DAY - 1:

PIMPL: Pointer implementation model.

```
class CA
{
private:
  CB *p;   //PIMPL model
public:

//RAII - Resource Acquistion Is Initialization
  CA()
  {
    p = new(nothrow) CB;  //Accquiring a heap resource
  }

  //...
  ~CA()
  {
    delete p;  //Releasing the resource
  }
};
```

An extension of PIMPL model & RAII design is SMARTPOINTER TECHNQIUE.

```
class CA{...};


class CB
{
private:
  CA* p;   //PIMPL model
public:
    CB()
    {
      p = new(nothrow) CA;
    }
    ~CB()
    {
      delete p;
    }

    CA* GetCAaddress()
    {
      return this->p;
    }

    CA* operator &()
    {
      return this->p;
    }

    CA* operator ->()
    {
      return this->p;
    }

    CA& operator *()
    {
      return *p;
    }
};

//consumer code
CB obj1;

CA* p = obj1.p;  //error, cannot access private data

CA* p = obj1.GetCAaddress();

CA* p = &obj1;   // CA* p = obj1.operator &();

obj1->fun();   // (obj1.operator ->())->fun();

(*obj1).print();   // (obj1.operator *()).print();
```

C language: FUNCTION CENTRIC
C++ language: DATA CENTRIC

```
struct STUDENT
{
  int stdid;
  char sname[25];
  float marks;
};


struct EMPLOYEE
{
  int empid;
  char empname[25];
  float salary;
};

//structure varaibles, memory gets defined...
struct STUDENT s1;   //It has only STATE, does not have a well-defined behaviour
struct EMPLOYEE e1;

//...
//...

s1.marks = e1.salary;
```

Now, in C++ we can have an extended struct, to ensure data safety...

```
struct EMPLOYEE
{
// DATA MEMBERS, ATTRIBUTES
  int empid;
  char empname[25];
  float salary;
  //member functions, methods or behaviours
  void input(){ }
  void print(){ }
};


EMPLOYEE e1;  //An object - It has both STATE as well as a well-defined behaviour.
------------------------------------------

CA* p = new(nothrow) CA[5];  //Array of heap objects

p[0].print();
p[1].print();



CA p[5];  //Array of stack objects
p[0].print();
p[1].print();
```

Syntax to the invoke the 'print' method is the same for both stack based and heap based instances. To avoid this conf usion or make it clear to the programmer, it would make sense to overload the '->' operator for heap based instances...

```
class CA
{

public:
  //..
  CA* operator ->()
  {
    return this;
  }
}:
```

Now use the foll: convention...

```
p[0]->print();    // (p[0].operator ->())->print();
p[1]->print();
----------------------------------
void fun()
{
  int *p = new(nothrow) int;  //throw version
  if(p == NULL)
  {
     cout <<"allocation failed";
     exit(1);
  }
  //..
}
```

```
void fun()
{
 try
 {
  int *p = new int;  //throw version
  //......
 }
  catch(bad_alloc &x)
  {
     cout << x.what();
  }

  //..
}
---------------------------------------------

CA* p =(int*) malloc(sizeof(int));

CA* p = new int;   // CA* p = (CA*) operator new(sizeof(int));

void* operator new(size_t size)
{
   if allocation successful
      return address
   else
      throw bad_alloc();
}

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

CA* p = new(nothrow) int;  // CA* p = (CA*) operator new(sizeof(int), nothrow());

struct nothrow {  };  //empty struct for overloading purpose

void* operator new(size_t size, nothrow& x)
{
  if allocation suceeds
     return heap address
  else
     return NULL;
}

class exception
{
public:
  virtual const char* what() = 0;
};

class bad_alloc:public exception
{
public:
  const char* what()
  {
    //...
  }
};
-------------------------------------
class CA
{
int a,b;
public:
  void fun1()    // void fun1(CA* const this)
  { }

  void fun2() const  // void fun2(const CA* const this)
  { }

  static void fun3()   // void fun3()
  {
  }

};
//class consumers
CA obj1;

obj1.fun1();     // CA::fun1(&obj1);   ===> CA::fun1(CA* );
obj1.fun2();     // CA::fun2(&obj1);   ===> CA::fun2(CA* );

const CA obj2;

//error, const objects will only call const member functions.
obj2.fun1();   //  CA::fun1(&obj2);   ===>  CA::fun1(const CA* );

CA::fun3();
---------------------------------------------------

class CA
{
private:
  int a,b,c;
  static int d;
public:
  //...

  //void Getd()  /Deals only with static data
  static void Getd()
  {
    return d;
  }
};
//define memory for the static data well before creating objects
int CA::d;

int main()
{
  CA obj1, obj2, obj3;
  //...

  obj1.Getd();   // CA::Getd();
```

```
}

------------------------
class CA
{
private:
  int a;
public:
  void fun(int a)
  {
    //int a;
    cout << a;  // local variable only
    cout << this->a;
  }
};
```

FUNCTIONS ARE THE BASIC BUILDING BLOCKS in a structured world.

```
void fun1()
{

}
void fun2(){ }
void fun3(){ }

int main()
{
  fun1();
  fun2();
  fun3();
}

-----------------------
//global function

void fun(){ }     // ?fun@@YAXXZ  --microsoft
                  // _Zfun3v

class CA
{
public:
  void fun()  //class member function  ---> ?fun@CA@YAXXZ
  { }
};
```

FUNCTORS:-
  Overloading of function-operator is called as FUNCTOR.

STL: Standard template library

1) container classes
  - These classes that exihibit HAS-A relationship (PIMPL model) help us contain or hold values of the desired type i n a specific mathematical model....

- stack, queue, list, vector
- map, pair, multimap etc... [KEY-VALUE RELATIONSHIP]

2) iterator classes
  - Are classes whose objects would act like pointers to the values contained in a container class instance.

3) allocator class
  - Are the classes which the container class instances use, to manage heap memory for the values getting populated onto them.

4) algorithms:
  These are global utility functions, that can help us solve some custom business on the values held in a container cla ss object.

```
template<typename T1, typename T2 = Allocator<T1>> class list
{
T1 data;
T2 mem_obj;

};


list<int> ls1;   // list<int, Allocator<int>> ls1;

//Custom allocator

template<typename T> class MyAllocator
{
public:
 void* Allocate(){... }
 void Deallocate(void* p){....}
 //...
};


list<int,MyAllocator<int>> ls1;

--------------------------------
template<typename T1, typename T2 = Allocator<T1>> class list
{
  friend class iterator;
  private:
T1 data;
T2 mem_obj;
public:
  //....



  class iterator
  {
    //...
  };
};
```

```
-------------------------
class student
{
 //...
 public:
   int Getid(){return id; }
   float GetMarks(){return marks;}
}:

bool SortId(student& s1, student& s2)
{
  if(s1.Getid() > s2.Getid())
     return true;
  else
     return false;
}

bool SortMarks((student& s1, student& s2)
{
  if(s1.GetMarks() > s2.GetMarks())
     return true;
  else
     return false;
}

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

int a;
float b;
double c;

//float result1 = a+b*c;
auto result1 = a+b*c;   //float result1=...;

//float result2 = b+c;
auto result2 = b+c;   //float result2=...;

//float result3 = a+c;
auto result3 = a+c;  //float result3=...;

---------------------------------------------------
```

While using the type-inference keyword 'auto', for defining a variable, that variable must and should be initialized.

It is based on the initialized value or info, the data-type of the variable would be decided by the compiler.

for eg:

```
auto x;        //error, not initialized
auto x = 10;   //ok
```

The keyword 'auto' cannot be used for declaring array type elements.
for eg:

```
auto arr[10] = {10,20,30...};  //error
```

Cannot use 'auto' for declaring of a data member in a class.

```
class CA
{
  private:
    auto x;  //error
    auto y;  //error
};
```

Cannot use 'auto' as a formal parameter of a function.

```
void fun(auto x)  //error
{
}
```

```
int a,b,c;

auto a = 10, b=30, c=20;  //ok

auto a = 10, b=34.12f, c=56.21;  //error
```

The extended qualifiers if any would be ignored by the keyword 'auto'

```
const int b = 200;

auto x = b;   // 'x' will be 'int' and not 'const int'
```

```
list<int> ls1;

list<int>::iterator itr = ls1.begin();
```

Now, modern approach

```
auto itr = ls1.begin();
```

```
int func(double x){....}

auto g = func;   // int(*g)(double);

//call the function
auto result = (*g)(45.12);


auto& f = func;

//call the function
auto result = f(45.12);
```

DECLTYPE:
  Unlike 'auto', the decltype keyword is used for defining variables of desired type without the need to initialize the s
ame. Also the extended qualifiers is not ignored like 'auto'.

```
auto x = 10;

//Define 'b' whose type has to be whatever is 'x' type.
decltype(x) b;   // int b;

decltype(x*10.32f)  c;
-----------------------------------------
const auto b1=200;


auto val1 = b1;   // int val1= 200;

decltype(b1) b2;  //error --> const int b2;
decltype(b1) b2=200;  //ok, const int b2 = 200;

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



void Add(int x, int y)
{
  //..
  //...
}


int main()
{
  //..
  //...
  Add(10,20);
  //..
  //...
}

|  10   |  <-- sp
|  20   |  <-- bp
|-------|
|  500  |  <--bp, sp
|-------| [main, @ 500]

After epilog

|  10   |
|  20   |
|-------|
|  500  |  <--bp, sp
|-------| [main, @ 500]
-----------------------------------------------

template<typename T> class Func
{
private:
  T *fp;
};
```

```
Func<void(void)>  obj1;   //   void(*fp)(void);
Func<int(float)> obj2;   //   int(*fp)(float);
```

There is a built-in header in traditional C++ library called 'functional'

```
#include<functional>

function<int(double)> obj1;
```

LAMBDA'S:
  Ananymous functions, can be locally scoped inside of another function or also defined globally.

```
  []()
  {
    //.....
  }();   //in-place call
```

  A lambda function with a handle

```
  auto lm=[]() ->void
  {
    //.....
  };


  lm(); //call the lambda


  auto lm = [](void)->void {....};

  function<void(void)> lm = ...;
  lm();   ==> lm.operator()();
```

DAY - 2:

LAMBDA EXPRESSIONS:-
  These are ananymous functions, can be scoped inside a function [global/member function] or in the global scope.

  Whenever we plan to define a lambda expression, it always starts with a pair of square-brackets...

```
  []()
  {
     //....
  };
```

A lambda if it has to be invoked, can be achieved in 2 different styles...
  - Can be invoked at the point of definition itself...

```
  []()
  {
    //...
  }();    //call here
```

- The defined lamba can be assigned to a lambda handle, and then use this handle to invoke it.

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

Invoke using the lambda handle...where 'lm' is actually an object of 'function' class.
And it calls the 'function operator' member function

```
  lm();    // lm.operator()();

  auto lm =[](){...};    // function<void(void)> lm =  [](){...};
```

Whenever we plan to define lambda's, this construct does not have a place-holder in its syntax to state it's return-typ
e if any..., this is where we use the special modern C++ syntax called 'trailing return-type syntax'.

for eg:

```
  auto lm =[]() ->void
  {
    //..
    //does not return anything...
  };


  auto lm1 =[]() -> int
  {
    int a;
    ///...
    return a;
  };
```

****
CAPTURE BY REFERENCE:

```
int a=10,b=20;   //outer-scope elements being captured by reference in a lambda function

auto lm = [&]()  ->void
    {
        int& lambda::a = main::a;
        int& lambda::b = main::b;
        //..
    };
```

****
CAPTURE BY VALUE:

```
int a=10,b=20;   //outer-scope elements being captured by reference in a lambda function

auto lm = [=]()  ->void
    {
        static const int lambda::a = main::a;
        //error changes not permitted on 'const' elements
        a=a+100;
        static const int lambda::b = main::b;
        //error changes not permitted on 'const' elements
        b=b+100;
    };
```

If, we as developers insist on making changes to the element that has been captured by value, then use the 'mutable' qualifier in the lambda definition, by doing so the compiler will now permit changes to these elements that have bee n captured by VALUE.

```
auto lm = [=]() mutable ->void
    {
        static const int lambda::a = main::a;
        //OK changes permitted on 'const' elements as it is mutable
        a=a+100;
        static const int lambda::b = main::b;
        //OK changes permitted on 'const' elements as it is mutable
        b=b+100;
    };
```

RECURSIVE LAMBDA:-
  Ability of a lambda function to call itself.

```
  auto lm = []()
  {
    //..
    //..
    lm(); //error, 'lm' is not captured
  };
```

For recursion, the lambda handle must be visible inside the lambda scope, thereby we need to capture the same.

```
auto lm = [&lm]()
{
  //...
  //...
  lm();  //FINE, It is visible inside the lambda as it has been captured
};
```

Yet, the compiler would throw an error, for the above lambda definition, why ?
We are trying to capture a function object instance by the name 'lm', even before the compiler could infer its type in f ull and define it.

  So, here we notice the type-inference keyword 'auto' will not be of any help. The only option is to explicitly state th e lambda handle type...

```
  function<int(int)> lm =[&lm](int x)
  {
    //..
    lm(...);
  };
```

UNIFORM UNITIALIZATION SYNTAX [brace initialization syntax]:-

- When used with a functions local variable declaration, it would automatically initialize to zero.
  for eg:

```
    int a;  //local variable on stack, will hold garbage info

    int a{};  //though a local variable on the stack, will be initialized zero.
```

```
int a=100;
int a(100);
Now,
int a{100};
*******************
CA obj1 = 100;
CA obj1(100);
Now,
CA obj1{100};
```

- When used with 'new' operator function, each instance of array can call respective constructors...

- When used for populating collections-STL containers, all values in the braces will treated as one class-type called 'initializer_list' type.

IN-CLASS INITIALIZERS FOR NON-STATIC DATA MEMBERS:-

In a traditional C++ code, while declaring data members in a class, there was no support for providing any default v alues for the same. Doing so it would flag error. For eg:

```
  class CA
  {
    private:
      int a =10;          //error
      float b= 45.12f;  //error
  };
```

On the contrary, now with in-class initializer feature in modern C++, we can achieve the same, where-in we can pro vide default values for each of the non-static data members, this value can be either a literal or an expression or a fun ction call.

It is just a syntatic sugar to explicitly avoid or define a default constructor by the class author.

The memory for data members do get defined only upon an object construction, this syntax is a message to the comp iler to initialize the data members with the stated values in the class declaration by defining its own default construct or.

```
class CA
  {
    private:
      int a =10;          //ok, in modern C++
      float b= 45.12f;  //ok, modern c++
  };
```

Traditional C++ code   Modern C++ code

```
    class CA                                       class CA
    {                                             {
      private:                                       private:
        int a;                                          int a = 10;
        float b;                                         float b = 45.12f;
        float c;                                         float c = a*b;
      public:                                         public:
        CA():a(10),b(45.12f),c(a*b)                              //no explicit def. constructor necessary
         { }                                            //compiler would assume on our behalf.
    };                                            };
```

DELEGATING CONSTRUCTORS:-

A provision by which a class constructor can be called in the initializion_list of the same class constructor.

In traditional C++, it was not possible to delegate constructors, such attempts would throw an error. For eg:

```
class CA
{
public:
  CA():CA(1,2)            //ERROR, not supported in traditional C++
   {}
  CA(int x)
   { }
  CA(int x, int y)
   {  }
};


CA obj1;
```

DEFAULT METHODS:-

The 'default' keyword is applicable to the foll: compiler generated methods
1) Default constructor
2) Copy constructor
3) Assignment function.

In traditional C++ class, if none of the above methods are defined, it is generally assumed compiler would generate o ne.

A default constructor when ?
- A default constructor would be assumed under the foll: scenarios
Traditional C++
a) Virtual inheritance model
b) A base has a default constructor, derived does not have any - then the compiler will assume one for the derived class.
c) A class contains a data member, which is an object of another class having a default constructor, then the compiler would assume one for the container class.
d) If the class is POLYMORPHIC by nature i.e. having OVER-RIDING and OVER-RIDABLE methods.
Modern C++
e) If the class is using in-class initializer feature, and no default constructor is defined, then the compiler will assume one.
****

A copy constructor when ?
If the class is not provided with any copy constructor, and the class consumer happens to copy construct

objects of such a class, then the compiler is expected to generate a COPY CONSTRUCTOR.

An Assignment function when ?
If the class it not provided with an assignment function, and the class consumer happends to assign one or objects, then the compiler would assume one.

NOTE: If the above stated contexts do not apply for a class, then compiler will not assume any of the above stated m ethods.
****
The default method in modern C++ world supports 2 things...
a) States the intentions of the class author that they plan to depend upon the compiler synthesized methods and they don't plan to define anything of their own.
In Traditional C++ class, if these methods are defined it is blindly assumed by the class consumer that the compil er will generate one.

Where as now in modern C++, we as class authors can make our intentions very clear to the class consumer, that we plan to depend upon the compiler generated one. for eg:

```
class CA
{
public:
  CA() = default;
  CA(const CA& x) = default;
  CA& operator =(const CA& x) = default;
};
```

The above class is more expressive in its intent.

b) It also supports the programmer to have a fine grained control over these compiler generated methods.

We generally understand or know, that in a traditional C++ class if we don't happen to define these methods, then the compiler will assume one and these compiler assumed methods will always be under 'public' access-specifier of the class.

But, now in modern C++ we as class authors can dictate or direct the compilers that they shall assume or provide these methods under the access-specifier of our choice not necessarily under 'public'.

How ?
Let the compiler assume a default constructor in 'public', the copy constructor and assignment function in private.

```
class CA
{
private:
 CA(const CA& x)=default;
 CA& operator =(const CA& x)=default;
public:
  CA()=default;
};
```

The above class is expressive to the class consumer, in its intent that the copy construction and assignment operation is not permitted outside the class scope, shall be allowed only inside the class scope and these methods will be compiler generated ones.

DELETE METHODS:
It is just the converse of default methods, dis-able certain operations, applicable to the foll: methods or proble

m scenarios...

1) default constructor
2) LVALUE copy constructor
3) RVALUE copy constructor
4) LVAUE assignment function
5) RVALUE assignment function
6) To dis-able the implementation of 'operator new' on certain class types
7) To dis-able certain conversions.
8) To dis-able certain template instantiations.

For Eg: in the traditonal C++ world, we as class authors wished, that instances of our class should not be allowed for copy construction nor assignment operation, then the only way was to define or declare our own methods under the ' private' access-specifier of the class.

```
class CA   //Traditional C++ class
{
private:
   CA(const CA& );
   CA& operator =(const CA& );
public:
    //all other methods will be defined under public
};
```

A modern approach to address the above stated problem:

```
class CA
{
public:
  CA(const CA& x) = delete;
  CA& operator =(const CA& x) = delete;
};
```

DELETE METHODS (contd...):-

In a modern C++ class, if a copy constructor or assignment function are tagged 'delete', it will not be possible to cop y construct or carry out an assignment operation even inside the class scope.

Design a class whose heap instances shall not be permitted:-

Traditional C++ design, overload the 'operator new' under the private access-specifier of that class.

```
class CA
{
private:
  //..
  void* operator new(size_t size);
public:
  //...
};

//consumer

CA* p = new CA;  //error, private member is not accessible.
```

****

Now, in modern C++ we refactor the above code, declare the method under public and dis-able the same with 'delete
'

```
class CA
{
private:
  //..

public:
  void* operator new(size_t size)=delete;
  //...
};

//consumer

CA* p = new CA;  //error, It is a deleted function
```

OVERRIDE AND FINAL KEYWORDS:-

- The override keyword is applicable to member functions of a POLYMORPHIC class family.
- The final keyword is applicable both at class level and function level.

In a traditional C++ environment it is not possible for any class author to prevent their classes being further inherited
. [Modern C++ we use 'final']

```
class CA{...};

class CB:public CA   //Now the author of 'class CB' cannot prevent a class from inherting it.
{...};

class CC:public CB
{...};
```

Also, there was no facility or construct available in a traditional C++ environment, to prevent any methods being ove
r-ridden in the derived class. For eg:
[Modern C++ we use 'final']

```
class Base
{
public:
  virtual void fun(){ }
};

class Derived1:public Base  //Now, what if Derived1 wants to prevent the 'fun' being over-ridden the derived class?
{                            //It is not possible in the traditional C++ world.
public:
   void fun()   //OVERRIDES the Base 'fun'
   { }
};

class Derived2:public Derived1
{
public:
  void fun()   //OVERRIDES THE Derived1 'fun'
   { }
};
```

What is the guarantee a derived class programmer, who attempts to override the base class virtual function(s), uses t
he correct or precise signature ?

```
class Base
{
public:
  virtual void fun(int x){ }
};

class Derived1:public Base
{
public:
   void fun(int x)   //Signature matches exactly, thereby no issues
   { }
};

class Derived2:public Base
{
public:
  void fun(float x)  //signature mis-match, does not over-ride the base class method.
  { }
};
```

When a such a takes place with function signature mis-match, a Traditional C++ compiler will never issue any error or warning, rather it will smoothly compile and consider the method 'void fun(float)' as a new member function of th
e derived class.

Hypothetically the above 'Derived2' class can be seen as follows:

```
class Derived2:public Base
{
public:
  void Base::fun(int x)
  {  }

  void fun(float x)  //signature mis-match, considered as a new member function in Derived2 class
  { }
};
```


EXTENDED FRIEND DECLARATIONS:
  This modern C++ class feature, provides 3 distinct advantages...
  1) More type safe [trying to qualify a non-existent class as a friend]
  2) Helps in using alias or typedef names to qualify it as a friend
  3) A typename in a generic code can also make use of it in order to qualify itself as a friend to someone.

1)

```
//Loop holes in a traditional approach.
class CB{ } ;
class CC{ };
//..
class CD
{
  friend class CA;    // This statement actually boils down to two diff. constructs...
                      a) class CA;   //acts as a forward declaration
                      b) friend CA; // now this CA becomes friend.
};
```

In the above example, the 'class CA' declaration does not precede the 'class CD' declaration, further there is no guara ntee there would be 'class CA' declaration following the 'class CD' declaration.

If the 'class CA' declaration exists foll: 'class CD', then it makes sense to qualify it as a friend.

What if 'class CA' declaration does not exist foll: 'class CD' ?
In traditional C++ code, it will happily compile, now there is an attempt to qualify a non-existent class a friend.

But,now in modern C++ we can trap such issues... by using the extended friend declaration syntax...

```
class CB{};
class CC{};
//..
class CD
{
  //Extended 'friend' declaration syntax, that avoids the 'class' keyword altogether
  friend CA;   //Error, at the point compiler has never encountered any class by the name 'CA',
          //whether the 'class CA' declaration follows 'class CD' or not, it is immaterial
};
```

What will work ?

```
class CA{};
class CB{};
//...

class CD
{
 friend CA;   //OK, will work as the declaration of 'CA' precedes this statement or 'class CD' declaration.
};
```

2)

In a traditional C++ code, it was not possible to use a typedef name for qualifying it as a friend, for eg:

```
class MyclassCB{..};

typedef MyclassCB CB;   //Now 'CB' is an alias to name to 'MyclassCB'

class CD
{
  friend class MyclassCB;   //ok, no problem,
  friend class CB;  //ERROR, cannot use alias to typedef names to qualify it as a friend.
};
```

Whereas now in modern C++, with extended friend declaration syntax, we can qualify a typedef also a friend if we wish.

```
class MyclassCB{..};
```

```
typedef MyclassCB CB;   //Now 'CB' is an alias to name to 'MyclassCB'

class CD
{
  friend MyclassCB;   //ok, no problem,
     (or)
  friend CB;  //ok, can use typedef names
};
```

3)

In a traditional C++ code it is never possible to qualify a typename as a friend for eg:

```
template<typename T1, typename T2> class CA
{
  //in C++98 code
  friend class T1;    //error, cannot use typenames for qualifying it as a friend
  friend class T2;    //error, cannot use typenames for qualifying it as a friend
};
```

But now, in modern C++ we can achieve the same with extended friend declaration syntax...

```
template<typename T1, typename T2> class CA
{
  //in C++11 code
  friend  T1;    //ok, can use typenames for qualifying it as a friend
  friend  T2;    //ok, can use typenames for qualifying it as a friend
};

CA<int,float> obj1;

  friend int;          //These statements will be ignored automatically as they are not class types
  friend float;        //These statements will be ignored automatically as they are not class types
***************************
CA<CB,CC> obj2;

   friend CB;
   friend CC;
************************
CA<CB, double> obj3;

   friend CB;
   friend double;   //will be ignored
```

DAY - 3:

NESTED CLASS ACCESS RIGHTS:-

In Traditional C++, if we happen to declare one class inside of another class, then we used to call them as nested c lass.

This nesting attempt is not considered as member of the outer class, it is just scoping, instead of global scope, we happen to declare it inside another class scope...As such both the classes are independant of each other.

for eg:

```
class Outer
{
  public:
    //...
  private:
    //..

  class Inner
  {
    private:
      //..
    public:
      //...
  };
};
```

In the above declaration, the class 'Inner' is scoped inside the class 'Outer', as such the public member functions of ' Inner' cannot access the private data of the 'Outer'.
If there is any compulsion, that this nested class called 'Inner' must and should have access to the private members of the 'Outer' class, then we have to qualify the 'Inner' as a friend to the 'Outer'.

```
class Outer
{
  friend class Inner;  //Now the 'Inner' can have access to the private data of 'Outer'
  public:
    //...
  private:
    //..

  class Inner
  {
    private:
      //..
    public:
      //...
  };
};
```

But, now modern C++ a mere nesting of class declaration, it becomes a friend... There is nothing like we have had th e friend declaration statement like in traditional C++ code.

```
class Outer
{
  public:
    //...
  private:
    //..

  class Inner     //By default this enclosed class [Inner] becomes a friend to the enclosing class [Outer]
  {
    private:
      //..
    public:
      //...
  };
};
```

RANGE-FOR CONSTRUCT:

This loop construct is primarily designed to work on collection types, types hold arbitrary or finite quantum of valu es having a definite starting address and an definite ending address.

for eg:
- Could an array type
- could be an instance of STL container class.

```
for(data_type var:collection_instance)
{
  //..
}

list<int> ls1;
//...

//Given the foll: statement the range-for construct will loop thru from 'begin' to 'end' of the list collection,
//for every looping or iteration, it would pick a value and initialize 'x'.
for(int x: ls1)
{
  //...
}
```

How do we specify a particular starting and ending location for a given collection.

```
list<int> ls1;
```

We wish to start from the 40th index and proceed till only the 80th index of the collection

```
auto itr = ls1.begin();

list<int>::iterator start(itr+=40);    // start(itr.operator +=(40));
itr = ls1.begin():
list<int>::iterator end(itr+=80);

for_each(start, end, function_name);
```

How can the following 'class Data' be made range-for compatible type ?

The class has multiple data members of collections types..

```
class Data
{
private:
  int arr[10];
  list<int> ls1;
  vector<float> v1;
  //...
public:
  begin()
  {
  }

  end()
  {
  }
};
```

Let us refactor the above code, such that each collection data-member is an instance of a seperate class...

```
class intArray
{
private:
  int arr[10];
public:
  auto begin(){ return arr; }
  auto end(){ return &arr[10]; }
};

class intList
{
private:
  list<int> ls1;
public:
    auto begin(){ return ls1.begin(); }
    auto end(){ return ls1.end(); }
};

class floatVector
{
private:
  vector<float> v1;
public:
    auto begin(){ return v1.begin(); }
    auto end(){ return v1.end(); }
};
```

Now the modified 'class Data' will look as follows:

```
class Data
{
private:
  intArray ob1;
  intList ob2;
  floatVector ob3;
public:
    //...
    //provide suitable getter functions
    intArray& GetArray()
    {
      return ob1;
    }

    intList& GetList()
    {
      return ob2;
    }

    floatVector& GetVector()
    {
      return obj3;
    }
};

//***consumer code**********
int main()
{
  Data obj1;
                //Data::ob1
  for(auto x: obj1.GetArray())
  {
    //....
  }
            // Data::ob2
  for(auto x:obj1.GetList())
  {
    //..
  }
            // Data::ob3
  for(auto x:obj1.GetVector())
  {
    //..
  }
}
```

FACILITATING A READ-ONLY OPERATION WHILE USING RANGE-FOR CONSTRUCT ON AN COLLEC TION:-

```
class CArray()
{
private:
  int arr[10];
public:
  const int* begin()
  {
    return arr;
  }

  const int* end()
  {
    return &arr[10];
  }
};

***********************
class MyList
{
private:
  list<int> ls1;
public:
    decltype(auto) begin()
    {
      //return ls1.begin();
      return ls1.cbegin();
    }

    decltype(auto) end()
    {
      //return ls1.end();
      return ls1.cend():
    }
};
```

INITIALIZER_LIST type:-

Now in modern C++11 standards onwards, we are provided with a light-weight wrapper class, whose instance can accomodate or hold arbitrary number of values of homogenous types.

An instance of this type becomes very handy providing lot of flexibilities both to the class author as well to the cla ss library consumer, particularly the data members of class are all collection types, and they have initialized or popul ated by the class consumer.

```
//Traditional C++ approach

class CA
{
private:
  int arr[100];
public:
  //need to provide a constructor to initialize the above array...

  CA(int* p)
  {
    for(int i=0;i<100;i++)
    {
      arr[i] = p[i];
      i++;
    }
  }

  CA(int x1, int x2, int x3... int x10):a[0](x1), a[1](x2),.... a[9](x10)
  {
  }
};

//***consumer code****
int main()
{
  CA obj1(10,20,30,40...);  //approach-1

  CA ar[10] ={....};
  CA obj1(ar);   //approach-2
}
```
****

Now in modern C++ it is far more easy to accept arbitrary number of parameters to initialize any collection type data members of a class.

```
class CA
{
private:
  list<int> ls1 = {10,20,30,40,50,50};
};
```

The brace initialization syntax or the uniform initialization syntax by the compiler is inferred in different ways depen ding on the context where and how it is used.

for eg:
CONTEXT-1 when declaring primitive type variables...

```
int a=10;
int a(10);
//modern approach
int a{10};   // here the brace syntax is inferred as initialization syntax for the primitive variable.

//local variables
int x1;    //will hold garbage value
int x2{};  // will hold zero.
```

CONTEXT-2, when used with new operator function, for creating an array of instances

```
CA* p = new(nothrow) CA[5]{{},{10},{10,20},{100},{} };
```

   Given the above statement the inference by the compiler is to pass each parameter to its respective constructors.

CONTEXT-3, When used with STL container objects...

When STL container instances are initialized with brace initialization syntax, then the inference by the compiler is to first convert the complete range of values in the braces to the type 'initializer_list'.

```
list<int> ls1 ={10,20,30,40,50,60};
            = initializer_list<int>{....};
list<int> ls1(initializer_list<int>{....});

vector<int> v1 ={100,200,300,400};
            = initializer_list<int>{....};
vector<int> v1(initializer_list<int>{....});
```

Note: Every STL container classes have now been provided with additional overloaded constructor in their classes, t hat take 'initializer_list' as its parameter

```
template<typename T> class list
{
public:
  //...
  list(initializer_list<T>& ob)
  {
    //...
  }
};
```

MODERN C++ TEMPLATE FEATURES:

Default arguments for function templates:-

In traditional C++ there was no support for default types while defining a function template, The default types were only permitted for class templates...

for eg:

```
    //function template
    template<typename T=double> void fun(T X)  // error,in the typename list we could not provide the default type
    {... }
```

But, the same was permitted only for class templates...

```
    template<typename T = double> class CA   //default type allowed for class templates in C++98/2003 standards
    { };
```

But, now in modern C++ we can not only provide default arguments for a generic function, as well we can also pro vide default type for a generic function, for eg:

The default type in the typename list and the default argument to the formal parameter of a generic function has to t ype compatible.

```
 template<typename T = double> void fun(T x=34.12)
    {... }
```

****
RVALUE & LVALUE types:

Entities that can be present either on the left hand side of an equation or on the right hand side of an equation is an L VALUE type [stands for LEFT HAND SIDE VALUE]. For eg: It could be any named entity.

Entities that can only be present on the right hand side of an equation are known as RVALUE types [stands for RIG HT HAND SIDE VALUE]. For eg: It could be any literal (numeric/non-numeric/class type), entities that generally d o not have a name. Even the return value of a function is RVALUE, because this temporary does not have any name.

```
int fun()
{
 int a;   //LVALUE
 //...
 return a;    //assume it is returning 100
}


int main()
{
 int result;   //LVALUE
 //..
 result = fun();    // result = 100;
}
```

Let us consider the foll: equations...

```
int a,b;
a = 100;           // LVALUE = RVALUE
b = a;             // LVALUE = LVALUE

100 = a;      //error
```

Now in modern C++ a reference type entity can be categorized into 2 diff. forms or types..

```
        REFERENCE TYPE
              |
      |----------------|
    lvalue           rvalue
```

What is the syntax for declaring a lvalue and rvalue reference types...

LVALUE REFERENCE TYPE: At the point of declaration of an entity, prefixing a single '&' is termed as LVALUE reference type.

```
int a;
int& ref1 = a;  //ok
int& ref = 100; //error
```

RVALUE REFERENCE TYPE: At the point of declaration, prefixing it with a 2 '&' is termed as RVALUE referenc e type.

for eg:

```
int&& ref1 = a;
int&& ref2 = 100;
```

So, in a modern C++ class we can have a pair of copy constructor and a pair of assignment function.

```
class CA
{
public:
  //LVALUE PAIR
  CA(const CA& x);
  CA& operator = (const CA& x);

  //RVALUE PAIR
  CA(CA&& x);
  CA& operator =(CA&& x);
};
```

REFERENCE COLLAPSING:-

```
template<typename T> void fun3(T& x)   // void fun3(int&& & & x)   //int&
{

}

template<typename T> void fun2(T& x)   // void fun2(int&& & x)     ===>int&
{
   fun3(x);
}

template<typename T> void fun1(T& x)  // void fun1(int&& x)    ===> int&
{
   fun2(x);
}

template<typename T> void fun(T& x)  // void fun(int& x)  T ===> int&
{
  fun1(x);
}

//***consumer code******
int main()
{
  int a;
  fun(a);
}
```

Now that in modern C++, both a single '&' and a double '&&' have special meaning, and during template instantiatio ns by the compiler yeilds more than 2 '&', then how many should they ignore and how many should be retained ?

The decision was simple:
  If the '&' count yeilded odd number - retain one
  if the '&' count yeilded even number - retain two

```
int a;
int& ref = a;  // The term 'int& ' means it is an alias type to an integer variable

int& ref = 100;  //error, 'ref' is declared to an alias to an integer variable, 100 is a integer const.
```

correction or solution for the above issue..

```
const int& ref = 100;  //ok, as 'ref' is declared to be an alias to an 'const int'
```

Note:
  An LVALUE reference type would only act as an alias to an LVALUE type, where as an RVALUE reference type will act as an alias to both LVALUE type as well as RVALUE types.
****
A function with RVALUE reference type would be more scalable when compared to LVALUE Reference type:

```
//Traditional approach

template<typename T1, typenameT2> class Mydata
{
  private:
   vector<T1> v1;
   vector<T2> v2;
 public:
  void Populate(T1& x, T2& y)
  {
  v1.push_back(x);
  v2.push_back(y);
  }
 void Populate(T1& x, const T2& y)
  {
  v1.push_back(x);
  v2.push_back(y);
  }
 void Populate(const T1& x, T2& y)
  {
  v1.push_back(x);
  v2.push_back(y);
 }
 void Populate(const T1& x, const T2& y)
 {
  v1.push_back(x);
  v2.push_back(y);
 }

};
```

Modern Approach - No overloading of 'Populate'

```
template<typename T1, typenameT2> class Mydata
{
  private:
   vector<T1> v1;
   vector<T2> v2;
 public:
  void Populate(T1& x, T2& y)
 {
  v1.push_back(x);
  v2.push_back(y);
 }
};
```

MOVE COPY AND MOVE ASSIGNMENT FUNCTION:-

In a traditional C++ class, we don't happen define a copy constructor or an assignment function, then the compiler is
expected to generate one for the class under the 'public' access specifier. This compiler generated methods employ a
business plan which is also termed as BIT-WISE / MEMBER-TO-MEMBER and at times also called SHALLOW o
peration.

If in-case the programmer decides to define their own copy constructor or assignment function, then the compiler wil
l not assume any. Further, the programmer defined methods will employ algorithms as defined by the programmer, t
hat may be
  -BIT-WISE / MEMBER-TO-MEMBER and at times also called SHALLOW operation (or)
  -DEEP COPY/ DEEP ASSIGNMENT

  The general practice is, if the programmer wishes for BIT-WISE operation, then they would generally depend the c
ompiler synthesized methods, and if the programmer wished for DEEP copy or DEEP assignment operation, then th
ey would be compelled to define their own custom copy constructor and assignment function.

  Compiler Generates: bit-wise/shallow
  Programmer defines: deep copy or deep assignment.

Now in modern C++ the programmer has the flexibility of having both pairs of COPY CONSTRUCTOR and ASSI
GNMENT function operations defined in a single class.

   Desires to go for DEEP copy or DEEP ASSIGNMENT operation ?
      LVALUE copy constructor and LVALUE ASSIGNMENT FUNCTION:

```
      class CA
      {
        public:
          //...
          /*
            - It is always a READ-ONLY business on the SOURCE, and a WRITE operation on the TARGET
          */
          CA(const CA& x);
          CA& operator =(const CA& x);
      };
```

Desires to go for shallow/bit-wise operation ?
RVALUE copy constructor and RVALUE ASSIGNMENT function:

```
    class CA
    {
      public:
        //...
        /*
          - It is READ cum WRITE business on the SOURCE, and a WRITE operation on the TARGET
        */
        CA(CA&& x);
        CA& operator =(CA&& x);
    };
```

LVALUE/RVALUE pairs of copy constructor/assignment functions (contd...):-

If the author of the class decides to provide an option to the class consumer, whether they want to go for a DEEP cop
y or DEEP assignment operation only. Then the class would be introduced only with LVALUE pairs of copy constru
ctor and assignment function.

If the author of the class decides to provide an option to the class consumer, whether they want to go for a SHALLO
W copy or SHALLOW assignment operation only. Then the class would be introduced only with RVALUE pairs of
copy constructor and assignment function.

At times the programmer of a class may provide both options to the class consumer.

Note: The decision of deciding between LVALUE pairs and RVALUE pairs always arises only in the context of a cl
ass exhibiting PIMPL model.

****

Design a class that is MOVE-ONLY type.

```
class CA
{
private:
  //...
public:
  //...
  CA(const CA& x) = delete;
  CA& operator =(const CA& x)=delete;

  CA(CA&& x){ }
  CA& operator =(CA&& x){  }
};
```

VARIADIC TEMPLATES:

- A variadic function template

```
   template<typename... T> void fun(T ...Args) { }
         (or)
   template<typename  ...  T> void fun(T ...Args) { }
         (or)
   template<typename ...T> void fun(T ...Args) { }

   //Implementation of the above function

   fun(10,45.12f, 'a', 89);
```

2nd form of a variadic function

```
    template<typename T, typename... PACK> void fun(T x, PACK ...Args)
    {
      // Where T is non-variadic and
      // PACK is a variadic
    }
```

In order to process the arbitrary number of values received by a variadic function, the process is as follows:

- Unpack the parameter pack (variadic element) and pick one value
  : We need to go for a recursive call.
- In order to hold or receive the value that just got unpacked from the parameter pack we provide a placeholder variable as the first formal parameter of the function.

The above 2 process should continue for how long ?
- As long as there is only 1 value in the parameter pack (or)
- The pack is empty..

When the above circumstances arise, we need to break or come out of the recursion process. For which purpose we provide an additional overload function that either takes a single formal parameter or no parameter.

****

FORM-1: Suitable to accept arbitrary number of arguments of homogenous type...

```
   template<typename... T> void fun(T ...Args) { }
```

FORM-2: Suitable to accept arbitrary number of arguments of heterogenous types...

```
   template<typename T, typename... PACK> void fun(T x, PACK ...Args)
    {
      // Where T is non-variadic and
      // PACK is a variadic
    }
```

A CALLBACK FUNCTION:-

```
void Callback(void(*ff)())
{
}
```

In the above function, the consumer has no choice with regard to the function, each time one tries make a call or implement the above function, the consumer is mandated to always pass address of a function of the form 'return-type void and input-type void'.

How about designing a callback function that provides the choice of the function form to the consumer ?
Answer is - make use of variadic function templates in modern C++.

Var args. function...
Now in modern C++ we have variadic templates...
We can design or write generic code that can handle arbitary numbers of parameters, these parameters may be homo genous or heterogenous types.

-Variadic function templates
-Variadic class templates

The form of a variadic function template can as follows:

```
template<typename... PACK> return_type function_name(PACK ...args)
{
        // PACK ...args   --> Arbitary numbers of inputs of the type PACK
}

(or)
template<typename ... PACK> return_type function_name(PACK ...args)
{

}

(or)
template<typename ...PACK> return_type function_name(PACK ...args)
{

}
```

// Processing the values received by a variadic function....

//Approach-1
-The idea is to have additional formal parameter that act's as place holder for every value that we plan to un-pack fr om the variadic collection.
- In-order to un-pack the elements of the variadic collection or list, we try to make a recursive call to the function, d uring each call it is expected to un-pack one value from the collection.
- The recursion has to break when either the pack is empty or is holding the last or a single value, for which we nee d to have another overloaded function to address this issue.

For eg:

```
  //overloaded function
  template<typename T> void Fun(T x)
  {

  }

  template<typename T, typename... PACK> void Fun(T x, PACK ...Args)
  {
     cout << x;
     Fun(Args...);  // Fun() (or) Fun(one_value)
  }
***************************************************************************
```

```
//consumer end
Print(10,56,89,43.21,'a','b',98.21f);
```

stage-1

```
void Fun(T x=10, PACK ...Args =(56,89,43.21,'a','b',98.21f))
{
  cout << x;
  Fun(Args...);  // Fun(56,89,43.21,'a','b',98.21f);
}
```

stage-2

```
void Fun(T x=56, PACK ...Args =(89,43.21,'a','b',98.21f))
{
  cout << x;
  Fun(Args...);  // Fun(89,43.21,'a','b',98.21f);
}
```

stage-3

```
void Fun(T x=89, PACK ...Args =(43.21,'a','b',98.21f))
{
  cout << x;
  Fun(Args...);  // Fun(43.21,'a','b',98.21f);
}
//...
```

Last but one stage:

```
void Fun(T x='b', PACK ...Args =(98.21f))
{
  cout << x;
  Fun(Args...);  // Fun(98.21f); --> we call the overloaded funct.
}
```

DAY - 4:

FOLD EXPRESSIONS:

```
template<typename... Args> auto SumVal(Args ...Data)
{
 return (Data + ...);
}
```

Hypothetical instantiation of the above variadic function for the call statement below:

```
cout << SumVal(10, 20, 30, 40, 50) << endl;

template<> auto SumVal(int x1, int x2, int x3, int x4, int x5)  // SumVal... HHHHH
{...}

cout << SumVal(10, 20.32f, 30, 40.67, 50) << endl;

template<> auto SumVal(int x1, float x2, int x3, double x4, int x5)  // SumVal... HMHNH
{...}
```

****
STAGE-1

```
template<typename... Args> auto SumVal(Args ...Data)   //  ...Data  ===> (10, 20, 30, 40, 50)
{
 return (Data + ...);    // return (10 + [20, 30, 40, 50]);
 //        x1 + ...);    // return (x1 +[x2,x3,x4,x5]);
}
```

STAGE-2

```
template<typename... Args> auto SumVal(Args ...Data)   //  ...Data  ===> (10, 20, 30, 40, 50)
{
 return (Data + ...);    // return (10 + 20 + [30, 40, 50]);
 //
}
```

stage-N

```
template<typename... Args> auto SumVal(Args ...Data)   //  ...Data  ===> (10, 20, 30, 40, 50)
{
 return (Data + ...);    // return (10 + 20 + 30 + 40 + [50]);
}
```

Unary fold:

Homegenous operation on all the values in the parameter pack.

Prefer Unary fold when the operation on the values held in the parameter pack is same, and the operation with a 2nd element ('z') is different from the operations on the values inside the pack.

USAGE: It is a addition on the elements in the pack
It is a multiplication or something else on the second element with total sum values of the pack

```
auto z=100;
//Unary fold with addition on pack elements and a product of 'z'
return z * (pack + ...);  // ---> z * (Args1 + (... + (ArgsN-1 + ArgsN)))
```

Binary Fold:
Prefer Binary fold when the operation on the values held in the parameter pack and the 2nd value is also the same.

```
//binary fold  with addition operation on both the pack elements and 'z'
return (z + ... + pack);   //Binary Left fold --> (((z + Args1) + Args2) + ...) + packN
```

FOLD EXPRESSION INSTANTIATIONS:

```
template<typename... Args> auto SumVal(Args ...Data)
{
 return (Data + ...);
}

template<int> auto SumVal(int x1, int x2, int x3, int x4, int x5)
{...}
```

G++:- _Z6SumValiiiii

Hypothetical instantiation of the above variadic function for the call statement below:

```
cout << SumVal(10, 20, 30, 40, 50) << endl;
****************************************************
template<typename... Args> auto SumVal(Args ...Data)
{
 return (Data + ...);
}

template<int> auto SumVal(int x1, int x2)
{...}
```

Hypothetical instantiation of the above variadic function for the call statement below:

```
cout << SumVal(10, 20) << endl;
-----------------------------------------------
template<typename... Args> auto SumVal(Args ...Data)
{
 return (Data + ...);
}

template<float> auto SumVal(float x1, float x2, float x3, float x4)
```

DECORATED NAMES FOR THE ABOVE INSTANTIATION
MSVC : ??SumVal@@YAMMMMM@Z
G++ : _Z6SumValffff

```
SumVal(45.12f, 56.12f, 89.21f, 99.45f);
------------------------------------------------------------
template<typename...Args> auto Compute(Args... pack)
{

}

template<typename T1, typename T2, typename T3, typename T4> auto Compute(T1 x1, T2 x2, T3 x3, T4 x4)
{

}

template<int, int, double, float> auto Compute(int x1, int x2, double x3, float x4)
{

}

auto result= Compute(20,49,12.34,5.12f);

---------------------------------------------
template<typename... PACK> auto SumVal(PACK... Data)
{
   return (Data + ...);      //  return  (10+(20+(30+(40+50))));
}

SumVal(10, 20, 30, 40, 50)
------------------------------------------------------------------------
```

```
template<typename... Pack> void Process(Pack... Args)
{
   auto lm =[](auto& x){cout << x << endl;};  //C++14 -Generic lambda

   (lm(Args), ...);   //Fold over comma
}


(lm(Args),...);   ===>  (lm(10),lm(20),lm(30),lm("cpp"),lm(50.56f))

Process(10,20,30,"cpp",50.56f);

------------------------------------------------------------
template<typename... Args> void VariadicFun1(Args... Data)
{
 (cout << ... << std::forward<Args>(Data));
 cout << endl;
}
**********************************************
                                      56
cout << (12, 56, 87.90, 'c') << std::forward<int>(Data));

                                   12
cout << (56, 87.90, 'c') << std::forward<int>(Data));

                                56
cout << (87.90, 'c') << std::forward<int>(Data));

                            87.90
cout << ('c') << std::forward<double>(Data));

                          c
cout << () << std::forward<char>(Data));
********************************************************
int main()
{
 VariadicFun1(56, 12, 56, 87.90, 'c');
}

------------------------------------------------------------------------
//Generic class

template<typename T> class CA   //primary template
{
  //...
};

//The above generic class can also be specialized...
template<typename T> class CA<T*>   //partial specialization
{
  //..
};

template<> class CA<char>   //full specialization
{
```

```
//...
};
---------------------------------
template<typename T> class CA {  };

//The specialized class for 'char', shall also derive from the primary template class
template<typename T> class CA<char> : public CA<T>
{ };
-----------------------------------------------------------
template<typename... PACK> class CA
{
   //A variadic class template
};
```

(or)

```
template<typename T, typename... PACK> class CA
{
  //A variadic class template
};
----------------------------------------------
//A variadic generic class that facilitates in realizing multi-level inheritance while instantiation.

template<typename... PARAMETER_PACK> class CA
{
public:
   CA(){cout <<"Base class CA" << endl;}
};

template<typename PARAMETER_TYPE, typename ... PARAMETER_PACK>
                class CA<PARAMETER_TYPE, PARAMETER_PACK...> : private CA<PARAMETER_PACK...>
{
private:
 PARAMETER_TYPE data;
public:

 CA(PARAMETER_TYPE h, PARAMETER_PACK... args) :CA<PARAMETER_PACK... >(args...), data(h)
 {
  cout << "No. of elements left further ..." << sizeof...(args) << endl;
  cout << "data " << data << endl;
 }
};

//consumer code
CA<int, float, double> obj1(10,12.34f, 56.12);
```

The above generic class instantiations by the compiler will be in the following order....

```
class CA<>
{
public:
  CA(){cout <<"Base class CA" << endl;}
};

class CA<double>:private CA<>
```

```
{
private:
  double data;  //56.12
public:
  CA(double h,[]):CA<>(),data(h)
  {
    //..
  }
};

class CA<float>:private CA<double>
{
private:
  float data;  //12.34f
public:
  CA(float h, [56.12]):CA<double>([56.12]),data(h)
  {
    //..
  }
};

class CA<int>:private CA<float,double>
{
private:
  int data;  //10
public:
  CA(int h, [12.34f,56.12]):CA<float,double>([12.34f,56.12]),data(h)
  {
    //..
  }
};
```

****

```
CA<int, float, double> obj1(10,12.34f, 56.12);

template<typename PARAMETER_TYPE, typename ... PARAMETER_PACK>
          class CA<PARAMETER_TYPE, PARAMETER_PACK...> : private CA<PARAMETER_PACK...>
          { };
```

Hypothetical view of the above generic declaration for the instance 'obj1' can be perceived like this....

```
template<int, [float,double]> class CA<int, [float,double]>:private CA<float,double>
{ };

//MULTILEVEL INHERITANCE to unwind the parameter-pack of a variadic class template...
class CA<double>:private CA<> { };
class CA<float>:private CA<double>{ };
class CA<int>:private CA<float> { };
```

In certain problem situations we can also go for MULTIPLE INHERITANCE to unwind the parameter-pack of a var iadic class template.

A variadic class template would be useful when we want to achieve compile-time strategy.

If the classes that is being consumed for achieving compile-time strategy comes from the same vendor,i.e all memb er functions across the different classes are same by name and signature, then
- Design the variadic class for MULTI-LEVEL inheritance.
for eg:

```
  template<typename...PARAMETER_PACK> class CA
  {
  };
  template<typename PARAMETER_TYPE, typename... PARAMETER_PACK>
     class CA<PARAMETER_TYPE,PARAMETER_PACK...>:private CA<PARAMETER_PACK...>
     {

     };
```

If the classes that is being consumed for achieving compile-time strategy comes from different vendors, i.e. the me mber functions across the different classes have different names or signatures..., then
- Design the variadic class for MULTIPLE-INHERITANCE.

For eg:

```
template<typename... PARAMETER_PACK> class CA:public PARAMETER_PACK...
{
};
```

****

```
template<typename...INSTRUMENT_TYPES> class Compose
{
public:
 void play_music() { cout <<"completed...." << endl;}
};

template<typename INSTRUMENT, typename ... INSTRUMENT_TYPES>
        class Compose<INSTRUMENT, INSTRUMENT_TYPES...> : private Compose<INSTRUMENT_TYPES...
>
{
private:
 INSTRUMENT object;
public:
 void play_music()
 {
  object.play();
  Compose<INSTRUMENT_TYPES...>::play_music();    //Compose<>::play_music();
 }
};

Compose<Drums, Piano, Flute, Guitar> music4;
```

The instantiation of the above variadic class template for the instance 'music4' above is as follows:

```
class Compose<>
{
private:

public:
  void play_music() { cout <<"completed...." << endl;}
};

class Compose<Guitar>:private Compose<>
{
private:
  Guitar object;
public:
  void play_music()
  {
    object.play();
    Compose<>::play_music();
  }
};

class Compose<Flute>:private Compose<Guitar>
{
private:
  Flute object;
public:
  void play_music()
  {
    object.play();
    Compose<Guitar>::play_music();
  }
};

class Compose<Piano>:private Compose<Flute,Guitar>
{
private:
  Piano object;
public:
  void play_music()
  {
    object.play();
    Compose<Flute,Guitar>::play_music();
  }
};

class Compose<Drums>:private Compose<Piano,Flute,Guitar>
{
private:
  Drums object;
public:
  void play_music()
  {
    object.play();
    Compose<Piano,Flute,Guitar>::play_music();
  }
};
```
****

A variadic class template for multiple-inheritance.

```
template<typename... Policy> class Music :public Policy...  //Derive from the parameter pack
{   };
```

```
Music<Flute, Drums,Piano> obj1; [instantiation for this object]

class Music:public Flute, public Drums,public Piano
{ };
-------------------------
Music<Flute> obj2;  [instantiation for this object]

class Music:public Flute
{ };
---------------------------------
Music<Guitar, Drums,Violin> obj3;  [instantiation for this object]

class Music:public Guitar, public Drums,public Violin
{ };
----------------------------------------------------
```

NON-TYPE parameters with class templates

```
template<typename T> class CA
{
private:
  T arr[10];   //No matter what the data type of the elements are, the size is always fixed.
};

CA<int> obj1;   // int arr[10];
CA<float> obj2; // float array[10];
```

What if the class consumer wants a provision for also deciding the size of the array.
The size of the array is signed integer value and a compile-time constant [Size of the array must be resolvable by the compiler during compile-time]

```
template<typename T, int SIZE> class CA
{
private:
  T arr[SIZE];   // 'SIZE' the non-type parameter acts as a COMPILE-TIME  constant.
};
//consumer...
CA<int, 10> obj1;   // int arr[10];
CA<char,5> obj2;    // char arr[5];

constexpr int SIZE=10;
int arr[SIZE];
//...
----------------------------------------------------------------
```

Memory management classes in Modern C++:-
  - These classes employ certain C++ language features along with OO design's.

    Language features:
        - Class templates
        - Function overloading
        - Operator overloading
        - Conversion functions
    Design features:
        - PIMPL model
        - RAII model

- Smartpointer design.

The modern C++ library now supports 3 different types of memory management classes...

1) unique_ptr
2) shared_ptr
3) weak_ptr

#include<memory>

Unique_ptr:

It is an instance that would help us hold the address of a heap instance.

Memory allocation request is explicitly made by the programmer, upon successful allocation this address is pr ovided to the unique_ptr instance.

There upon this unique_ptr instance becomes the exclusive owner of that heap. This heap address is not allow ed to be shared across different unique_ptr instance.

In other-words a unique_ptr instance is a move only type, it cannot be lvalue copy constructed.

By applying a move operation on this instance, the ownership gets transferred, at any given point of time the h eap resource would be pointed to by only one unique_ptr instance.

The de-allocation happens automatically when the unique_ptr instance perishes.
[The heap memory life-time is directly equal to the life-time of unique_ptr instance]

```
template<typename T> class unique_ptr
{
private:
  T* pointee;
  void(*dl)(T*) = &default_delete<T>;  //in-class initializer feature
public:
    unique_ptr():pointee(nullptr){ }
    unique_ptr(T* x):p(x){ }
    //...
    //...
    auto& get_deleter()
    {
      return dl;
    }

    T* get()
    {
      return pointee;
    }

    unique_ptr(const unique_ptr& x)=delete;
    unique_ptr& operator =(const unique_ptr& x)=delete;

    //move operations
    unique_ptr(unique_ptr&& x):pointee(x.pointee)
    {
      x.pointee = nullptr;
    }

    unique_ptr& operator =(unique_ptr&& x)
    {
      if(pointee != nullptr)
      {
        delete pointee;
      }
       pointee=x.pointee;
       x.pointee = nullptr;
    }

    //..
    operator bool(){ }
    operator->();
    //...
    ~unique_ptr()
    {
     (*dl)(pointee);
    }
};

template<typename T> void default_delete(T *x)
{
 if(p != nullptr)
    delete p;
}
```

```
  Traditional usage:
      int* p1 = new(nothrow) int;
      //..
      delete p1;

  Modern usage:
       unique_ptr<int> p1 = make_unique<int>();   // make_unique() is C++14 function
              (or)
       unique_ptr<int> p2 = new int{};  //error, deprecated

       unique_ptr<int> p2 = unique_ptr<int>(new int{});
```

When a class exhibits PIMPL model, it also a good practice to introduce a conversion function 'operator bool'. All memory management classes have 'operator bool' overloaded as member function.

```
class Pointee
{
private:
  int* p;
public:
    Pointee():p(nullptr){ }
    Pointee(int* q):p(q){ }
    ~Pointee()
    {
      if(p != nullptr)
      {
        delete p;
      }
    }
    void operator(int* q){ p = q; }

    void print(){ cout <<"data is :" << *p << endl; }

    operator bool()
    {
      if(p != nullptr)
        return true;
      else
        return false;
    }
};

//***consumer code***********
int main()
{
   Pointee obj1(new(nothrow) int(100));
   //if(obj1.p != nullptr)
   if(obj1)          // if(obj1.operator bool())
      obj1.print();
   else
      //...
   //...

   Pointee obj2;
   //...
   //...
   if(obj2)
      obj2.print();
   else
      //....
return 0;
}
```

From a unique_ptr as source to a raw_pointer as target, we can employ
  - get() function (or)
  - release() function of the unique_ptr class.

From a raw_pointer as the source and a unique_ptr as the target, we can employ
    - reset() function of unique_ptr class.

When we happen to define unique_ptr instances that should point or delegate to a custom delete handler, and the hea p memory resource is incidentally being accquired with the help of make_unique function and not the 'new' operator function, then we need to use the member function called 'get_deleter' to provide the address of the custom delete ha ndler function with respect to the unique_ptr instance.

Why make_unique call is better than 'new' operator call ?

```
void fun(int* p1, float* p2, double* p3)
{
  //...
  delete p1;
  delete p2;
  delete p3;
}

int main()
{
    fun(new(nothrow) int, new(nothrow) float, new(nothrow) double);  //NOT SAFE, memory leaks
          [Suc..]              [Suc..]          [Broke...]
    //...
}
```

Alternate, better or modern idea...

```
void fun(unique_ptr<int> p1, unique_ptr<float> p2, unique_ptr<double> p3)
{
  //...

}

int main()
{
    fun(make_unique<int>(), make_unique<float>(), make_unique<double>());  //Very safe, no memory leaks
          [Suc..]              [Suc..]              [Broke...]
    //...
}
```

A call to make_unique will always yield a nameless 'unique_ptr' type instance
----
CONSTANT EXPRESSION:

```
  int Sum(int x)
{
   //...
   return x+100;
}


//consumer code
//...
int result = Sum(a);    //int result = call Sum(a)

     @@@@@@@          //Prolog
     @@@@@@
       $$$$$$$$$$     //Business logic
       $$$$$$$$$$
     !!!!!!!         // Epilog
     !!!!!!!

int result2 = Sum(10);
     @@@@@@@          //Prolog
     @@@@@@
       $$$$$$$$$$     //Business logic
       $$$$$$$$$$
     !!!!!!!         // Epilog
     !!!!!!!
```

```
constexpr int Sum(int x)
```

```
{
   //...
   return x+100;
}


//consumer code
//...
int result = Sum(a);    //int result = call Sum(a)

     @@@@@@@          //Prolog
     @@@@@@
       $$$$$$$$$$     //Business logic
       $$$$$$$$$$
     !!!!!!!          // Epilog
     !!!!!!!

constepxr int result2 = Sum(10); /// int result2 = 110;
```

DAY - 5:
----
unique_ptr(contd..);

  - get() method of unique_ptr, extracts the raw_pointer content from the unique_ptr instance, useful or recommende
d only under circumstances when the heap resource owned by this unique_ptr instance should serve as a parameter t
o a legacy code, whose formal parameter is a raw_pointer element. Further this business plan of this legacy code is s
uch that it wishes to only access the heap resource to carry out the desired business, as such this function is not keen
acquiring ownership of this heap resource owned by the unique_ptr.

  - release() method of unique_ptr, not only extracts or provides the raw_pointer content of the unique_ptr instance, a
lso dis-owns or transfers the ownership of the heap resource to a legacy code, taking a raw_pointer as its formal para
meter, now it is the duty of this legacy code which has acquired the ownership of the heap resource to de-allocate the
 same at a suitable time.

Note: In General the above 2 methods help us extract the raw_pointer info from an unique_ptr instance.

Normally when we have an unique_ptr instance, fetching heap memory in a modern approach would be like...

```
  1) call the make_unique<> function
     unique_ptr<IData> ptr;
     ptr = make_unique<DataA>();

  2) call the new operator and cast the return type to a unique_ptr type
     unique_ptr<IData> ptr;
     //...
     ptr = new DataA;  //error
     ptr = unique_ptr<DataA>(new DataA);  //ok
  3) As an application developer we wished to consume a helper member function of a legacy class code, then what ?


     template<typename T> class Factory
     {
     public:
       static T* GetData()
       {
        return new T;
      }
    };
```

since the above static method returns a data_type* kind, such return result cannot be directly accomodated onto a unique_ptr instance, we need cast it to a unique_ptr kind.

```
    unique_ptr<IData> ptr;

    ptr = Factory<DataA>::GetData();  //error, equivalent to : ptr = new DataA;
    ptr = unique_ptr<DataA>(Factory<DataA>::GetData());
```

The above statement, can be avoided with the help of 'reset' member function of unique_ptr.

```
     unique_ptr<IData> ptr;
     //..
     ptr.reset(Factory<DataA>::GetData());
```

The 'reset()' has 2 overloaded forms...
1) reset() member function accepts a raw_pointer as input and suitabily converts this raw_pointer to the actual targe t type that the unique_ptr has been constructed for.

2) A 'reset()' call without any parameter, this member function call actually helps in de-allocating the heap memory owned by him, well before the unique_ptr instance could perish.

A scenario where a call to the 'reset()' function with no parameter of unique_ptr class would be helpful:

Let us assume we have a function, whose life-time is very large. Within the scope of such a function a request for heap memory is quite intensive, every heap resource that is acquired, its handle is provided to distinct unique_ptr ins tances, as it is very safe, ensuring no memory leak or dangling pointer and even exception safe.

Given the nature of this function being heap-intensive in its operations, if there a circumstance or situation where at a particular stage of the function execution we realize the earlier heap allocations made and owned by a few uniqu e_ptr instances is of no more use, as the desired function or business on those heap resources have already been achi eved. It would be a good-idea and under such situations, we programmers take control of the life-time of the heap in stances owned by the unique_ptr handles and not wait for the unique_ptr to have its natural destruction or death,as th e functions life-time is very huge.

MANAGING ARRAY TYPE INSTANCES with unique_ptr handles:-

When acquiring an array of heap instances, which are likely to be owned by a unique_ptr instance, these instances will automatically call the suitable handler functions in their destructor methods to de-allocate the array of heap reso urces owned by them, while they are about to perish on the stack.

If there are problem situations where we programmers would like to take control of this de-allocation process, pri marily for a reason, that we wished to accomplish some very important business just before the array of heap instanc es owned by the unique_ptr is about to be de-allocated, then we define our own delete-handler for the same, and furt her at the time of creating the unique_ptr instance we provide the custom delete-handler as a parameter, so that when the unique_ptr instance is about to perish, it would call our own custom delete-handler and not the built-in to achiev e the desired goal.

```
   template<typename T1, typename T2 = default_delete<T1>> class unique_ptr
   {
```

```
  public:
    unique_ptr(T1* x, T2 x)
    {  }

    //Expected, but does not have one for move operation
    unique_ptr(T1&& x, T2 x)
    { }
  };


 unique_ptr<...>  ptr2(ptr1.release(),Arr_deleter);

 unique_ptr<...>  ptr2(std::move(ptr1),Arr_deleter);  //error, suitable constructor
```

Member function 'get_deleter' of unique_ptr class returns a reference to the default_delete handler data member.

With the help of this member function, we can provide or assign custom delete handlers for any unique_ptr object alr eady constructed with default_delete handlers.

SHARED_PTR:-
A memory management class that can share or multiple shared_ptr instances can point to the same heap resource. An instance of shared_ptr is LVALUE copyable and LVALUE assignable.
When multiple instances of shared_ptr happen to point to the same heap instance then, which shared_ptr will de-all ocate the heap memory when they are losing scope or perishing ?

The design of the shared_ptr memory management class takes care of this issue. The shared_ptr instances are also c alled as reference-counted types. Meaning, there is a count internally maintained indicating as to how many shared_ ptr instances are pointing to a common heap resource.

When a shared_ptr instances happens to perish, this instance will decrement this reference-count value by one, and further a check would be made to see if this decremented value is zero, if so a de-allocation of the heap resource will take place, else not.

A shared_ptr type instance handle would be recomended particularly when an expensive heap resource scope has to be extended across different functions, without duplicating this expensive heap resource and still be exception safe, the last shared_ptr instance that is about to perish would de-allocate the heap instance.

This shared_ptr instances apart from allocation heap memory for the type of data we desire, they also do additional heap memory allocation for internal book-keeping of the reference count. This book keeping information block of he ap memory is also called as CONTROL-BLOCK.

The control-block is designed to hold few informations...
- handle to the actual heap instance which it is supposed to manage.
- A count on how many shared_ptr instances are sharing the same heap resource [strong count].
- A count on how many weak_ptr instances are sharing the same shared_ptr state [weak count].
- A handle to the delete-handlers...

What is the life-time of the actual heap instance that we desire to manage on the heap w.r.to a shared_ptr?
The actual life of the heap instance that we wished the shared_ptr manages, depends upon the strong-count value in the control block. If the strong count is one or greater, the heap instance would be residing on the heap memory. I f the strong count drops to zero, the heap instance will get de-allocated.

What is the life-time of the control-block on the heap, shared by different shared_ptr/weak_ptr instances ?

The actual life-time of the control block is, the moment both strong-count and weak-count fall to zero the control -block will get de-allocated.
****

When will the shared_ptr instances have a seperate or distinct control blocks for them ?

- A shared_ptr instance upon creation or construction is being initialized with a 'new' operator call.
- A shared_ptr instance upon creation or construction is being initialized with a raw_pointer
- A shared_ptr instance upon creation is being initialized with a make_shared<>() function call.

```
shared_ptr<datatype> ptr(new...);
shared_ptr<datatype> ptr(raw_pointer);
shared_ptr<datatype> ptr = make_shared<datatype>();
```

Note:
Please be cautious while providing a raw_pointer as a parameter for a shared_ptr object under construction, providing the same raw_pointer address for multiple shared_ptr instances will multiply the number of 'delete' attempts when these shared_ptr instances are perishing..

for eg:

```
int* fun()
{
  static int* p = new int;
  //..
  return p;
}
```

Trying to hold the return address of the above function with multiple shared_ptr is dangerous.

```
shared_ptr<int> p1(fun());  //once, fine
//...more than once dangerous
```

****

When will the shared_ptr instances share a common a control block ?

- When shared_ptr instances are being LVALUE copy constructed.
- When one shared_ptr is being LVALUE assigned with another shared_ptr instance

```
shared_ptr<CA> ptr1 = make_shared<CA>();
shared_ptr<CA> ptr2(ptr1);  // will share same control block

shared_ptr<CA> ptr3;
ptr3 = ptr1;  //will share same control block
```

****

WEAK_PTR:
An instance of type weak_ptr will be of use only when it is associated with a shared_ptr instance. A weak_ptr instance seperately or by itself is of no practical use.

During some problem design it would lead to circular reference of shared_ptr types, under such circumstances it would lead to a situation called dead-lock, neither shared_ptr's will drop a handle to resource, each waiting for one another to the same. It is during this kind of problem situation, a weak_ptr type instance becomes handy.

A weak_ptr type instance when associated with some shared_ptr instance, will also point to the same heap resource, including the control block. As such this instance when associated with a shared_ptr type, does not disturb or alter the strong_count in the control, rather it mains it's use count in the control block.

Thereby we say, an instance of weak_ptr will bring about a weak-link in a circular reference model.

Note: A weak_ptr instance shall be initialized or assigned with a shared_ptr types only. No external heap resource ac quired thru 'new' or make_shared call will be provided.

  The state of the HEAP resource owned by a shared_ptr instance, is influenced by the state of the strong-count in the control block, the state of weak_count does not impact the life-time of the HEAP resource.

Multithreading library in modern C++:-
  The multi-threading library which is a standard, is platform neutral. While compiling the code, depending the platfo rm and tool-chain, this plat-form neutral library will map to its native API.

  In order to brake a single executable into multiple sub-tasks, we spawn or create that many threads...For which we use the 'thread' class library.

```
  #include<thread>

  thread thread_instance_name(&function_address,[parameters..]);
```

A thread instance upon being spawned needs to joined. This join statement must minimum be encountered by the chi ld thread while the parent thread is exiting or terminating. If at this moment a thread does not find a join for himself, then it will throw an exception.

There is no way where we can directly hold the return value of a function, that has been spawned as a seperate threa d by using thread class instances.

  Rather, wrap these functions return a value onto a packaged_task type instance, and then provide this packaged_tas k object as a parameter to the thread class instance.

  The packaged_task class has a member function called "get_future". A call to this function would return an instanc e of the type 'future'. This 'future' type instance would actually point to a shared_memory, where the thread functions return result would be stored, To access this data from the shared_memory, we can employ a call to the 'get' membe r function of the 'future' class w.r.to the 'future' type object.

  A call to the 'get' must be made only once, anything in excess is in-valid. The shared_memory will be active till the first call to the 'get' function, thereafter this memory will get de-allocated.

If any function that we plan to spawn as a thread, and that function designed to generate an exception, then such func tion addresses should not serve directly as thread parameters...

```
void fun()
{
 //..
 if(condition)
    throw 100;
 //....
}

thread th1(&fun);  //bad-idea, catching the exception is impossible
```

It is a good practice to wrap a call to functions that are designed to throw exceptions in a lambda function, and then p rovide these lambda instances as thread parameter for spawning threads, only then we can hold or catch the exceptio ns thrown by the function if any.

```
auto lm =[]()
{
  try
  {
    fun();
  }
  catch(...)
  {
    //..
  }
};

thread th1(lm);
```

DataRace:

Part of the code that employes are uses global entities or entities that is common to more than one function, then thes e statements consuming global entities could cause a RACE-CONDITION. The part of the code that causes RACE-CONDITION is also termed as CRITICAL-SECTION.

Whenever we define any function, and such function if it is likely to used or consumed while developing a multi-thr eaded application, the CRITICAL-SECTION area of the code must synchronize this part of the code to avoid RACE -CONDITION.

Thereby we try and use a synchronization primitive called as a 'mutex' type instance.

LOCK_GUARD: Uses RAII technique, it is exception safe, meaning the 'unlock' on the mutex is always guaranteed, either the function exits gracefully or pre-maturely terminates due to an exception.

```
template<typename T> class lock_guard
{
private:
  T mx;
 public:
   lock_guard(T x):mx(x)
   {
     mx.lock();
   }

   ~lock_guard()
   {
     mx.unlock();
   }
};
```

Scenario-1 : Where lock_guard instance is preferred over explicit call to lock/unlock calls on a mutex instance.

```
std::mutex mx;

void fun1()
{
   //Critical section starts from line-1 of the code and proceeds till the last-statement of the function
   lock_guard<mutex> guard1(mx);
   //***************
   //*****************
   //***************
   //*****************
}
```

Scenario-2 : Where lock_guard instance is preferred over explicit call to lock/unlock calls on a mutex instance.

```
std::mutex mx;

void fun1()
{

  //#############
  //##############
  //############
   //Critical section starts now and proceeds till the last-statement of the function
   lock_guard<mutex> guard1(mx);
   //***************
   //*****************
   //***************
   //*****************
}
```

The unlocking of the mutex will only happen upon the destruction of the 'guard1' instance.

Scenario-3: lock_guard is not the preferred choice.

```
std::mutex mx;

void fun1()
{

  //#############
  //##############
  //############
   //Critical section starts here
   lock_guard<mutex> guard(mx);   //a call to lock takes place here on the 'mx' and remain so till function
   //***************              // terminates...
   //*****************
   //***************
   //*****************
   //Critical section ends here

  //#############
  //##############
  //############
   //Critical section starts here

   //***************
   //*****************
   //***************
   //*****************
   //Critical section ends here
   //....
}
```

Solution-1: For the above (not exception safe)

```
std::mutex mx;

void fun1()
{

  //############
  //##############
  //###########
   //Critical section starts here
   mx.lock();
   //***************
   //*****************
   //***************
   //*****************
   //Critical section ends here
   mx.unlock();

  //############
  //##############
  //###########
   //Critical section starts here
   mx.lock();
   //***************
   //*****************
   //***************
   //*****************
   //Critical section ends here
   mx.unlock();
   //....
}
```

Solution-2: For the above problem (exception safe), prefer to use 'unique_lock' handler

```
std::mutex mx;

void fun1()
{
   unique_lock<mutex> ul(mx,defer_lock);

  //############
  //##############
  //###########
   //Critical section starts here
   ul.lock();
   //***************
   //*****************
   //***************
   //*****************
   //Critical section ends here
   ul.unlock();
  //############
  //##############
  //############
   //Critical section starts here
   ul.lock();
   //***************
   //*****************
   //***************
   //*****************
   //Critical section ends here
   ul.unlock();
   //....
}
```

```
std::mutex mtx;

void fun()
{
  std::unique_lock<mutex> ul(mtx, defer_lock);
  //NON-CRITICAL SECTION...

 //##########
 //##########

 //CRITICAL SECTION
 ul.lock();
 //...
 //...
 ul.unlock();
 //NON-CRITICAL SECTION

 //##########
 //##########

 //CRITICAL SECTION
 ul.lock();
 //...
 //...
 ul.unlock();
}
```

With respect to a thread class object or instance, It is not possible for us to hold the return value of the function spaw ned as a thread.

The work-around for the same is to wrap that function which is returning a value and is likely to be spawned as threa d into an object of type 'packaged_task'. And then provide this packaged_task object as a input to the thread instance under construction.

Thereupon upon the thread completing it's job of invoking the function, the return result of the function will be store d onto a special memory called 'shared_memory'. In order to fetch the value from the this shared memory, we associ ate a future object with respect to this packaged_task instance.

This 'future' object will establish a link or channel to the shared_memory. By invoking the 'get' member function of t
he future object we can retrieve the return result held in this shared_memory.

The life-time of this shared_memory will be till the first 'get' function call with respect to a future object, and thereaf
ter this shared_memory will get de-allocated. Thereby a second 'get' call will fail or we say the future object is not in
 a VALID state.

For eg:

```
std::string getDataFromDB( std::string token)
{ ..}
```

By creating a thread instance and providing the above function as unit-work as direct parameter, it will not be possib
le for us to hold the return value.

```
thread th1(&getDataFromDB, "hello from main");
```

We recommend wrap the above function onto a packaged_task object...

```
std::packaged_task<std::string(std::string)> task(getDataFromDB);

//Now employ a channel to point to the shared_memory
//Call the 'get_future' member function of the packaged_task class, which in-turn would return a 'future' type instanc
e. The 'future' type instance is also get getter.

std::futurue<string> ft = task.get_future();

thread th1(task,"hello from main"); //error, a packaged_task instance cannot be LVALUE copied.

thread th1(move(task),"hello from main");
//Compiler will translate the above statement to the foll: form....
// thread th(packaged_task<string(string)>::operator()(), std::move(task), "Arg");

th1.join();

auto str = ft.get();  // now the value will be fetched from shared_memory, and thereafter memory will de-allocated.
cout << str;
```

Promise and future:
  A promise instance or an object helps store a value onto a shared memory.
    Also called as INPUT channel... (SETTER)

  A future type instance or an object helps us to retrieve the value from the shared memory.
    Also called as OUTPUT channel...(GETTER)

  If a writing or storing a value onto a shared memory is taking place in a seperate thread, then this thread should use the 'promise' object for the same.

  A reading thread which attempts to fetch the value from a shared memory should use the 'future' object.

Async function:
  Using this function we can spawn threads the way we spawn threads using the thread class library.

  There are subtle differences between the them,..

Thread library:
- It is compulsory for every thread spawned using thread class objects, we need to have a join statement.
- If the function being spawned as a thread using thread class objects, we cannot hold the return value. We have to make use of 'packaged_task' type instances for the same.
- The moment thread instances gets constructed with a function address, the thread will get immediately spawned. Their is no option to delay this feature.

Async function:
- By spawning threads using the 'async' function call we don't need any 'join' statements.
- We can directly hold the return values of a function with the help of 'async' call, because by default the 'async' function returns a 'future' type object.
- We can direct the 'async' function to spawn a thread either immediately or later at point as decided by the programmer.

Note:
Holding the returned 'future' instance by the 'async' call would be necessary under 2 different circumstances..
- If the function spawned as thread is returning a value..
- If the launching of the thread has to be delayed and not immediate.

```
  async(launch::async | launch::deferred);
*************************
void fun1(){ }

async(launch::async | launch::deferred, &fun1);  //launch immediately or later

async(launch::async, &fun1);  //launch immediately

future<void> ft = async(launch::deferred, &fun1);  //spawn later when told


//Now inform async to spawn the thread...

ft.get();  //Now thread starts...
--------------------
int fun2(){ ... }

future<int> ft = async(&fun2);

cout << ft.get();
----------------------------------------
----------------------------
```

COMPILE-TIME ASSERTIONS:
Facilitates in checking for a condition or a pre-requisite for the code to compile. If the condition or the stated pre-re quisite is not met, then the compiler would be prompted to flag-off an error. We use the 'static_assert' function from t he 'type_traits' library.

The 'static_assert' call can be employed inside of a function or at a class level.

```
  template<typename T> void fun()
  {
    static_assert(...., "error_message");   //can be employed inside a global function
  }
```

```
 template<typename T> class CA
 {
     static_assert(...., "error_message");   //can also be employed at class-level
 private:
  //..
 public:
  //..
  void fun()
  {
    static_assert(...., "error_message");  //can be employed inside a member function
  }
 };

 //C++98 approach
 class CA
 {
 public:
   template<typename T> void Fun(T x)
   {
     if(typeid(T).name() == typeid(int).name())
     {
      //..
     }
     else
     {
       cout <<"skipped the value:" << x << endl;
     }
   }
 };


 //C++11 approach
 class CA
 {
 public:
   template<typename T> void Fun(T x)
   {
     static_assert(is_INT<T>::value, "the input must be integer type");
   }
 };



 CA obj1;
 obj1.Fun(100); //ok
 obj1.Fun(34.12f); //error
```

----

static_assert:
A static_assert is an assertion that is checked at compile-time.
The static_assert can be global scope, function scope or class scope.

Tuples in C++

A tuple is an object that can hold more than one element. These elements can be of heterogenous data types. The ele ments of tuples are initialized as arguments in order in which they will be accessed.

----
std::optional [c++17]

The class template std::optional manages an optional contained value, i.e. a value that may or may not be present.

----
std::exception_ptr is a nullable pointer-like type that manages an exception object which has been thrown and captur
ed with std::current_exception. An instance of std::exception_ptr may be passed to another function, possibly on ano
ther thread, where the exception may be rethrown and handled with a catch clause. It is reference counted type, the e
xception pointed will be deleted only upon the reference count falling to zero.

When functions like current_exception (or) make_exception_ptr get called they return an instance of exception_ptr, and can be accessed with rethow_exception.
