/\*###################################################################

**\* Learn C++ 11**

**1. Initializer List**

**\*/**

//C++ 03 initializer list:

int arr[4] = {3, 2, 4, 5}; // We can initialize array like this, but no way to

// vector.

vector<int> v;

v.push\_back(3);

v.push\_back(2);

v.push\_back(4);

v.push\_back(5);

// C++ 11 extended the support

vector<int> v = {3, 4, 1, 9}; // Calling initializer\_list constructor

We can create light weight object of **std::initializer\_list<T>** that will refer to an array of elements of type T.

std::initialzer\_list<int> data = {1,2,4,5};

So, where ever compiler see elements in braces i.e. {a, b, c} it creates a  **std::initialzer\_list<T>**, where T is the type of elements in the list. Now, all containers i.e. vector & list etc. has a parameterized constructor that accepts this **std::initializer\_list<T>** as an argument and insert them i.e.

vector<T>::vector<T>(initializer\_list<T> elements){

......

}

**std::initialzer\_list<T>** also detects the narrowing of data i.e. it will detect the narrowing:

std::vector<int> vec= {1,2,3,4.5,5}; // Will give compile time warnings

// All the relevant STL containers have been updated to accept initializer\_list.

// Define your own initializer\_list constructor:

#include <initializer\_list>

class BoVector {

vector<int> m\_vec;

public:

BoVector(const initializer\_list<int>& v) {

for (initializer\_list<int>::iterator itr = v.begin(); itr!=v.end(); ++ itr)

m\_vec.push\_back(\*itr);

}

};

BoVector v = {0, 2, 3, 4};

BoVector v{0, 2, 3, 4}; // effectively the same

// Automatic normal Initialization

class Rectangle {

public:

Rectangle(int height, int width, int length){ }

};

void draw\_rect(Rectangle r);

int main() {

draw\_rect({5, 6, 9}); // Rectangle{5,6,9} is automatically called

}

// Note: use it with caution.

// 1. Not very readable, even with the help of IDE. Funcion name rarely indicates

// the type of parameter the function takes.

// 2. Function could be overloaded with differenct parameter types.

void draw\_rect(Triangle t);

**/\***

**\* 2. Uniform Initialization**

**\*/**

// C++ 03

class Dog { // Aggregate class or struct

public:

int age;

string name;

};

Dog d1 = {5, "Henry"}; // Aggregate Initialization

// C++ 11 extended the scope of curly brace initialization

class Dog {

public:

Dog(int age, string name) {...};

};

Dog d1 = {5, "Henry"};

/\* Uniform Initialization Search Order:

\* 1. Initializer\_list constructor

\* 2. Regular constructor that takes the appropriate parameters.

\* 3. Aggregate initializer.

\*/

Dog d1{3};

class Dog {

public:

int age; // 3rd choice

Dog(int a) { // 2nd choice

age = a;

}

Dog(const initializer\_list<int>& vec) { // 1st choice

age = \*(vec.begin());

}

};

**/\***

**\* 3. auto type**

**\*/**

std::vector<int> vec = {2, 3, 4, 5};

// C++ 03

for (std::vector<int>::iterator it = vec.begin(); it!=vec.end(); ++ it)

m\_vec.push\_back(\*it);

// C++ 11: use auto type, It can reduce the code:

for (auto it = vec.begin(); it!=vec.end(); ++ it)

m\_vec.push\_back(\*it);

auto a = 6; // a is a integer

auto b = 9.6; // b is a double

auto c = a; // c is an integer

auto const x = a; // int const x = a

auto& y = a; // int& y = a

// It's static type, no run-time cost, fat-free.

// It also makes code easier to maintain.

// 1. Don't use auto when type conversion is needed

// 2. IDE becomes more important

We can story ant type in auto even if its a function or some iterator. Storing Lambda function inside a auto variable

// Storing Lambda function inside a auto variable

auto fun\_sum = [](int a , int b){

return a+b;

};

std::cout<<fun\_sum(4,5)<<std::endl;

**Important points about auto variable in C++11**

Once you have initialized the auto variable then you can change the value but you cannot change the type

auto x = 1;

// Cannot change the type of already initialized auto variable

// Error will occur at compile time

// x = "dummy";

It cannot be left uninitialized i.e. auto a;

**/\***

**\* 4. Bind**

**\*/**

**std::bind** is a Standard Function Objects that acts as a Functional Adaptor i.e. it takes a function as input and returns a new function Object as an output with one or more of the arguments of passed function bound or rearranged. Suppose We have a function to add two numbers i.e.

int add(int first, int second){

return first + second;

}

std::bind takes a function as its first parameter and then that function’s argument as its parameter.

auto add\_func = std::bind(&add, \_1, \_2);

**std::bind** took the first parameter a function i.e. &add and then its arguments as \_1 & \_2 as his own arguments. ***So, add\_func(4,5) is equivalent to add(4,5).***

auto add\_func = std::bind(&add, 12, \_1); //First argument is fixed: 12

### Use of std::bind in STL algorithms

As std::bind acts as a functional adaptor and gives a new function objects, hence it is very useful with many STL algorithms. Example, list of numbers and we want to count the numbers which are multiple of 5. To achieve this, we have an existing function i.e.

bool divisible(int num , int den){

if(num % den == 0)

return true;

return false;

}

int main()

{

int arr[10] = {1,20,13,4,5,6,10,28,19,15};

return std::count\_if(arr, arr + sizeof(arr)/sizeof(int) ,

std::bind(&divisible, \_1, 5));

}

Here, std::bind(&divisible, \_1, 5) Will return a new Function Object that takes only One Argument and checks if it is divisible by 5 or not.

### What std::bind returns ?

std::bind returns a function object. In above examples we have either save this new function object in auto variable or used it directly. But we can also store them using std::function Function object i.e.

### std::function<bool (int) > mod\_add\_funcObj = std::bind(&add, 20, \_1);

### Class template std::function is a general-purpose polymorphic function wrapper. Instances of std::function can store, copy, and invoke any [*Callable*](https://en.cppreference.com/w/cpp/named_req/Callable) *target* -- functions, [lambda expressions](https://en.cppreference.com/w/cpp/language/lambda), [bind expressions](https://en.cppreference.com/w/cpp/utility/functional/bind), or other function objects, as well as pointers to member functions and pointers to data members.

**/\***

**\* 5. foreach**

**\*/**

// C++ 03:

for (vector<int>::iterator itr = v.begin(); itr!=v.end(); ++ itr)

cout << (\*itr);

// C++ 11:

for (auto i: v) { // works on any class that has begin() and end()

cout << i ; // readonly access

}

for (auto& i: v) {

i++;} // changes the values in v and avoids copy

construction

auto x = begin(v); // Same as: int x = v.begin();

int arr[4] = {3, 2, 4, 5};

auto y = begin(arr); // y == 3

auto z = end(arr); // z == 5

// How this worked? Because begin() and end() are defined for array.

// Adapt your code to third party library by defining begin() and end()

// for their containers.

**/\***

**\* 5. nullptr**

**\***

**\* To replace NULL in C++ 03**

**\*/**

void foo(int i) { cout << "foo\_int" << endl; }

void foo(char\* pc) { cout << "foo\_char\*" << endl; }

int main() {

foo(NULL); // Ambiguity

// C++ 11

foo(nullptr); // call foo(char\*)

}

**/\***

**\* 6. enum class**

**\*/**

// C++ 03

enum apple {green\_a, red\_a};

enum orange {big\_o, small\_o};

apple a = green\_a;

orange o = big\_o;

if (a == o)

cout << "green apple and big orange are the same\n";

else

cout << "green apple and big orange are not the same\n";

// C++ 11

enum class apple {green, red};

enum class orange {big, small};

apple a = apple::green;

orange o = orange::big;

if (a == o)

cout << "green apple and big orange are the same\n";

else

cout << "green apple and big orange are not the same\n";

// Compile fails because we haven't define ==(apple, orange)

**/\***

**\* 7. static\_assert**

**\*/**

// run-time assert

assert( myPointer != NULL );

// Compile time assert (C++ 11)

static\_assert( sizeof(int) == 4 );

**/\***

**\* 8. Delegating Constructor**

**\*/**

class Dog {

public:

Dog() { ... }

Dog(int a) { Dog(); doOtherThings(a); }

};

// C++ 03:

class Dog {

init() { ... };

public:

Dog() { init(); }

Dog(int a) { init(); doOtherThings(); }

};

/\* Cons:

\* 1. Cumbersome code.

\* 2. init() could be invoked by other functions.

\*/

// C++ 11:

class Dog {

int age = 9;

public:

Dog() { ... }

Dog(int a) : Dog() { doOtherThings(); }

};

// Limitation: Dog() must be called first.

**/\***

**\* 9. override (for virtual function)**

**\***

**\* To avoid inadvertently create new function in derived classes.**

**\*/**

// C++ 03

class Dog {

virtual void A(int);

virtual void B() const;

}

class Yellowdog : public Dog {

virtual void A(float); // Created a new function

virtual void B(); // Created a new function

}

// C++ 11

class Dog {

virtual void A(int);

virtual void B() const;

void C();

}

class Yellowdog : public Dog {

virtual void A(float) override; // Error: no function to override

virtual void B() override; // Error: no function to override

void C() override; // Error: not a virtual function

}

**/\***

**\* 10. final (for virtual function and for class)**

**\*/**

class Dog final { // no class can be derived from Dog

...

};

class Dog {

virtual void bark() final; // No class can override bark()

};

**/\***

**\* 11. Compiler Generated Default Constructor**

**\*/**

class Dog {

Dog(int age) {}

};

Dog d1; // Error: compiler will not generate the default constructor

// C++ 11:

class Dog {

Dog(int age);

Dog() = default; // Force compiler to generate the default constructor

};

**/\***

**\* 12. delete**

**\*/**

class Dog {

Dog(int age) {}

}

Dog a(2);

Dog b(3.0); // 3.0 is converted from double to int

a = b; // Compiler generated assignment operator

// C++ 11:

class Dog {

Dog(int age) {}

Dog(double ) = delete;

Dog& operator=(const Dog&) = delete;

}

**/\***

**\* 13. constexpr**

**\*/**

int arr[6]; //OK

int A() { return 3; }

int arr[A()+3]; // Compile Error

// C++ 11

constexpr int A() { return 3; } // Forces the computation to happen

// at compile time.

int arr[A()+3]; // Create an array of size 6

// Write faster program with constexpr

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

int y = cubed(1789); // computed at compile time

//Function cubed() is:

//1. Super fast. It will not consume run-time cycles

//2. Super small. It will not occupy space in binary.

**/\***

**\* 14. New String Literals**

**\*/**

// C++ 03:

char\* a = "string";

// C++ 11:

char\* a = u8"string"; // to define an UTF-8 string.

char16\_t\* b = u"string"; // to define an UTF-16 string.

char32\_t\* c = U"string"; // to define an UTF-32 string.

char\* d = R"string \\" // to define raw string.

**/\***

**\* 15. lambda function**

**\*/**

cout << [](int x, int y){return x+y}(3,4) << endl; // Output: 7

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

cout << f(3,4) << endl; // Output: 7

template<typename func>

void filter(func f, vector<int> arr) {

for (auto i: arr) {

if (f(i))

cout << i << " ";

}

}

int main() {

vector<int> v = {1, 2, 3, 4, 5, 6 };

filter([](int x) {return (x>3);}, v); // Output: 4 5 6

...

filter([](int x) {return (x>2 && x<5);}, v); // Output: 3 4

int y = 4;

filter([&](int x) {return (x>y);}, v); // Output: 5 6

//Note: [&] tells compiler that we want variable capture

}

This article we will discuss, what is a C++11 Lambda Functions,  how to use Lambda Functions as Callbacks.

## What is a Lambda Function?

Lambda functions are a kind of anonymous functions in C++. These are mainly used as callbacks in C++. Lambda function is like a normal function i.e.

* You can pass arguments to it
* It can return the result

But it doesn’t have any name. It’s mainly used when we create very small functions to pass as a callback to another API.

[](int x) {

        std::cout<<x<<" ";

    }

Here,

* [] is used to pass the outer scope elements
* (int x) shows argument x is passed

## How to pass outer scope elements inside lambda functions

### Case 1: Using [=]

     [=](int &x) {

        // All outer scope elements have been passed by value }

### Case 2: Using [&]

    [&](int &x) {

        // All outer scope elements has been passed by reference

    }

Example:

int main() {

    int arr[] = { 1, 2, 3, 4, 5 };

    int mul = 5;

    std::for\_each(arr, arr + sizeof(arr) / sizeof(int), [&](int x) {

        std::cout<<x<<" ";

        // Can modify the mul inside this lambda function because

        // all outer scope elements have write access here.

            mul = 3;

        });

    std::cout << std::endl;

    std::for\_each(arr, arr + sizeof(arr) / sizeof(int), [=](int &x) {

        x= x\*mul;

        // Cannot modify the mul inside this lambda function because

        // all outer scope elements have read only access here.

        // mul = 9;

        });

    std::cout << std::endl;

    std::for\_each(arr, arr + sizeof(arr) / sizeof(int), [](int x) {

        // No access to mul inside this lambda function because

        // all outer scope elements are not visible here.

        //std::cout<<mul<<" ";

        });

    std::cout << std::endl;

}

**To capture the local variables by value, specify their name in capture list i.e.**

std::string msg = "Hello";

int counter = 10;

// Defining Lambda function and

// Capturing Local variables by Value

auto func = [msg, counter] () {

//...

};

Now, the variables specified in capture list will be copied inside lambda by value. Inside lambda they can be accessed but cannot be changed, because they are const. To modify the we need to add mutable keyword i.e.

auto func = [msg, counter] () mutable { };

**To capture the local variables by reference, specify their name in capture list with prefix & i.e.**

// Local Variables

std::string msg = "Hello";

int counter = 10;

// Defining Lambda function and Capturing Local variables by Reference

auto func = [&msg, &counter] () {

//...

};

**How to capture member variables from outer scope.**

Suppose we have a class **OddCounter** to keep the track of odd numbers read by the object. Inside its member function we are creating a lambda function and passing to a STL algorithm std::for\_each. Now this lambda function needs to capture member variable mCounter. How to do it?

If we try to capture member variable directly by value or reference, then it will **not work.**

class OddCounter{

// tracks the count of odd numbers encountered

int mCounter = 0;

public:

int getCount(){

return mCounter;

}

void update(std::vector<int> & vec){

// Capturing member variable by value will not work

// Will result in Compile Error

std::for\_each(vec.begin(), vec.end(), [mCounter](int element){

if(element % 2)

mCounter++; // Accessing member variable from outer scope

});

}

};

To capture the member variables inside lambda function, capture the “**this**” pointer by value.

std::for\_each(vec.begin(), vec.end(), [this](int element){

if(element % 2)

mCounter++; // Accessing member variable from outer scope

});

}

int main()

{

std::vector<int> vec = {12,3,2,1,8,9,0,2,3,9,7};

OddCounter counterObj;

//Passing the vector to OddCounter object

counterObj.update(vec);

int count = counterObj.getCount();

}

**/\***

**\* 17. User defined Literals**

**\*/**

// C++ went a long way to make user defined types (classes) to behave same as buildin types. User defined literals pushes this effort even further

//Old C++:

long double height = 3.4;

// Remember in high school physics class?

height = 3.4cm;

ratio = 3.4cm / 2.1mm;

//Why we don't do that anymore?

// 1. No language support

// 2. Run time cost associated with the unit translation

// C++ 11:

long double operator"" \_cm(long double x) { return x \* 10; }

long double operator"" \_m(long double x) { return x \* 1000; }

long double operator"" \_mm(long double x) { return x; }

int main() {

long double height = 3.4\_cm;

cout << height << endl; // 34

cout << (height + 13.0\_m) << endl; // 13034

cout << (130.0\_mm / 13.0\_m) << endl; // 0.01

}

//Note: add constexpr to make the translation happen in compile time.

// Restriction: it can only work with following paramters:

char const\*

unsigned long long

long double

char const\*, std::size\_t

wchar\_t const\*, std::size\_t

char16\_t const\*, std::size\_t

char32\_t const\*, std::size\_t

// Note: return value can be of any types.

// Example:

int operator"" \_hex(char const\* str, size\_t l) {

// Convert hexdecimal formated str to integer ret

return ret;

}

int operator"" \_oct(char const\* str, size\_t l) {

// Convert octal formated str to integer ret

return ret;

}

int main() {

cout << "FF"\_hex << endl; // 255

cout << "40"\_oct << endl; // 32

}

**/\***

**\* Variadic Template**

**\***

**\* A template that can take any number of template arguments of any type.**

**\* Both class and function templates can be variadic.**

**\*/**

template<typename... arg>

class BoTemplate;

BoTemplate<float> t1;

BoTemplate<int, long, double, float> t2;

BoTemplate<int, std::vector<double>> t3;

BoTemplate<> t4;

// Combination of variadic and non-variadic argument

template<typename T, typename... arg>

class BoTemplate;

BoTemplate<> t4; // Error

BoTemplate<int, long, double, float> t2; // OK

// Define a default template argument

template<typename T = int, typename... arg>

class BoTemplate;

**/\***

**\* Template Alias**

**\*/**

template<class T> class Dog { /\* ... \*/ };

template<class T>

using DogVec = std::vector<T, Dog<T>>;

DogVec<int> v; // Same as: std::vector<int, Dog<int>>

**/\***

**\* decltype**

**\***

**\* It is equivalent of GNU typeof**

**\*/**

const int& foo(); // Declare a function foo()

decltype(foo()) x1; // type is const int&

struct S { double x; };

decltype(S::x) x2; // x2 is double

auto s = make\_shared<S>();

decltype(s->x) x3; // x3 is double

int i;

decltype(i) x4; // x4 is int

float f;

decltype(i + f) x5; // x5 is float

// decltype turns out to be very useful for template generic programming

template<type X, type Y>

void foo(X x, Y y) {

...

decltype(x+y) z;

...

}

// How about return type needs to use decltype?

template<type X, type Y>

decltype(x+y) goo(X x, Y y) { // Error: x & y are undefined

return x + y;

}

// Combining auto and decltype to implement templates with trailing return type

template<type X, type Y>

auto goo(X x, Y y) -> decltype(x+y) {

return x + y;

}

**Shared Pointer**



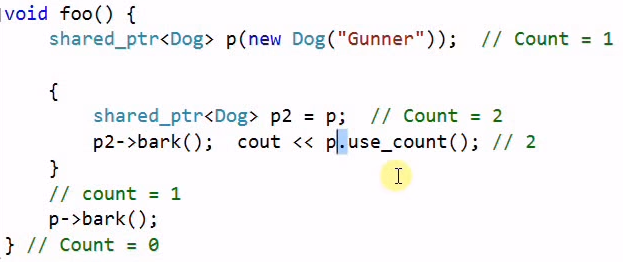
/\*\*\*\*\*\*\*\*\*\*\* Shared\_ptr \*\*\*\*\*\*\*\*\*\*\*/

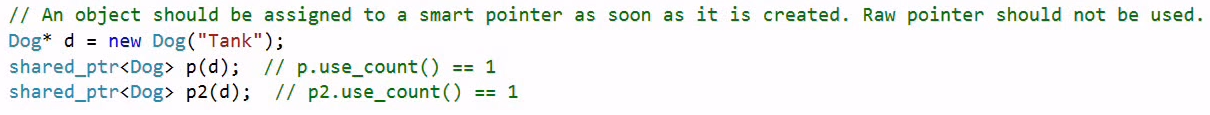
// 1. When a pointer outlives pointee: danling pointer

// 2. When a pointee outlives all its pointers: resource leak

//

// Smart Pointers: Make sure the lifetime of a pointer and pointee match.



**foo{**

**}**

When control goes out of foo() “d” is deleted twice and it is disastrous.

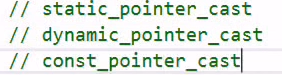
shared\_ptr<Dog> pD = make\_shared<Dog>(new Dog("Gunner")); // faster and safer

In compare to shared\_ptr<Dog> pD(new Dog("Gunner")); above one faster and safer:

Why Faster? In first approach 1. Gunner is created then 2. smart pointer pD is created which manage the gunner. In other part make\_shared combine both steps.

Why Safer? Exception free code. Suppose Gunner is created but problem with smart pointer creation then who will delete pD.

We can use smart pointer like raw pointer: pD->bark(); or \*(pD).bark(); Also we can perform casting operation using below cast:



**Void foo(){**

shared\_ptr<Dog> pD1 = make\_shared<Dog>(new Dog("Gunner"));

shared\_ptr<Dog> pD2 = make\_shared<Dog>(new Dog("Rockey"));

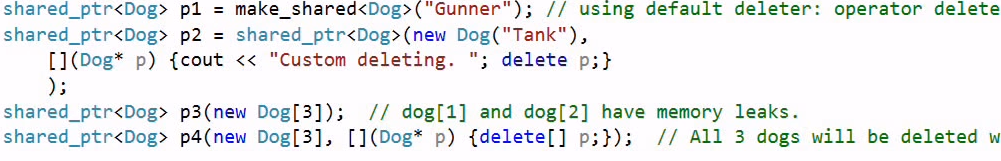
pD1=pD2; //Gunner Deleted

pD1.reset(); //Gunner Deleted

pD1=nullptr; //Gunner Deleted

**}**

Custom delete we can’t used with make\_shared, it always uses default delete operator. Shared pointer offers custom deleter using lambda function as we can passed it second parameter.



**Void foo(){**

shared\_ptr<Dog> pd = make\_shared<Dog>(new Dog("Gunner"));

Dog \*dog= pd.get(); //get the raw pointer.

Delete dog; //We can also delete it.

………

} What happed when pd goes out of scope of foo()? It again deleted by shared\_pointer which is already deleted – **Crashed or undefined behaviour**.

**Note: usually avoid playing with raw pointer when using smart\_pointer interface.**

// Don't use shared pointer for object on stack.

auto pD2 = make\_shared<Dog>( Dog("Smokey") );

**Problem of cyclic reference:**

Main advantage of shared\_ptr is that it automatically releases the associated memory when not used any more. But if we don’t use shared\_ptr carefully then this advantage can turn into a disadvantage.

class Dog {

shared\_ptr<Dog> m\_pFriend;

string m\_name;

public:

Dog(string name) { cout << "Dog is created: " << name.c\_str() << endl;

m\_name = name;

}

~Dog() { cout << "dog is destroyed: " << m\_name.c\_str() << endl; }

void makeFriend(shared\_ptr<Dog> f) { m\_pFriend = f; }

void bark() { cout << "Dog " << m\_name.c\_str() << " rules!" << endl; }

};

int main()

{

shared\_ptr<Dog> pD1(new Dog("Gunner"));

shared\_ptr<Dog> pD2(new Dog("Smokey"));

pD1->makeFriend(pD2);

pD2->makeFriend(pD1);

} //Memory Leak, no destructor called.

Reason of this problem with shared\_ptr is **cyclic references** i.e.

If two objects refer to each other using shared\_ptrs, then no one will delete the internal memory when they go out of scope.

It happens because shared\_ptr in its destructor after decrementing the reference count of associated memory checks if count is 0 then it deletes that memory and if it’s greater than 1 then it means that any other shared\_ptr is using this memory.

But in this kind of scenario these shared\_ptrs will always found count greater than 0 in destructor.

In above example both pD1 and pD2 having m\_pFriend with reference count 2. When it goes for destruction both decrement ref by 1 but not 0 because one is having ref of another. In such case none of them are deleted.

**Now How to fix this problem?**

Answer is using **weak\_ptr.** Weak\_ptr allows sharing but not owning an object. It only allows sharing, but deletion is not his business. Its object is created by a shared\_ptr.

    std::shared\_ptr<int> ptr = std::make\_shared<int>(4);

    std::weak\_ptr<int> weakPtr(ptr); weak\_ptr<int>

With weak\_ptr object we cannot directly use operators \* and -> to access the associated memory. First we have to create a shared\_ptr through weak\_ptr object by calling its lock() function,  then only we can use it.

int main()

{

    std::shared\_ptr<int> ptr = std::make\_shared<int>(4);

    std::weak\_ptr<int> weakPtr(ptr);

    std::shared\_ptr<int> ptr\_2 =  weakPtr.lock();

    if(ptr\_2)

        std::cout<<(\*ptr\_2)<<std::endl;

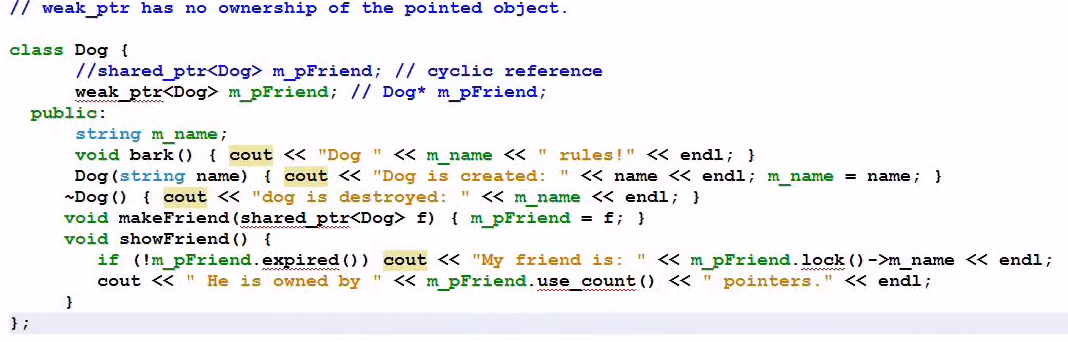
std::cout<<"Reference Count :: "<<ptr\_2.use\_count()<<std::endl;

if(weakPtr.expired() == false)

        std::cout<<"Not expired yet"<<std::endl;

}

**Important Point:** lock() can return empty shared\_ptr if that shared\_ptr us already deleted.



# **unique\_ptr**

unique\_ptr<> is one of the Smart pointer implementations provided by c++11 to prevent memory leaks. A unique\_ptr object wraps around a raw pointer and its responsible for its lifetime. When this object is destructed then in its destructor it deletes the associated raw pointer.

unique\_ptr has its -> and \* operator overloaded, so it can be used like normal pointer.

So, even if function is exited normally or abnormally (due to some exception), destructor of taskPtr will always be called. Hence, raw pointer will always get deleted and prevent the memory leak.

# **Unique Ownership of unique pointer**

A unique\_ptr object is always the unique owner of associated raw pointer. We can not copy a unique\_ptr object, its only movable.

As each unique\_ptr object is sole owner of a raw pointer, therefore in its destructor it directly deletes the associated pointer. There is no need of any reference counting, therefore its very light.

We can create an empty pointer like : unique\_ptr<int> object

here are two ways to check if a unique\_ptr<> object is empty or it has a raw pointer associated with it i.e.

// Method-1

if(!ptr1)

std::cout<<"ptr1 is empty"<<std::endl;

// Method 2

if(ptr1 == nullptr)

std::cout<<"ptr1 is empty"<<std::endl;

### Reseting a unique\_ptr

Calling reset() function on a unique\_ptr<> object will reset it i.e. it will delete the associated raw pointer and make unique\_ptr<> object empty i.e.

// Reseting the unique\_ptr will delete the associated

// raw pointer and make unique\_ptr object empty

ptr.reset();

### unique\_ptr object is not copyable

// 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

taskPtr = taskPtr2; // compile error

Note: Both copy constructor and assignment operator are deleted in unique\_ptr<> class.

### Transfering the ownership of unique\_ptr object

A unique\_ptr object can transfer the owner ship of associated raw pointer to another unique\_ptr object.

std::unique\_ptr<Task> taskPtr2(new Task(55));

{ // Transfer the ownership

std::unique\_ptr<Task> taskPtr4 = std::move(taskPtr2);

if(taskPtr2 == nullptr)

std::cout<<"taskPtr2 is  empty"<<std::endl;

//taskPtr4 goes out of scope and deletes the associated raw pointer

}

std::move() will convert the taskPtr2 to a RValue Reference. So that move constructor of unique\_ptr is invoked and associated raw pointer can be transferred to taskPtr4.

taskPtr2 will be empty after transferring the ownership of its raw pointer to taskPtr4.

### 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;

Unordered\_map

Unordered\_map provides a functionality of map i.e. it stores the elements in key value pair and with unique key only.

Unordered\_map internally uses the hashing to achieve this. It internally uses a hash table to implement this hashing feature. In an unordered\_map elements are stored in a key value pair combination. But elements are stored in arbitrary order unlike associative containers where elements were stored in sorted order of keys.

string str = "Rajeev Kumar Sharma";

map<char, int> mp;

unordered\_map<char, int> um;

for (char c : str) mp[c]++; //Count number of char in str.

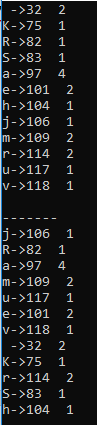
for (char c : str) um[c]++; //Count number of char in str.

for (auto &iter : mp) cout << iter.first <<"->" << charToShort(iter.first)

<< " " << iter.second << endl;

for (auto &iter : um) cout << iter.first <<"->" <<charToShort(iter.first)

<< " " << iter.second << endl;



Here we can clearly see the oput of Map in sorted order while

Unordered\_map in random order.

| map | unordered\_map

---------------------------------------------------------

Ordering | increasing order | no ordering

| (by default) |

Implementation | Self balancing BST | Hash Table

| like R & B tree |

search time | log(n) | O(1) -> Average

| | O(n) -> Worst Case

Insertion time | log(n) + Rebalance | Same as search

Deletion time | log(n) + Rebalance | Same as search

## How unordered\_map store elements?

Whenever we try to insert an element in a unordered\_map, it internally does the following steps:

* First hash of key is calculated using Hasher function and then based on that hash an appropriate bucket is choose.
* Once bucket is identified then it compares the key with key of each element inside the bucket using Comparator function to identify if given element is a duplicate or not.
* If it’s not a duplicate, then only it stores the element in that bucket.

Insert, find and delete into unordered\_map

//Diffrence between operator[] and insert method.

//operator[] perform insert & update operation based on key

//While MAP::insert() perform only insert. Refuse to insert or

//update when key available in container.

typedef map<char, int> CharMapType;

map<char, int> charMap;

pair<map<char, int>::iterator, bool> isInsert;

charMap['A'] = 65; //Key Not found, insert new node

charMap['B'] = 66; //Key Not found, insert new node

charMap['C'] = 67; //Key Not found, insert new node

//charMap['A'] = 100; //Key found, Update the value only.

isInsert = charMap.insert({ 'A',100 }); //Refuse to update value.

//OR, isInsert = charMap.insert(CharMapType::value\_type({ 'A',100 }));

**Like map, unordered\_map has same behavior to insert, find and erase.**

typedef std::unordered\_map<std::string, int>::iterator UOMIterator;

// Pair of Map Iterator and bool value

std::pair< UOMIterator, bool> result;

// Inserting an element through pair

result = wordMap.insert(std::make\_pair<std::string, int>("Second", 6));

//result=wordMap.insert({“Second”,6});

if (result.second == false)

cout<<”Element not inserted\n”

unordered\_map<char,int>::iterator it=um.find('R');

if (it != um.end()) {

cout << "Element found\n";

}

int &x = charMap['A']; //If Key ‘A’ is available in map, return its

x = 100; //value reference, we can also change its value.

//If key not found then create

**Erasing via key and iterator.**

*// Initialize an unordered\_map through initializer\_list*

std::unordered\_map<std::**string**, **int**> wordMap( { { "First", 1 }, { "Second",

2 }, { "Third", 3 }, { "Fourth", 4 }, { "Fifth", 5 } });

*//Erase element by key*

**if** (wordMap.erase("Second") == 1)

std::**cout** << "Element Deleted" << std::endl;

*// Iterator pointing to first element of unordered\_map*

std::unordered\_map<std::**string**, **int**>::iterator it = wordMap.find("Fourth");

*// Erase the element pointed by iterator it*

**if** (it != wordMap.**end**())

wordMap.erase(it);

**Erase while iterating**

// Erase all element whose key starts with letter 'F' in an iteration

while (it != wordMap.end()) {

// Check if key's first character is F

if (it->first[0] == 'F') {

// erase() function returns the iterator of the next

// to last deleted element.

it=wordMap.erase(it); // Or wordMap.erase(it++)

}

else

it++;

}

**Map vs unordered\_map | When to choose one over another ?**

Both std::map & std::unordered\_map store elements in key value pair & provide member functions to efficiently insert, search & delete key value pairs, but they are different in following areas:

* **Internal Implementation:**
* **Memory Usage:** Memory usage is more in unordered\_map as compared to map because unordered\_map need space for storing hash table too.
* **Time Complexity:** Time complexity for searching elements in **std::map** is

O(log n). Even in worst case it will be O(log n) because elements are stored internally as Balanced Binary Search tree (BST).

Whereas, in **std::unordered\_map** best case time complexity for searching is O(1).

If hash code function is not good then, worst case complexity can be O(n) (In case

all keys are in same bucket).

* **Using user defined objects as keys:**

For std::map to use user defined object as keys, we need to override either < operator or pass external comparator i.e. a functor or function pointer that can be used by map for comparing keys. Whereas, For **std::unordered\_map** we need to provide definition of function std::hash<K> for our key type K. Also, we need to override == operator.

**std::map** Internally store elements in a balanced BST. Therefore, elements will be stored in sorted order of keys.

**std::unordered\_map** store elements using hash table. Therefore, elements will not be stored in any sorted order. They will be stored in arbitrary order.

**When to choose map instead of unordered\_map**

* **When we need Low Memory:** Unordered\_map consumes extra memory for internal hashing, so if you are keeping millions and billions of data inside the map and want to consume less memory then choose std::map instead of std::unordered\_map.
* **When we are interested in Ordering too:**
* **When you need guaranteed Performance:** For searching an element, unordered\_map gives the complexity O(1) in best case but O(n) in worst case (if hash implementation is not perfect).

**When to choose unordered\_map instead of map**

* **When we have good hasher and no memory limitation:** Unordered\_map consumes extra memory for internal hashing. But to due to this searching complexity is O(1), if hasher function is good.

Using unordered\_set with custom hasher and comparison function