# C++11/C++14 EXPLANATION WITH EXAMPLE AND FAQ

C++11/C++14


## ABSTRACT

C++11/C++14 new Features



## Haramohan Sahu


C++11/C++14 Explanation with Example and FAQ

# Contents

**Design goals:**

- Maintain stability and compatibility with C++98 and possibly with C
- Prefer introducing new features via the standard library, rather than extending the core language
- Improve C++ to facilitate systems and library design, rather than introduce new features useful only to specific applications
- Increase type safety by providing safer alternatives to earlier unsafe techniques

## Extensions to the C++ core language

### Core language runtime performance enhancements

These language features primarily exist to provide some kind of performance benefit, either of memory or of computational speed

#### Rvalue references and move constructors:

The primary purpose of introducing an rvalue is to implement move semantics. rvalue references enable us to distinguish an lvalue from an rvalue. Rvalue references solve at least two problems:

1. Implementing move semantics
2. Perfect forwarding

An **lvalue** is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An **rvalue** is an expression that is not an lvalue
If X is any type, then X&& is called an **rvalue reference** to X. For better distinction, the ordinary reference X& is now also called an **lvalue reference.**
void foo(X& x); // lvalue reference overload
void foo(X&& x); // rvalue reference overload

X x;
X foobar();

foo(x); // argument is lvalue: calls foo(X&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
Rvalue references allow a function to branch at compile time (via overload resolution) on the condition "Am I being called on an lvalue or an rvalue?"
A reference to T can be initialized with an object of type T, a function of type T, or an object implicitly convertible to T. Once initialized, a reference cannot be changed to refer to another object.
.
Example:

```
int main()
{

    int x = 4;
    int& j = x;
    cout << j << endl;
    int y = 5;
    j = y;  // ----------→ point 1
    cout << j << endl;
```

```
}
```

Here point 1, is j is not changed, But the value of x is changed. That is why Once reference initialized, a reference cannot be changed to refer to another object.

**Example of RVALU reference**

a. int&& rvalue_ref = 99;

b.
```
#include <iostream>

void f(int& i) { std::cout << "lvalue ref: " << i << "\n"; }
void f(int&& i) { std::cout << "rvalue ref: " << i << "\n"; }

int main ()
{
int i = 77;
f(i);    // lvalue ref called
f(99);   // rvalue ref called
f(std::move(i));  // rvalue ref called
return 0;
}
```

Now we have a way (overloading functions - taking lvalue or rvalue) to determine if a reference variable refers to a temporary object or to a permanent object. So, how it can be used?
The main usage of rvalue references is to create a move constructor and move assignment operator. A move constructor, like a copy constructor, takes an instance of an object as its argument and creates a new instance from original object. However, the move constructor can avoid memory reallocation because we know it has been provided a temporary object, so rather than copy the fields of the object, we will move them.

In other words, the rvalue references and move semantics allows us to avoid unnecessary copies when working with temporary objects. We do not want to copy the temporary which will go away. So, the resources needed for the temporary objects can be used for other objects.
the move constructor — to construct new objects by stealing data from temporaries;
the move assignment operator — to replace existing objects by stealing data from temporaries.
A move constructor of class T is a non-template constructor whose first parameter is

a. T&&,
b. const T&&,
c. volatile T&&,
d. const volatile T&&, and
e. either there are no other parameters, or
f. the rest of the parameters all have default values.

The move constructor is typically called when an object is initialized from rvalue of the same type, including

a. initialization: T a = std::move(b)
b. function argument passing: f(std::move(a))
c. function return: return a;

Move constructors typically "steal" the resources held by the argument rather than make copies of them, and leave the argument in some valid state.

**Implicitly-declared move constructor**

If no user-defined move constructors are provided for a class type (`struct`, `class`, or `union`), then all of the following is true:

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there is no user-declared destructor;

If a class defines any of the following, then it should probably explicitly define all five.

a. destructor
b. copy constructor
c. copy assignment operator
d. move constructor
e. move assignment operator

These five functions are special member functions. If one of these functions is used without being declared by the programmer, then it will be implicitly implemented by the compiler with the default semantics.
The Rule of Five claims that if one of these had to be defined by the programmer, it means that the compiler-generated version does not fit the needs of the class, hence programmer has to define all Five.

- If parent class has move constructor, then derive class does have implicitly the move constructor.
- If derive class does explicitly declare any of the Rule Five, say example derive class declare explicit Destructor, then that destructor prevents implicit move constructor.

**Example:**

```
struct A
{
   std::string s;
   int k;
   A() : s("test"), k(-1) { }
   A(const A& o) : s(o.s), k(o.k) { std::cout << "move failed!\n"; }
   A(A&& o) noexcept :
        s(std::move(o.s)),       // explicit move of a member of class type
        k(std::exchange(o.k, 0)) // explicit move of a member of non-class type
   { }
};

A f(A a)
{
   return a;
}
struct B : A
{
   std::string s2;
   int n;
   // implicit move constructor B::(B&&)
   // calls A's move constructor
   // calls s2's move constructor
   // and makes a bitwise copy of n
};

struct C : B
```

```
{
   ~C() { } // destructor prevents implicit move constructor C::(C&&)
};

struct D : B
{
   D() { }
   ~D() { }         // destructor would prevent implicit move constructor D::(D&&)
   D(D&&) = default; // forces a move constructor anyway
};
```

**Another example:**

```
class Holder
{
  public:
    Holder(int size)         // Constructor
    {
      m_data = new int[size];
      m_size = size;
    }
    ~Holder()                // Destructor
    {
      delete[] m_data;
    }
  private:
    int*   m_data;
    size_t m_size;
};
Holder createHolder(int size)
{
  return Holder(size);
}
int main ()
{
  Holder h = createHolder(1000); // Copy constructor
  h = createHolder(500);         // Assignment operator
}
```

Too many expensive copies! We already have a fully-fledged object, the temporary and short-lived one returning from createHolder(), built for us by the compiler.
Now, we will steal existing data from temporary objects instead of making useless clones. Don't copy, just move, because moving is always cheaper.

```
Holder(Holder&& other)     // <-- rvalue reference in input
{
  m_data = other.m_data;   // (1)
  m_size = other.m_size;
  other.m_data = nullptr;  // (2)
  other.m_size = 0;
}
```

So let's steal its data first (1), then set it to null (2). No deep copies here, we have just moved resources around! It's important to set the rvalue reference data to some valid state (2) to prevent it from being accidentally deleted when the temporary object dies: our Holder destructor calls delete[] m_data, remember

```
Holder& operator=(Holder&& other)     // <-- rvalue reference in input
{
  if (this != &other)
  {
  delete[] m_data;         // (1)
  m_data = other.m_data;   // (2)
  m_size = other.m_size;
  other.m_data = nullptr;  // (3)
  other.m_size = 0;
  }
  return *this;
}

int main()
{
  Holder h1(1000);               // regular constructor
  Holder h2(h1);                 // copy constructor (lvalue in input)
  Holder h3 = createHolder(2000); // move constructor (rvalue in input) (1)

  h2 = h3;                       // assignment operator (lvalue in input)
  h2 = createHolder(500);        // move assignment operator (rvalue in input)
}
```

## Perfect forwarding:

When you combine rvalue references with function templates you get an interesting interaction: if the type of a function parameter is an rvalue reference to a template type parameter then the type parameter is deduce to be an lvalue reference if an lvalue is passed, and a plain type otherwise.

**Example:**

```
void g(X&& t); // A
void g(X& t);      // B
template<typename T>
void f(T&& t)
{
   g(std::forward<T>(t));
}
void h(X&& t)
{
   g(t);
}
int main()
{
   X x;
   f(x);   // 1
   f(X()); // 2
   h(x);
   h(X()); // 3
}
```

This time our function f forwards its argument to a function g which is overloaded for lvalue and rvalue references to an X object. g will therefore accept lvalues and rvalues alike, but overload resolution will bind to a different function in each case.

At line "1", we pass a named X object to f, so T is deduced to be an lvalue reference: X&, as we saw above.
When T is an lvalue reference, std::forward<T> is a no-op: it just returns its argument. We therefore call the overload of g that takes an lvalue reference (line B).

At line "2", we pass a temporary to f, so T is just plain X. In this case, std::forward<T>(t) is equivalent to static_cast<T&&>(t): it ensures that the argument is forwarded as an rvalue reference. This means that the overload of g that takes an rvalue reference is selected (line A).

This is called perfect forwarding because the same overload of g is selected as if the same argument was supplied to g directly.
It is essential for library features such as std::function and std::thread which pass arguments to another (user supplied) function.

Forwarding references allow a reference to bind to either an lvalue or rvalue depending on the type.
Forwarding references follow the rules of **reference collapsing**:

- `T& &` becomes `T&`
- `T& &&` becomes `T&`
- `T&& &` becomes `T&`
- `T&& &&` becomes `T&&`

auto type deduction with lvalues and rvalues:
int x = 0; // `x` is an lvalue of type `int`
auto && al = x; // `al` is an lvalue of type `int&` -- binds to the lvalue, `x`
auto && ar = 0; // `ar` is an lvalue of type `int&&` -- binds to the rvalue temporary, `0`

## Template type parameter deduction with lvalues and rvalues:

```
// Since C++14 or later: we can use auto 14 onwards
void f(auto && t) {
  // ...
}

// Since C++11 or later:
template <typename T>
void f(T && t) {
  // ...
}

int x = 0;
f(0); // deduces as f(int&&)
f(x); // deduces as f(int&)

int& y = x;
f(y); // deduces as f(int& &&) => f(int&)

int&& z = 0; // NOTE: `z` is an lvalue with type `int&&`.
f(z); // deduces as f(int&& &) => f(int&)
f(std::move(z)); // deduces as f(int&& &&) => f(int&&)
```

## constexpr – Generalized constant expressions

C++11 introduced the keyword constexpr, which allows the user to guarantee that a function or object constructor is a compile-time constant.[10] The above example can be rewritten as follows:

```
constexpr int get_five() {
return 5;
}
int some_value[get_five() + 7]; // Create an array of 12 integers. Valid C++11
```

This allows the compiler to understand, and verify, that get_five() is a compile-time constant.

```
const= “I won’t change  this”
constexpr= “I will evaluate  this  at compile  time”  if I can
constexpr size_t size = sizeof(int) + sizeof(Foo);
```

The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time. It must not be virtual.

## constexpr vs inline functions:

Both are for performance improvements, inline functions request compiler to expand at compile time and save time of function call overheads. In inline functions, expressions are always evaluated at run time. constexpr is different, here expressions are evaluated at compile time.

Declaring something as constexpr does not necessarily guarantee that it will be evaluated at compile time. It can be used for such, but it can be used in other places that are evaluated at run-time, as well.

```
constexpr const int N = 5;
```

is the same as

```
constexpr int N = 5;
```

Constexpr tells the compiler that this expression results in a compile time constant value, so it can be used in places like array lengths, assigning to const variables, etc

A const int var can be dynamically set to a value at runtime and once it is set to that value, it can no longer be changed.

A constexpr int var cannot be dynamically set at runtime, but rather, at compile time. And once it is set to that value, it can no longer be changed.

```
int main(int argc, char*argv[]) {
    const int p = argc;
    // p = 69; // cannot change p because it is a const
    // constexpr int q = argc; // cannot be, bcoz argc cannot be computed at compile time
    constexpr int r = 2^3; // this works!
    // r = 42; // same as const too, it cannot be changed
}
```

## Trailing return type

```
// Trailing return type is used to represent
// a fully generic return type for a+b.
template <typename FirstType, typename SecondType>
auto add(FirstType a, SecondType b) -> decltype(a + b){
   return a + b;
}

int main(){
   // The first template argument is of the integer type, and
   // the second template argument is of the character type.
   add(1, 'A');

   // Both the template arguments are of the integer type.
   add(3, 5);
}
```

In the following example, the auto keyword is put before the function identifier add. The return type of add is decltype(a + b), which depends on the types of the function arguments a and b.

## Noexcept

The noexcept operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions. The noexcept operator does not evaluate expression.
There are two things i.e `noexcept` specifier and the `noexcept` operator.
**They are implicitly 6 non-throwing** special member functions**.**

- Default constructor and destructor
- Move and copy constructor
- Move and copy assignment operator

This special six members such as the destructor can only be non-throwing if all destructors of the attributes and the bases-classes are non-throwing.

What happens when you throw an exception in a function which is declared as non-throwing? In this case, std::terminate is called. std::terminate calls the currently installed std::terminate_handler which calls std::abort by default.The result is an abnormal program termination.

**Example:**

1. **Noexcept specifier**

By declaring a function, a method, or a lambda-function as noexcept, you specify that these does not throw an exception and if they throw, you do not care and let the program just crash.
The noexcept specifier specifies whether a function could throw exceptions. It is an improved version of throw().
The noexcept specification is equivalent to the noexcept(true) specification. throw() is equivalent to noexcept(true) but was deprecated with C++11 and will be removed with C++20.
In contrast, noexcept(false) means that the function may throw an exception.
The noexcept specification is part of the function type but cannot be used for function overloading.

There are two good reasons for the use of `noexcept`:

- First, an exception specifier documents the behavior of the function. If a function is specified as `noexcept`, it can be safely used in a non-throwing function.
- Second, it is an optimization opportunity for the compiler. `noexcept` may not call `std::unexpected` and may not unwind the stack.

The initialization of a container may cheap move the elements into the container if the move constructor is declared as noexcept. If not declared as noexcept, the elements may be expensive copied into the container.

```
void func1() noexcept;        // does not throw
void func2() noexcept(true);  // does not throw
void func3() throw();         // does not throw

void func4() noexcept(false); // may throw
```

### 2. noexcept as operator

The **noexcept** operator checks at compile-time if an expression does not throw an exception.
The noexcept operator does not evaluate the expression. It can be used in a noexcept specifier of a function template to declare that the function may throw exceptions depending on the current type.
**Example:**

```
#include <iostream>
#include <utility>
#include <vector>

void may_throw();
void no_throw() noexcept;
auto lmay_throw = []{};
auto lno_throw = []() noexcept {};
class T{
public:
  ~T(){} // dtor prevents move ctor
         // copy ctor is noexcept
};
class U{
public:
  ~U(){} // dtor prevents move ctor
         // copy ctor is noexcept(false)
  std::vector<int> v;
};
class V{
public:
  std::vector<int> v;
};

int main()
{
 T t;
 U u;
 V v;

 std::cout << std::boolalpha
           << "Is may_throw() noexcept? " << noexcept(may_throw()) << '\n'
           << "Is no_throw() noexcept? " << noexcept(no_throw()) << '\n'
           << "Is lmay_throw() noexcept? " << noexcept(lmay_throw()) << '\n'
           << "Is lno_throw() noexcept? " << noexcept(lno_throw()) << '\n'
           << "Is ~T() noexcept? " << noexcept(std::declval<T>().~T()) << '\n'
```

```
        // note: the following tests also require that ~T() is noexcept because
        // the expression within noexcept constructs and destroys a temporary
        << "Is T(rvalue T) noexcept? " << noexcept(T(std::declval<T>())) << '\n'
        << "Is T(lvalue T) noexcept? " << noexcept(T(t)) << '\n'
        << "Is U(rvalue U) noexcept? " << noexcept(U(std::declval<U>())) << '\n'
        << "Is U(lvalue U) noexcept? " << noexcept(U(u)) << '\n'
        << "Is V(rvalue V) noexcept? " << noexcept(V(std::declval<V>())) << '\n'
        << "Is V(lvalue V) noexcept? " << noexcept(V(v)) << '\n';
}
```

Output:

```
Is may_throw() noexcept? false
Is no_throw() noexcept? true
Is lmay_throw() noexcept? false
Is lno_throw() noexcept? true
Is ~T() noexcept? true
Is T(rvalue T) noexcept? true
Is T(lvalue T) noexcept? true
Is U(rvalue U) noexcept? false
Is U(lvalue U) noexcept? false
Is V(rvalue V) noexcept? true
Is V(lvalue V) noexcept? false
```

## Core language build-time performance enhancements

### Extern template

In C++03, the compiler must instantiate a template whenever a fully specified template is encountered in a translation unit. If the template is instantiated with the same types in many translation units, this can dramatically increase compile times. There is no way to prevent this in C++03, so C++11 introduced extern template declarations, analogous to extern data declarations.

We should only use extern template to force the compiler to not instantiate a template when we know that it will be instantiated somewhere else. It is used to reduce compile time and object file size.

**For Example**:

```
// header.h
template<typename T>
void ReallyBigFunction()
{
   // Body
}

// source1.cpp
#include "header.h"
void something1()
{
   ReallyBigFunction<int>();
}

// source2.cpp
```

```
#include "header.h"
void something2()
{
    ReallyBigFunction<int>();
}
```

This will result in the following object files:

**source1.o**

```
    void something1()
    void ReallyBigFunction<int>()    // Compiled first time
```

**source2.o**

```
    void something2()
    void ReallyBigFunction<int>()    // Compiled second time
```

Now If both files are linked together, one void ReallyBigFunction<int>() will be discarded, that mean unnecessarily we had wasted compile time & object size too.
So in c++11, if we use extern then this object size & compile time will be reduced.

**// source2.cpp**

```
#include "header.h"
extern template void ReallyBigFunction<int>();
void something2()
{
    ReallyBigFunction<int>();
}
```

Will result in the following object files:

**source1.o**

```
     void something1()
    void ReallyBigFunction<int>() // compiled just one time
```

**source2.o**

```
     void something2()
    // No ReallyBigFunction<int> here because of the extern
```

## Core language usability enhancements

### Initializer lists:

C++11 binds the concept to a template, called std::initializer_list. This allows constructors and other functions to take initializer-lists as parameters

```
struct X {
    X() = default;
    X(const X&) = default;
};

struct Q {
    Q() = default;
    Q(Q const&) = default;
    Q(std::initializer_list<Q>) {}
};
```

```
int main() {
  X x;
  X x2 = X { x }; // copy-constructor (not aggregate initialization)
  Q q;
  Q q2 = Q { q }; // initializer-list constructor (not copy constructor)
}
```

## Uniform initialization

C++11 attempts to overcome the problems of C++03 initialization by introducing a universal initialization notation that applies to every type—whether a POD variable, a class object with a user-defined constructor, a POD array, a dynamically allocated array, or even a Standard Library container. Initialization in C++03 is tricky, with four different initialization notations.

- No way to initialize a member array
- No convenient form of initializing containers
- No way to initialize dynamically allocated POD types

```
//C++11 brace-init
int a{0};
string s{"hello"};
string s2{s}; //copy construction
vector <string> vs{"alpha", "beta", "gamma"};

map<string, string> stars
 { {"Superman", "+1 (212) 545-7890"},
  {"Batman", "+1 (212) 545-0987"}};

double *pd= new double [3] {0.5, 1.2, 12.99};

class C
{
int x[4];
public:
C(): x{0,1,2,3} {}
};

//C++11: default initialization using {}
int n{}; //zero initialization: n is initialized to 0
int *p{}; //initialized to nullptr
double d{}; //initialized to 0.0
char s[12]{}; //all 12 chars are initialized to '\0'
string s{}; //same as: string s;
char *p=new char [5]{}; // all five chars are initialized to '\0'
Uniform Initialization
----------------------
struct BasicStruct
{
    int x;
    double y;
};
struct AltStruct
{
    AltStruct(int x, double y)
        : x_{x}
        , y_{y}
    {}
```

```
private:
    int x_;
    double y_;
};

BasicStruct var1{5, 3.2};
AltStruct var2{2, 4.3};
```

## Type inference

Auto One of the most common and notable examples of type derivation using auto in the iterator.

```
// Example: before C++11
cbegin() returns vector <int >:: const_iterator and therefore itr is type vector <int >:: const_iterator
for ( vector <int >:: const_iterator it = vec. cbegin(); itr != vec. cend(); ++ it)
In C++11:
for ( auto it = vec. cbegin(); itr != vec. cend(); ++ it)
auto i = 5; // i as int 2
auto arr = new auto(10); // arr as int *
Note: auto cannot be used for function arguments &auto cannot be used to derive array types:
        auto auto_arr2[10] = arr; // illegal, can't infer array type

int add(auto x, auto y);
error: 'auto' not allowed in function prototype
In addition, auto cannot be used to derive array types:
```

## Range-based for loop:

Range-based for loop in C++ is added since C++ 11. It executes a for loop over a range. Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

```
// Iterating over whole array
std::vector<int> v = {0, 1, 2, 3, 4, 5};
for (auto i : v)
```

## Lambda functions and expressions

C++11 supports anonymous functions, called lambda expressions, which have the form:

```
[capture](parameters) -> return_type { function_body }

An example lambda function is defined as follows:
[](int x, int y) -> int { return x + y; }
```

C++11 also supports closures. Closures are defined between square brackets [and ] in the declaration of lambda expression.

```
[ ]       //no variables defined. Attempting to use any external variables in the lambda is an error.
[x, &y] //x is captured by value, y is captured by reference
[&]       //any external variable is implicitly captured by reference if used
[=]       //any external variable is implicitly captured by value if used
[&, x]   //x is explicitly captured by value. Other variables will be captured by reference
[=, &z] //z is explicitly captured by reference. Other variables will be captured by value
```

Example:

```
#include <string>

int main(int argc, char **argv)
{
	std::string msg = "Hello";
	int counter = 10;

	// Defining Lambda function and
	// Capturing Local variables by Reference
	auto func = [&msg, &counter]  () {
		std::cout<<"Inside Lambda :: msg = "<<msg<<std::endl;
		std::cout<<"Inside Lambda :: counter = "<<counter<<std::endl;

		// Change the counter & msg
		// Change will affect the outer variable because counter variable is
		// captured by Reference in Lambda function
		msg = "Temp";
		counter = 20;

		std::cout<<"Inside Lambda :: After changing :: msg = "<<msg<<std::endl;
		std::cout<<"Inside Lambda :: After changing :: counter = "<<counter<<std::endl;

		};
		std::cout<<"Before Lambda msg = "<<msg<<std::endl;
		std::cout<<"counter = "<<counter<<std::endl;

		//Call the Lambda function
		func();

		std::cout<<"After ambda msg = "<<msg<<std::endl;
		std::cout<<"counter = "<<counter<<std::endl;

		return 0;
}
```

**Capture All Local Variables from outer scope by Value:**

```
auto func = [=] () {
		//...
		};
```

**Capture all local variables from outer scope by Reference:**

```
// Capturing All Local variables by value except counter, which is
// captured by reference here.
auto func = [=, &counter] () mutable {};
```

**Variables captured by value are constant by default. Adding mutable after the parameter list makes them non-constant.**

Some More Examples:

```
int x = 1;
auto getX = [=] { return x; };
getX(); // == 1
auto addX = [=](int y) { return x + y; };
addX(1); // == 2
auto getXRef = [&]() -> int& { return x; };
getXRef(); // int& to `x`
```

```
int x = 1;
auto f1 = [&x] { x = 2; }; // OK: x is a reference and modifies the original
auto f2 = [x] { x = 2; }; // ERROR: the lambda can only perform const-operations
//! on the captured  value
// vs.
auto f3 = [x]() mutable { x = 2; }; // OK: the lambda can perform any
// operations on the captured value
```

## Alternative function syntax

```
template<class Lhs, class Rhs>
auto adding_func(const Lhs &lhs, const Rhs &rhs) -> decltype(lhs+rhs) {return lhs + rhs;}
```

For more Info, please refer to **2.1.3 Trailing return type.**

## Object construction improvement

In C++03, constructors of a class are not allowed to call other constructors in an initializer list of that class. Each constructor must construct all of its class members itself or call a common member function.

```
class SomeType
{
public:
   SomeType(int new_number)
   {
      Construct(new_number);
   }
private:
   void Init(int new_number)
   {
      number = new_number;
   }

   int number;
         // int number =90; not allowed in C++03
};
```

For base-class constructors, C++11 allows a class to specify that base class constructors will be inherited. Thus, the C++11 compiler will generate code to perform the inheritance and the forwarding of the derived class to the base class.

This is an all-or-nothing feature: either all of that base class's constructors are forwarded or none of them are. Also, an inherited constructor will be shadowed if it matches the signature of a constructor of the derived class, and restrictions exist for multiple inheritance:

class constructors cannot be inherited from two classes that use constructors with the same signature.

```
class BaseClass
```

```
{
int number;

public:
   BaseClass(int value): number(value) {}
         BaseClass() : BaseClass(42) {} //! In C++11, we can call other peer cotr
                                        //! This is called Delegating constructors

};

class DerivedClass : public BaseClass
{
public:
   using BaseClass::BaseClass;
};
Now I can call DerivedClass obj(90); // as  base class cotr parameterized inherited to derive class
This was not possible in C++03.

struct B2 {
   B2(int = 13, int = 42);
};
struct D2 : B2 {
   using B2::B2;
// The set of inherited constructors is
// 1. B2(const B2&)
// 2. B2(B2&&)
// 3. B2(int = 13, int = 42)
// 4. B2(int = 13)
// 5. B2()

// D2 has the following constructors:
// 1. D2()
// 2. D2(const D2&)
// 3. D2(D2&&)
// 4. D2(int, int) <- inherited
// 5. D2(int) <- inherited
};
```

Some more example:

```
#include <iostream>
struct B {
   virtual void f(int) { std::cout << "B::f\n"; }
   void g(char)        { std::cout << "B::g\n"; }
   void h(int)         { std::cout << "B::h\n"; }
 protected:
   int m; // B::m is protected
   typedef int value_type;
};

struct D : B {
   using B::m; // D::m is public
   using B::value_type; // D::value_type is public

   using B::f;
   void f(int) { std::cout << "D::f\n"; } // D::f(int) overrides B::f(int)
```

```
    using B::g;
    void g(int) { std::cout << "D::g\n"; } // both g(int) and g(char) are visible
                                          // as members of D
    using B::h;
    void h(int) { std::cout << "D::h\n"; } // D::h(int) hides B::h(int)
};

int main()
{
    D d;
    B& b = d;

//    b.m = 2; // error, B::m is protected
    d.m = 1; // protected B::m is accessible as public D::m
    b.f(1); // calls derived f()
    d.f(1); // calls derived f()
    d.g(1); // calls derived g(int)
    d.g('a'); // calls base g(char)
    b.h(1); // calls base h()
    d.h(1); // calls derived h()
}
```

Output:

```
D::f
D::f
D::g
B::g
B::h
D::h
```

**Delegating constructors**

Constructors can now call other constructors in the same class using an initializer list.

## Explicit overrides and final

The override special identifier means that the compiler will check the base class(es) to see if there is a virtual function with this exact signature. And if there is not, the compiler will indicate an error.

```
struct Base
{
    virtual void some_func(float);
};
struct Derived : Base
{
    virtual void some_func(int) override; // ill-formed - doesn't override a base class method
};
```

**If the virtual function does not override a parent's virtual function, throws a compiler error.**

C++11 also adds the ability to prevent inheriting from classes or simply preventing overriding methods in derived classes. This is done with the special identifier final. For example:

```
struct Base1 final { };
struct Derived1 : Base1 // ill-formed because the class Base1 has been marked final
 {
```

```
};
```
Like this we can declare final with virtual function which will prevent to override
**Example:**
```
struct Base2
{
   virtual void f() final;
};
struct Derived2 : Base2
{
   void f(); // ill-formed because the virtual function Base2::f has been marked final
};
```

When overriding a virtual function, introducing the override keyword will explicitly tell the compiler to overload, and the compiler will check if the base function has such a virtual function, otherwise it will not compile:
```
struct Base {
virtual void foo( int );
};
struct SubClass: Base {
virtual void foo( int ) override; // legal
virtual void foo( float ) override; // illegal , no virtual function in super class
};
```
final is to prevent the class from being continued to inherit and to terminate the virtual function to continue to be overloaded.

C++11 also adds the ability to prevent inheriting from classes or simply preventing overriding methods in derived classes. This is done with the special identifier final. For example:

```
struct Base1 final { };
struct Derived1 : Base1 { }; // ill-formed because the class Base1 has been marked final
struct Base2
{
   virtual void f() final;
};
struct Derived2 : Base2
{
   void f(); // ill-formed because the virtual function Base2::f has been marked final
};
```

Class cannot be inherited from.
```
struct A final {};
struct B : A {}; // error -- base 'A' is marked 'final'
```

## Deleted functions

A more elegant, efficient way to provide a deleted implementation of a function. Useful for preventing copies on objects.
```
class A {
  int x;

public:
  A(int x) : x{x} {};
  A(const A&) = delete;
  A& operator=(const A&) = delete;
};
```

```
A x {123};
A y = x; // error -- call to deleted copy constructor
y = x; // error -- operator= deleted
```

## Null pointer constant

The purpose of nullptr appears to replace NULL. In a sense, traditional C++ treats NULL and 0 as the same thing, depending on how the compiler defines NULL, and some compilers define NULL as ((void*)0) Some will define it directly as 0.

| C++03 | C++11 |
|---|---|
| void foo(char*); void foo(int); | void foo(char*); void foo(int); |
| foo(NULL); //calls second foo | foo(nullptr); //calls first foo |

C++11 introduced the nullptr keyword, which is specifically used to distinguish null pointers, 0. The type of nullptr is nullptr_t, which can be implicitly converted to any pointer or member pointer type, and can be compared equally or unequally with them.

## Strongly typed enumerations

In C++03, enumerations are not type-safe. They are effectively integers, even when the enumeration types are distinct. This allows the comparison between two enum values of different enumeration types. The only safety that C++03 provides is that an integer or a value of one enum type does not convert implicitly to another enum type.
This enumeration is type-safe. Enum class values are not implicitly converted to integers. Thus, they cannot be compared to integers either (the expression Enumeration::Val4 == 101 gives a compile error).
The underlying type of enum classes is always known. The default type is int;
this can be overridden to a different integral type as can be seen in this example:

```
enum class Enum2 : unsigned int {Val1, Val2};
enum Enum1;                 // Invalid in C++03 and C++11; the underlying type cannot be
//determined.
enum Enum2 : unsigned int;      // Valid in C++11, the underlying type is specified explicitly.
enum class Enum3;               // Valid in C++11, the underlying type is int.
enum class Enum4 : unsigned int; // Valid in C++11.
enum Enum2 : unsigned short; // Invalid in C++11, because Enum2 was formerly declared with a
different underlying type
```

## Right angle bracket

- In the traditional C++ compiler, >> is always treated as a right shift operator. But actually we can easily write the code for the nested template:
- std:: vector < std:: vector <int >> matrix;

```
template<bool Test> class SomeType;
std::vector<SomeType<1>2>> x1;  // Interpreted as a std::vector of SomeType<true>,
   // followed by "2 >> x1", which is not valid syntax for a declarator. 1 is true.
std::vector<SomeType<(1>2)>> x1;  // Interpreted as std::vector of SomeType<false>,
   // followed by the declarator "x1", which is valid C++11 syntax. (1>2) is false.
```

## Explicit conversion operators

In C++11, the explicit keyword can now be applied to conversion operators. As with constructors, it prevents using those conversion functions in implicit conversions.

## Template aliases

```
template <typename First, typename Second, int Third>
class SomeType;

template <typename Second>
typedef SomeType<OtherType, Second, 5> TypedefName; // Invalid in C++03 & This will not compile.

C++11 adds this ability with this syntax:
template <typename First, typename Second, int Third>
class SomeType;

template <typename Second>
using TypedefName = SomeType<OtherType, Second, 5>;
```

The using syntax can also be used as type aliasing in C++11:

```
typedef void (*FunctionType)(double);       // Old style
using FunctionType = void (*)(double); // New introduced syntax

typedef int (* process)( void *);
using NewProcess = int (*)( void *);
template < typename T>
using TrueDarkMagic = MagicType < std:: vector <T>, std:: string >;
int main() {
TrueDarkMagic < bool > you;
}
```

## Template

Templates are used to generate types. In traditional C++, typedef can define a new name for the type, but there is no way to define a new name for the template. Because the template is not a type.

### Default Template parameter:

A convenience is provided in C++11 to specify the default parameters of the template:

```
template < typename T = int , typename U = int >
auto add(T x, U y) -> decltype(x+y) {
return x+y;
}
template<class T = float, class U=int> class A;
template<class T, class U> class A {
  public:
    T x;
    U y;
};
A<> a;
```

The type of member a.x is float, and the type of a.y is int.

If one template parameter has a default argument, then all template parameters following it must also have default arguments. For example, the compiler will not allow the following:
template<class T = char, class U, class V = int> class X { };
Template parameter U needs a default argument or the default for T must be removed.

## Unrestricted unions

In C++03, there are restrictions on what types of objects can be members of a union. For example, unions cannot contain any objects that define a non-trivial constructor or destructor.
C++11 lifts some of these restrictions.
If a union member has a non-trivial special member function, the compiler will not generate the equivalent member function for the union and it must be manually defined.
This is a simple example of a union permitted in C++11:

```
#include <new> // Needed for placement 'new'.
struct Point
{
   Point() {}
   Point(int x, int y): x_(x), y_(y) {}
   int x_, y_;
};
union U
{
   int z;
   double w;
   Point p; // Invalid in C++03; valid in C++11.
   U() {} // Due to the Point member, a constructor definition is now needed.
   U(const Point& pt) : p(pt) {} // Construct Point object using initializer list.
   U& operator=(const Point& pt) { new(&p) Point(pt); return *this; } // Assign Point object
using placement 'new'.
```

## decltype

decltype is an operator which returns the declared type of an expression passed to it. cv-qualifiers and references are maintained if they are part of the expression.
**Examples of decltype:**

```
int a = 1; // `a` is declared as type `int`
decltype(a) b = a; // `decltype(a)` is `int`
const int& c = a; // `c` is declared as type `const int&`
decltype(c) d = a; // `decltype(c)` is `const int&`
decltype(123) e = 123; // `decltype(123)` is `int`
int&& f = 1; // `f` is declared as type `int&&`
decltype(f) g = 1; // `decltype(f) is `int&&`
decltype((a)) h = g; // `decltype((a))` is int&

template <typename X, typename Y>
auto add(X x, Y y) -> decltype(x + y) {
  return x + y;
}
add(1, 2.0); // `decltype(x + y)` => `decltype(3.0)` => `double`
```

```
template < typename R, typename T, typename U>
R add(T x, U y) {
return x+y
}
```

using decltype to derive the type of x+y, write something like this:

```
decltype(x+y) add(T x, U y)

template < typename T, typename U>
auto add2 (T x, U y) -> decltype(x+y){
return x + y;
}
```

The good news is that from C++14 it is possible to directly derive the return value of a normal function, so the following way becomes legal:

```
template < typename T, typename U>
auto add3 (T x, U y){
    return x + y;
}
```

## decltype(auto):

The **decltype(auto)** type-specifier also deduces a type like auto does. However, it deduces return types while keeping their references and cv-qualifiers, **while auto will not.**

```
const int x = 0;
auto x1 = x; // int
decltype(auto) x2 = x; // const int
int y = 0;
int& y1 = y;
auto y2 = y1; // int
decltype(auto) y3 = y1; // int&
int&& z = 0;
auto z1 = std::move(z); // int
decltype(auto) z2 = std::move(z); // int&&

// decltype of a parenthesized variable is always a reference
decltype((i)) d; // error: d is int& and must be initialized
decltype(i) e;   // ok: e is an (uninitialized) int
// Return type is `int`.
auto f(const int& i) {
 return i;
}
// Return type is `const int&`.
decltype(auto) g(const int& i) {
 return i;
}
```

## Variadic templates

Before C++11, templates had a fixed number of parameters that must be specified in the declaration of the templates with the variadic templates feature, you can define class or function templates that have any number (including zero) of parameters.

Template parameter packs

A template parameter pack is a template parameter that represents any number (including zero) of template parameters.

```
template<class...A> struct container{};
template<class...B> void func();
```

In this example, A and B are template parameter packs.

```
template<class...T>
class X{ };
X<> a;                  // the parameter list is empty
X<int> b;               // the parameter list has one item
X<int, char, float> c;    // the parameter list has three items

template<class...A>
void func(A...args)
```

In this example, A is a template parameter pack, and args is a function parameter pack. You can call the function with any number (including zero) of arguments:

```
func();                 // void func();
func(1);                // void func(int);
func(1,2,3,4,5);        // void func(int,int,int,int,int);
func(1,'x', aWidget);    // void func(int,char,widget);

#include <cassert>

template<class...A, class...B> void func(A...arg1,int sz1, int sz2, B...arg2)
{
  assert( sizeof...(arg1) == sz1);
  assert( sizeof...(arg2) == sz2);
}

int main(void)
{
  //A:(int, int, int), B:(int, int, int, int, int)
  func<int,int,int>(1,2,3,3,5,1,2,3,4,5);

  //A: empty, B:(int, int, int, int, int)
  func(0,5,1,2,3,4,5);
  return 0;
}
```

In this example, function template func has two function parameter packs arg1 and arg2. arg1 is a non-trailing function parameter pack,

and arg2 is a trailing function parameter pack. When func is called with three explicitly specified arguments
as func<int,int,int>(1,2,3,3,5,1,2,3,4,5), both arg1 and arg2 are deduced successfully.
When func is called without explicitly specified arguments as func(0,5,1,2,3,4,5),
arg2 is deduced successfully and arg1 is empty. In this example, the template parameter packs of function template func can be deduced,
so func can have more than one template parameter pack.

## New string literals

C++11 supports three Unicode encodings: UTF-8, UTF-16, and UTF-32. Along with the formerly noted changes to the definition of char, C++11 adds two new character types: char16_t and char32_t. These are designed to store UTF-16 and UTF-32 respectively.
Examples:

```
u8"I'm a UTF-8 string."
u"This is a UTF-16 string."
U"This is a UTF-32 string."
u8"This is a Unicode Character: \u2018."
u"This is a bigger Unicode Character: \u2018."
U"This is a Unicode Character: \U00002018."
```

## User-defined literals

Allows integer, floating-point, character, and string literals to produce objects of user-defined type by defining a user-defined suffix.
Example:

```
long double operator "" _w(long double);
std::string operator "" _w(const char16_t*, size_t);
unsigned operator "" _w(const char*);

int main() {
    1.2_w; // calls operator "" _w(1.2L)
    u"one"_w; // calls operator "" _w(u"one", 3)
    12_w; // calls operator "" _w("12")
    "two"_w; // error: no applicable literal operator
}

int main() {
    L"A" "B" "C"_x; // OK: same as L"ABC"_x
    "P"_x "Q" "R"_y;// error: two different ud-suffixes (_x and _y)
}
```

Some More Example:

```
#include <iostream>
 // used as conversion
constexpr long double operator"" _deg ( long double deg )
{
    return deg * 3.14159265358979323846264L / 180;
}

// used with custom type
struct mytype
{
```

```
    unsigned long long m;
};
constexpr mytype operator"" _mytype ( unsigned long long n )
{
    return mytype{n};
}

// used for side-effects
void operator"" _print ( const char* str )
{
    std::cout << str;
}

int main(){
    double x = 90.0_deg;
    std::cout << std::fixed << x << '\n';
    mytype y = 123_mytype;
    std::cout << y.m << '\n';
    0x123ABC_print;
}
```

Output:
1.570796
123
0x123ABC

## Multithreading memory model

A memory model, a.k.a memory consistency model, is a specification of the allowed behavior of multithreaded programs executing with shared memory.
Different threads trying to access the same memory location participate in a data race if at least one of the operations is a modification (also known as store operation). These data races cause undefined behavior. To avoid them one needs to prevent these threads from concurrently executing such conflicting operations.

Synchronization primitives (mutex, critical section and the like) can guard such accesses. The Memory Model introduced in C++11 defines two new portable ways to synchronize access to memory in multi-threaded environment: atomic operations and fences.
https://stackoverflow.com/questions/31978324/what-exactly-is-stdatomic
https://riptutorial.com/cplusplus/topic/7975/cplusplus11-memory-model

## Thread-local storage

Thread-local storage duration is a term used to refer to data that is seemingly global or static storage duration (from the viewpoint of the functions using it) but in actual fact, there is one copy per thread.
When you declare a variable thread_local then each thread has its own copy. When you refer to it by name, then the copy associated with the current thread is used.

```
#include <iostream>
#include <string>
```

```
#include <thread>
#include <mutex>

thread_local unsigned int rage = 1;
std::mutex cout_mutex;

void increase_rage(const std::string& thread_name)
{
   ++rage; // modifying outside a lock is okay; this is a thread-local variable
   std::lock_guard<std::mutex> lock(cout_mutex);
   std::cout << "Rage counter for " << thread_name << ": " << rage << '\n';
}

int main()
{
   std::thread a(increase_rage, "a"), b(increase_rage, "b");

   {
      std::lock_guard<std::mutex> lock(cout_mutex);
      std::cout << "Rage counter for main: " << rage << '\n';
   }

   a.join();
   b.join();
}
```

## Explicitly defaulted and deleted special member functions

A defaulted default constructor is specifically defined as being the same as a user-defined default constructor with no initialization list and an empty compound statement.

If we have declared a user defined cotr, then compiler will not generate default cotr.
But If you want your class to be an aggregate or a trivial type (or by transitivity, a POD type), then you need to use = default.

- An aggregate is an array or a class with no user-provided constructors
- A default constructor is trivial if it is not user-provided
- A trivial class is a class that has a trivial default constructor

```
#include <type_traits>
struct X {
   X() = default;
};

struct Y {
   Y() { };
};

int main() {
   static_assert(std::is_trivial<X>::value, "X should be trivial");
   static_assert(std::is_pod<X>::value, "X should be POD");
```

```
    static_assert(!std::is_trivial<Y>::value, "Y should not be trivial");
    static_assert(!std::is_pod<Y>::value, "Y should not be POD");
}
```

**Some More example**

```
struct A
{
    int x;
    A(int x = 1): x(x) {} // user-defined default constructor
};

struct B: A
{
    // B::B() is implicitly-defined, calls A::A()
};

struct C
{
    A a;
    // C::C() is implicitly-defined, calls A::A()
};

struct D: A
{
    D(int y): A(y) {}
    // D::D() is not declared because another constructor exists
};

struct E: A
{
    E(int y): A(y) {}
    E() = default; // explicitly defaulted, calls A::A()
};

struct F
{
    int& ref; // reference member, which prevent default constructor
    const int c; // const member, which prevent default constructor
    // F::F() is implicitly defined as deleted
};
int main()
{
    A a;
    B b;
    C c;
//  D d; // compile error
    E e;
//  F f; //error: use of deleted function 'F::F()'
}
```

**The = delete specifier** can be used to prohibit calling any function, which can be used to disallow calling a member function with particular parameters.

For example:

```
struct NoInt
{
   void f(double i);
   void f(int) = delete;
};
```

An attempt to call f() with an int will be rejected by the compiler, instead of performing a silent conversion to double.

This can be generalized to disallow calling the function with any type other than double as follows:

```
struct OnlyDouble
{
   void f(double d);
   template<class T> void f(T) = delete;
}
```

Certain features can be explicitly disabled. For example, this type is non-copyable:

```
struct NonCopyable
{
   NonCopyable() = default;
   NonCopyable(const NonCopyable&) = delete;
   NonCopyable& operator=(const NonCopyable&) = delete;
};
```

## Type long long int

In C++03, the largest integer type is long int. C++11 adds a new integer type long long int. It is guaranteed to be at least as large as a long int, and have no fewer than 64 bits.

## Static assertions

C++03 provides two methods to test assertions: the macro assert and the preprocessor directive #error. However, neither is appropriate for use in templates: the macro tests the assertion at execution-time, while the preprocessor directive tests the assertion during preprocessing, which happens before instantiation of templates. Neither is appropriate for testing properties that are dependent on template parameters.

C++11 introduces a new way to test assertions at compile-time, using the new keyword **static_assert**.

Example:

```
template<class T>
struct Check
{
   static_assert(sizeof(int) <= sizeof(T), "T is not big enough!");
};
```

## Allow sizeof to work on members of classes without an explicit object

In C++03, the sizeof operator can be used on types and objects. But it cannot be used to do this:

```
struct SomeType { OtherType member; };

sizeof(SomeType::member); // Does not work with C++03. Okay with C++11
```
This should return the size of OtherType.

## Control and query object alignment

C++11 allows variable alignment to be queried and controlled with alignof and alignas.
The alignof operator takes the type and returns the power of 2 byte boundary on which the type instances must be allocated (as a std::size_t). When given a reference type alignof returns the referenced type's alignment; for arrays it returns the element type's alignment.
The alignas specifier controls the memory alignment for a variable. The specifier takes a constant or a type; when supplied a type alignas(T) is shorthand for alignas(alignof(T)).

For example,
To specify that a char array should be properly aligned to hold a float:

```
alignas(float) unsigned char c[sizeof(float)]
```

```
#include <stdalign.h>
#include <stdio.h>
// every object of type struct sse_t will be aligned to 16-byte boundary
// (note: needs support for DR 444)
struct sse_t
{
  alignas(16) float sse_data[4];
};

// every object of type struct data will be aligned to 128-byte boundary
struct data {
  char x;
  alignas(128) char cacheline[128]; // over-aligned array of char,
                                    // not array of over-aligned chars
};

int main(void)
{
    printf("sizeof(data) = %zu (1 byte + 127 bytes padding + 128-byte array)\n",
           sizeof(struct data));

    printf("alignment of sse_t is %zu\n", alignof(struct sse_t));

    alignas(2048) struct data d; // this instance of data is aligned even stricter
}
```

Output:

```
sizeof(data) = 256 (1 byte + 127 bytes padding + 128-byte array)
alignment of sse_t is 16
```

**Example of alignof**

```
#include <iostream>
```

```
struct Foo {
    int   i;
    float f;
    char  c;
};

struct Empty {};

struct alignas(64) Empty64 {};

int main()
{
    std::cout << "Alignment of"  "\n"
        "- char             : " << alignof(char)    << "\n"
        "- pointer          : " << alignof(int*)    << "\n"
        "- class Foo        : " << alignof(Foo)     << "\n"
        "- empty class      : " << alignof(Empty)   << "\n"
        "- alignas(64) Empty: " << alignof(Empty64) << "\n";
}
```

Possible output:

```
Alignment of
- char             : 1
- pointer          : 8
- class Foo        : 4
- empty class      : 1
- alignas(64) Empty: 64
```

## Attributes

Attributes are one of the key features of modern C++ which allows the programmer to specify additional information to the compiler to enforce constraints(conditions), optimise certain pieces of code or do some specific code generation. In simple terms, an attribute acts as an annotation or a note to the compiler which provides additional information about the code for optimization purposes and enforcing certain conditions on it. Introduced in C++11, they have remained one of the best features of C++ and are constantly being evolved with each new version of C++.

Example: 1

**To enforce constraints on the code:**

Old ways:

```
int f(int i)
{
    if (i > 0)
        return i;
    else
        return -1;

    // Code
}
```

In C++11

```
int f(int i)[[expects:i > 0]]
```

```
{
    // Code
}
```

**To give additional information to the compiler for optimization purposes:**

```
int f(int i)
{
    switch (i) {
    case 1:
        [[fallthrough]];
        [[likely]] case 2 : return 1;
    }
    return -1;
}
```

**Suppressing certain warnings and errors that programmer intended to have in his code:**

```
int main()
{

    // Set debug mode in compiler or 'R'
    [[maybe_unused]] char mg_brk = 'D';

    // Compiler does not emit any warnings
    // or error on this unused variable
}
#include <iostream>
#include <string>

[[noreturn]] void f()
{
    // Some code that does not return
    // back the control to the caller
    // In this case the function returns
    // back to the caller without a value
    // This is the reason why the
    // warning "noreturn' function does return' arises
}
void g()
{
    std::cout << "Code is intented to reach here";
}
int main()
{
    f();
    g();
}
```

## Threading facilities

std::thread is the thread class that represents a single thread in C++.
To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable.

A callable can be either of the three

- A function pointer
- A function object
- A lambda expression

```
//Using Function
void foo(param)
{
   // Do something
}
// The parameters to the function are put after the comma
std::thread thread_obj(foo, params);

// Define a lamda expression
auto f = [](params) {
   // Do Something
};
 // Pass f and its parameters to thread   object constructor as below
std::thread thread_object(f, params);
/We can also pass lambda functions directly to the constructor.
std::thread thread_object([](params) {
   // Do Something
};, params);

// Define the class of function object
class fn_object_class {
   // Overload () operator
   void operator()(params)
   {
      // Do Something
   }
}
// Create thread object
std::thread thread_object(fn_class_object(), params)
int main()
{
   // Start thread t1
   std::thread t1(callable);

   // Wait for t1 to finish
   t1.join();
```

```
    // t1 has finished do other stuff

    ...
}
```

**Some more example:**
==========

```
// CPP program to demonstrate multithreading
// using three different callables.
#include <iostream>
#include <thread>
using namespace std;

// A dummy function
void foo(int Z)
{
    for (int i = 0; i < Z; i++) {
        cout << "Thread using function"
             " pointer as callable\n";
    }
}

// A callable object
class thread_obj {
public:
    void operator()(int x)
    {
        for (int i = 0; i < x; i++)
            cout << "Thread using function"
                 " object as  callable\n";
    }
};
int main()
{
    cout << "Threads 1 and 2 and 3 "
         "operating independently" << endl;

    // This thread is launched by using
    // function pointer as callable
    thread th1(foo, 3);

    // This thread is launched by using
    // function object as callable
    thread th2(thread_obj(), 3);

    // Define a Lambda Expression
    auto f = [](int x) {
        for (int i = 0; i < x; i++)
```

```
        cout << "Thread using lambda"
         " expression as callable\n";
    };

    // This thread is launched by using
    // lamda expression as callable
    thread th3(f, 3);

    // Wait for the threads to finish
    // Wait for thread t1 to finish
    th1.join();
    // Wait for thread t2 to finish
    th2.join();

    // Wait for thread t3 to finish
    th3.join();

    return 0;
}
```

## Joining and Detaching Thread:

Once a thread is started then another thread can wait for this new thread to finish.
For this another need to call join() function on the std::thread object.
Joining Threads with std::thread::join()

**example:**

```
void thread_function()
{
    for(int i = 0; i < 10000; i++);
        std::cout<<"thread function Executing"<<std::endl;
}
std::thread th(thread_function);
// Some Code
th.join();
```

Detaching Threads using std::thread::detach()

```
std::thread th(thread_function);
th.detach();
```

Be careful with calling detach() and join() on Thread Handles

```
th.join();
th.join(); // this will lead to crash
th.detach();
th.detach(); // This will lead to crash
```

In case again join() function is called on same object again then it will cause the program to Terminate. same with detach().

```
class DummyClass {
public:
    DummyClass() {}
    DummyClass(const DummyClass & obj)   {}
    void sampleMemberFunction(int x)
```

```
    {
        std::cout<<"Inside sampleMemberFunction "<<x<<std::endl;
    }
};
int main() {
     DummyClass dummyObj;
    int x = 10;
    std::thread threadObj(&DummyClass::sampleMemberFunction,&dummyObj, x);
    threadObj.join();
}
```

or if the thread function signature is like below:

```
void threadCallback(int x, std::string str); →signature
    int x = 10;   std::string str = "Sample String";
    std::thread threadObj(threadCallback, x, str); →Call like this
```

## Data Sharing and Race Condition

When two or more threads perform a set of operations in parallel, that access the same memory location.Also, one or more thread out of them modifies the data in that memory location, then this can lead to an unexpected results some times.

```
class Wallet
{
    int mMoney;
public:
    Wallet() :mMoney(0){}
    int getMoney() { return mMoney; }
    void addMoney(int money)
    {
        // in MT ENV, this will lead to problem, will be solved by using Mutex.
       for(int i = 0; i < money; ++i)       {
          mMoney++;
       }
    }
};
```

Now Let's create 5 threads and all these threads will share a same object of class Wallet and add 1000 to internal money using it's addMoney() member function in parallel. So, if initially money in wallet is 0.
Then after completion of all thread's execution money in Wallet should be 5000.
As addMoney() member function of same Wallet class object is executed 5 times.
hence it's internal money is expected to be 5000. But as addMoney() member function is executed in parallel
hence in some scenarios mMoney will be much lesser than 5000.

```
int testMultithreadedWallet()
{
   Wallet walletObject;
```

```
   std::vector<std::thread> threads;
   for(int i = 0; i < 5; ++i){
       threads.push_back(std::thread(&Wallet::addMoney, &walletObject, 1000));
   }

   for(int i = 0; i < threads.size() ; i++)
   {
      threads.at(i).join();
   }
   return walletObject.getMoney();
}

int main()
{

  int val = 0;
  for(int k = 0; k < 1000; k++)
  {
     if((val = testMultithreadedWallet()) != 5000)
     {
       std::cout << "Error at count = "<<k<<" Money in Wallet = "<<val << std::endl;
     }
  }
  return 0;
}


void addMoney(int money)
   {
      mutex.lock();
      for(int i = 0; i < money; ++i)
                {
                        mMoney++;
                }
      mutex.unlock();
   }
```

There are two important methods of mutex:
1) lock()
2) unlock()
It's guaranteed that it will not found a single scenario where money in wallet is less than 5000.
Because mutex lock in addMoney(), makes sure that once one thread finishes the modification of money,
then only any other thread modifies the money in Wallet.

But what if we forgot to unlock the mutex at the end of function. In such scenario, one thread will exit without releasing the lock and
other threads will remain in waiting. This kind of scenario can happen in case some exception came after locking the mutex.
To avoid such scenarios we should use std::lock_guard.

## STD::LOCK_GUARD

A lock guard is an object that manages a mutex object by keeping it always locked.
On construction, the mutex object is locked by the calling thread, and on destruction, the mutex is unlocked.
It guarantees the mutex object is properly unlocked in case an exception is thrown.

```
std::mutex mtx;

void print_even (int x) {
  if (x%2==0) std::cout << x << " is even\n";
  else throw (std::logic_error("not even"));
}

void print_thread_id (int id) {
  try {
    // using a local lock_guard to lock mtx guarantees unlocking on destruction / exception:
    std::lock_guard<std::mutex> lck (mtx);
    print_even(id);
  }
  catch (std::logic_error&) {
    std::cout << "[exception caught]\n";
  }
}
int main ()
{
  std::thread threads[10];
  // spawn 10 threads:
  for (int i=0; i<10; ++i)
    threads[i] = std::thread(print_thread_id,i+1);

  for (auto& th : threads) th.join();

  return 0;
}
```

## Difference between std::lock_guard and std::unique_lock

One of the differences between std::lock_guard and std::unique_lock is that the programmer is able to unlock std::unique_lock,
but she/he is not able to unlock std::lock_guard.
std::lock_guard guard1(mutex);
Then the constructor of guard1 locks the mutex. At the end of guard1's life, the destructor unlocks the mutex.
There is no other possibility to unlock it & also it does not have any other function too.

On the other hand, we have an object of std::unique_lock. It has guard2.unlock(); & guard2.lock();

## CONDITION VARIABLE

The condition_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the condition),
and notifies the condition_variable.

Condition variables allow us to synchronize threads via notifications. So, you can implement workflows like sender/receiver or producer/consumer.
In such a workflow, the receiver is waiting for the sender's notification. If the receiver gets the notification,
it continues its work. The thread that intends to modify the variable has to acquire a
std::mutex (typically via std::lock_guard)
perform the modification while the lock is held execute notify_one or notify_all on the
std::condition_variable
(the lock does not need to be held for notification).

Condition variables allow one to atomically release a held mutex and put the thread to sleep. Then, after being signaled, atomically re-acquire the mutex and wake up.
condition variables always take a mutex. The mutex must be held when wait is called. You should always verify
that the desired condition is still true after returning from wait. The mutex protects the shared state.
The condition lets you block until signaled. Uunique_lock is an RAII (Resource Acquisition Is Initialization) wrapper for locking
and unlocking the given mutex.

condition_variable  which is shared by threads.
condition_variable .wait() function releases this mutex before suspending the thread and obtains it again before returning.
condition_variable .wait() function waits until a notify_one or notify_all is received.

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
   // Wait until main() sends data
   std::unique_lock<std::mutex> lk(m);
   cv.wait(lk, []{return ready;});
```

```
    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one();
}

int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

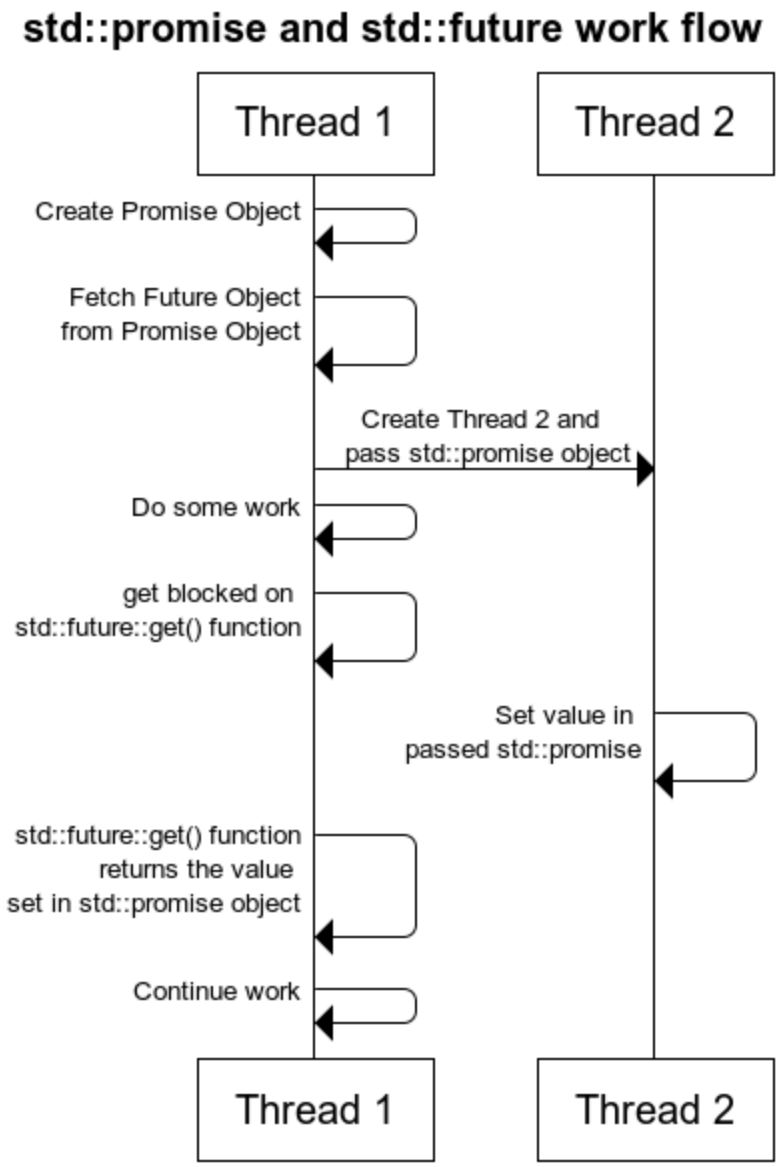
Example:

conditionVariable.cpp

```
std::mutex mutex_;
std::condition_variable condVar;
void doTheWork(){
  std::cout << "Processing shared data." << std::endl;
}
void waitingForWork(){
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
    doTheWork();
    std::cout << "Work done." << std::endl;
```

```
}
void setDataReady(){
    std::cout << "Sender: Data is ready."  << std::endl;
    condVar.notify_one();
}
int main(){
std::thread t1(waitingForWork);
  std::thread t2(setDataReady);
  t1.join();  t2.join();
  std::cout << std::endl;
  }
```

The program has two child threads: t1 and t2. They get their callable payload (functions or functors) waitingForWork and setDataReady . The function setDataReady notifies  - using the condition variable condVar - that it is done with the preparation of the work: condVar.notify_one(). While holding the lock, thread t2 is waiting for its notification: condVar.wait(lck). The waiting thread always performs same steps. It wakes up, tries to get the lock, checks if he's holding the lock,if the notifications arrived and, in case of failure, puts himself  back to sleep. In case of success, the thread leaves the endless loop and continues with its work.




```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
```

```
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;});

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one();
}

int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

## std::future & std::promise

Actually a std::future object internally stores a value that will be assigned in future and it also provides a mechanism to access that value i.e. using get() member function. But if somebody tries to access this associated value of future through get() function before it is available, then get() function will block till value is not available.

Every std::promise object has an associated std::future object, through which others can fetch the value set by promise.
So, Thread 1 will create the std::promise object and then fetch the std::future object from it before passing the std""promise object to thread

2 i.e.std::promise is also a class template and its object promises to set the value in future. Each std::promise object has an associated std::future object that will give the value once set by the std::promise object.
A std::promise object shares data with its associated std::future object.

std::future<int> futureObj = promiseObj.get_future();

Now Thread 1 will pass the promiseObj to Thread 2.

Then Thread 1 will fetch the value set by Thread 2 in std::promise through std::future's get function,

int val = futureObj.get();

But if value is not yet set by thread 2 then this call will get blocked until thread 2 sets the value in promise object i.e.

promiseObj.set_value(45);

```
#include <thread>
#include <future>
void initiazer(std::promise<int> * promObj)
{
   std::cout<<"Inside Thread"<<std::endl;      promObj->set_value(35);
}
int main()
{
//As of now this promise object doesn't have any associated value.
//But it gives a promise that somebody will surely set the value
//in it and once its set then you can get that value through associated std::future object.
   std::promise<int> promiseObj;
   std::future<int> futureObj = promiseObj.get_future();
   std::thread th(initiazer, &promiseObj);
   std::cout<<futureObj.get()<<std::endl;
   th.join();
   return 0;
}
```

If std::promise object is destroyed before setting the value the calling get() function on associated std::future object will throw exception. A part from this, if you want your thread to return multiple values at different point of time then just pass multiple std::promise objects in thread and fetch multiple return values from their associated multiple std::future objects.

## Tuple types

Tuples are a fixed-size collection of heterogeneous values. Tuples are collections composed of heterogeneous objects of pre-arranged dimensions. A tuple can be considered a generalization of a struct's member variables.

```
// `playerProfile` has type `std::tuple<int, const char*, const char*>`.
auto playerProfile = std::make_tuple(51, "Frans Nielsen", "NYI");
std::get<0>(playerProfile); // 51
std::get<1>(playerProfile); // "Frans Nielsen"
std::get<2>(playerProfile); // "NYI"
```

std::tuple is a fixed-size collection of heterogeneous values. It is a generalization of std::pair.

```
// Declaring tuple
 tuple <char, int, float> geek;
 // Initializing 1st tuple
 tuple <int,char,float> tup1(20,'g',17.5);

 // Assigning values to tuple using make_tuple()
 geek = make_tuple('a', 10, 15.5);
 // Use of size to find tuple_size of tuple
 cout << tuple_size<decltype(geek)>::value << endl;
 tie() :- The work of tie() is to unpack the tuple values into separate variables
  // Use of tie() without ignore
 tie(i_val,ch_val,f_val) = tup1;
 tuple_cat() :- This function concatenates two tuples and returns a new tuple.
 // Concatenating 2 tuples to return a new tuple
 auto tup3 = tuple_cat(tup1,tup2);
```

```
#include <tuple>
#include <iostream>
int main()
{
  auto t = std::make_tuple("String",5.2, 1);
  std::cout << std::get<0>(t) << ' '
            << std::get<1>(t) << ' '
            << std::get<2>(t) << '\n';
}
```

## Hash tables

Including hash tables (unordered associative containers) in the C++ standard library is one of the most recurring requests. It was not adopted in C++03 due to time constraints only. Although hash tables are less efficient than a balanced tree in the worst case (in the presence of many collisions), they perform better in many real applications.
Collisions are managed only via linear chaining because the committee didn't consider it to be opportune to standardize solutions of open addressing that introduce quite a lot of intrinsic problems (above all when erasure of elements is admitted). To avoid name clashes with non-standard libraries that developed their own hash table implementations, the prefix “unordered” was used instead of “hash”.

## Regular expressions

regex_match() -This function return true if the regular expression is a match against the given string otherwise it returns false.

```
string a = "GeeksForGeeks";

    // Here b is an object of regex (regular expression)
    regex b("(Geek)(.*)"); // Geeks followed by any character

    // regex_match function matches string a against regex b
    if ( regex_match(a, b) )
        cout << "String 'a' matches regular expression 'b' \n";


   string s = "I am looking for GeeksForGeek \n";

    // matches words beginning by "Geek"
    regex r("Geek[a-zA-z]+");

    // regex_replace() for replacing the match w
```

## std::to_string

Converts a numeric argument to a std::string.
std::to_string(1.2); // == "1.2"
std::to_string(123); // == "123"
The chrono library contains a set of utility functions and types that deal with durations, clocks, and time points. One use case of this library is benchmarking code:
**std::chrono**
==========
std::chrono::time_point<std::chrono::steady_clock> start, end;
start = std::chrono::steady_clock::now();
// Some computations...
end = std::chrono::steady_clock::now();
std::chrono::duration<double> elapsed_seconds = end - start;
double t = elapsed_seconds.count(); // t number of seconds, represented as a `double`

## General-purpose smart pointers

C++11 provides **std::unique_ptr**, **std::shared_ptr and std::weak_ptr**. **std::auto_ptr** is deprecated.

1. **std::unique_ptr**
2. **std::shared_ptr**
3. **std::weak_ptr**

## Std::unique_ptr:

If a C++ function returns a pointer, there is no way to know whether the caller should delete the memory of the referent when the caller is finished with the information.

Example:
SomeType* AmbiguousFunction(); **// What should be done with the result?**
C++11 introduced a way to ensure correct memory management in this case by declaring the function to return a unique_ptr.

std::unique_ptr<SomeType> ObviousFunction();

The declaration of the function return type as a unique_ptr makes explicit the fact that the caller takes ownership of the result, and the C++ runtime ensures that the memory will be reclaimed automatically. **Before C++11, unique_ptr can be replaced with auto_ptr.**
std::auto_ptr now becomes deprecated and then eventually removed in C++17.

std::unique_ptr is a non-copyable, movable pointer that manages its own heap-allocated memory. **Note: Prefer using the** std::make_X **helper functions as opposed to using constructors. See the sections for** std::make_unique **and** std::make_shared**.**

**Examples:**

```
std::unique_ptr<Foo> p1 { new Foo{} };  // `p1` owns `Foo`
if (p1) {
  p1->bar();
}

{
  std::unique_ptr<Foo> p2 {std::move(p1)};  // Now `p2` owns `Foo`
  f(*p2);

  p1 = std::move(p2);  // Ownership returns to `p1` -- `p2` gets destroyed
}

if (p1) {
  p1->bar();
}
// `Foo` instance is destroyed when `p1` goes out of scope
```

**Question on Unique_ptr:**

- **What happens when a std::unique_ptr is passed by value to a function?**

```
#include <memory>
auto f(std::unique_ptr<int> ptr)
{
 *ptr = 42;
return ptr;
}
int main()
{
auto ptr = std::make_unique<int>();
ptr = f(ptr);
}
```

**Ans:**
The correct answer is that this code won't even compile. The **std::unique_ptr** type cannot be copied, so passing it as a parameter to a function will fail to compile.
To convince the compiler that this is fine, **std::move** can be used:

- Write a copy constructor, move constructor, copy assignment operator, and move assignment operator for the following class.

```
class DirectorySearchResult {
public:
DirectorySearchResult( std::vector<std::string> const& files,
size_t attributes,
SearchQuery const* query) : files(files), attributes(attributes), query(new
SearchQuery(*query)) { }

~DirectorySearchResult() {
```

```
delete query;
}

private:
std::vector<std::string> files;
size_t attributes;
SearchQuery* query;
};
```

**Ans:**

**Copy constructor:**

```
DirectorySearchResult(DirectorySearchResult const& other)
  : files(other.files),
    attributes(other.attributes),
    query(other.query ? new SearchQuery(*other.query) : nullptr)
{ }
```

**Move constructor:**

```
DirectorySearchResult(DirectorySearchResult&& other)
  : files(std::move(other.files)),
    attributes(other.attributes),
    query(other.query)
{
  other.query = nullptr;
}
```

**Assignment operator:**

```
DirectorySearchResult& operator=(DirectorySearchResult const& other)
{
  if (this == &other)
    return *this;
  files = other.files;
  attributes = other.attributes;
  delete query;
  query = other.query ? new SearchQuery(*other.query) : nullptr;
  return *this;
}
```

**Move assignment operator:**

```
DirectorySearchResult& operator=(DirectorySearchResult&& other)
{
  files = std::move(other.files);
  attributes = other.attributes;
  std::swap(query, other.query);
  return *this;
}
```

What is the difference between std::make_shared<T>(…) and std::shared_ptr<T>(new T(…)).?

**Ans:**

std::make_shared is more exception-safe when used as a function argument.

**Some nice example of unique_ptr:**

## unique_ptr object is not copyable

As unique_ptr<> is not copyable, only movable. Hence we cannot create copy of a unique_ptr object either through copy constructor or assignment operator.

```
// Create a unique_ptr object through raw pointer
std::unique_ptr<Task> taskPtr2(new Task(55));

// Compile Error : unique_ptr object is Not copyable
std::unique_ptr<Task> taskPtr3 = taskPtr2; // Compile error

// Compile Error : unique_ptr object is Not copyable
taskPtr = taskPtr2; //compile error

 unique_ptr<A> ptr1 (new A);

 // Error: can't copy unique_ptr
 unique_ptr<A> ptr2 = ptr1;
// Works, resource now stored in ptr2
unique_ptr<A> ptr2 = move(ptr1);
```

## Releasing the associated raw pointer

Calling release() on unique_ptr object will release the ownership of associated raw pointer from the object.
It returns the raw pointer.

```
// Create a unique_ptr object through raw pointer
std::unique_ptr<Task> taskPtr5(new Task(55));
if(taskPtr5 != nullptr)
std::cout<<"taskPtr5 is not empty"<<std::endl;
// Release the ownership of object from raw pointer
Task * ptr = taskPtr5.release();
if(taskPtr5 == nullptr)
std::cout<<"taskPtr5 is empty"<<std::endl;


#include <iostream>
#include <memory>

struct Task
{
    int mId;
    Task(int id ) :mId(id)
    {
            std::cout<<"Task::Constructor"<<std::endl;
    }
    ~Task()
    {
```

```
        std::cout<<"Task::Destructor"<<std::endl;
    }
};

int main()
{
    // Empty unique_ptr object
    std::unique_ptr<int> ptr1;

    // Check if unique pointer object is empty
    if(!ptr1)
        std::cout<<"ptr1 is empty"<<std::endl;

    // Check if unique pointer object is empty
    if(ptr1 == nullptr)
        std::cout<<"ptr1 is empty"<<std::endl;

    // can not create unique_ptr object by initializing through assignment
    // std::unique_ptr<Task> taskPtr2 = new Task(); // Compile Error

    // Create a unique_ptr object through raw pointer
    std::unique_ptr<Task> taskPtr(new Task(23));

    // Check if taskPtr is empty or it has an associated raw pointer
    if(taskPtr != nullptr)
        std::cout<<"taskPtr is  not empty"<<std::endl;

    //Access the element through unique_ptr
    std::cout<<taskPtr->mId<<std::endl;

    std::cout<<"Reset the taskPtr"<<std::endl;
    // Reseting the unique_ptr will delete the associated
    // raw pointer and make unique_ptr object empty
    taskPtr.reset();

    // Check if taskPtr is empty or it has an associated raw pointer
    if(taskPtr == nullptr)
        std::cout<<"taskPtr is  empty"<<std::endl;


    // Create a unique_ptr object through raw pointer
    std::unique_ptr<Task> taskPtr2(new Task(55));

    if(taskPtr2 != nullptr)
        std::cout<<"taskPtr2 is  not empty"<<std::endl;

    // unique_ptr object is Not copyable
    //taskPtr = taskPtr2; //compile error
```

```
    // unique_ptr object is Not copyable
    //std::unique_ptr<Task> taskPtr3 = taskPtr2;

    {
        // Transfer the ownership

        std::unique_ptr<Task> taskPtr4 = std::move(taskPtr2);


        if(taskPtr2 == nullptr)
            std::cout<<"taskPtr2 is  empty"<<std::endl;

        // ownership of taskPtr2 is transfered to taskPtr4
        if(taskPtr4 != nullptr)
            std::cout<<"taskPtr4 is not empty"<<std::endl;

        std::cout<<taskPtr4->mId<<std::endl;

        //taskPtr4 goes out of scope and deletes the assocaited raw pointer
    }

    // Create a unique_ptr object through raw pointer
    std::unique_ptr<Task> taskPtr5(new Task(55));

    if(taskPtr5 != nullptr)
        std::cout<<"taskPtr5 is not empty"<<std::endl;

    // Release the ownership of object from raw pointer
    Task * ptr = taskPtr5.release();

    if(taskPtr5 == nullptr)
        std::cout<<"taskPtr5 is empty"<<std::endl;

    std::cout<<ptr->mId<<std::endl;

    delete ptr;

    return 0;
}
```

## shared_ptr:

A shared_ptr is a container for raw pointers. It is a reference counting ownership model i.e. it maintains the reference count of its contained pointer in cooperation with all copies of the shared_ptr.
So, the counter is incremented each time a new pointer points to the resource and decremented when the destructor of the object is called.

**Reference Counting**: It is a technique of storing the number of references, pointers or handles to a resource such as an object, block of memory, disk space or other resources.

An object referenced by the contained raw pointer will not be destroyed until reference count is greater than zero i.e. until all copies of shared_ptr have been deleted.

So, we should use shared_ptr when we want to assign one raw pointer to multiple owners.

**Example -1:**

```
std::shared_ptr<int[]> p1(new int[5]);  // Valid, allocates 5 integers.
std::shared_ptr<int[]> p2 = p1;  // Both now own the memory.

p1.reset();  // Memory still exists, due to p2.
p2.reset();  // Deletes the memory, since no one else owns the memory.
```

**Example -2:**

```
void baz(std::shared_ptr<int> p3) {
 //Stage 3
 std::cout << "@3 Ref Count: " << p3.use_count() << "\n"; //@3 Ref Count: 3
}

int main () {
 //Create a shared_ptr<int>
 auto p1 = std::make_shared<int>(0);
 //Stage 1
 std::cout << "@1 Ref Count: " << p1.use_count() << "\n"; //@1 Ref Count: 1

 { // Block
  //Create copy
  auto p2 = p1;
  //Stage 2
  std::cout << "@2 Ref Count: " << p2.use_count() << "\n"; //@2 Ref Count: 2
  //Will create another copy
  baz(p2);
  //Stage 4
  std::cout << "@4 Ref Count: " << p2.use_count() << "\n"; //@4 Ref Count: 2
 }

 //Stage 5
 std::cout << "@5 Ref Count: " << p1.use_count() << "\n"; //@5 Ref Count: 1
 //reset
 p1.reset();
 //Stage 6
 std::cout << "@6 Ref Count: " << p1.use_count() << "\n"; //@6 Ref Count: 0
 return 0;
}
```

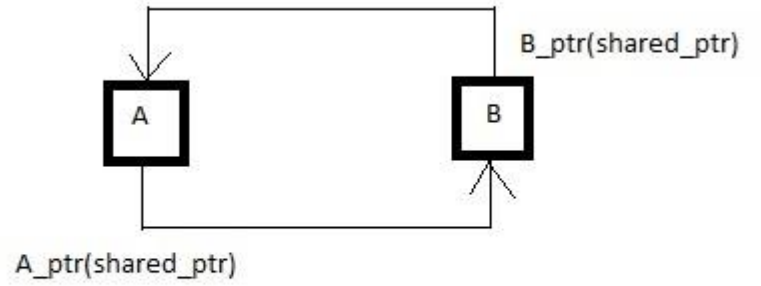
## weak_ptr:

A weak_ptr is created as a copy of shared_ptr. It provides access to an object that is owned by one or more shared_ptr instances but does not participate in reference counting. The existence or destruction of weak_ptr has no effect on the shared_ptr or its other copies. It is required in some cases to break circular references between shared_ptr instances.

Where does weak pointer required most?
It is highly required in Cyclic Dependency with shared pointer.

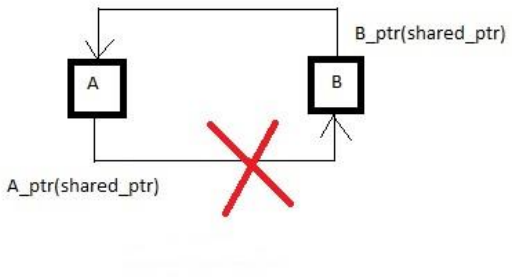
## Cyclic Dependency (Problems with shared_ptr):

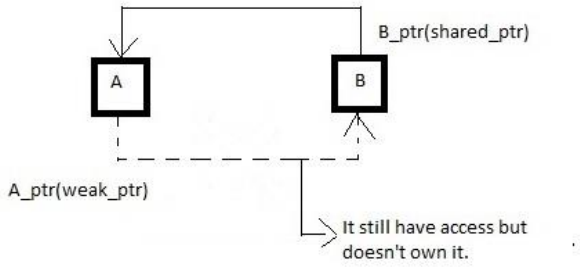
Example:
Let's consider a scenario where we have two classes A and B, both have pointers to other classes. So, it's always like A is pointing to B and B is pointing to A. Hence, use_count will never reach zero and they never get deleted.




**Circular Reference**

This is the reason we use **weak pointers**(weak_ptr) as they are not reference counted. So, the class in which weak_ptr is declared doesn't have a stronghold of it i.e. the ownership isn't shared, but they can have access to these objects.




So, in case of shared_ptr because of cyclic dependency use_count never reaches zero which is prevented using weak_ptr, which removes this problem by declaring A_ptr as weak_ptr, thus class A does not own it, only have access to it and we also need to check the validity of object as it may go out of scope. In general, it is a design issue.

**Some more example:**

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
```

```
public:
std::shared_ptr<Resource> m_ptr; // initially created empty
Resource() { std::cout << "Resource acquired\n"; }
~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
auto ptr1 = std::make_shared<Resource>();
 //! Create cyclic reference "cyclical ownership"
ptr1->m_ptr = ptr1; // m_ptr is now sharing the Resource that contains it

return 0;
}
```

In the above example, when ptr1 goes out of scope, it doesn't deallocate the Resource because the Resource's m_ptr is sharing the Resource. Then there's nobody left to delete the Resource (m_ptr never goes out of scope, so it never gets a chance). Thus, the program prints:

```
Resource acquired
```

and that's it.
How to solve it?

```
std::shared_ptr<Resource> m_ptr; // initially created empty
```

Instead, declare like below:

```
std::weak_ptr<Resource> m_ptr; // initially created empty
```

```
#include <iostream>
#include <memory> // for std::shared_ptr and std::weak_ptr
#include <string>

class Person
{
std::string m_name;
//! This below yellow colored line will create cyclical ownership, resolved by weak_ptr
//! std::shared_ptr<Person> m_partner; // initially created empty
std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr

public:

Person(const std::string &name) : m_name(name)
{
std::cout << m_name << " created\n";
```

```
}
~Person()
{
std::cout << m_name << " destroyed\n";
}

friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
{
if (!p1 || !p2)
return false;

p1->m_partner = p2;
p2->m_partner = p1;

std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";

return true;
}

const std::shared_ptr<Person> getPartner() const { return m_partner.lock(); } // use
lock() to convert weak_ptr to shared_ptr
const std::string& getName() const { return m_name; }
};

int main()
{
auto lucy = std::make_shared<Person>("Lucy");
auto ricky = std::make_shared<Person>("Ricky");

partnerUp(lucy, ricky);

auto partner = ricky->getPartner(); // get shared_ptr to Ricky's partner
std::cout << ricky->getName() << "'s partner is: " << partner->getName() << '\n';

return 0;
}
```

**Weak_ptr:**

A weak_ptr is a container for a raw pointer. It is created as a copy of a shared_ptr. The existence or destruction of weak_ptr does have no effect on the shared_ptr or its other copies. If all shared_ptr have been destroyed, then all weak_ptr pointing the same shared_ptr, become empty.

std::unique_ptr is meant for exclusive ownership, std::shared_ptr is meant for shared ownership and std::weak_ptr is basically for temporary ownership because it borrows the resource from a std::shared_ptr.

The main reason for having a std::weak_ptr in C++: is to break the cyclic references of std::shared_ptr's. It must be converted to std::shared_ptr in order to access the referenced object.

**One more example of cyclic reference:**

```
class B;
class A
{
public:
 A(  ) { };
 ~A( )
 {
  cout<<" A is destroyed"<<endl;
 }
 weak_ptr<B> m_sptrB;
//! shared_ptr<B> m_sptrB; // This would have been lead to cyclic reference
};
class B
{
public:
 B(  ) { };
 ~B( )
 {
  cout<<" B is destroyed"<<endl;
 }
 weak_ptr<A> m_sptrA;
//! shared_ptr<A> m_sptrA;  // This would have been lead to cyclic reference
 };

void main( )
{
 shared_ptr<B> sptrB( new B );
 shared_ptr<A> sptrA( new A );
 sptrB->m_sptrA = sptrA;
 sptrA->m_sptrB = sptrB;
 }
```

## Extensible random number facility

the C++11 mechanism will come with three base generator engine algorithms:

- linear_congruential_engine,
- subtract_with_carry_engine, and
- mersenne_twister_engine.

**C++11 also provides a number of standard distributions**:

- uniform_int_distribution,
- uniform_real_distribution,

- bernoulli_distribution,
- binomial_distribution,
- geometric_distribution,
- negative_binomial_distribution,
- poisson_distribution,
- exponential_distribution,
- gamma_distribution,
- weibull_distribution,
- extreme_value_distribution,
- normal_distribution,
- lognormal_distribution,
- chi_squared_distribution,
- cauchy_distribution,
- fisher_f_distribution,
- student_t_distribution,
- discrete_distribution,
- piecewise_constant_distribution and
- piecewise_linear_distribution.

The generator and distributions are combined as in this example:

```
#include <random>
#include <functional>

std::uniform_int_distribution<int> distribution(0, 99);
std::mt19937 engine; // Mersenne twister MT19937
auto generator = std::bind(distribution, engine);
int random = generator(); // Generate a uniform integral variate between 0 and 99.
int random2 = distribution(engine); // Generate another sample directly using the distribution and the engine objects.
```

## Wrapper reference

A wrapper reference is obtained from an instance of the template class reference_wrapper. Wrapper references are similar to normal references ('&') of the C++ language. To obtain a wrapper reference from any object the function template ref is used (for a constant reference cref is used).

Wrapper references are useful above all for function templates, where references to parameters rather than copies are needed:

```
// This function will obtain a reference to the parameter 'r' and increment it.
void func (int &r)  { r++; }

// Template function.
template<class F, class P> void g (F f, P t)  { f(t); }

int main()
{
   int i = 0;
   g (func, i); // 'g<void (int &r), int>' is instantiated
                // then 'i' will not be modified.
```

```
    std::cout << i << std::endl; // Output -> 0

    g (func, std::ref(i)); // 'g<void(int &r),reference_wrapper<int>>' is instantiated
                           // then 'i' will be modified.
    std::cout << i << std::endl; // Output -> 1
}
```

This new utility was added to the existing <utility> header and didn't need further extensions of the C++ language.

## Polymorphic wrappers for function objects

Polymorphic wrappers for function objects are similar to function pointers in semantics and syntax, but are less tightly bound and can indiscriminately refer to anything which can be called (function pointers, member function pointers, or functors) whose arguments are compatible with those of the wrapper.
An example can clarify its characteristics:

```
std::function<int (int, int)> func; // Wrapper creation using
                                    // template class 'function'.
std::plus<int> add; // 'plus' is declared as 'template<class T> T plus( T, T ) ;'
                    // then 'add' is type 'int add( int x, int y )'.
func = add;  // OK - Parameters and return types are the same.

int a = func (1, 2); // NOTE: if the wrapper 'func' does not refer to any function,
                     // the exception 'std::bad_function_call' is thrown.

std::function<bool (short, short)> func2 ;
if (!func2)
{
    // True because 'func2' has not yet been assigned a function.

    bool adjacent(long x, long y);
    func2 = &adjacent; // OK - Parameters and return types are convertible.

    struct Test
    {
        bool operator()(short x, short y);
    };
    Test car;
    func = std::ref(car); // 'std::ref' is a template function that returns the wrapper
                          // of member function 'operator()' of struct 'car'.
}
func = func2; // OK - Parameters and return types are convertible.
```

The template class function was defined inside the header <functional>, without needing any change to the C++ language.

## Type traits for metaprogramming

Metaprogramming consists of creating a program that creates or modifies another program (or itself). This can happen during compilation or during execution. The C++ Standards Committee has decided to introduce a library that allows metaprogramming during compiling via templates.

```
//  primaryTypeCategories.cpp
```

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

struct A{
  int a;
  int f(int){return 2011;}
};

enum E{
  e= 1,
};

union U{
  int u;
};


int main(){
  std::cout <<  std::boolalpha <<  std::endl;
  std::cout << std::is_void<void>::value << std::endl;
  std::cout << std::is_integral<short>::value << std::endl;
  std::cout << std::is_floating_point<double>::value << std::endl;
  std::cout << std::is_array<int []>::value << std::endl;
  std::cout << std::is_pointer<int*>::value << std::endl;
  std::cout << std::is_null_pointer<std::nullptr_t>::value << std::endl;
  std::cout << std::is_member_object_pointer<int A::*>::value <<  std::endl;
  std::cout << std::is_member_function_pointer<int (A::*)(int)>::value << std::endl;
  std::cout << std::is_enum<E>::value << std::endl;
  std::cout << std::is_union<U>::value << std::endl;
  std::cout << std::is_class<std::string>::value << std::endl;
  std::cout << std::is_function<int * (double)>::value << std::endl;
  std::cout << std::is_lvalue_reference<int&>::value << std::endl;
  std::cout << std::is_rvalue_reference<int&&>::value << std::endl;

  std::cout <<  std::endl;
}
```

## Miscellaneous features

### std::bind and std::placeholder

One of the main use of std::function and std::bind is as more generalized function pointers.

You can use it to implement callback mechanism. And std::bind is used to bind the parameters of the function call.

```
int foo( int a, int b, int c) {
}
struct Foo {
    Foo(int num) : num_(num) {}
    void print_add(int i) const { std::cout << num_+i << '\n'; }
    int num_;
};
int main() {
// bind parameter 1, 2 on function foo, and use std:: placeholders::_1 as placeholder
// for the first parameter.
auto bindFoo = std:: bind(foo , std:: placeholders::_1 , 1 ,2);
// when call bindFoo , we only need one param left
bindFoo(1);
const Foo foo(314159);

// store a call to a member function and object
sing std::placeholders::_1;
 std::function<void(int)> f_add_display2 = std::bind( &Foo::print_add, foo, _1 );
   f_add_display2(2);
}
```

Container:

## Linear Container

### std::array

The array is a container for constant size arrays. This container wraps around fixed size arrays and

also doesn't loose the information of its length when decayed to a pointer.

std::array<int,10> arr: The 10 elements are not initialized.

std::array<int,10>arr{}. The 10 elements are value-initialized.

std::array<int,10>arr{1,2,3,4): The remaining elements are value-initialized.

### std::forward_list

Forward lists are implemented as singly-linked lists; Singly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept by the association to each element of a link to the next element in the sequence. Provides element insertion of O(1) complexity, does not support fast random access (this is also a feature of linked lists), Forward lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence.

It does not provide the size() method

### Unordered Container

In traditional, elements are internally implemented by red-black trees. The average complexity of inserts and searches is O(log(size)). While inserting into ordered container, the element size is compared according to the < operator & then stored in right place. That is why

it is in sorted order or called ordered container. The elements in the unordered container are not sorted, and the internals are implemented by the Hash table.
The average complexity of inserting and searching for elements is O(constant), Significant performance gains can be achieved without concern for the order of the elements inside the container. C++11 introduces two sets of unordered containers: std::unordered_map/std::unordered_multimap and std::unordered_set/std::unordered_multiset.Their usage is basically similar to the original std::map/std::multimap/std::set/set::multiset

```
# include  <iostream >
#  include <string  >
# include < unordered_map >
# include <map >
int main() {
// initialized in same order
std:: unordered_map <int , std:: string > u = {
{1 , "1"},
{3 , "3"},
{2 , "2"}
};
std:: map <int , std:: string > v = {
{1 , "1"},
{3 , "3"},
{2 , "2"}
};
// iterates in the same way
std:: cout << " std:: unordered_map" << std:: endl;
for ( const auto & n : u)
std:: cout << " Key:[" << n. first << "] Value:[" << n. second << "]\ n";
std:: cout << std:: endl;
std:: cout << " std:: map" << std:: endl;
for ( const auto & n : v)
std:: cout << " Key:[" << n. first << "] Value:[" << n. second << "]\ n";
}
```

# C++11 Faqs

## decltype vs auto

Ans:

auto that deduces types based on values being assigned to the variable, decltype deduces the type from an expression passed to it

decltype gives the declared type of the expression that is passed to it. auto does the same thing as template type deduction. So, for example, if you have a function that returns a

reference, auto will still be a value (you need auto& to get a reference), but decltype will be exactly the type of the return value.
Example:

```
#include <iostream>
int global{};
int& foo()
{
        return global;
}

int main()
{
decltype(foo()) a = foo(); //a is an `int&`
auto b = foo(); //b is an `int`
b = 2;

std::cout << "a: " << a << '\n'; //prints "a: 0"
std::cout << "b: " << b << '\n'; //prints "b: 2"

std::cout << "---\n";
decltype(foo()) c = foo(); //c is an `int&`
c = 10;

std::cout << "a: " << a << '\n'; //prints "a: 10"
std::cout << "b: " << b << '\n'; //prints "b: 2"
std::cout << "c: " << c << '\n'; //prints "c: 10"
}
```

Auto can be used in trailing return types for functions:

```
auto foo() -> int;
```

In generic programming it is useful:

```
template <typename T, typename U>
auto sum( T t, U u ) -> decltype(t+u)
```

## Explain about unordered_set:

An unordered_set is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the unordered_set takes constant time O(1) on an average which can go up to linear time O(n) in worst case.

## Sets vs Unordered Sets

Set is an ordered sequence of unique keys whereas unordered_set is a set in which key can be stored in any order, so unordered. Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of set operations is O(log n) while for unordered_set, it is O(1).

## How to create an unordered_set of user defined class or struct in C++?

Ans:The unordered_set internally implements a hash table to store elements. By default we can store only pre definded type as int, string, float etc.
We can not store user defined type like class/structure object in unordered_set just like POD. We need to supply a hash function/class which implements bool operator () (). This class is responsible to store elements in in unordered_set.
We create a structure/class type and define a comparison function inside that structure/class that will be used to compare two structure/class type objects.
Example:
**unordered_set(userDefinedType, MyHashType) us;**
//! MyHashType is class implementing hash function i.e
//! bool operator () (const struct &s) const{ //! return (this->memberVar == s. memberVar) ;}
Example in C++:

```
struct Test {
     int id;
     // This function is used by unordered_set to compare
   // elements of Test.
   bool operator==(const Test& t) const
   {
      return (this->id == t.id);
   }
};

// class for hash function
class MyHashFunction {
public:
   // id is returned as hash function , This function we can define in our main Test class too.
//! So that we don't have to define it as a separate class, But best practice is to have a special claas like java
   size_t operator()(const Test& t) const
   {
      return t.id;
   }
};
// Driver method
int main()
{
   // put values in each
   // structure define below.
   Test t1 = { 110 }, t2 = { 102 },
        t3 = { 101 }, t4 = { 115 };

   // define a unordered_set having
   // structure as its elements.
   unordered_set<Test, MyHashFunction> us;

   // insert structure in unordered_set
   us.insert(t1);
   us.insert(t2);
   us.insert(t3);
   us.insert(t4);

   // printing the elements of unordered_set
   for (auto e : us) {
      cout << e.id << " ";
```

```
    }

    return 0;
}
```

## The programmer has decided to store objects of a user-defined type(a structure) in an unordered_set. Which of the following are steps that must be taken in order for this to work properly**?**

A. The structure will have to define a default constructor
B. The structure will have to overload operator<() so that the elements can be stored in the proper palce within the collection.
C. The structure will have to create a specialization of std::hash for the user defined type.
D. The structure will have to overload operator==() in order for this type to be supported by this collection.
E. The structure will have to overload operator() so that elements may be locatedin the collection.

Ans: C,D and E.

## How can we inject custom specialization of std::hash in namespace std

Ans:

```
struct S {
    std::string first_name;
    std::string last_name;
};
bool operator==(const S& lhs, const S& rhs) {
    return lhs.first_name == rhs.first_name && lhs.last_name == rhs.last_name;
}

// custom specialization of std::hash can be injected in namespace std
namespace std
{
    template<>
    struct hash<S>
    {
        std::size_t operator()(S const& s) const noexcept
        {
            std::size_t h1 = std::hash<std::string>{}(s.first_name);
            std::size_t h2 = std::hash<std::string>{}(s.last_name);
            return h1 ^ (h2 << 1);
        }
    };
}
```

## How to create an unordered_map of user defined class in C++?

Ans:

unordered_map is used to implement hash tables. It stores key value pairs. For every key, a hash function is computed and value is stored at that hash entry. Hash functions for standard data types (int, char, string, ..) are predefined.

**//! UserDefinedType => key, MyHashType=> has function**

Syntax: unordered_map<UserDefinedType, ValueType, MyHashType> um;

We need to do two things

a. Unordered_map expects us to define operator ==() for our class
b. It also expects us to define bool operator()(const UserDefinedType&obj)const

```
// CPP program to demonstrate working of unordered_map
// for user defined data types.
#include <bits/stdc++.h>
using namespace std;

// Objects of this class are used as key in hash
// table. This class must implement operator ==()
// to handle collisions.
struct Person {
   string first, last; // First and last names

   Person(string f, string l)
   {
                first = f;
                last = l;
   }

   // Match both first and last names in case
   // of collisions.
   bool operator==(const Person& p) const
   {
                return first == p.first && last == p.last;
   }
};
class MyHashFunction {
public:

   // Use sum of lengths of first and last names
   // as hash function.
   size_t operator()(const Person& p) const
   {
                return p.first.length() + p.last.length();
   }
};

// Driver code
int main()
{
   unordered_map<Person, int, MyHashFunction> um;
   Person p1("kartik", "kapoor");
   Person p2("Ram", "Singh");
   Person p3("Laxman", "Prasad");

   um[p1] = 100;
   um[p2] = 200;
   um[p3] = 100;

   for (auto e : um) {
                cout << "[" << e.first.first << ", "
                                << e.first.last
                                << "] = > " << e.second << '\n';
   }

   return 0;
```

}

## When did move constructor comes into use?

Ans:
In C++11, in addition to copy constructors, objects can have move constructors.
(And in addition to copy assignment operators, they have move assignment operators.)
The move constructor is used instead of the copy constructor, if the object has type "rvalue-reference" (Type &&).
std::move() is a cast that produces an rvalue-reference to an object, to enable moving from it.
It's a new C++ way to avoid copies. For example, using a move constructor, a std::vector could just copy its internal pointer to data to the new object, leaving the moved object in an moved from state, therefore not copying all the data. This would be C++-valid.

## What does std::move do ?

Ans:
The first thing to note is that std::move() **doesn't actually move anything**. It converts an expression from being an lvalue (such as a named variable) to being an xvalue.
An xvalue tells the compiler:
You can plunder me, **move** anything I'm holding and use it elsewhere (since I'm going to be destroyed soon anyway)".
In C++11 std::vector::push_back will use a move constructor if passed an rvalue (and a move constructor exists for the type), If you declare move() constructor for this type as delete, then it will not be compiled.