```
std:sort(books.begin(), books.end() );                          //operator <
std:minelement(books.begin(), books.end() );        //operator<

2 arguments    ==> operator < defined for the class
3 arguments    ==> 3rd arg as unary predicate
--------------
std::sort/std::minelement/std::maxelement
        ==> 2 args, operator <
        ==> 3 args, 3rd arg as normal function
        ==> 3rd arg as lambda
        ==> 3rd arg as function object
------------------------------

bool bcompare(int x , int y ) {
    return x > y;
}

auto ucompare = std::bind( bcompare, std::placeholders::_1, 30);
ucompare(x)    ==> bcompare(x,30);

std::function<bool(int)> ucomp =  std::bind( bcompare, std::placeholders::_1, 30);
```

```
bool bcompare(int x , int y ) {
  return x > y;
}

std::vector<int> v1{12, 37, 25, 56, 48};
int tmin = 30;

std::function< bool(int) > ucompare = std::bind(bcompare, ::_1, tmin);
//auto ucompare = std::bind(bcompare, ::_1, tmin);
std::count_if(v1.begin(), v1.end(), ucompare);

//std::count_if(v1.begin(), v1.end(), std::bind(bcompare, ::_1, tmin) );
----------------------------


class MyCompare {
   public:
   bool operator()  (int x,int y) {       //overloading function call operator
        return x > y;
   }
};
MyCompare mcomp;
mcomp(a,b);                    //a>b,      mcomp.operator() (a,b)
```

```
template<typename T>
class GCompare {
   public:
   bool operator()  (T x,T y) {   //overloading function call operator
        return x > y;
   }
};

GCompare<int> icomp;
mcomp(a,b);
GCompare<float> fcomp;
fcomp(p,q);

std::greater<int>()

auto sum = std::plus<int>();
sum(10,20);
-----------------
std::map<int, std::string> cities;

std::map<int, std::string, std::greatet<int>  > cities;
std::map<int, std::string, GCompare<int>  > cities;
std::map<int, std::string, MyCompare  > cities;
```

```
bool checkTemperature(Weather& wref, tint tmin, int tmax) { //3 args
return wref.getTemperature() > tmin && wref.getTemperature() < tmax;
};
auto isValidTemperature = std::bind(checkTemperature, _1, 18, 30 );//1 arg
auto minTemperature = std::bind(checkTemperature, _1, 18, ::_2 ); //2 args
auto maxTemperature = std::bind(checkTemperature, _1, _2, 30 );

Weather w1(  ....  );
isValidTemperature(w1)          ==> checkTemperatur(w1, 18, 30);
isMinTemperature(w1, tmax)   ==> checkTemperature(w1, 18, tmax);
isMaxTemperature(w1, tmin)   ==> checkTemperature(w1, tmin, 30);

fupdate(10,12,5)      ==> b1.update(10,12,5)
fzoom(1.25)           ==> b1.zoom(1.25)
fvolume()             ==> b1.volume()
```

```
template<typename T>
class Point {


};

std::list< Point<int>    >   points;    //prior to C++11, space required
std::list< Point<int>>   points;        //from C++11, no space required

using IPoint = Point<int>;
using IVector = std::vector<int>;
using IPVector = std::vector<Point<int>>;
//above three can be managed with typedef also

template<typename T>
using PVector = std::vector<Point<T>>;              //typedef can't help here

PVector<int> ipvector;
PVector<float> fpvector;
```

```
template<typename T1, template T2>
class Sample {
  T1 x;
  T2 y;

};

Sample<int,float>    s1;
Sample<float,char> s2;

using IFSample = Sample<int,float>;
using DISample = Sample<double,int>;

template<typename T>
using ISample = Sample<int,T>;

template<typename T>
using FSample = Sample<T,float>;
```

variable length arguments (C concepts)

...   any no.of any type of args
va_arg, va_list, va_start,va_end

TODO:-
vsum(3, a, b, c);
vsum(2, a, b);
vsum(4, a, b, c, d);
vsum(1, a);
vsum(0);

TODO:-
extern Templates
noexcept -- exception handling

Self Study:- STL Improvements (Except Reference Wrapper, std::ref/std::cref)

```
//Box *ptr=new Box(10,12,5);                                  //raw ptr
std::unique_ptr<Box> uptr(new Box(10,12,5) );                 //smart ptr
uptr->volume();              //uptr.operator->().volume();
uptr->zoom(1.5);
uptr->update(11,13,8);
//*uptr                      //uptr.operator*()
//no need of delete
```

One scenario:-

```
Box *rawptr = new Box(10,12,5);
std::unique_ptr<Box> up1(rawptr);
std::unique_ptr<Box> up2(rawptr);
up1->volume();
delete rawptr;
up2->volume();
----------------------
std::unique_ptr<Box> up1 = std::make_unique<Box>(10,12,5);  //C++14

std::unique_ptr<Box> up2;
up2 = std::make_unique<Box>(10,12,5);  //std::unique_ptr<Box>(new Box(10,12,5));

//up2 = up1;                        //error
//up2=std::move(up1);               //ok
```

Note:- std::make_unique returns anonymous object, compatible
with r-value references (move operations)

```
template<typename T, /*TODO*/ >
std::unique_ptr<T>&&  my_make_unique( /*TODO*/  ) {
    return std::unique_ptr<T>( new T( /*TODO*/ ) );
}
```

/*TODO*/ syntax related to variadic args

Members of std::unique_ptr:-

default ctor
parameterized ctor -- address of managed object
move ctor, move operator=
//copy ctor, copy operator= not allowed

release
reset
get

operator*
operator->

```
std::list<Point*> points;
points.push_back(new Point(3,4));
points.push_back(new Point(5,6));
points.push_back(new Point(1,2));
```

better way with smart ptr:-

```
std::list< std::unique_ptr<Point> > points;
points.push_back ( std::unique_ptr<Point>(new Point(3,4));
points.push_back ( std::make_unique<Point>(5,6));

std::;unique_ptr<Point> temp(new Point(7,8));
points.push_back(temp);                  //error, copy operations not allowed
points.push_back(std::move(temp));
```

Skip:-
* Customer Deleter

Activities:-
* post read of covered topics + practice examples
* continue coding tasks
* refresh thread & ipc concepts (linux os, posix apis)
    * threads
    * mutex
    * semaphore
    * producer consumer problem
    * deadlocks

Further:-
* concurrency in C++ ==> std::thread, std::async
* IPC techniques ==> std::mutex, some locks, std::condition_variable etc.