C++11 New Features

Prepared By Haramohan Sahu

New Language Features

- **Language Usability Enhancements**
  - Constants:
    - Nullptr
    - Constexpr

```
char *pc = nullptr;     // OK
int  *pi = nullptr;     // OK
bool   b = nullptr;     // OK. b is false.
int    i = nullptr;     // error
```

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

Language Usability Enhancements

- **Nullptr**
  - C++11 introduced the `nullptr` keyword, which is specifically used to distinguish null pointers, 0. The type of `nullptr` is `nullptr_t`, which can be implicitly converted to any pointer or member pointer type, and can be compared equally or unequally with them.
- **Constexpr:**
  - The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time. The main idea is performance improvement of programs by doing computations at compile time rather than run time. Note that once a program is compiled and finalized by developer, it is run multiple times by users. The idea is to spend time in compilation and save time at run time

Variables and initialization

- **Initializer list**

std::vector v = { 1, 2, 3, 4 };

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

```
class MyNumber
{
public:
   MyNumber(const std::initializer_list<int> &v;) {
      for (auto itm : v) {
         mVec.push_back(itm);
      }
   }
private:
   std::vector<int> mVec;
};
int main()
{
   MyNumber m = { 1, 2, 3, 4 };
}
```

- **in-class member initializers**

Class A{private: string h = "text1"; };

Type inference

auto

One of the most common and notable examples of type derivation using auto is the iterator

```
// before C++11
// cbegin() returns vector <int >:: const_iterator
// and therefore itr is type vector <int >:: const_iterator
for ( vector <int >:: const_iterator it = vec. cbegin(); itr != vec. cend(); ++ it)
```

In C++11:

```
for ( auto it = vec. cbegin(); itr != vec. cend(); ++ it)
 auto i = 5; // i as int 2
auto arr = new auto(10); // arr as int *
```

Note: auto cannot be used for function arguments &auto cannot be used to derive array types:

```
 auto auto_arr2[10] = arr; // illegal, can't infer array type
```

decltype

decltype is a keyword used to query the type of an expression.
decltype is an operator which returns the declared type of an expression passed to it.
cv-qualifiers and references are maintained if they are part of the expression.

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

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

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

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

decltype(auto)

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

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

decltype(auto)

```
// decltype of a parenthesized variable is always a reference
decltype((i)) d; // error: d is int& and must be initialized
decltype(i) e;   // ok: e is an (uninitialized) int


// Return type is `int`.
auto f(const int& i) {
 return i;
}

// Return type is `const int&`.
decltype(auto) g(const int& i) {
 return i;
}
```

Range-based for loop

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

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

Templates

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

If use should not instantiation in current file/ translation unit

```
extern template class std::vector<MyClass>;
```

which tells the compiler not to instantiate the template in this translation unit.

The “>”

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

Template

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

Template aliases

```
typedef int (* process)( void *);

using NewProcess = int (*)( void *);

template < typename T>

using TrueDarkMagic = MagicType < std:: vector <T>, std:: string >;

int main() {

TrueDarkMagic < bool > you;

}
```

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

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


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

Default template parameters

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

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

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

Variadic templates

Before C++11, templates had a fixed number of parameters that must be specified in the declaration of the templates

With the variadic templates feature, you can define class or function templates that have any number (including zero) of parameters.

Template parameter packs

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

template<class...A> struct container{};

template<class...B> void func();

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

Continue Variadic Template

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

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

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

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

- **Delegate constructor**

C++11 introduces the concept of a delegate construct, which allows a constructor to call another constructor in a constructor in the same class.

```
class Base {
public :
int value1 ; int value2 ; Base() {
value1 = 1;
}
Base( int value) : Base() { // delegate Base() constructor
value2 = value;
}
};
int main() {
Base b(2);
std:: cout << b. value1 << std:: endl; std:: cout << b. value2 << std:: endl;
}
```

Inheritance constructor

- In C++03, constructors of a class are not allowed to call other constructors in an initializer list of that class. Each constructor must construct all of its class members itself or call a common member function.
- Constructors for base classes cannot be directly exposed to derived classes; each derived class must implement constructors even if a base class constructor would be appropriate. Non-constant data members of classes cannot be initialized at the site of the declaration of those members. They can be initialized only in a constructor.
- C++11 allows constructors to call other peer constructors (termed delegation). This allows constructors to utilize another constructor's behavior with a minimum of added code.

```
class BaseClass {
public: BaseClass(int value);
 };
class DerivedClass : public BaseClass
{
public:
using BaseClass::BaseClass;
 };
```

Inheriting constructors cont…

If the using-declaration refers to a constructor of a direct base of the class being defined (e.g. using Base::Base;), all constructors of that base (ignoring member access) are made visible to overload resolution
when initializing the derived class.

```
struct B1 {  B1(int, ...) { } };
int get();
struct D1 : B1 {
  using B1::B1;  // inherits B1(int, ...)
  int x;
  int y = get();
};

void test() {
  D1 d(2, 3, 4); // OK: B1 is initialized by calling B1(2, 3, 4),
                 // then d.x is default-initialized (no initialization is performed),
                 // then d.y is initialized by calling get()
  D1 e;          // Error: D1 has no default constructor
}
```

Explicit overrides and final

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

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

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

```
struct Base1 final { };

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

Like this we can declare final with virtual function which will prevent to ovverride

Cont…

When overriding a virtual function, introducing the override keyword will explicitly tell the compiler to overload, and the compiler will check if the base function has such a virtual function, otherwise it will not compile:

```
struct Base {
virtual void foo( int );
};
struct SubClass: Base {
virtual void foo( int ) override; // legal
virtual void foo( float ) override; // illegal , no virtual function in super class
};
```

final

final is to prevent the class from being continued to inherit and to terminate the virtual function to continue to be overloaded

Continue final

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

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

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

final is to prevent the class from being continued to inherit and to terminate the virtual function to continue to be overloaded.

Explicit delete default function

C++11 provides explicit declarations to take or reject functions that come with the compiler.

```
class Magic {
public :
Magic() = default ; // explicit let compiler use default constructor
Magic& operator =( const Magic&) = delete ; // explicit declare refuse constructor
Magic( int magic_number);
}
```

OVERRIDE

In C++03, it is possible to accidentally create a new virtual function, when one intended to override a base class function. When overriding a virtual function, introducing the override keyword will explicitly tell the compiler to overload, and the compiler will check if the base function has such a virtual function, otherwise it will not compile.

```
 struct Base {
 virtual void foo(int);
 };

struct SubClass: Base {
 virtual void foo(int) override; // legal
 virtual void foo(float) override; // illegal, no virtual function in super class
 };
```

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

Strongly typed enumerations

- In C++03, enumerations are not type-safe. They are effectively integers, even when the enumeration types are distinct. This allows the comparison between two enum values of different enumeration types. The only safety that C++03 provides is that an integer or a value of one enum type does not convert implicitly to another enum type.
- C++11 allows a special classification of enumeration that has none of these issues. This is expressed using the enum class (enum struct is also accepted as a synonym) declaration.

```
enum class Enumeration
{
    Val1,
    Val2,
    Val3 = 100,
    Val4 // = 101
};
```

This enumeration is type-safe. Enum class values are not implicitly converted to integers. Thus, they cannot be compared to integers either
(the expression Enumeration::Val4 == 101 gives a compile error)

Language Runtime Enhancements

- **Lambda Expression**
- C++11 provides the ability to create anonymous functions, called lambda functions

Anonymous functions are used when a function is needed, but you don't want to use name to call a function.

Example:

```
[](int x, int y) -> int { return x + y; }
```

[capture list] (parameter list) mutable(optional) exception attribute -> return type {

// function body

}

Lambda continue

A lambda is an unnamed function object capable of capturing variables in scope. It features: a capture list; an optional set of parameters with an optional trailing return type; and a body.
Examples of capture lists:
[] - captures nothing.
[=] - capture local objects (local variables, parameters) in scope by value.
[&] - capture local objects (local variables, parameters) in scope by reference.
[this] - capture this pointer by value.
[a, &b] - capture objects a by value, b by reference.

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

Lambda continue

By default, value-captures cannot be modified inside the lambda because the compiler-generated method is marked as const. The **mutable** keyword allows modifying captured variables.
The keyword is placed after the parameter-list (which must be present even if it is empty).

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

Unrestricted unions

In C++03, there are restrictions on what types of objects can be members of a union. For example, unions cannot contain any objects that define a non-trivial constructor or destructor. C++11 lifts some of these restrictions.

If a union member has a non trivial special member function, the compiler will not generate the equivalent member function for the union and it must be manually defined.

This is a simple example of a union permitted in C++11:

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

Type aliases

Semantically similar to using a typedef however, type aliases with using are easier to read and are compatible with templates.

```
template <typename T>
using Vec = std::vector<T>;
Vec<int> v; // std::vector<int>

using String = std::string;
String s {"foo"};
```

String Literal example

From C++ 11, we can use raw strings in which escape characters (like \n \t or \") are not processed.

```
// String literals
    auto s0 =   "hello"; // const char*
    auto s1 = u8"hello"; // const char*, encoded as UTF-8
    auto s2 =  L"hello"; // const wchar_t*
    auto s3 =  u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 =  U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 =   R"("Hello \ world")"; // const char*
    auto R1 = u8R"("Hello \ world")"; // const char*, encoded as UTF-8
    auto R2 =  LR"("Hello \ world")"; // const wchar_t*
    auto R3 =  uR"("Hello \ world")"; // const char16_t*, encoded as UTF-16
    auto R4 =  UR"("Hello \ world")"; // const char32_t*, encoded as UTF-32
```

```
                                                    // literals (UDLs)
#include<iostream>
#include<iomanip>
using namespace std;
  // user defined literals
  // KiloGram
long double operator"" _kg( long double x )
{
    return x*1000;
}
  // Driver code
int main()
{
    long double weight = 3.6_kg;
    cout << weight << endl;
    cout << ( 32.3_mg *2.0_g ) << endl;
    return 0;
}
unsigned int uint = 25U; // for unsigned integer with value 25
auto i = 0x30;
```

Attributes are one of the key features of modern C++ which allows the programmer to specify additional information to the compiler
to enforce constraints(conditions), optimize certain pieces of code or do some specific code generation.

Attributes represent a standardized alternative to vendor-specific extensions such as
#pragma directives, __declspec() (Visual C++), or __attribute__ (GNU).
However, you will still need to use the vendor-specific constructs for most purposes.
The standard currently specifies the following attributes that a conforming compiler should recognize:

[[noreturn]] Specifies that a function never returns; in other words it always throws an exception.
The compiler can adjust its compilation rules for [[noreturn]] entities.

[[carries_dependency]] Specifies that the function propagates data dependency ordering with respect to thread synchronization.

[[deprecated]] Specifies that a function is not intended to be used, and
might not exist in future versions of a library interface.

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

[[noreturn]] void f()
{
    // Some code that does not return
    // back the control to the caller
    // In this case the function returns
    // back to the caller without a value
    // This is the reason why the
    // warning "noreturn' function does return' arises
}

void g()
{
    std::cout << "Code is intented to reach here";
}

int main()
{
    f();
    g();
}
```

Function Object Wrapper

Class template std::function is a general-purpose polymorphic function wrapper.
Instances of std::function can store, copy, and invoke any Callable target -- functions, lambda expressions,
bind expressions, or other function objects,
as well as pointers to member functions and pointers to data members.

```
void print_num(int i)
{
   std::cout << i << '\n';
}

int main()
{
   // store a free function
   std::function<void(int)> f_display = print_num;
   f_display(-9);

   // store a lambda
   std::function<void()> f_display_42 = []() { print_num(42); };
   f_display_42();

}
```

Some more example

```
# include < functional >
# include <iostream >

int foo( int para) {
return para;
}

int main() {
// std:: function wraps a function that take int paremeter and returns int value
std:: function < int ( int )> func = foo;

int important = 10;
std:: function < int ( int )> func2 = [&]( int value) -> int {
return 1+ value+ important;
};
std:: cout << func(10) << std:: endl; std:: cout << func2 (10) << std:: endl;
}
```

std::bind and std::placeholder

One of the main use of std::function and std::bind is as more generalized function pointers.
You can use it to implement callback mechanism. And std::bind is used to bind the parameters of the function call.

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

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

Linear Container

std::array

The array is a container for constant size arrays. This container wraps around fixed size arrays and also doesn't loose the information of its length when decayed to a pointer.

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

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

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

std::forward_list

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

It does not provide the size() method

Unordered Container

In traditional, elements are internally implemented by red-black trees. The average complexity of inserts and searches is O(log(size)). While inserting into ordered container, the element size is compared according to the < operator & then stored in right place. That is why it is in sorted order or called ordered container. The elements in the unordered container are not sorted, and the internals are implemented by the Hash table.

The average complexity of inserting and searching for elements is O(constant), Significant performance gains can be achieved without concern for the order of the elements inside the container. C++11 introduces two sets of unordered containers: std::unordered_map/std::unordered_multimap and std::unordered_set/std::unordered_multiset.Their usage is basically similar to the original std::map/std::multimap/std::set/set::multiset

TUPLE

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

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

// Assigning values to tuple using make_tuple()
geek = make_tuple('a', 10, 15.5);
// Use of size to find tuple_size of tuple
cout << tuple_size<decltype(geek)>::value << endl;
```

tie() :- The work of tie() is to unpack the tuple values into separate variables

```
// Use of tie() without ignore
tie(i_val,ch_val,f_val) = tup1;
```

tuple_cat() :- This function concatenates two tuples and returns a new tuple.

```
// Concatenating 2 tuples to return a new tuple
auto tup3 = tuple_cat(tup1,tup2);
```

Regular expression

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

```
string a = "GeeksForGeeks";

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

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


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

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

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

std::to_string

Converts a numeric argument to a std::string.

std::to_string(1.2); // == "1.2"
std::to_string(123); // == "123"
The chrono library contains a set of utility functions and types that deal with durations, clocks, and time points. One use case of this library is benchmarking code:
std::chrono
==========
std::chrono::time_point<std::chrono::steady_clock> start, end;
start = std::chrono::steady_clock::now();
// Some computations...
end = std::chrono::steady_clock::now();

std::chrono::duration<double> elapsed_seconds = end - start;
double t = elapsed_seconds.count(); // t number of seconds, represented as a `double`

THREADS

std::thread is the thread class that represents a single thread in C++.

To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object)

into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable.

A callable can be either of the three

1. A function pointer
2. A function object
3. A lambda expression

//Using Function

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

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

// Define the class of function object
class fn_object_class {
    // Overload () operator
    void operator()(params)
    {
        // Do Something
    }
}
// Create thread object
std::thread thread_object(fn_class_object(), params)
```

Thread Cont.

```
int main()
{
    // Start thread t1
    std::thread t1(callable);

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

    // t1 has finished do other stuff

    ...
}
```

Joining and Detaching Threads

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

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

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

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

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

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

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

How to pass argument to threads.

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

or if the thread function signature is like below:

void threadCallback(int x, std::string str); → signature

```
    int x = 10;   std::string str = "Sample String";
```

std::thread threadObj(threadCallback, x, str); → Call like this

Data Sharing and Race Conditions:

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

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

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

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

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

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

std::lock_guard

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

```
std::mutex mtx;

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

void print_thread_id (int id) {
  try {
    // using a local lock_guard to lock mtx guarantees unlocking on destruction / exception:
    std::lock_guard<std::mutex> lck (mtx);
    print_even(id);
  }
  catch (std::logic_error&) {
    std::cout << "[exception caught]\n";
  }
}
```

```
int main ()
{
  std::thread threads[10];
  // spawn 10 threads:
  for (int i=0; i<10; ++i)
    threads[i] = std::thread(print_thread_id,i+1);

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

  return 0;
}
```

Difference between std::lock_guard and std::unique_lock

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

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

CONDITION VARIABLE

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

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

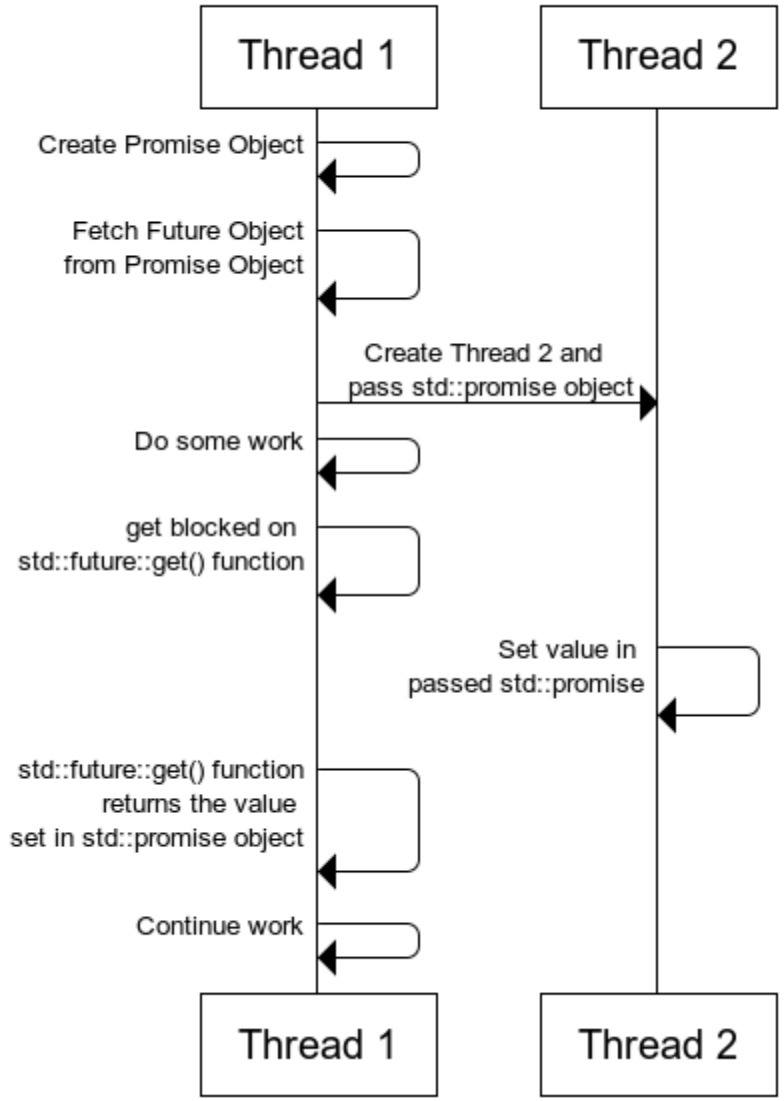
Condition variables allow one to atomically release a held mutex and put the thread to sleep.
Then, after being signaled, atomically re-acquire the mutex and wake up.
condition variables always take a mutex. The mutex must be held when wait is called. You should always verify that the desired condition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

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

conditionVariable.cpp

```
std::mutex mutex_;
std::condition_variable condVar;
void doTheWork(){
  std::cout << "Processing shared data." << std::endl;
}
void waitingForWork(){
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
    doTheWork();
    std::cout << "Work done." << std::endl;
}
void setDataReady(){
    std::cout << "Sender: Data is ready."  << std::endl;
    condVar.notify_one();
}
int main(){
std::thread t1(waitingForWork);
  std::thread t2(setDataReady);
  t1.join();  t2.join();
  std::cout << std::endl;
  }
```

std::promise and std::future work flow



std::future & std::promise

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

Every std::promise object has an associated std::future object, through which others can fetch the value set by promise.
So, Thread 1 will create the std::promise object and then fetch the std::future object from it before passing the std””promise object to thread 2 i.e.
std::promise is also a class template and its object promises to set the value in future. Each std::promise object has an associated std::future object that will give the value once set by the std::promise object.
A **std::promise** object shares data with its associated **std::future** object.

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

Now Thread 1 will pass the promiseObj to Thread 2.

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

```
int val = futureObj.get();
```

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

```
promiseObj.set_value(45);
```

std::future & std::promise

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

Every std::promise object has an associated std::future object, through which others can fetch the value set by promise.
So, Thread 1 will create the std::promise object and then fetch the std::future object from it before passing the std””promise object to thread 2 i.e.
std::promise is also a class template and its object promises to set the value in future. Each std::promise object has an associated std::future object that will give the value once set by the std::promise object.
A **std::promise** object shares data with its associated **std::future** object.

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

Now Thread 1 will pass the promiseObj to Thread 2.

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

```
int val = futureObj.get();
```

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

```
promiseObj.set_value(45);
```

std::future & std::promise:

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

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