# Modern C++ (C++11/14) Study Guide

## 1. Building Blocks

### 1.1 auto [since C++11]

Automatic compile-time type deduction, great for complex template types, reduces repetition and improves maintainability.

auto x = 10; // int  
auto y = 3.14; // double

**Note:** auto cannot be used for function parameters (use templates instead), and cannot deduce initializer\_list element type without hint.

### 1.2 nullptr [since C++11]

A type-safe null pointer literal replacing NULL macros. In C++98, NULL was just 0, which could confuse overloads.

int\* p1 = NULL; // old style, ambiguous is actually 0  
int\* p2 = nullptr; // new, type-safe

### 1.3 enum class [since C++11]

Strongly scoped enums avoid name clashes and implicit conversions. In C++98 enums leaked their names globally.

enum Color {Red, Green}; // leaks 'Red' and 'Green'  
enum Color c=3 ; // Works! Its just a number

//It is not a known Scoped type

enum class Color2 {Red, Green}; // scoped

### 1.4 range-based for [since C++11]

Simplifies iteration for containers and arrays.

int arr[] = {1,2,3};  
for(auto x : arr) {}

### 1.5 binary literals and digit separators [since C++14]

Improves numeric readability.

int mask = 0b1010; // Binary Literals. Oct and Hex were there earlier  
int big = 1'000'000; // Makes reading large number easy

### 1.6 constexpr [since C++11, relaxed C++14]

Compile-time constant evaluation. In C++14, constexpr supports multiple statements and ifs.

constexpr int square(int x) { return x\*x; }

**Note:** if called with runtime data, it behaves like a normal function.

### 1.7 decltype(auto) [since C++11/14]

Captures exact type, including reference category.

int i=5;  
int& r=i;  
decltype(auto) x=r; // x is int&

## 2. std::function and std::bind [since C++11]

std::function is a flexible type-erased wrapper to store any callable target (free functions, function pointers, functors). In C++98, you had to rely on plain function pointers, which cannot store stateful functors.

Example with a regular function:

int sum(int a, int b) { return a + b; }  
std::function<int(int,int)> f = sum;

Example with a function object:

struct Mult {  
 int operator()(int a, int b) { return a \* b; }  
};  
Mult m;  
std::function<int(int,int)> g = m;

### std::bind

std::bind lets you fix arguments in advance, creating a new callable with fewer parameters.

int add(int a, int b) { return a+b; }  
auto add5 = std::bind(add, 5, std::placeholders::\_1);  
add5(10); // 15

### Using with class methods

You can store class member functions in std::function using bind to fix the object:

struct Calculator {  
 int multiply(int x, int y) { return x\*y; }  
};  
Calculator c;  
std::function<int(int,int)> h = std::bind(&Calculator::multiply, c, std::placeholders::\_1, std::placeholders::\_2);

## 3. Lambdas [since C++11]

Lambdas are function objects (functors) with syntax sugar. They are closer to functors than to free functions because they can carry captured data just like constructor parameters carry data into a functor. When you write a lambda with captures, conceptually you are building a small anonymous class with an operator().

First, using std::function:

std::function<int(int,int)> add = [](int a,int b){ return a+b; };

Then with auto to avoid repeating the type:

auto add = [](int a,int b){ return a+b; };

### Lambda parameters

You declare them like normal functions:

auto f = [](int x, int y){ return x+y; };

### Capture

A capture is like the constructor of a function object, carrying external variables:

int factor = 2;  
auto mul = [factor](int x){ return x\*factor; };

### Capture by reference

int counter = 0;  
auto inc = [&counter](){ counter++; };

### mutable lambdas

Allow modification of captured-by-value variables inside the lambda body:

int total = 0;  
auto add5 = [total](int x) mutable { total += x; return total; };

### this capture

Inside a member function:

class X {  
 int val = 42;  
 void show() {  
 auto f = [this]() { std::cout << val; };  
 f();  
 }  
};

### auto and decltype(auto) returns

You can return a lambda using auto or specify decltype(auto) for perfect forwarding.

You can return lambdas using auto return type or decltype(auto) in generic code. For example:

auto make\_adder(int y) {  
 return [y](int x) { return x + y; };  
}

decltype(auto) is more common for perfect forwarding return values. You can also combine lambdas with a trailing return type syntax if needed, for example:

auto f = [](int x, int y) -> int { return x + y; };

which explicitly specifies the return type after the arrow.

## 4. Class Enhancements [since C++11]

Modern C++ introduced several class-related features:

* **Delegating constructors**: one constructor can call another to reuse initialization logic.

class Point {  
 int x, y;  
 Point(int a, int b) : x(a), y(b) {}  
 Point(int a) : Point(a, 0) {} // delegates to two-arg constructor  
};

* **Inheriting constructors**: derived class automatically inherits base constructors.

struct Base { Base(int){} };  
struct Derived : Base {  
 using Base::Base; // inherits Base(int)  
};

* **Rvalue references**: allow you to distinguish temporary objects from lvalues, enabling move semantics.
* **Move semantics**: transfer resources from temporaries instead of deep copying.

std::vector<int> v1 = {1,2,3};  
std::vector<int> v2 = std::move(v1);

**Note**: v1 is now valid but unspecified.

* **Rule of 0/3/5**:
  + 0: let compiler provide defaults
  + 3: define copy constructor, copy assignment, destructor
  + 5: also define move constructor, move assignment

### Move constructor and assignment operator syntax

In C++11 you can explicitly write these:

MyClass(MyClass&& other) noexcept { /\* steal resources \*/ }  
MyClass& operator=(MyClass&& other) noexcept { /\* steal resources \*/ return \*this; }

### default and delete

Use = default to let compiler generate, or = delete to disable:

MyClass() = default;  
MyClass(const MyClass&) = delete; // no copying allowed

## 5. Smart Pointers [since C++11]

C++11 added smart pointers to manage dynamic resources safely:

* unique\_ptr is sole ownership; cannot be copied, only moved.

auto p = std::make\_unique<int>(42);

* shared\_ptr supports shared ownership with reference counting.

auto sp1 = std::make\_shared<int>(100);  
auto sp2 = sp1; // both share ownership

* weak\_ptr observes a shared\_ptr without increasing its refcount; prevents cycles.

std::weak\_ptr<int> wp = sp1;

### make\_unique and make\_shared

Preferred to raw new, provides exception safety.

### When to use which smart pointer

* use unique\_ptr when ownership is unique
* use shared\_ptr if you truly need shared ownership
* use weak\_ptr to break cycles

In most modern C++ code, start with unique\_ptr by default.

## 6. Variadic Templates and initializer\_list [since C++11]

Variadic templates allow you to accept any number of parameters:

### Variadic Templates in C++11 style (recursive)

Before fold expressions (C++17), you had to write recursion:

template<typename T>  
T sum(T v) { return v; }  
  
template<typename T, typename... Args>  
T sum(T first, Args... rest) {  
 return first + sum(rest...);  
}

### C++ 14 Enhancements

template<typename... Args>  
void print(Args... args) {  
 (std::cout << ... << args) << "  
"; // C++17 fold example  
}

initializer\_list is simpler but fixed-type:

void sum(std::initializer\_list<int> list) {  
 int total=0;  
 for(auto x: list) total+=x;  
}  
sum({1,2,3,4});

### Left and right fold examples

For subtraction:

auto left = [](auto... args) { return (args - ...); };  
left(20,5,3); // ((20-5)-3) = 12  
  
auto right = [](auto... args) { return (... - args); };  
right(20,5,3); // 20-(5-3) = 18

**Comparison**:

* variadic templates are flexible for any types
* initializer\_list is simpler for homogenous values

### Zero initializer in fold expressions

Using a zero base value prevents errors if no args passed:

auto left = [](auto... args) { return (args - ... - 0); };  
auto right = [](auto... args) { return (0 - ... - args); };

This makes folds safe even with empty packs.

## 7. Concurrency [since C++11]

C++11 standardized concurrency with threads, mutexes, and futures. It introduced std::thread, std::mutex, std::condition\_variable, std::future, and std::async for portable multithreading. For example:

std::thread t([]{ std::cout << "Hello from thread"; });  
t.join();

Use std::mutex and std::lock\_guard for critical sections:

std::mutex m;  
{  
 std::lock\_guard<std::mutex> lock(m);  
 // safely modify shared data  
}

Use std::async to launch background tasks:

auto fut = std::async([]{ return 42; });  
int result = fut.get();

Use std::condition\_variable for producer/consumer style:

std::condition\_variable cv;  
bool ready = false;  
std::mutex m;  
std::unique\_lock<std::mutex> lk(m);  
cv.wait(lk, [&]{ return ready; });

### More Concurrency Patterns

* Waiting for multiple threads: use join on each thread in a loop.
* Collecting results: store futures in a vector and get() each result.
* Thread pool: a collection of worker threads pulling tasks from a queue.
* Thread communication: prefer condition\_variable or message queues.
* Locks: use lock\_guard or unique\_lock to manage std::mutex.

**Best practices**: use RAII for locks, avoid data races, prefer futures for result collection, and prefer thread pools or task frameworks to raw threads for scalable designs.

### Advanced Concurrency Patterns

* Example of waiting for multiple threads:

std::thread t1(func1);  
std::thread t2(func2);  
t1.join();  
t2.join();

* Collecting results from multiple tasks:

std::vector<std::future<int>> futures;  
for(int i=0;i<5;++i)  
 futures.push\_back(std::async([i]{ return i\*i; }));  
for(auto& f: futures)  
 std::cout << f.get();

* Simple thread pool pattern: use a queue of tasks and a fixed number of threads pulling from it (example available on request)
* Communicating between threads: use condition\_variable to signal events; avoid shared bool flags without locks.
* Locks: prefer lock\_guard for scope-based locking to avoid forgetting unlock.

## 8. Other Features

### attributes [since C++11]

Standard way to give compiler hints, e.g. [[deprecated]]

### inline namespace [since C++11]

Helps manage library versions by making one namespace version default

### variable templates [since C++14]

Templated variables like template<typename T> constexpr T pi = T(3.14);

### make\_unique [since C++14]

A safer alternative to raw new, for unique\_ptr

### attributes [since C++11]

Attributes provide portable metadata to compilers and tools, such as [[deprecated]] or [[nodiscard]].

### inline namespace [since C++11]

Inline namespaces help manage versioning by making one namespace version visible by default.

namespace Lib {  
 inline namespace v1 { void foo(); }  
 namespace v2 { void foo(); }  
}

Lib::v1::foo(); //works

Lib::foo(); //same as above

Lib::v2::foo(); //v2 must be explicitly specified.

### variable templates [since C++14]

Provide type-dependent constants with one definition, avoiding macros.

template<typename T> constexpr T pi = T(3.1415926535);

### make\_unique [since C++14]

A safer alternative to raw new, ensures exception safety with unique\_ptr:

auto p = std::make\_unique<int>(42);

### Summary and Examples

Modern C++ adds consistent tools to write safer, more expressive, and cleaner code.

* attributes example:

[[deprecated]] void old\_function() {}

* variable templates example:

template<typename T> constexpr T pi = T(3.1415926535);

* inline namespace example:

namespace MyLib { inline namespace v1 { void foo(); } }

These features replace error-prone macros and improve maintainability. Modern C++ adds consistent tools to write safer, more expressive, and cleaner code.