std::is_trivial<Box>::value
std::is_trivial<MyStack>::value

std::is_trivially_copyable
std::is_trivial

TODO:- take some classes and check if they are trivial or not, trivially copyable

Move constructor
Move operator=
R-value reference

```
MyString getString(const char *ps) {
  //process ps
  return MyString(ps);
}
```

```
MyString res = getString("abcdxyz")
```

RVO - Return value optimization

```
std::list<MyString> names;

names.push_back( MyString("hello123") );
```

Anonymous objects will match move operations, because they
are compatible with r-value references.

```
MyString ts("Welcome");
//names.push_back(ts);                    //copy
names.push_back(std::move(ts));           //move
```

Anonymous objects (or) objects type casted with std::move
will match move operations, i.e. compatible with r-value references

TODO:- check that copy ctor is not provided by compiler
if any of the following is implemented - dtor, move ctor, move operator=

```
std::list<Point> points;

Point p1;
points.push_back(p1);
points.push_back(Point(3,4));
points.push_back(std::move(p1));
```

Additional/Advanced:-
* Universal References
* Reference Collapsing
* Perfect Forwarding

```
int (*fptr)(int,int);

fptr=sum;
(or)
fptr=multiply;

fptr(a,b);
------------------
//Array of pointers
int (*fparr[4])(int,int);
fparr[0]=sum;
fparr[1]=diff;
fparr[2]=multiply;
fparr[3]=custom;
res = (fparr[i])(a,b);

int (*fsum)(int,int);
fsum=sum;

std::function<int(int,int)> fsum=sum;
```

```
int Banking::countAccountsWithMinBal(double minval) {
  int count=0;
  count = std::count_if(accounts.begin(), accounts.end(),
                                [minval](const Account& ref) {
    return ref.getBalance() > minval;
  });
  return count;
}
--------------------

Unary Predicate
Binary comparator

bool bcompare(const Account& r1, const Account& r2) {
    return r1.getBalance() < r2.getBalance();
}
void Banking::sortByBalance() {
   //std::sort(accounts.begin(), accounts.end(), bcompare);
   std::sort(accounts.begin(), accounts.end(),
              [](const Account& r1, const Account& r2) {
    return r1.getBalance() < r2.getBalance();
   });
}
```

```
int Banking::countAccountsByRange(double minval,double maxval) {
  int count=0;
  /*std::list<Account>::iterator iter;
  for(iter=accounts.begin();iter!=accounts.end();++iter)
    if(iter->getBalance() >= minval && iter->getBalance() <= maxval)
       count++;*/
  count=std::count_if(accounts.begin(), accounts.end(),
                                [minval,maxval](const Account& ref) {
    return ref.getBalance() > minval && ref.getVal() <= maxval;
  });

  return count;
}
```

Understand STL algorithms:-

std::find
std::count
std::find_if
std::count_if
std::copy_if
std::remove_if

std::sort
std::minval / std::maxval

```
double Banking::findAccountWithMaBalance() {

  auto iter = std::maxval(accounts.begin(), accounts.end(),
              [](const Account& r1, const Account& r2) {
                 return r1.getBalance() < r2.getBalance();
   });
}
```

TODO:-

* refresh move semantics, lambdas (covered topics)
* explore STL algorithms, apply in coding tasks with lambdas
* self study (try) -- file handling -- ofstream, ifstream
* pre-read
    - std::bind
    - some topics on templates
    - smart pointers