output: 1 0 5  } |

### Relaxed Requirements for constexpr Member Functions

* What is constexpr?
  + The constexpr keyword is used to define functions and variables that can be evaluated at compile-time. This can help optimize your code by performing calculations during compilation rather than at runtime.
* What are Member Functions?
  + Member functions are functions that belong to a class. They can access and modify the data members of the class.

#### Changes in C++14

* Before C++14, constexpr member functions were implicitly const. This means that they couldn't modify the object's state (i.e., they couldn't change the values of data members).
* In C++14, the requirement that constexpr member functions be const was relaxed, allowing constexpr member functions to modify the object's state if necessary.
* The constexpr keyword ensures that the member functions can be evaluated at compile time if the context allows it, but it doesn't prevent them from being used at runtime.

|  |
| --- |
| // A class with constexpr member functions  class Counter {  private:  int count;  public:  // Constructor to initialize count  constexpr Counter(int initialCount) : count(initialCount) {}  // constexpr member function to increment the count  constexpr Counter increment() const {  return Counter(count + 1);  }  // constexpr member function to get the count  constexpr int getCount() const {  return count;  }  };  int main() {  // Create a Counter object using a constexpr expression  constexpr Counter counter = Counter(0);  counter.increment().increment();  // Get the count at compile time  constexpr int result = counter.getCount();  cout << "Count1: " << result << endl;  cout << "Enter a value: ";  int i; cin >> i;  Counter counter2 = Counter(i);  counter2.increment().increment();  auto result1 = counter2.getCount();  cout << "Count2: " << result1 << endl;  return 0;  } |

### Inheriting Constructors

* In **C++14**, the ability to **inherit constructors** from a **base class** was introduced, making it easier to write classes that need to inherit constructors from their base classes.

#### What is Constructor Inheritance?

* In C++11, when you create a derived class, you had to explicitly define constructors for the derived class, even if the constructor was just calling the base class constructor. This could lead to redundant code.
* With constructor inheritance (introduced in C++11) and improved in C++14, the derived class can inherit the constructor(s) from the base class, reducing redundancy.

|  |
| --- |
| class Base {  public:  Base(int x) {  cout << "Base constructor called with value: "  << x << endl;  }  };  class Derived : public Base {  public:  // In C++11, we used to write a constructor in Derived  // class that calls the constructor of the Base class.  //Derived(int x) : Base(x) {  // cout << "Derived constructor called with value: "  // << x << endl;  //}  // C++14: Inherit constructors from Base class  using Base::Base;  };  int main() {  Derived obj(10);  return 0;  } |

### Aggregate Initialization with Member Initializers

* In **C++14**, aggregate classes (simple structures or classes without constructors) can now **use default member initializers** while still allowing aggregate initialization.

#### What is Aggregate Initialization?

* Aggregate initialization is a way to initialize structs or `es **using braces** {} **without defining a constructor**.

|  |
| --- |
| struct Point {  int x = 10; // Default member initializer  int y = 20; // Default member initializer  };  int main() {  // This was NOT allowed in C++11:  // ERROR: Default member initializers prevent aggregate initialization.  // Point p = {5, 6};  Point p; // Works (uses default values: x=10, y=20)  std::cout << p.x << ", " << p.y << "\n";  // C++14 allows aggregate initialization with default member initializers  Point p1; // Uses default values: x = 10, y = 20  Point p2 = { 5 }; // x = 5, y = 20 (default is used for missing values)  Point p3 = { 3, 4 }; // x = 3, y = 4 (all values provided)  } |

# Standard Library Features

* In **C++14**, **standard user-defined literals** were introduced, allowing developers to create more readable and expressive code with **custom literals** for common types like std::string, std::chrono, and std::complex.

#### What Were the Limitations Before C++14?

* Before C++14, it was possible to define custom literals, but there were no built-in support for literals of common types like std::string, std::chrono, or std::complex. This meant you had to define your own literal operators for these types.

|  |
| --- |
| using namespace std;  using namespace std::chrono;  int main() {  std::string s = "Hello World!"s;  auto duration = 5s;  std::complex<double> c = 3.0 + 4.0i;  cout << "String: " << s << endl;  cout << "Duration: " << duration.count()  << " seconds" << endl;  cout << "Complex: " << c << endl;  return 0;  } |

### The std::make\_unique

* In **C++14**, the std::make\_unique function was introduced as a **safe, efficient, and convenient** way to create std::unique\_ptr objects.

|  |
| --- |
| int main() {  // C++11: But using new manually is error-prone and less readable.  std::unique\_ptr<int> ptr1(new int(42)); // Manually allocating memory  std::cout << \*ptr1 << std::endl; // Output: 42  // C++14: std::make\_unique() is a better way.  auto ptr2 = std::make\_unique<int>(42); // Creates a unique\_ptr<int>  std::cout << \*ptr2 << std::endl; // Output: 42  return 0;  } |

#### Summary

* std::make\_unique (introduced in C++14) is a safer, cleaner, and more efficient way to create std::unique\_ptr.
* It eliminates the need for new, preventing memory leaks and improving exception safety.
* Supports single objects (std::make\_unique<int>(42)) and arrays (std::make\_unique<int[]>(5)).
* Use it whenever possible, except when a custom deleter is required.

#### **Quick Comparison:** new **vs.** std::make\_unique

| **Feature** | new **with** std::unique\_ptr | std::make\_unique |
| --- | --- | --- |
| **Memory Safety** | ❌ Manual new/delete needed | ✅ No need for new or delete |
| **Exception Safety** | ❌ Risk of leaks if an exception occurs | ✅ Ensures safe memory allocation |
| **Code Readability** | ❌ Verbose and error-prone | ✅ Concise and clean |
| **Array Support** | ✅ Supports new[] but requires delete[] | ✅ Supports arrays automatically |

### The std::exchange

* The std::exchange is a utility function introduced in C++14 that:
  + Replaces the value of an object with a new value.
  + Returns the old value of the object before the replacement.

|  |
| --- |
| #include <utility> // exchange  int main() {  int x = 10;  // x becomes 20, and old\_value stores 10  int old\_value = exchange(x, 20);  cout << "Old Value: " << old\_value << "\n"; // Output: 10  cout << "New Value of x: " << x << "\n"; // Output: 20  return 0;  } |

### Heterogeneous Lookup in Associative Containers

#### What is Heterogeneous Lookup?

* Before C++14, when searching for a key in associative containers (std::set, std::map, std::unordered\_map, etc.), you had to use exactly the same key type.
* C++14 introduced heterogeneous lookup, which allows searching using different but compatible types (e.g., looking up a std::string key using a const char\*).

#### Why is This Useful?

* **Efficiency**: Avoids unnecessary conversions when searching.
* **Flexibility**: You can search using a different but compatible type.
* **Performance**: Reduces object creation overhead.

|  |
| --- |
| int main() {  std::set<std::string> mySet = { "apple", "banana", "cherry" };  // Searching using const char\* (does NOT work in C++11)  //auto it = mySet.find("banana"); // ERROR: Must convert to std::string  auto it1 = mySet.find(std::string("banana")); // Works but inefficient  // C++14: Searching using const char\*  auto it2 = mySet.find("banana");  if (it2 != mySet.end())  std::cout << "Found: " << \*it2 << "\n";  } |

#### How Does This Work?

* C++14 Adds Transparent Comparators (std::less<>)
* C++14 allows std::set, std::map, etc., to use **transparent comparators**, such as std::less<> instead of std::less<T>.

#### But what are transparent comparators?

* A **transparent comparator** is a special type of comparator that allows **heterogeneous lookup**—meaning you can **search** for elements in a std::set or std::map using different but compatible types **without unnecessary conversions**.

#### **Summary**

| **Feature** | std::less<T> | std::less<> **(Transparent)** |
| --- | --- | --- |
| Requires exact type match? | ✅ Yes | ❌ No |
| Supports const char\* vs std::string? | ❌ No | ✅ Yes |
| Unnecessary object creation? | ✅ Yes | ❌ No |
| Performance overhead? | ✅ High | ❌ Low |
| Introduced in | **C++98** | **C++14** |

### The std::quoted

* The std::quoted is a C++14 feature that simplifies handling quoted strings when using input (>>) and output (<<) streams.
* It automatically adds or removes quotes (") when reading or writing strings.
* It escapes special characters like " and \ properly.

#### Why Do We Need std::quoted?

* Before C++14, managing quoted strings manually was difficult.
  + Printing strings with quotes manually:

std::cout << "\"" << myString << "\"";

* + Example:

|  |
| --- |
| #include <iomanip> // Required for std::quoted  int main() {  std::string name = "John Doe";  // Writing to output stream (with quotes)  std::cout << std::quoted(name) << "\n";  }  // Output: "John Doe" |

#### **Summary: Why Use** std::quoted**?**

| **Feature** | **Without** std::quoted | **With** std::quoted |
| --- | --- | --- |
| Adding Quotes on Output | ❌ Manual (" + str + ") | ✅ Automatic (std::quoted(str)) |
| Reading Quoted Input | ❌ Reads only first word | ✅ Reads full string with spaces |
| Handling Escaped Quotes | ❌ Manual (\") | ✅ Automatic (" → \") |
| Handling Backslashes | ❌ Manual (\\) | ✅ Automatic (\ → \\) |
| Improves Code Readability? | ❌ No | ✅ Yes |