/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*C++11 Features

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Contents

[01 Learn C++ 11 in 20 Minutes - Part I and Part II 3](#_Toc392496601)

[a. Initializer List 3](#_Toc392496602)

[b. initializer\_list Constructor 4](#_Toc392496603)

[c. Uniform Initialization 4](#_Toc392496604)

[d. Uniform Initialization Search Order 5](#_Toc392496605)

[e. auto Type 6](#_Toc392496606)

[f. Ranged based for 6](#_Toc392496607)

[g. nullptr 7](#_Toc392496608)

[h. enum class 7](#_Toc392496609)

[i. static\_assert 8](#_Toc392496610)

[j. Delegating Constructor 8](#_Toc392496611)

[k. Override for Virtual Functions 10](#_Toc392496612)

[l. final 11](#_Toc392496613)

[m. Compiler Generated Default Constructor 12](#_Toc392496614)

[n. delete 13](#_Toc392496615)

[o. constexpr 14](#_Toc392496616)

[p. New String Literals 14](#_Toc392496617)

[q. lambda function 15](#_Toc392496618)

[02 Learn C++ 11 in 20 Minutes - Part II 15](#_Toc392496619)

[03 C++ 11 Rvalue Reference - Move Semantics 16](#_Toc392496620)

[04 C++ 11 Rvalue Reference - Perfect Fowarding 19](#_Toc392496621)

[05 C++ 11 User Defined Literals 22](#_Toc392496622)

[06 C++ 11 Compiler Generated Functions 25](#_Toc392496623)

[07 C++ 11 Library Shared Pointer – I 28](#_Toc392496624)

[08 C++ 11 Library Shared Pointer – II 31](#_Toc392496625)

[09 C++ 11 Library Weak Pointers 32](#_Toc392496626)

[10 C++ 11 Library Unique Pointers 34](#_Toc392496627)

[11 C++ 11 Resource Managing Class 37](#_Toc392496628)

[12 C++ 11 Library Regular Expression 1 40](#_Toc392496629)

[13 C++ 11 Library Regular Expression 2 – Submatch – Not there 41](#_Toc392496630)

[14 C++ 11 Library Regular Expression 3 – Iterators – Not there 41](#_Toc392496631)

[15 C++ 11 Library Clocks and Timers - Part I 42](#_Toc392496632)

[16 C++ 11 Library Clocks and Timers - Part II 43](#_Toc392496633)

[17 C++ 11 Library Random Number Engine 44](#_Toc392496634)

[18 C++ 11 Library Random Number Distribution 46](#_Toc392496635)

[19 C++ 11 Library Tuple 47](#_Toc392496636)

[20 C++ 11 Library When to Use Tuple 49](#_Toc392496637)

# 01 Learn C++ 11 in 20 Minutes - Part I and Part II

## Initializer List

#pragma region **Initializer List**

int main(void)

{

// In C++03, { 3, 2, 4, 5 } is called initializer list

int arr[4] = { 3, 2, 4, 5 };

// If we wanted to initialize a vector we have to use

// push\_bach() API of vector.

// Example:

vector<int> vCPP03;

vCPP03.push\_back(3);

vCPP03.push\_back(2);

vCPP03.push\_back(4);

vCPP03.push\_back(5);

// C++11 Extends the initializer list support to vector and

// other classes

// Calling initializer\_list constructor and initializing vector

vector<int> vCPP11 = { 3, 2, 4, 5 };

/\* All the relevant STL containers have been updated to accept

initializer\_list!!! \*/

return 0;

}

#pragma endregion **Initializer** **List**

## initializer\_list Constructor

#pragma region **initializer\_list Constructor**

/\* Defining our own initializer\_list constructor for our class\*/

class boVector

{

vector<int> vec\_;

public:

boVector(const initializer\_list<int>& v)

{

for (auto itr = v.begin();

itr != v.end();

itr++)

{

vec\_.push\_back(\*itr);

}

}

};

int main(void)

{

// boVector constructor taking initializer\_list as argument!!!

boVector v = { 0, 2, 4, 6 };

boVector av{ 0, 2, 4, 6 };

return 0;

}

#pragma endregion **initializer\_list Constructor**

## Uniform Initialization

#pragma region **Uniform Initialization**

// C++03

class dog

{

public:

int age;

string name;

};

dog d1 = { 5, "Bruce" }; // Aggregate Initialization

// C++11

class dogCPP11

{

public:

dogCPP11(int age, string name){ }

};

int main(void)

{

dogCPP11 d2 = { 5, "A Dog" };

return 0;

}

#pragma endregion **Uniform Initialization**

## Uniform Initialization Search Order

#pragma region **Uniform Initialization Search Order**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Uniform Initialization Search Order:

1. initializer\_list constructor.

2. Regular constructor that takes the appropriate parameters.

3. Aggregate initialization.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class dog

{

public:

int age\_;

dog(int a)

{

cout << "Inside dog(int a)" << endl;

age\_ = a;

}

dog(const initializer\_list<int>& vec) // 3 is taken as single item array

{

cout << "Inside dog(const initializer\_list<int>& vec)" << endl;

age\_ = \*(vec.begin());

}

};

int main(void)

{

// Uniform Initialization will search for

// initializer\_list constructor first and it will construct.

// If initializer\_list is not found, it will search for

// constructor that will take single integer as parameter.

// If that is also not found, it will try to treat dog as

// aggregate class and will call aggregate initializer.

dog{ 3 };

return 0;

}

#pragma endregion **Uniform Initialization Search Order**

## auto Type

#pragma region **auto Type**

int main(void)

{

vector<int> vec = { 2, 3, 4, 5 };

// In C++03, to traverse a vector, we would have done...

for (vector<int>::iterator itr = vec.begin();

itr != vec.end();

itr++)

{

cout << \*itr << " ";

}

cout << endl;

// C++11, we can use auto

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

cout << \*itr << " ";

cout << endl;

return 0;

}

#pragma endregion **auto Type**

## Ranged based for

#pragma region **Ranged based for**

int main(void)

{

vector<int> vec = { 2, 3, 4, 5 };

// In C++03, to traverse a vector, we would have done...

for (vector<int>::iterator itr = vec.begin();

itr != vec.end();

itr++)

{

cout << \*itr << " ";

}

cout << endl;

// C++11, we can use range based for

for (auto i : vec) // Works on any class of which has begin() and end()

cout << i << " ";// read only access

cout << endl;

for (auto& i : vec)

cout << ++i << " ";

cout << endl;

return 0;

}

#pragma endregion **Ranged based for**

## nullptr

#pragma region **nullptr**

// To replace NULL in C++03

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

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

int main(void)

{

// This will call foo\_int

foo(NULL);

// In C++11 it will call foo\_char\*

foo(nullptr);

return 0;

}

#pragma endregion **nullptr**

## enum class

#pragma region **enum** **class**

int main(void)

{

{

// C++03 enumerators are basically integers

enum apple { green\_a, red\_a };

enum orange { big\_o, small\_o };

apple a = green\_a;

orange o = big\_o;

if (a == o) // We can compare APPLE with ORANGE

cout << "Green APPLE and big ORANGE are the same" << endl;

else

cout << "Green APPLE and big ORANGE are not same" << endl;

}

{

// C++11 introduces enumerator class

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

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

apple a = apple::green\_a;

orange o = orange::big\_o;

// Compilation fails here cos, APPLE is different from ORANGE

// or else we have to define operator==

//if (a == o) // We can compare APPLE with ORANGE

//cout << "Green APPLE and big ORANGE are the same" << endl;

//else

//cout << "Green APPLE and big ORANGE are not same" << endl;

}

return 0;

}

#pragma endregion **enum** **class**

## static\_assert

#pragma region **static\_assert**

int main(void)

{

int\* myPointer = NULL;

// Run-time assert. C++03

assert(myPointer != NULL);

// C++11 provides compile time assert

// This code will not compile if the sizeof(int) is not 4

static\_assert(sizeof(int) == 4, "Message");

return 0;

}

#pragma endregion **static\_assert**

## Delegating Constructor

#pragma region **Delegating** **Constructor**

// This will only work in JAVA

//class dog

//{

//public:

// dog(){}

// dog(int a)

// {

// dog(); // Reuse the code of 1st constructor and do something else

// cout << "Inside dog(int a) " << endl;

// }

//};

// To achieve the same in C++03 is like below

class dogCPP03

{

void init() { cout << "Inside init()" << endl; }

public:

dogCPP03() { init(); }

dogCPP03(int a)

{

init();

cout << "Inside dogCPP03(int a)" << endl;

}

/\* PROBLEM

1. Cumbersome code.

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

\*/

};

// C++11 provides a new way to share the code of constructor!!!

class dogCPP11

{

public:

dogCPP11()

{

cout << "Inside dogCPP11()" << endl;

cout << "My age is: " << age\_ << endl;

}

// The first constructor can be called at the initializer list part

// of the second constructor. This will invoke the first constructor

// before the second constructor starts!!!

// Limitation is, first constructor can't be invoked in the middle

// or at the end of the second constructor.

dogCPP11(int a) : dogCPP11()

{

age\_ = a;

cout << "Inside dogCPP11(int a)" << endl;

cout << "My age is: " << age\_ << endl;

}

private:

// C++11 also supports "in class data member initialization"

// age\_ will be initialized to 9, at every constructor!!!

int age\_ = 9;

};

int main(void)

{

dogCPP11 a;

dogCPP11 b(10);

return 0;

}

#pragma endregion **Delegating** **Constructor**

## Override for Virtual Functions

#pragma region **Override** **for Virtual Functions**

// C++03

class dogCPP03

{

public:

virtual void A(int) {}

virtual void B() const {}

};

class yellowDog : public dogCPP03

{

public:

virtual void A(float) {} // C++03, In Polymorphism, A(float) will not be called

virtual void B(){}

};

// C++11

class dogCPP11

{

public:

virtual void A(int) { cout << "dogCPP11::void A(int) " << endl; }

virtual void B() const { cout << "dogCPP11::void B() const " << endl; }

void c() { cout << "dogCPP11::void c() " << endl; }

};

class yellowDogCPP11 : public dogCPP11

{

// ERRORS, compiler will through error

//virtual void A(float) override { cout << "yellowDogCPP11::void A(float) " << endl; }

//virtual void B() override { cout << "yellowDogCPP11::void B() " << endl; }

//void c() override { cout << "yellowDogCPP11::void c() " << endl; }

};

int main(void)

{

return 0;

}

#pragma endregion **Override for Virtual Functions**

## final

#pragma region **final**

class dog final

{

public:

dog() { cout << "dog::dog() " << endl; }

};

// Below code will through error since the base class dog is final!!!

//class yellowDog : public dog { };

class anotherDog

{

public:

anotherDog() { cout << "anotherDog::anotherDog() " << endl; }

virtual void bark(void) final

{

cout << "I am barking!!!" << endl;

}

};

class yellowDog : public anotherDog

{

public:

yellowDog() { cout << "yellowDog::yellowDog() " << endl; }

// Below code will give error as the bark() is final!!!

//void bark(void)

//{

// cout << "I am barking!!!" << endl;

//}

};

int main(void)

{

return 0;

}

#pragma endregion **final**

## Compiler Generated Default Constructor

#pragma region **Compiler Generated Default Constructor**

class dogCPP03

{

int age\_;

public:

dogCPP03(int a) : age\_(a) { cout << "dogCPP03::dogCPP03(int a) " << endl; }

};

class dogCPP11

{

int age\_;

public:

dogCPP11(int a) : age\_(a) { cout << "dogCPP11::dogCPP11(int a) " << endl; }

// This will force the compiler to generate the

// default constructor!!!

dogCPP11() = default;

};

int main(void)

{

// Below code will be ERROR, because we have already provided

// a constructor and the compiler will not generate any default

// constructor for us.

//dogCPP03 d1;

dogCPP11 d1; // This is OK

return 0;

}

#pragma endregion **Compiler** **Generated** **Default** **Constructor**

## delete

#pragma region **delete**

class dogCPP03

{

int age\_;

public:

dogCPP03(int a) : age\_(a) { cout << "dogCPP03::dogCPP03(int a) " << endl; }

};

class dogCPP11

{

int age\_;

public:

dogCPP11(int a) : age\_(a) { cout << "dogCPP11::dogCPP11(int a) " << endl; }

dogCPP11(double) = delete;

dogCPP11& operator=(const dogCPP11&) = delete;

};

int main(void)

{

// C++03

// This will work

dogCPP03 d1(2);

// This will also work since the double is converted to int

dogCPP03 d2(3.0);

dogCPP03 d3(9);

// This will also work, because compiler generates operator= for us

d3 = d2;

// This is OK

dogCPP11 d1\_11(9);

// This will not work, because we have deleted the function

//dogCPP11 d2\_11(3.0);

dogCPP11 d3\_11(9);

// This will not work, because we have deleted operator=

//d3\_11 = d1\_11;

return 0;

}

#pragma endregion **delete**

## constexpr

#pragma region **constexpr**

int return3(void) { return 3; }

constexpr int Aconstexpr(void) { return 3; }

int main(void)

{

int arr[6];

// This will not compile!!!

//int arr6[return3() + 3];

// C++11

// This will work and the compiler will compute the expression

// during compile time!!!

int arr6[Aconstexpr() + 3];

return 0;

}

#pragma endregion **constexpr**

## New String Literals

#pragma region **New String Literals**

int main(void)

{

// C++03

char\* a = "string";

// C++11

char\* aa = u8"string"; // to define UTF-8 string

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

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

char\* d = R"foo(string )foo";// to define raw string.

return 0;

}

#pragma endregion **New String Literals**

## lambda function

#pragma region **lambda function**

template<typename func>

void filter(func f, vector<int> arr)

{

for (auto i : arr)

{

if (f(i))

cout << i << " ";

}

cout << endl;

}

int main(void)

{

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

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

cout << f(3, 4) << endl;

vector<int> v = { 1, 2, 3, 4, 5, 6 };

   /\*\*\*\*\*\*\*\*\* LAMBDA \*\*\*\*\*\*\*\*\*/

filter([](int x) { return (x > 3); }, v);

filter([](int x) { return ((x > 2) && (x < 5)); }, v);

int y = 4;

filter([&](int x) { return (x > y); }, v);

return 0;

}

#pragma endregion **lambda function**

# 02 Learn C++ 11 in 20 Minutes - Part II

# 03 C++ 11 Rvalue Reference - Move Semantics

#include <iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Rvalue Reference - Move Semantics

1. Moving Semantics

2. Perfect Forwarding

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Introduction

void printInt(int& i) { cout << "lvalue reference " << i << endl; }

void printInt(int&& i){ cout << "rvalue reference " << i << endl; }

//void printInt(int i){} // Will not work with above overloads!!!

int main(void)

{

int a = 5; // a is lvalue

int& b = a; // b is lvalue reference

//int&& c; // c is rvalue reference

printInt(a); // Will call printInt(int& i)

printInt(6); // Will call printInt(int&& i)

return 0;

}

#pragma endregion Introduction

#pragma region The Problem

class boVector

{

int size\_;

double\* arr\_; // A big array

public:

boVector(const boVector& rhs) // Copy constructor

{

size\_ = rhs.size\_;

arr\_ = new double[size\_];

for (int i = 0; i < size\_; ++i)

{

arr\_[i] = rhs.arr\_[i];

}

}

~boVector(){ delete arr\_; }

};

void foo(boVector v);

boVector createBoVector(); // Creates a boVector

int main(void)

{

boVector resuable = createBoVector();

// This will invoke the costly copy constructor!!!

foo(resuable);

// createBoVector returns rvalue(temporary), what's the

// use of copying a temporary while passing it foo function

// since that temporary is going to be destroyed!!!

foo(createBoVector());

return 0;

}

#pragma endregion The Problem

#pragma region Move Constructor

class boVector

{

int size\_;

double\* arr\_; // A big array

public:

boVector(const boVector& rhs) // Copy constructor

{

size\_ = rhs.size\_;

arr\_ = new double[size\_];

for (int i = 0; i < size\_; ++i)

{

arr\_[i] = rhs.arr\_[i];

}

}

boVector(boVector&& rhs) // Move constructor

{

size\_ = rhs.size\_;

arr\_ = rhs.arr\_;

rhs.arr\_ = nullptr;

}

~boVector(){ delete arr\_; }

};

void foo(boVector v);

boVector createBoVector(); // Creates a boVector

int main(void)

{

boVector resuable = createBoVector();

// This will invoke the costly copy constructor!!!

foo(resuable);

// Calls Move constructor!!!

foo(createBoVector());

return 0;

}

#pragma endregion Move Constructor

#pragma region std::move

class boVector

{

int size\_;

double\* arr\_; // A big array

public:

boVector() = default;

boVector(const boVector& rhs) // Copy constructor

{

size\_ = rhs.size\_;

arr\_ = new double[size\_];

for (int i = 0; i < size\_; ++i)

{

arr\_[i] = rhs.arr\_[i];

}

}

boVector(boVector&& rhs) // Move constructor

{

size\_ = rhs.size\_;

arr\_ = rhs.arr\_;

rhs.arr\_ = nullptr;

}

~boVector(){ delete arr\_; }

};

void foo(boVector v){}

boVector createBoVector(){ boVector a; return a; } // Creates a boVector

void foo\_by\_reference(boVector& v){}

int main(void)

{

boVector reusable = createBoVector();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The situation that we are dealing now is,

once the foo(reusable) is called, reusable object

will be destroyed. since foo(reusable) calls

copy constructor and we know that it is costly.

Why to call a copy constructor on an object which

is going to be destroyed soon (reusable)?

Standard provides a special construct for

this kind of situation. It is std::move

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

foo(reusable);

 This will move the object to foo with the move

 constructor!!! We need to be very careful cos

 once we call the std::move, reusable.arr will

 be nullptr.

foo(std::move(reusable));

 Calls Move constructor!!!

foo(createBoVector());

/\*

foo\_by\_reference(reusable); - Calls no constructor

foo(std::move(reusable)); - Calls move constructor

foo(reusable); - Calls copy constructor

Efficiency:

foo\_by\_reference(reusable); - Most Efficient

foo(std::move(reusable)); - All most efficient as pass by reference

foo(reusable); - Most expensive!!!

\*/

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Note 1:

The most useful place for rvalue reference is overloading

copy constructor and copy assignment operator to achieve

move semantics.

X& X::operator=(X const& rhs); // Copy Assignment

X& X::operator=(X&& rhs); // Move assignment

Note 2:

Move semantics is implemented for all STL container. This means

1. Move to C++11, your code will be faster without changing a thing.

2. Passing-by-value can always be used for STL container.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion std::move

# 04 C++ 11 Rvalue Reference - Perfect Fowarding

#include <iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Perfect Forwarding

- 2nd usage of rvalue reference is "Perfect Forwarding"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Perfect Forwarding

class boVector

{

int size\_;

double\* arr\_; // A big array

public:

boVector() = default;

boVector(const boVector& rhs) // Copy constructor

{

size\_ = rhs.size\_;

arr\_ = new double[size\_];

for (int i = 0; i < size\_; ++i)

{

arr\_[i] = rhs.arr\_[i];

}

}

boVector(boVector&& rhs) // Move constructor

{

size\_ = rhs.size\_;

arr\_ = rhs.arr\_;

rhs.arr\_ = nullptr;

}

~boVector(){ delete arr\_; }

};

boVector createBoVector(){ boVector a; return a; } // Creates a boVector

void foo(boVector arg)

{

}

template<typename T>

// We have a function relay which take an argument and pass it

// to foo function. This is called argument forwarding!!!

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. No costly and unnecessary copy construction of boVector is made;

2. rvalue is forwarded as rvalue, lvalue is forwarded as lvalue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//void relay(T arg)

//{

// foo(arg);

//}

// To achieve perfect forwarding,

template<typename T>

void relay(T&& arg)

{

foo(std::forward<T>(arg));

}

int main(void)

{

boVector reusable = createBoVector();

relay(reusable); // lvalue is passed

relay(createBoVector()); // rvalue is passed

return 0;

}

#pragma endregion Perfect Forwarding

#pragma region Reference Collapsing Rules

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C++11 defines reference collapsing rules for

type deduction.

1. T& & ==> T& // We can't write a code like this, but compiler

// generates code like the above and deduce the type

2. T& && ==> T&

3. T&& & ==> T&

4. T&& && ==> T&&

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

return 0;

}

#pragma endregion Reference Collapsing Rules

#pragma region Remove Reference

template< typename T>

struct remove\_reference; // It removes reference on type T

// T is int&

remove\_reference<int&>::type i; // int i;

// T is int

remove\_reference<int>::type i; // int i;

#pragma endregion Remove Reference

#pragma region Universal Reference

template<typename T>

void relay(T&& arg)

{

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

rvalue reference is specified with type&&

type&& is rvalue reference?

Answer: No. The type of type&& depends on how the

arg is initialized.

If arg is initialized with rvalue, then T&& is rvalue reference

If arg is initialized with lvalue, then T&& is lvalue reference

How does this happen?

Example 1:

relay(9); => T = int&& => T&& = int&& = int&&

1. 9 rvalue is integer, then T is replaced with int&&.

2. Then T&& will become int&& &&.

3. Applying reference collapsing rule, int&& && will become int &&

Example 2:

int x = 9;

relay(x); => T = int& => T&& = int& && = int&

1. x is lvalue and is integer, then T is replaced with int&

2. Then T&& will become int& &&.

3. Applying reference collapsing rule, int& && will become int&

- T&& is universal reference, this means it can take

rvalue, lvalue, const, non-const etc...

- T&& will be universal reference if and only if the following

conditions are met:

1. T is a template type.

2. Type deduction (reference collapsing) happens to T.

- T is a function template type, not class template type.

In any other case, apart from the above, wherever we see T&&, it is

rvalue reference!!!.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Universal Reference

#pragma region Perfect Forwarding Problem

template< typename T>

 relay() function is taking universal reference of arg

void relay(T&& arg)

{

 Perform forward on arg

foo(std::forward<T>(arg));

}

 What does the forward() function does?

template< typename T>

T&& forward(typename remove\_reference<T>::type& \_Arg)

{

return static\_cast<T&&>(arg);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

std::move<T>(arg); // Turns arg into an rvalue type

std::forward<T>(arg); // Turns arg to type T&&

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Perfect Forwarding Problem

# 05 C++ 11 User Defined Literals

#include <iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

User Defined Literals

What are literals>

- Literals are constants

C++ has 4 kinds of literals...

1. Integer Literal - 4

2. Floating Point Literal - 4.5

3. Character Literal - 'z'

4. String Literal - "Hello"

Literals can have suffix to specify the exact type...

45; // int

45u; // unsigned int

45l; // long

\*\*\* User defined literals allows us to define our own suffix \*\*\*

But why do we want to do that?

Look at the examples...

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Problem

int main(void)

{

// C++99

// height is Meter? Centimeters? Inches? If it is a big

// project, we have to look 1000s of line to figure it out

// whether height is in meters or centimeters or in inches.

long double height = 3.4;

//Ideally, we would have done like this

// height = 3.4cm;

// ratio = 3.4cm / 2.1mm => ration = 3.4\*10/2.1

return 0;

}

#pragma endregion Problem

#pragma region Example 1

// User defined literals

// operator"" means, it is the declaration of USER DEFINED LITERALS

// \_cm, \_m and \_mm means the suffice. Both centimeter and meter will

// be translated to millimeter.

// Visual Studio 2013 doesn't support USER DEFINED LITERALS

// constexpr make the calculation to happen at compile time!!!

// constexpr is not supported by Visual Studio 2013

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

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

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

int main(void)

{

long double height = 3.4\_cm;

cout << height << endl; // Prints 34

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

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

return 0;

}

#pragma endregion Example 1

#pragma region Example 2

 String representation of binary number into integer!!!

int operator""\_bin(const char\*, size\_t l)

{

int ret = 0;

for (int i = 0; i < l; ++i)

{

ret = ret << 1;

if (str[i] == '1')

ret += 1;

}

return ret;

}

int main(void)

{

cout << "110"\_bin << endl; // 6

cout << "1100110"\_bin << endl; // 102

cout << "110100010001001110001"\_bin << endl;// 1712753

return 0;

}

#pragma endregion Example 2

#pragma region Restriction

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

User Defined Literals can work only with the

following parameters...

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

Return value can be of any type

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Restriction

# 06 C++ 11 Compiler Generated Functions

#include <iostream>

using namespace std;

#pragma region Compiler Generated Functions

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C++03

Compiler generates 4 functions for us

1. Default constructor (Generated only if no constructor is declared by user)

2. Copy constructor

3. Copy assignment operator

4. Destructor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C++11

C++11 compiler generates 6 functions for us

1. Default constructor (Generated only if no constructor is declared by user)

2. Copy constructor

3. Copy assignment operator

4. Destructor

5. Move constructor

6. Move assignment operator

C++11 has added more rules on how to generate these functions

Copy constructor:

- Copy constructor will not generated if we have declared our own

1. Move constructor

2. Move assignment operator

3. Copy assignment operator - (Deprecated rule of C++03)

4. Destructor - (Deprecated rule of C++03)

Copy assignment operator:

- Copy assignment operator will not generated if we have declared our own

1. Move constructor

2. Move assignment operator

3. Copy constructor - (Deprecated rule of C++03)

4. Destructor - (Deprecated rule of C++03)

Move constructor:

-  Move constructor will not generated if we have declared our own

1. Copy constructor

2. Copy assignment operator

3. Destructor

4. Move assignment operator

Move assignment operator:

-  Move assignment operator will not generated if we have declared our own

1. Copy constructor

2. Copy assignment operator

3. Destructor

4. Move constructor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class EmptyDog{};

class EquivalentEmptyDog

{

/\*\*\*\*\*\*\*\*\* C++03 \*\*\*\*\*\*\*\*\*/

// Default constructor

EquivalentEmptyDog();

// Copy constructor

EquivalentEmptyDog(const EquivalentEmptyDog& rhs);

// Copy assignment operator

EquivalentEmptyDog& operator=(const EquivalentEmptyDog& rhs);

// Destructor

~EquivalentEmptyDog();

/\*\*\*\*\*\*\*\*\* C++11 \*\*\*\*\*\*\*\*\*/

// Move constructor

EquivalentEmptyDog(EquivalentEmptyDog&& rhs);

// Move assignment operator

EquivalentEmptyDog& operator=(EquivalentEmptyDog&& rhs);

};

#pragma endregion Compiler Generated Functions

#pragma region Example 1

class Cat

{

// Copy constructor

Cat(const Cat& rhs){ }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What are the function generate for 'class Cat' if we have Copy

constructor as in the above example?

1. Default constructor - This will not be generated since we

already have a constructor (Copy constructor).

2. Copy assignment operator  - YES (Deprecated rule of C++03)

3. Destructor - YES

4. Move constructor - This will not be generated since we

have declared Copy constructor.

5. Move assignment operator - This will not be generated since we

have declared Copy constructor.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Cat& operator=(const Cat& rhs) = default;

};

#pragma endregion Example 1

#pragma region Example 2

class Duck

{

// Move constructor

Duck(Duck&& rhs) { }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What are the function generate for 'class Duck' if we have Move

constructor defined?

1. Default constructor - NO, this will not be generated since we

already have a constructor (Move constructor).

2. Copy constructor - NO

3. Copy assignment operator  - NO

4. Destructor - YES

5. Move assignment operator - NO

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

};

#pragma endregion Example 2

#pragma region Example 3

class Frog

{

// Move constructor - because if the second parameter is not

// specified, Frog can be constructed using another Frog!!!

Frog(Frog&&, int = 0){ }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What are the function generate for 'class Frog' if we have Move

constructor defined?

1. Default constructor - NO, this will not be generated since we

already have a constructor (Move constructor).

2. Copy constructor - NO

3. Copy assignment operator  - NO

4. Destructor - YES

5. Move assignment operator - NO

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Side note:

//Frog(int = 0){ } - Default constructor

//Frog(const Frog&, int = 0) {} - Copy constructor

};

#pragma endregion Example 3

#pragma region Example 4

class Fish

{

~Fish() {}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What are the function generate for 'class Fish' if we have

Destructor defined?

1. Default constructor - YES

2. Copy constructor - YES (Deprecated rule of C++03)

3. Copy assignment operator  - YES (Deprecated rule of C++03)

4. Move constructor - NO this will not be generated since we

have defined Destructor.

5. Move assignment operator - NO this will not be generated since we

have defined Destructor.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Bring it back, since it is deprecated

Fish(const Fish&) = default;

Fish& operator=(const Fish&) = default;

};

#pragma endregion Example 4

#pragma region Example 5

class Cow

{

Cow& operator=(const Cow&) = delete;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What are the function generated for 'class Cow' if we have

copy assignment operator that is being DELETED!!!?

1. Default constructor - YES

2. Copy constructor - YES (Deprecated rule of C++03)

3. Destructor - YES

4. Move constructor - NO this will not be generated since we

have declared Copy assignment operator even though it is deleted.

5. Move assignment operator - NO this will not be generated since we

have declared Copy assignment operator even though it is deleted.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// This will bring back the compiler generated copy constructor

Cow(const Cow&) = default;

};

#pragma endregion Example 5

# 07 C++ 11 Library Shared Pointer – I

#include <iostream>

#include <string>

#include <memory>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Shared Pointer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Dog

{

public:

Dog(string name) : name\_(name) { cout << "Dog is created: " << name << endl; }

Dog() { cout << "Nameless dog created: " << endl; name\_ = "nameless"; }

~Dog() { cout << "Dog is destroyed: " << name\_ << endl; }

void bark(void){ cout << "Dog " << name\_ << " rules!" << endl; }

private:

string name\_;

};

#pragma region Problem

void foo(void)

{

Dog\*p = new Dog("Gunner");

//...

//delete p;

//...

p->bark(); // p is a dangling pointer if we delete the Dog early - Undefined behavior

} // If we forget to delete the Dog, then there is memory leak!!!

#pragma endregion Problem

#pragma region Solution

void foo(void)

{

shared\_ptr<Dog>p(new Dog("Gunner")); // Count == 1

{

shared\_ptr<Dog> p2 = p; // Count == 2

p2->bark();

// To print the count.

// Notice that to use object members we use ->

// and to use shared\_ptr members we use '.'

cout << p.use\_count() << endl;

}// Count == 1

p->bark();

} // Count == 0

int main(void)

{

foo();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An object should be assigned to a smart pointer

as soon as it is created. Raw pointer should not

be used again.

Example:

{// Pitfall

Dog\* d = new Dog("Tank");

shared\_ptr<Dog> p(d); // p.use\_count() == 1

shared\_ptr<Dog> p2(d); // p2.use\_count() == 1

}// CRASH

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

// We are creating an object but not assigning to smart pointer.

// This is incorrect way of using smart pointer

Dog\* d = new Dog("Tank");

// We are using raw pointer again here. This is incorrect way of

// using smart pointer.

shared\_ptr<Dog> p(d); // p.use\_count() == 1

shared\_ptr<Dog> p2(d); // p2.use\_count() == 1

}// CRASH

return 0;

}

#pragma endregion Solution

#pragma region make\_shared

void foo(void)

{

 make\_shared is FASTER and SAFER!!! Exception safe!!!

shared\_ptr<Dog> ptr = make\_shared<Dog>("Bruce");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Use make\_shared instead of new. What happens if

we use new?

Two steps in creating a shared pointer...

1. "Gunner" is created.

2. p is created with Dog "Gunner".

Also,

- What if "Gunner" has been created successfully

and the make\_shared has failed?

Answer: Dog "Gunner" will be created but not managed

by share pointer. Memory will be leaked!!!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

shared\_ptr<Dog>p(new Dog("Gunner")); // Not exception safe

p->bark();

}

int main(void)

{

foo();

 Also we can use below mentioned cast operators to cast smart pointers

 static\_pointer\_cast

 dynamic\_pointer\_cast

 const\_pointer\_cast

return 0;

}

#pragma endregion make\_shared

# 08 C++ 11 Library Shared Pointer – II

#include <iostream>

#include <string>

#include <memory>

using namespace std;

class Dog

{

public:

Dog(string name) : name\_(name) { cout << "Dog is created: " << name << endl; }

Dog() { cout << "Nameless dog created: " << endl; name\_ = "nameless"; }

~Dog() { cout << "Dog is destroyed: " << name\_ << endl; }

void bark(void){ cout << "Dog " << name\_ << " rules!" << endl; }

private:

string name\_;

};

#pragma region Releasing Memory

void foo(void)

{

shared\_ptr<Dog> p1 = make\_shared<Dog>("Gunner");

shared\_ptr<Dog> p2 = make\_shared<Dog>("Tank");

//p1 = p2; // "Gunner" will be deleted

//p1 = nullptr; // "Gunner" will be deleted

//p1.reset(); // "Gunner" will be deleted

}

#pragma endregion Releasing Memory

#pragma region Custom Deleter

void foo(void)

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What is custom deleter?

- When shared\_ptr (p1) goes out of scope, it will call

some delete function to delete the Dog "Gunner" and this

delete function is called the "DELETER!!!".

- By default the "DELETER" is operator 'delete'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Below code uses default "DELETER": operator delete

shared\_ptr<Dog> p1 = make\_shared<Dog>("Gunner");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- Sometimes we want to use different "DELETER" and if

that is the case, we have to use the constructor of the

shared\_ptr to create a shared pointer.

Example:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

shared\_ptr<Dog> p2 = shared\_ptr<Dog>(

/\* Object \*/

new Dog("Bruce"),

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DELETER \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

[](Dog\* p){cout << "Custom Deleting" << endl; delete p; }

);

// When we create an array of Dog as shown below, p3 (shared\_ptr)

// will be pointing to dog[0]. When p3 goes out of scope, only

// dog[0] gets deleted and dog[1] and dog[2] are leaked. In this

// case we need to use custom deleter.

//shared\_ptr<Dog> p3(new Dog[3]);

// Custom deleter to delete the array of objects when p4 goes out

// of scope

shared\_ptr<Dog> p4(new Dog[4],

[](Dog\* p){ cout << "Deleting array of DOGs!!!" << endl; delete[] p; }

);

}

#pragma endregion Custom Deleter

int main(void)

{

foo();

return 0;

}

# 09 C++ 11 Library Weak Pointers

#include <iostream>

#include <string>

#include <memory>

using namespace std;

#pragma region Cyclic Reference

class Dog

{

shared\_ptr<Dog> mpFriend\_;

public:

string name\_;

Dog(string name) : name\_(name) { cout << "Dog is created: " << name << endl; }

Dog() { cout << "Nameless dog created: " << endl; name\_ = "nameless"; }

~Dog() { cout << "Dog is destroyed: " << name\_ << endl; }

void bark(void){ cout << "Dog " << name\_ << " rules!" << endl; }

void makeFreind(shared\_ptr<Dog> f)

{

mpFriend\_ = f;

}

};

int main(void)

{

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

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

pD->makeFreind(pD2);

pD2->makeFreind(pD);

return 0;

}

#pragma endregion Cyclic Reference

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

weak\_ptr:

- We know shared\_ptr provides shared ownership

of the object.

- weak\_ptr doesn't owns the object. Its not

the business of weak\_ptr to delete the object.

- weak\_ptr is just similar to raw pointer with

one level of protection. The protection is that

nobody can do delete on weak\_ptr.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Dog

{

weak\_ptr<Dog> mpFriend\_;

public:

string name\_;

Dog(string name) : name\_(name) { cout << "Dog is created: " << name << endl; }

Dog() { cout << "Nameless dog created: " << endl; name\_ = "nameless"; }

~Dog() { cout << "Dog is destroyed: " << name\_ << endl; }

void bark(void){ cout << "Dog " << name\_ << " rules!" << endl; }

void makeFreind(shared\_ptr<Dog> f)

{

mpFriend\_ = f;

}

void showFriend()

{

// Below code will not compile. weak\_ptr can't be used

// like regular pointer.

//cout << "My friend is: " << mpFriend\_->name\_ << endl;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In order to use it, we need to call lock(), what the lock()

function does is, it creates a shared\_ptr out of weak\_ptr.

Why it does that?

It does that to make sure 2 things...

1. It checks if the weak\_ptr is still pointing to a valid object.

2. It makes sure that, while it is accessing the object, the object

will not be deleted.

If weak\_ptr is a empty pointer, lock() will throw an exception.

If you don't want the lock() to throw an exception we have to

check the weak\_ptr using expired()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (!mpFriend\_.expired())

cout << "My friend is: " << mpFriend\_.lock()->name\_;

/\*use\_count() gives the count of shared\_ptr that is pointing to the object \*/

cout << ", He is owned by " << mpFriend\_.use\_count() << " pointers" << endl;

}

};

int main(void)

{

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

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

pD->makeFreind(pD2);

pD2->makeFreind(pD);

pD->showFriend();

return 0;

}

# 10 C++ 11 Library Unique Pointers

#include <iostream>

#include <string>

#include <memory>

using namespace std;

class Dog

{

public:

string name\_;

Dog(string name) : name\_(name) { cout << "Dog is created: " << name << endl; }

Dog() { cout << "Nameless dog created: " << endl; name\_ = "nameless"; }

~Dog() { cout << "Dog is destroyed: " << name\_ << endl; }

void bark(void){ cout << "Dog " << name\_ << " rules!" << endl; }

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Unique Pointer:

It represents exclusive ownership of an object.

One object can only be owned by one pointer and when

that one pointer goes out of scope or some other object

is assigned to the unique pointer, the old object is

deleted!!!.

Unique pointer is light weight smart pointer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region unique\_ptr Example 1

void test()

{

{

unique\_ptr<Dog> pD(new Dog("Gunner"));

pD->bark();

// Raw pointer for unique\_ptr. release() function

// returns the raw pointer from unique\_ptr similar

// to get() function of shared\_ptr

// After the pD.release(), pD has given up the ownership

// of the Dog and as the result nobody owns the Dog anymore

// nobody is going to delete the Dog. "Gunner" is not deleted here.

Dog\* ptr = pD.release();

// Inference is that, shared\_ptr.get() doesn't give up the

// ownership of the object. But the unique\_ptr.release()

// function gives up the ownership of the object.

if (!pD)

cout << "pD is empty!" << endl;

}

{

unique\_ptr<Dog> pD(new Dog("Gunner"));

// Gunner is destroyed when we call reset() function

// because pD starts to own another Dog

pD.reset(new Dog("Smokey"));

pD.reset(); // Reseting pD to nullptr

if (!pD)

cout << "pD is empty!" << endl;

// So the unique\_ptr.release() doesn't destroy the object

// but the unique\_ptr.reset() does!!!

}

}

#pragma endregion unique\_ptr Example 1

#pragma region unique\_ptr Example 2

void f(unique\_ptr<Dog> p)

{

p->bark();

}// Dog Gunner is destroyed here

unique\_ptr<Dog> getDog()

{

unique\_ptr<Dog> p(new Dog("Smokey2"));

return p; // This will automatically use move semantics

}

void test()

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We said unique\_ptr can't share the object at the

same time. But they can share the object at

different time.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unique\_ptr<Dog> pD(new Dog("Gunner"));

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

pD2->bark();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Smokey is destroyed.

2. pD becomes empty.

3. pD2 owns Gunner.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

pD2 = move(pD);

pD2->bark();

f(move(pD2));

if (!pD2)

cout << "pD2 is empty" << endl;

unique\_ptr<Dog> pdd = getDog();

// With shared\_ptr, we provided the custom deleter, but with

// unique\_ptr, we need not provide.

//shared\_ptr<Dog> arrofDog(new Dog[3], [](Dog\* p){delete[] p; });

// Creating array of dogs, here the template parameter is array

unique\_ptr<Dog[]> dogd(new Dog[3]);

}

int main(void)

{

test();

return 0;

}

#pragma endregion unique\_ptr Example 2

#pragma region unique\_ptr as Class Member

class Bone { };

class Toy { };

class UniqueDog

{

// This is the old way of managing resource.

// In the constructor, lets say we created new Bone() and assigned

// the object to pBone pointer. But in the rest of the constructor,

// an exception is thrown. This means, the object of UniqueDog is

// not created successfully so the destructor will not be called.

// Observe that, we have created the Bone object successfully, but

// the same Bone object, we are destroying it in the destructor.

// Since the destructor is not called, Bone's memory is leaked.

Bone\* pBone\_;

// Its better to use unique\_ptr instead of Raw pointer as a class member.

unique\_ptr<Toy> pToy\_;

public:

string name\_;

UniqueDog(string name) : name\_(name) { cout << "Dog is created: " << name << endl; }

UniqueDog() : pToy\_(new Toy()) // Initializing unique\_ptr

{

pBone\_ = new Bone();

cout << "Nameless dog created: " << endl; name\_ = "nameless";

}

~UniqueDog()

{

delete pBone\_;

cout << "Dog is destroyed: " << name\_ << endl;

}

void bark(void){ cout << "Dog " << name\_ << " rules!" << endl; }

};

#pragma endregion unique\_ptr as Class Member

# 11 C++ 11 Resource Managing Class

#include<iostream>

#include <string>

#include <vector>

#include <memory>

using namespace std;

#pragma region Resource Managing C++03

class Person

{

string\* pName\_;

public:

Person(string name) : pName\_(new string(name)) { }

~Person() { delete pName\_; }

void printName(void) { cout << \*pName\_ << endl; }

};

int main(void)

{

vector<Person> persons;

persons.push\_back(Person("George"));

persons.front().printName(); // CRASH

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To solve the crash:

1. Define copy constructor and copy assignment operator

2. Delete copy constructor and copy assignment operator

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

cout << "Good bye!!!" << endl;

return 0;

}

#pragma endregion Resource Managing C++03

#pragma region Resource Managing C++11

#pragma region Solution 1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To solve the crash:

1. C++11 provides delete keyword to delete copy

constructor and copy assignment operator.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Person

{

string\* pName\_;

public:

Person(string name) : pName\_(new string(name)) { }

Person(const Person&) = delete;

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

~Person() { delete pName\_; }

void printName(void) { cout << \*pName\_ << endl; }

};

int main(void)

{

vector<Person> persons;

persons.push\_back(Person("George"));

//persons.front().printName(); // CRASH

// C++11 provides a way to construct an object in place

// in a container. Instead of using push\_back(), we can use

// emplace\_back()

// This will construct the object in place

persons.emplace\_back("George");

cout << "Good bye!!!" << endl;

return 0;

}

#pragma endregion Solution 1

#pragma region Solution 2

// What if we want to use copy constructor and

// copy assignment operator?

// We can use shared\_ptr

class Person

{

// With shared\_ptr we don't need to define copy constructor

// or copy assignment operator

shared\_ptr<string> pName\_;

public:

Person(string name) : pName\_(new string(name)) { }

//Person(const Person&) = delete;

//Person& operator=(const Person&) = delete;

//~Person() { delete pName\_; }

void printName(void) { cout << \*pName\_ << endl; }

};

int main(void)

{

vector<Person> persons;

persons.push\_back(Person("George"));

persons.front().printName(); // CRASH

cout << "Good bye!!!" << endl;

return 0;

}

#pragma endregion Solution 2

#pragma region Solution 3

class Person

{

// We can use unique\_ptr since shared\_ptr pName\_

// is really not shared and also unique\_ptr is light weight.

//shared\_ptr<string> pName\_;

// But the code will not compile. Since persons.push\_back() will

// copy the person "George" into vector. Here we are using compiler

// generated copy constructor and compiler generated copy

// constructor does the member by member copy and we have a member

// pName\_ which is a unique\_ptr and which can't be copied.

// unique\_ptr can only be moved.

unique\_ptr<string> pName\_;

public:

Person(string name) : pName\_(new string(name)) { }

//Person(const Person&) = delete;

//Person& operator=(const Person&) = delete;

~Person() { }

// Since the destructor has been declared, compiler will not

// provide move constructor. So we have to bring it back.

// Visual Studio 2013 doesn't support defaulting move constructor

// and move assignment operator.

Person(Person&&) = default;

void printName(void) { cout << \*pName\_ << endl; }

};

int main(void)

{

vector<Person> persons;

Person p = Person("George");

persons.push\_back(std::move(p));

persons.front().printName();

return 0;

}

#pragma endregion Solution 3

#pragma endregion Resource Managing C++11

# 12 C++ 11 Library Regular Expression 1

#include <iostream>

#include <regex> // Regular Expression

#include <string>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Regular Expression

- What is "Regular Expression"?

Answer: Regular Expression is a specific pattern that provides

concise and flexible means to match strings of text, such as

particular characters, words or patterns of characters.

- In short, Regular Expression is a way of matching strings.

- It is supported by almost all languages.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Simple Example

int main(void)

{

string str;

while (true)

{

cin >> str;

/\*Ignore Case\*/

regex e("abc", regex\_constants::icase);

bool match = regex\_match(str, e);

cout << (match ? "Matched" : "Not matched") << endl;

}

return 0;

}

#pragma endregion Simple Example

#pragma region Special Pattern Characters

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Special Pattern Characters - has special meaning when it appears

in regular expression.

dot (.) - Any character except newline.

? - 0 or 1 preceding character. regex e("abc?") means, here

preceding character is 'c' and it means 0 or 1 'c'. So 'ab' is match

\* - 0 or more preceding character.

+ - 1 or more (can't be 0)

[] - any characters in the [] can be matched.

^ - any characters which are in [] is not a match

{3} - exact 3 appearance of preceding character

{3,}- 3 or more preceding character

{3,5} - either 3 or 4 or 5 of preceding character

| - mean OR

\ - escape sign

() - indicates the group

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

string str;

while (true)

{

cin >> str;

regex e("abc", regex\_constants::icase);

//regex e("abc."); // Means abcd is match

//regex e("abc?"); // ab is a match

//regex e("abc\*"); // abcdef is a match

//regex e("abc+"); //

//regex e("ab[cd]\*"); // abcccdd is a match

//regex e("ab[^cd]\*"); // abeee is a match

//regex e("ab[cd]{3}"); // abddd is a match

//regex e("ab[cd]{3,}"); // abcdddd is a match

//regex e("ab[cd]{3,5}"); // abcdddd is a match

//regex e("abc|de[fg]"); // def is a match

//regex e("abc|de[\[fg]");

//regex("(abc)de+\\1"); // abddeeabc is a match

//regex("(ab)c(de+)\\2\\1"); //

// To search a email address

//regex e("[[:w:]]+@[[:w:]]+\.com"); // [[:w:]] word char: digit, number or underscore

//bool match = regex\_match(str, e);

bool match = regex\_search(str, e);

cout << (match ? "Matched" : "Not matched") << endl;

}

return 0;

}

#pragma endregion Special Pattern Characters

# 13 C++ 11 Library Regular Expression 2 – Submatch – Not there

# 14 C++ 11 Library Regular Expression 3 – Iterators – Not there

# 15 C++ 11 Library Clocks and Timers - Part I

#include <iostream>

#include <chrono>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

chrono

- is the library to deal with time and date!!!

- A precision - neutral library for time and date.

Concepts needs clarification:

Clock: chrono provides 3 clocks!!!

1. System Clock - std::chrono::system\_clock

- Current time according to system and is not steady

2. Steady Clock - std::chrono::steady\_clock

- Goes at a uniform rate.

3. High Resolution Clock - std::chrono::high\_resolution\_clock

- provides smallest possible tick period.

Duration:

chrono::duration<> - represents time duration.

Example: If we want to represent "2 hours", here we need number (2) and

a unit (hours). These are the 2 things the template class duration<> needs.

duration<int, ratio<1,1>> - Number of seconds in a int

duration<double, ratio<60,1>> - Number of minutes in a double

typedefs provided by chrono

nanoseconds, microseconds, milliseconds, seconds, minutes, hours

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

#pragma region ratio

std::ratio<2, 10>r;

cout << r.num << "/" << r.den << endl;

// All the clock period is represented with ration

// Period of system clock

cout << chrono::system\_clock::period::num << "/"

<< chrono::system\_clock::period::den << endl; // 1/10000000

#pragma endregion ratio

chrono::microseconds mi(2700); // 2700 microseconds

chrono::nanoseconds na = mi; // 2700000 nanoseconds

// This will not compile. Because we are converting higher resolution

// duration to lower resolution and we might lose some data.

//chrono::milliseconds mill = mi;

// 700 microseconds are truncated!!!

chrono::milliseconds mill = chrono::duration\_cast<chrono::milliseconds>(mi);

mi = mill + mi; // 2000 + 2700 = 4700

mi.count(); // 4700

return 0;

}

# 16 C++ 11 Library Clocks and Timers - Part II

#include <iostream>

#include <chrono>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

time\_point: we need clock and duration concepts to know time\_point

- Represents a point of time.

In oder to represent a point of time, we need a reference.

Example: 04-July-2014. From where 2014 came from? 2014 number of

years after Jesus birth.

Similarly we need a reference to represent a point of time. In

programming world, the reference is

00:00 January 1, 1970 (Coordinated Universal Time - UTC) - Epoch of a clock

Examples:

time\_point<system\_clock,milliseconds> : According to system\_clock, the

elapsed time since epoch in milliseconds!!!

system\_clock::time\_point -> time\_point < system\_clock, system\_clock::duration >

steady\_clock::time\_point -> time\_point < steady\_clock, steady\_clock::duration >

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

// Gives current time of system clock.

chrono::system\_clock::time\_point tp = chrono::system\_clock::now();

// This returns the duration from now and the time of epoch

cout << tp.time\_since\_epoch().count() << endl;

tp = tp + chrono::seconds(2);

cout << tp.time\_since\_epoch().count() << endl;

chrono::steady\_clock::time\_point start = chrono::steady\_clock::now();

cout << "I am bored" << endl;

chrono::steady\_clock::time\_point endPoint = chrono::steady\_clock::now();

chrono::steady\_clock::duration d = endPoint - start;

if (d == chrono::steady\_clock::duration(0))

cout << "No time elapse!!!" << endl;

cout << chrono::duration\_cast<chrono::microseconds>(d).count() << endl;

return 0;

}

# 17 C++ 11 Library Random Number Engine

#include <iostream>

#include <random>

#include <sstream>

#include <chrono>

#include <vector>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Library Random Number Engine

- Random number library is made of 2 parts...

1. Random number engine.

2. Random number distribution.

Random number engine.

- Stateful generator that generates random values within predefined

min and max.

- The numbers generated are not truly random but they are pseudo random.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void printRandom(std::default\_random\_engine e)

{

for (int i = 0; i < 10; i++)

cout << e() << " ";

cout << endl;

}

int main(void)

{

{

std::default\_random\_engine eng;

cout << "Min: " << eng.min() << endl;

cout << "Max: " << eng.max() << endl;

// This will generate 1 random value

cout << eng() << endl;

// This will generate another random value

cout << eng() << endl;

//Engine has internal state which determines what kind of

// random value it generates

std::stringstream state;

// Saves the current state into variable state.

state << eng;

cout << eng() << endl;

cout << eng() << endl;

// Will restore the state.

state >> eng;

// If the engine is in the same state, it will always generates the

// same values!!!

cout << eng() << endl;

cout << eng() << endl;

}

{

default\_random\_engine e;

default\_random\_engine e2;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Two different random engine are printing the same set of values

Often times we don't want this. This is why we need seed!!!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printRandom(e);

printRandom(e2);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

So we will create a 3rd Random engine with seed!!!

What is a seed then?

- Seed is a number which determines the internal state of

the engine!!!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned seed = std::chrono::steady\_clock::now().time\_since\_epoch().count();

std::default\_random\_engine e3(seed);

printRandom(e3);

// This will reset the engine 'e' to initial state!!!

e.seed();

// This will set the engine 'e' to a state according to seed 109

e.seed(109);

e2.seed(109);

if (e == e2)

cout << "e and e2 are in the same state!!!" << endl;

}

return 0;

}

# 18 C++ 11 Library Random Number Distribution

#include <iostream>

#include <random>

#include <sstream>

#include <chrono>

#include <vector>

#include <string>

using namespace std;

int main(void)

{

unsigned seed = chrono::steady\_clock::now().time\_since\_epoch().count();

default\_random\_engine e(seed);

// This generates the random number in the range e.min to e.max()

cout << e() << endl;

// What if I want to generate the random number between 0 to 5?

// How can we do that?

cout << e() % 6 << endl;

// 1. Bad quality of randomness.

// 2. Can only provide uniform distribution

// What if we want values to be more distributed between 2 and 5

// and less distributed at 2 and 5? This is why we need library

// of distribution.

cout << e() % 6 << endl;

// This will generate the random numbers between 0 and 5 (including 0 and 5)

std::uniform\_int\_distribution<int> dist(0, 5);

// Here 3 provides a sauce of randomness and dist provides a way to

// distribute the random number within certain range and with certain

// probability.

cout << dist(e) << endl;

// Range is half open [0,5)

std::uniform\_real\_distribution<double> distr(0, 5);

cout << distr(e) << endl;

std::poisson\_distribution<int> disp(1.0);

cout << disp(e) << endl;

cout << "Normal Distribution.." << endl;

// Mean and standard deviation

std::normal\_distribution<double> disNormal(10.0, 6.0);

vector<int> v(20);

for (int i = 0; i < 800; i++)

{

// Convert double to int

int num = disNormal(e);

if (num >= 0 && num < 20)

v[num]++;

}

for (int i = 0; i < 20; i++)

cout << i << ": " << std::string(v[i], '\*') << endl;

cout << endl;

return 0;

}

# 19 C++ 11 Library Tuple

#include <iostream>

#include <tuple>

#include <string>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- Pair is used to store a pair of values of different types.

- A Tuple can be considered as an extended pair. Instead of

storing 2 values, it can be used to store arbitrary number of

values with arbitrary data types.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

std::pair<int, string> p = make\_pair(23, "hello");

cout << p.first << " " << p.second << endl;

// Creating a tuple

tuple<int, string, char> t(32, "Penny wise", 'a');

// To get the values stored in the tuple

cout << get<0>(t) << endl;

cout << get<1>(t) << endl;

cout << get<2>(t) << endl;

// To change the values in the tuple

// The get<1>(t) returns a reference to tuple!!!

get<1>(t) = "Pound Foolish";

string& s = get<1>(t);

s = "Patience is virtue!!!";

cout << get<0>(t) << endl;

cout << get<1>(t) << endl;

cout << get<2>(t) << endl;

// get<3> will not work. It wont compile. get<> needs a compile

// time constant. get<i> also will not compile!!!

// cout << get<3>(t) << endl;

int i = 1;

//cout << get<i>(t) << endl;

tuple<int, string, char> t2;

t2 = make\_tuple(12, "Cat", 'd');

// This does lexicographical comparison!!!

if (t > t2)

{

cout << "t is larger than t2" << endl;

// Member by member copy!!!

t = t2;

}

// Tuple can store references!!!

string st = "In for a penny";

tuple<string&> t3(st);

cout << "The string (before) : " << st << endl;

get<0>(t3) = "In for a pound!!!";

cout << "The string (after) : " << st << endl;

// To extract the values of tuple, we can use below technique

{

int x;

string y;

char z;

make\_tuple(ref(x), ref(y), ref(z)) = t2;

cout << "X: " << x << endl;

cout << "Y: " << y << endl;

cout << "Z: " << z << endl;

// Same as the above

std::tie(x, y, z) = t2;

// We can ignore a value using tie()

std::tie(x, std::ignore, z);

}

// Concatenating tuple

{

// t4 is a tuple<int, string, char, string> since t3 is tuple of

// string&

auto t4 = std::tuple\_cat(t2, t3);

}

return 0;

}

# 20 C++ 11 Library When to Use Tuple

#include <iostream>

#include <tuple>

#include <string>

#include <map>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

When to Use Tuple?

- We already have structure which does the same thing as

tuple. Which one should we use and when?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

tuple<string, int> getNameAge();

int main(void)

{

{

struct Person

{

string name;

int age;

};

tuple<string, int> t;

Person p = { "Nishith", 32 };

get<0>(t) = "Nishith";

get<1>(t) = 32;

// By looking at the code we can say that p.name is name

// and p.age is age and if we mouse hover over p, it says

// p is person. But tuple doesn't tell anything!!!.

// So by using struct, we can make the code more readable.

cout << p.name << " " << p.age << endl;

// If the code is 1000s of line, I have to think about what is

// store at position 0 of tuple!!! Then why do we need a tuple?

cout << get<0>(t) << " " << get<1>(t) << endl;

}

{

string name;

int age;

// Tuple can be used as a one-time only structure to transfer

// a group of data from one function to another instead of

// creating a dedicated structure for this purpose.

tie(name, age) = getNameAge();

cout << name << " " << age << endl;

// We can take advantage of comparison function of tuple

tuple<int, int, int> time1, time2;

if (time1 > time2)

cout << "time1 is greater than time2" << endl;

// Multi index map

map<tuple<int, char, float>, string>m;

m[make\_tuple(2, '1', 2.3)] = "Faith will move mountains";

// If we want to rotate values

int a = 5, b = 6, c = 7;

cout << "A: " << a << " B: " << b << " C: " << c << endl;

tie(b, c, a) = make\_tuple(a, b, c);

cout << "A: " << a << " B: " << b << " C: " << c << endl;

}

return 0;

}

tuple<string, int> getNameAge()

{

//....

//....

return make\_tuple("Nishith", 32);

}