Course : Update to Modern C++

Section : 2 Review of C++

* Operators :

```
int main()
{
    string str("Hello")
    string :: iterator it = str.begin();
    while (it != str.end())
    {
        cout << *it << ",";
        ++it;
    }
```

→ H, e, l, l, o

or

```
for(string:: iterator it= str.begin() ; it != str.end() ; ++i)
        cout << *it* << ","
```

* Algorithm

```
string str("Hello world");
char c = 'l';
string:: iterator res = find(str.begin(), str.end(), c);
                        //searching string for first occurance of 'l'.

// check if we found it
if(res != str.~~find~~end())
        cout << "found at index" << res - str.begin() << endl;
else
        cout << "string does not contain" << c << endl;
```

* Operator Overloading (Needs to prepare)

```
class Rectangle
{   int i;
public:
    bool Operator == (const Rectangle & other)
                    { return other.i == i ; }
};

Rectangle R1, R2;
if (R1 == R2)          // '==' operator is overloaded
{                      //  to compare the objects
  :
}
```

* function call operator

```
class evenp {
public:
    bool operator () (int n)
      {  return (n % 2 == 0);
      }
};

int main()
{
    evenp is_even;      // We are making object as calleble.
    if (is_even(6))
        cout << " 6 is an even number " <<endl;
    else
        cout << " 6 is not even number" <<endl;
}
```

Topics : 1. Operator overloading
2. functors
3. Algorithms with Predicates.

* Classes and Inheritance

→ When a derived class object is created, the base class's constructor is called first, then the derived class's constructor

→ When it is destroyed, the derived class destructor is called before the base class's.

```
class Vehicle {
public:
    void accelerate() { };
};
class Aeroplane : public Vehicle
{ public:
    void accelerate(int height){ }   → // overload hide parent's accelerate().
    // using Vehicle::accelerate;    → // Makes parent's version of accelerate() available.
};
int main()
{
    Aeroplane plane;
    plane.accelerate();
}
```

* ★ Imp

⇒ If member is protected, then it can only be accessed by objects of the same class and classes which are derived from it.

```
class Vehicle
{ protected:
    void accelerate() { };
};
class Aeroplane : public Vehicle
{ public:
    void accelerate() { Vehicle::accelerate(); }
};                                  ↑ (Accessible)
int main()
{   Aeroplane plane;
    plane.accelerate();
    Vehicle vehicle;
    Vehicle.accelerate(); ← Not accessible as protected.
}
```

* Exceptions

①

```
int main()
{
    vector <int> v;
    try
    {
        cout << v.at(2) <<endl;          // Throw an exception
    }
    catch (const std::exception& e)  // will handle all subclasses of std:: exception
    {
        cout << " Exception caught : " << e.what() <<endl;
    }
}
```

②

```
int main()
{
    vector <int> v;
    try
    {
        if (v.size() < 3)
            throw std::out_of_range("OOps");
        cout << v[2] <<endl;
    }
    catch (const std:: exception & e)
    {
        cout << " Exception caught : " << e.what() <<endl;
    }
}
```

* Smart Pointer

→ Smart pointers is a class which wraps a raw pointer, to manage the life time of the pointer.

→ The most fundamental job of smart pointer is to remove the chances of memory leak.

→ It makes sure that the object is deleted if it is not reference any more.

```
#include <iostream>
using namespace std;

class MyInt
{ public:
    explicit MyInt (int *p=nullptr) { data =p;}
    ~ MyInt () { delete data; }
    int& operator * () { return *data }
  private:
    int *data;
}

int main()
{ int *p = my int (10);
  MyInt myint = MyInt (p);
  cout << *myint <<endl;
  return 0;
}
```

① Unique Pointer in C++

① Unique_ptr is a class template.

② Unique_ptr is one of the smart pointer provided by c++11 to prevent memory leaks.

③ Unique_ptr wraps a raw pointer in it and de-allocates the raw pointer.

④ Similar to actual pointer we can use → and * on the object of unique_ptr.

⑤ When exception comes then also it will de-allocate the memory hence no memory leak.

⑥ Not only object, we can create array of objects of unique_ptr.

// Operations
release, reset, swap, get, get_deleter.

Code

```
class foo
{   int x;
public:
    explicit foo(int x) : x{x} {}
    int getx()
    { return x;
    }
    ~foo() { cout << "foo Dest" << endl; }
};
int main()
{
    // foo *f = new foo(10);
    // cout << f->getx() << endl;

    std::unique_ptr<foo> p(new foo(10));
    cout << p->getx() << endl;

    return 0;
}
```

← new pointer is created, (on heap)
Now after its use if we don't delete it (delete f).
memory leak problem will arise as memory is not freed.

↑ Here we are creating object of a class "unique_ptr".
It is created on stack. Once the scope of object ends, its destructor will be called.
and inside the destructor of unique_ptr, the pointer is deleted.

```
int main ()
{
⇒ // different ways to create unique pointer
   unique_ptr <foo> p1 (new foo(10));   ← Not exception safe.
or unique_ptr <foo> p2 = make_unique <foo> (20);  ← Exception safe.
```

* make_unique is exception safe. hence prefer using it

or `Foo *f = new foo(10);` ← This way can also be used but should not be use as it violates the basic unique rule only.
`unique_ptr <foo> p1 (f);`
`unique_ptr <foo> p6 (f)`

⇒ // Accessing the element of class
`cout << p1 -> getx() << (*p2).getx() << endl;`
'->' & '*' are overloaded.

⇒ // Different operations
`// p1=p2 ;` Fail: This will fail because you can not copy ownership.

`unique_ptr <foo> p3 = std::move(p1);`
Pass: Because moving ownership is allowed.

`foo * p = p3.get();` .get() will give the managed object by p3 i.e. (new foo(10)) address.

`foo * p4 = p3.release();` .release() will give up ownership of managed object and p3 will be nullptr after that.

`p2.reset(p4);` // the previously manged object by p2 will be delcted and now it will point to object managed by p4.

`p1.swap(p2);` // will just swap the ownerships.

}

Q. When to use unique_ptr ?

→ Use unique_ptr when ~~yow~~ you want to have single ownership (Exclusive) of the resources. Only one unique_ptr can point to one resource. Since there can be one unique_ptr for a single resource it's not possible to copy one unique_ptr to another.

② Shared Pointer

① shared_ptr is a smart pointer which can share the ownership of object (managed object).

② several shared_ptr can point to the same object (managed object)

③ It keep a reference count to maintain how many shared_ptr are pointing to the same object.

④ shared_ptr is threads safe and not thread safe.

i.e. a) control block is thread safe. (reference count)

b) managed object is not thread safe. (object managed)

(we can make it safe using mutex in class).

⑤ There are three ways shared_ptr will destroy managed object.

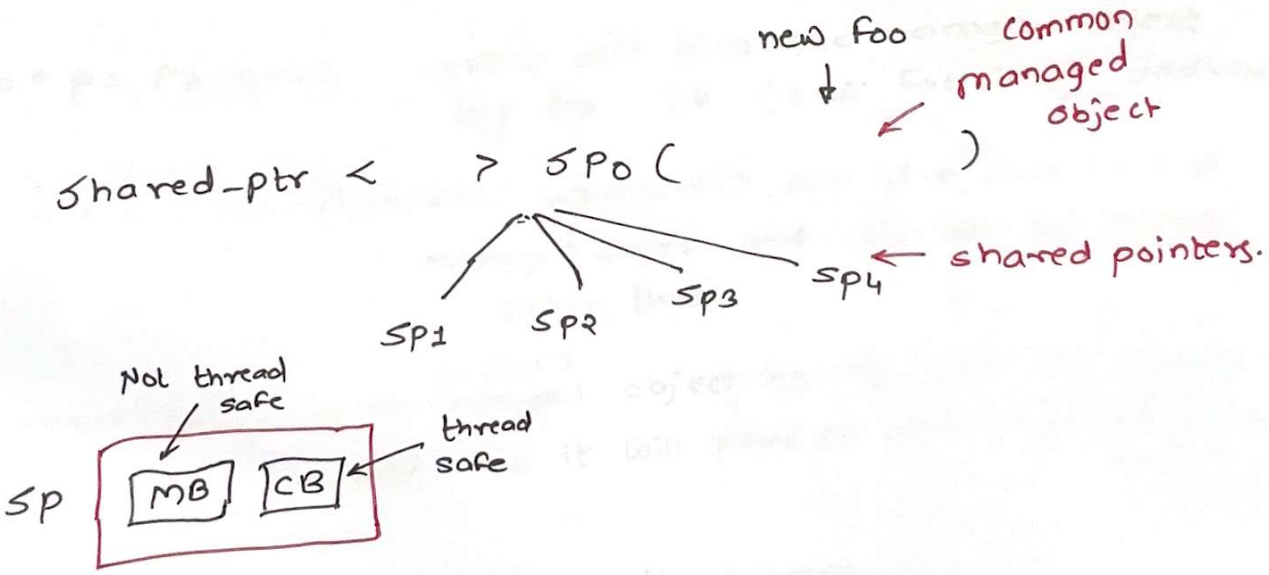
i) If the last shared_ptr goes out of the scope.

ii) If you initialize shared_ptr will some other shared_ptr.

iii) If you reset shared_ptr.

⑥ Reference count doesn't work when we use reference or pointer of shared_ptr.

Imp



```
#include <iostream>
# include <memory>
Using namespace std;

class foo
{  int x;
 public:
    foo(int x) : x{x} {};
    int getx () { return x;}
    ~foo() { cout << "~foo" <<endl;}
};

int main()                    shared_ptr<foo> sp1 = make_shared (20);
{
   std:: shared_ptr <foo> sp (new foo(100));
   cout << sp->getx() <<endl;                    ⟶ 100
   cout << sp. use_count () <<endl;              ⟶ 1

   std:: shared_ptr <foo> sp1 = sp;  ⟶ ref. count is increase to 2

   cout << sp. use_count() <<endl;               ⟶ 2
   cout << sp1 -> use_count() <<endl;            ⟶ 2

   return 0;                      O/p: 100, 1, 2, 2, ~foo
}

// shared_ptr is mostly useful in multithreading / threading


int main()
{
   std:: shared_ptr <foo> sp (new foo(100));
   thread  t1 (fun, sp), t2 (fun, sp) , t3 (fun, sp) ;
   cout <<"main: " << sp.use_count() <<endl;
   t1.join() ; t2.join() ; t3.join();
   return 0;
}
```

③ Weak Pointer

① If we say unique_ptr is for unique ownership and shared_ptr is for shared ownership then weak_ptr is for non-ownership smart pointer.

② It actually reference to an object which is managed by shared_ptr.

③ A weak_ptr is created as a copy of shared_ptr.

④ We have to convert weak_ptr to shared_ptr in order to use the managed object.

⑤ It is used to remove cyclic dependency between shared_ptr.

* *

```
# include <iostream>
# include <memory>   using namespace std;
int main()
{
  auto sharedptr = std:: make_shared <int> (100);
  std:: weak_ptr <int> weakptr (sharedptr);

std :: cout <<"weakptr. use_count():" << weakptr. use_count() << endl;
cout << " sharedptr. use_count():" << sharedptr. use_count() << endl;
cout << " WeakPtr. expired():" << weakptr.expired() << endl;

if(shared_ptr <int> sharedPtr1 = weakptr. lock())
  { cout << " *sharedPtr: " << * sharedPtr << endl;
    cout << " sharedptr1. use_count(): " << shandPtr1. use_count()
                                                    << endl;
  }
else
    cout << " Don't get the resources ! " << endl;

weakptr. reset();
```

Now, shared_ptr <int> sharedptr1 = weakPtr. lock()

Will give false.
bc

```
struct Son;
struct Daughter;
struct Mother
{  ~Mother()
    { cout << "Mother gone" << endl;
   }
   void setSon (const shared_ptr <son> s)
    {  myson = s;
    }
   void setDaughter (const shared_ptr <Daughter> d) { myDaughter = d;}

   shared_ptr <son> my Son;
   shared_ptr <Daughter> my Daughter;
}
struct Son
{  son (shared_ptr <Mother> m) : myMother (m) { }
   ~son () { cout << "son gone" << endl;
    shared_ptr <const Mother> myMother;
}
struct Daughter
{
   Daughter (shared_ptr < Mother > m) : myMother (m) { }
   ~Daughter () { cout << "Daughter gone" << endl;
   shared_ptr < const Mother > my Mother;
}

int main()
{  shared_ptr <Mother> mother = shared_ptr <Mother> (new Mother);
   shared_ptr <Son> son = shared_ptr <son> (new Son (mother));
   shared_ptr <Daughter> daughter = shared_ptr < daughter > (new daughter (mother));

   mother → setSon (son);
   mother → set Daughter (daughter);
}
```

⇒ If we run this program, No Destructure is called as it is in cyclic dependency.

⇒ To break cyclic dependency we have to use weak pointer.

⇒ Replace all "shared_ptr" in "struct Mother" by "weak_pointer."

⇒ Now if we run, then all Destructor are called i.e. objects are deleted, once the scope is over.

* Multi-threading in C++

Q. what do you understand by thread and give one example in C++ ?

→ ① In every application there is a default thread which is main(). Inside this main() we create other threads.

② A thread is also known as lightweight process.
Idea is achieving parallesim by dividing a process into multiple threads.

ex:
1) The browser has multiple tabs that can be different threads.
2) MS Word must be using multiple threads, one thread to format the text, another thread to process inputs (spell checker)
3) Visual studio code editor would be using threading for auto completeing the code (Intellicence).

// Ways to create threads in C++ 11.
① Function Pointers
② Lambda functions
③ Functors
④ Member functions
⑤ static Member functions

// Find the addition of all odd Numbers from 1 to 1900000000 and all even numbers from 1 to 1900000000.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <Algorithm>
using namespace std;
using namespace std::chrono;
typedef unsigned long long ull;
```

```
ull Oddsum=0;
ull Evensum=0;

void FindEven(ull start, ull end)
{ for(ull i=start ; i<=end ; i++)
  { if( (i&1)==0)
      Evensum += i;
  }
}

void FindOdd(ull start , ull, end)
{ for(ull i=start; i<=end ; i++)
  { if((i&1)==1)
      Oddsum+=i;
  }
}

int main()
{
  ull start= 0 , end= 1900000000;
  auto startTime= high_resolution_clock::now();
  std:: thread t1(FindEven, start, end);
  std:: thread t2( FindOdd, start, end);
  t1.join();
  t2.join();
  //findOdd(start,end);
  //FindEven(start,end);
  auto stopTime= high_resolution_clock::now();
  auto duration= duration_cast<microseconds>(stopTime- startTime);

  cout << "Oddsum: " << oddsum << endl;
  cout << " Evensum: " << Evensum << endl;

  cout << "sec: " << duration.count()/1000000 << endl;

  return 0;
}
```

} Time : 2 sec.

} Time : 4 sec

// Ways to create threads in C++ 11.

① Function Pointer

```
void fun(int x)
{
  while (x-- > 0)
    cout << x << endl;
}

int main()
{
  std:: thread t(fun, 10);
  t.join();
  return 0;
}
```

② Lambda function

```
int main()
{
  auto fun = [] (int x)
  {
    while (x-- > 0)
      cout << x << endl;
  };
  std:: thread t(fun, 10)
  t.join();
  return 0;
}
```

or We can directly inject lambda at thread creation time.

```
int main()
{
  std:: thread t ([] (int x)
    { while (x-- > 0)
        cout << x << endl;
    }, 10);
  t.join();
  return 0;
}
```

③ Functor (function Object)

```
class Base
{
public:
    void operator () (int x) {
        while (x-- > 0)
            cout << x << endl;
    }
};

int main()
{
  std:: thread t ((Base()), 10);
  t.join();
  return 0;
}
```

④ Non-static member function

```
class Base
{
public:
    void run(int x)
    {
        while(x-- > 0)
            cout << x << endl;
    }
};
int main()
{
    Base b;
    std::thread t(&Base::run, &b, 10);
    t.join();
    return 0;
}
```

⑤ static member function

```
class Base
{
public:
    static void run(int x)
    {
        while(x-- > 0)
            cout << x << endl;
    }
};
int main()
{
    std::thread t(&Base::run, 10);
    t.join();
    return 0;
}
```

If it is a static member function then no need to create the class object and pass the address.

* Topic: Use of join(), detach(), and joinable() in thread.

// join() Notes:

1) Once a thread is started we wait for this thread to finish by calling join() function on thread object.
2) Double join will result into program termination.
3) If needed we should check thread is joinable before joining. (using joinable() function.)

// Detach() Notes:

1) This is used to detach newly created thread from parent thread.
2) Always check before detaching a thread, that is joinable. otherwise we may end up double detaching and double detach() will result into program termination.
3) If we have detached thread and main function is returning then the detached thread execution is suspended.

// Note: Either join() or detach() should be called on thread object. Otherwise during thread object's destructor it will terminate the program. Because inside destructor it will check if thread is still joinable? if yes then it terminates the program.

```
void run(int count)
{
  while(count-->0)
      cout << count <<endl;
  std::this_thread::sleep_for(chrono::seconds(3));
}
// join()
int main()
{ std::thread t1(run,10);
  cout << "main()" <<endl;
  t1.join();
  if(t1.joinable())
      t1.join();
  cout <<"main() after"<<endl;
  return 0;
}
```

```
// detach()
int main()
{
  std::thread t1(run,10);
  cout << "main()" <<endl;
  t1.detach();
  if(t1.joinable())
      t1.detach();
  cout << "main() after" <<endl;
//std::this_thread::sleep_for(.....)
  return 0;
}
```

Topic: Mutex in C++ threading

Mutex: Mutual Exclusion

Race condition:

① Race condition is a situation where two or more threads/process happened to change a common data at the same time.

② If there is a race condition then we have to protect it and protected section is called 'Critical section/region.

Mutex:

① Mutex is used to avoid race condition.

② We use lock(), unlock() on mutex to avoid race condition.

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

int myAmount=0;
std:: mutex m;

void addMoney()
{
    m.lock();
    myAmount++;
    m.unlock();
}

int main()
{
    std:: thread t1(addmoney());
    std:: thread t2(addMoney());

    t1.join();
    t2.join();

    cout<< myAmount<< endl;

    return 0;
}
```

- Here, Let's say thread t1 reaches before t2.
- then t1 will lock the mutex.
- when t2 reaches mutex is locked so t2 will wait until mutex is unlocked.
- Now t1 will complete its process and unlock mutex.
- Once it is unlock then t2 will lock and will do its process and then will unlock mutex

* Topic: std::mutex :: try_lock()

1) try_lock() Tries to lock the mutex. Returns immediately. On succesful lock acquisition return true otherwise return false.

2) If try_lock() is not able to lock mutex, then it doesn't get blocked that's why it is called "non-blocking."

- It is a dead lock situation with undefined behaviour.

(If you want to be able to lock the same mutex by same thread more than one time then go for recurssive_mutex :)

```
#include <iostream>
#include <thread>
#include <mutex>
Using namespace std;
int counter = 0;
std::mutex mtx;

Void increaseThecounterfor100000Time()
{
  for(int i=0; i< 100000; i++)
    {
      if (mtx.try_lock())
        {
          counter++;
          mtx.unlock();
        }
    }
}
int main()
{
  std:: thread t1(increasTheCounterFor10000 0Time());
  std:: thread t2(increaseTheCounterfor 100000 Time());

    t1.join();
    t2.join();

    Cout << " Counter could increase upto:" << counter << endl;

    return 0;
}
```

Here, thread won't wait to unlock mutex. It will try to lock if unable to lock then it will go to next iteration. and it goes on

std:: try_lock() in c++ 11 threading :→

① std:: try_lock tries to lock all the lockable objects passed in it one by one in given order.

Syntax: std:: try_lock (m1, m2, m3, m4, ---- mn);

② On success this function returns -1 otherwise it will return 0-based mutex index number which it could not lock.

③ If it fails to lock any of the mutex then it will release all the mutex it locked before.

④ If a call to try_lock results in an exception, unlock is called for any locked objects before rethrowing.

⇒ The actual use of std:: try_lock() function is, it can ~~throw~~ try to lock multiple mutex objects at the same time.

* Topic :→ Timed Mutex in C++ :→

std:: timed_mutex

→ std:: timed_mutex is blocked till "timeout_time" or lock is aquired and returns true if success otherwise false.

① try_lock_for();

- waits untill specified timeout_duration has elasped or the lock is acquired, whichever comes first.
- On succesful lock acquisition returns true, otherwise returns false.

② try_lock_untill()

- waits untill specified timeout_time has been reached or the lock is acquired.

```
auto now = std:: chrono:: steady_clock :: now();
if(m.try_lock_until(now + std:: chrono :: seconds(1)));
```

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;

int myAmount = 0;
std::timed_mutex m;
void increment (int i)
{
  if (m.try_lock_for (std::chrono::seconds (1))
  {
      myAmount++;
      std::this_thread::sleep_for (std::chrono::seconds(2));
      cout << "Thread" << i << "Entered" <<endl;
      m.unlock();
  }
  else
  {
    cout << "Thread" << i << " Couldn't Enter" <<endl;
  }
}

int main()
{
  std::thread t1 (increment, 1);
  std::thread t2 (increment, 2);
    t1.join();
    t2.join();

    cout << myAmount <<endl;
    return 0;
}
```

Topic: Recursive Mutex in C++

→ It is same as mutex, but same thread can lock one mutex multiple times using recursive_mutex.

→ If thread t1 first call lock/try_lock on recursive mutex m1, ~~them~~ then m1 is locked by T1. now as T1 is running in recursion T1 can call lock/try_lock any number of times there is no issue.

→ But if T1 have acquired 10 times lock/try_lock on mutex m1 then thread T1 will have to unlock it 10 times otherwise no other thread will be able to lock mutex m1.

→ It means recursive_mutex keeps count how many times it was locked so that many times it should be unlocked.

→ How many time we can lock recursive_mutex is not defined but when that number reaches (ie. stack memory is full) and if we are calling lock() it will return std::system_error OR if we are calling try_lock() then it will return false.

// Bottom Line

→ It is similar to mutex but have extra facility that it can be locked multiple time by same thread.

→ If we can avoid recursive_mutex then we should because it brings overhead to the system.

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

std::recursive_mutex m1;

int increment = 0;

void recursion(int i, int count)
{
    if (count < 0)
        return;

    m1.lock();
    cout << "Thread: " << i << " = " << ++increment << endl;
    recursion(i, --count);
    m1.unlock();
    cout << " Unlock by thread " << i << endl;
}

int main()
{
    thread t1(recursion, 1, 10);
    thread t2(recursion, 2, 10);

    t1.join();
    t2.join();

    return 0;
}
```

T_1 will lock thread m1 for 10 times

T_1 will Unlock thread m1 for 10 times.

Once T_1 unlocks 10 times.
then T_2 will start locking recursively.

Topic : Lock Guard in C++

→ It is very light weight wrapper for owning mutex on scoped basis.

→ This is used when developer explicitly wants to tell that I want to lock this mutex object untill the scope of the function is over.

→ It aquires the mutex lock the moment you create the object of lock_guard.

→ It automatically removes the lock while goes out of scope.

→ you can not explicitly unlock the lock_guard.

→ You can not copy lock_guard.

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

std::mutex m1;
int increment=0;

void task(int i, int count)
{
  std::lock_guard<mutex> lock(m1);
  for(int i=0; i<count; i++)
  {
    increment++;
    cout << "Thread: " << i << increment <<endl;
  }
}  ← Here scope of the lock_guard object ends so destructor will be called. and in destructor "Unlock()" will be called.

int main()
{
  thread t1(task, 1, 10);
  thread t2(task, 2, 10);

  t1.join();
  t2.join();

  return 0;
}
```

Topic: Unique_lock

- The class unique_lock is a mutex ownership wrapper.
- It allows
  * Can Have different locking strategies.
  * time-constrained attempts at locking (try_lock_for , try_lock_until)
  * recursive locking
  * condition variables
- Locking strategies. Type
  1. defer_lock → do not acquire ownership of the mutex.
  2. try_to_lock → try to lock the ownership of the mutex without blocking
  3. adopt_lock → assume the calling thread already has ownership of the mutex.

```
#include <iostream>          #include <mutex>
#include <thread>            using namespace std;
std:: mutex m1;
int buffer = 0;
void task(const char* threadnumber, int& loopfor)
{ std:: unique_lock <mutex> lock(m1); // Automatically calls lock
                                      on mutex m1.
  for(int i=0 ; i< loopfor ; i++)
  { buffer ++;
    cout << threadnumber << buffer << endl;
  }
}
int main()
{  thread t1(task,"T1", 10);   thread t2(task,"T2",10);
   t1.join(); t2.join(); return 0;
}
void task2(        —11—          )
{ std:: unique_lock <mutex> lock(m1, std::defer_lock);
                  // Does not call lock on mutex m1, because
                     used defer_lock.
  lock.lock(); // But then we will have to explicitly tell to lock
               when ever we want to lock mutex m1.
for(              )
      —11—
      —11—
 }
// lock.unlock(); ← This is not needed as it will be unlocked in
}                    destructor of unique_lock.
```

*Topic: Condition Variable (Imp)

→ Condition variables are used for two purpose.

① Notify other thread
② Waiting for some condition.

→ Condition variable allows running threads to wait on some condition and once those conditions are met the waiting thread is notified using.

a. notify_one();
b. notify_all();

→ You need mutex to use condition variable.

→ If some thread want to wait on some condition then it has to do these things.

a. Acquire the mutex lock using std::unique_lock<std::mutex> lock(m);

b. Execute wait, wait_for, or wait_until. The wait operations automatically release the mutex and execution of thread waits/sleep until it is notified by condition variable.

c. When the condition variable is notified, the thread is awakened, and the mutex is automatically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

Note: ① Condition variables are used to synchronize two or more threads.

② Best use case of condition variable is Producer/Consumer Problem.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;
std::condition_variable cv;
std::mutex m;
long balance=0;

void addMoney(int money)
{
    std::lock_guard<mutex> lg(m);
    balance += money;
    cout << "Amount Added, Current Balance: " << balance << endl;
    cv.notify_one();
}

void withdrawMoney(int money)
{
    std::unique_lock<mutex> ul(m);
    cv.wait(ul, []{return (balance!=0) ? true : false;});
    if(balance >= money)
    {
        balance -= money;
        cout << "Amount Deducted: " << money << endl;
    }
    else
        cout << "Amount Can't be deducted, Balance is less" << endl;
}
int main()
{
    std::thread t1(withdrawMoney, 500); thread t2(addMoney, 500);
    t1.join(); t2.join(); return 0;
}
```

⇒ Suppose t1 reaches before t2. and now acquires the lock in withdraw money. then it executes the condition in wait, now money is not yet added hence condition becomes false. then t1 unlocks the mutex and sleeps there.

→ Now mutex is unlocked for thread t2. It adds the money and once it done. it notifies t1 via condition variable cv.

→ t1 gets the awaken call on it again checks the condition now condition is satisfied so it performs further action.

* Topic: Deadlock in threading

```
#include <iostream>
# include <thread>
# include < mutex >
Using namespace std;

 std:: mutex m1;
 std:: mutex m2;

 Void thread 1 ()
{
    m1.lock ();
    m2 . lock ();   ← or will wait here.
    cout << " Critical section of thread one "<<endl ;
    m1. unlock ();
    m2 . unlock ();
}

 Void thread 2 ()
{
    m2 . lock ();
    m1 . lock ();   ← will wait here
    cout << " Critical section of thread two " <<endl;
    m2 . unlock ();
    m1 . unlock ();

 }

 int main ()
{
    thread t1 (thread 1);
    thread t2 (thread 2);
     t1 . join ();
     t2 . join ();
      return 0;
 }
```

Topic: std::lock()

It is used to lock multiple mutex at the same time.

Syntax: `std::lock(m1, m2, m3, m4);`

→ All arguments are locked via a sequence of calls to lock(), try_lock() or unlock() on each argument.

→ Order of locking is not defined (it will try to lock provided mutex in any order and ensure that there is no dead lock 😊).

→ It is a blocking call.

Ex-1 No deadlock

Thread 1
```
std::lock(m1, m2);
```

Thread 2
```
std::lock(m1, m2);
```

Ex:2 No deadlock Imp (Understand)

Thread 1
```
std::lock(m1, m2)
```

Thread 2
```
std::lock(m2, m1)
```

Ex:-3 No deadlock

Thread 1
```
std::lock(m1, m2, m3, m4)
```

Thread 2
```
std::lock(m3, m4)
std::lock(m1, m2)
```

Ex-4 Yes, the below can be deadlock

Thread 1
```
std::lock(m1, m2)
std::lock(m3, m4)
```

Thread 2
```
std::lock(m3, m4)
std::lock(m1, m2)
```

```
#include <iostream>
#include < thread>
# include < mutex>

std:: mutex m1, m2;

void task_a ()
{
    while (1)
      {
        std:: lock (m1, m2);
        std:: cout<< " task a" <<endl;

        m1.unlock();
        m2.unlock();
      }
}


void  task_b ()
{
   while(1)
      {
        std:: lock (m2, m1);
        std:: cout << "task b" <<endl;
        m2.unlock();
        m1.unlock();
      }
}

int main()
{
    std:: thread  t1 (~~thread~~ task_a);
    std :: thread   t2 ( task_b);

    t1.join() ;
    t2.join();

    return 0;

}
```

* Diff. between Sleep and Wait:

① Sleep : Slip is

"I'm done with my timeslice, and please don't give me another one for atleast 'n' miliseconds."

The OS doesn't even try to schedule the sleeping thread until requested time has passed.

Points:

① It will keep the lock and sleep.

② sleep is directly to thread, it is a thread function.

② Wait: Wait is

"I'm done with my timeslice. Don't give me another timeslice until someone calls Notify() or NotifyAll()."

The OS won't even try to schedule your task unless someone calls notify() (or one of a few other wakeup scenarios occurs ☺).

Points:

① It releases the lock and wait.

② Wait is on condition variable, it is like there is a condition variable in thread and wait is applied to that CV but it ends up putting in waiting state.

Section: Move Semantic

Problem/Motivation

e.g: 1

```
T foo(T obj)
{
    T temp;
    :
    :
    return temp;
}
T f = foo(ob1);
```

← As it is passed by value hence new argument obj variable will be created and passed value will be copied.

← new temp variable is created whose scope will be for this function until returned. once it returned it will be deleted.

← and returned value is again deep copied in new variable 'f'.

e.g: 2

```
vector<int> fun(____)
{
    vector<int> *temp;
    1000 times {
        - - - - -
        - - - -
        - - - -
        *temp.push_back(Obj);
    }
    return temp;
}
vector<int> ans = fun(...);
```

← Here again temp vector is filled 1000 times consuming valuable space and time

← It is returned and then deleted so all the efforts are kind of wasted.

← Complete returned expensive vector needs to deep copy in new vector 'ans'.

⇒ Argument problem can be solved by passing by reference.

⇒ and temp copy problem is addressed by move semantics.

* LValue and RValue in C++

LValue: If you can take address of expression then it is LValue. and they last extended period of time.

Rvalue: RValues are such expressions which you can't take addres and they are temporary, they don't exist after one line.

e.g.:

int (i) = 5; ← i is lvalue

int *p = &i; ✓

*p2 = &5 ✗

class A {...};

A a; ← a is lvalue.

int x = 10 → x is lvalue and 10 is rvalue.

int a=10, b=20;

int x = (a+b); → x is lvalue and (a+b) is rvalue.

int *p = &(a+b); → Error: (a+b) is not lvalue.

class Cat {...};

Cat c = cat(); → Cat() is rvalue as it is constructor and it is tempory/limited only for that line.

int square(int x) {return x*x;};

int sq = square(10); → square(10) is rvalue. is it is a function call and it is temporary.

* lvalue reference & rvalue reference

① lvalue reference:

- Normal/Old references
- Can only be bound to lvalues but not r values.

Imp However we can bind an rvalue to a const lvalue reference

e.g:
```
int i = 10;          ✓
int &x = i;          ✓  ← Normal lvalue reference
int &x = 10;         ×  ← Assinging r value to lvalue reference which is not allowed.
Const int& x = 10;   ✓
```
↑ Binding r value ref to l value reference using const. (compiler do this).

② rvalue reference:

- Introduced in c++ 11 standard.
- Bind only to rvalues.
- Represented with &&
- An expression is an rvalue if it results in a tempory object.

e.g:
```
int x = 10;
int &rr = 10;        ×   & → represents lvalue reference
int && rv = 10;      ✓   && → represents rvalue reference

int &x = foo(10);    ×
int && x = foo(10);  ✓
```

e.g ③

```
void f (int & l) { cout << " lvaluef " <<endl; }
void f (int && r) { cout << " rvalue f "<<endl; }

main()
{
   int i = 10;   // i: lvalue, 10: rvalue
   f(i);  // lvaluef
   f(10);  // rvaluef
   f(std:: move (i)); // rvaluef   (move takes lvalue and convert it to rvalue).
}
```

e.g ④

```
① void f(int & f) { cout << " lvalue " <<endl; }
② void f (const int & f) { cout << " const lvalue " <<endl; }
③ void f (int && f) { cout << " rvalue " <<endl; }

main()
{
   int i = 10;   // i: lvalue, 10: rvalue.
   f(i);  // lvalue
   f(10);  → If ③ is commented then ⇒ const lvalue.
   f(move(i));
   f(static_cast<int &&>(i));   If ③ is not commented ⇒ rvalue.
}
```

⇒ " int && f " get preference over " const int & f ".

* std::move()

⇒ std::move() is a function from C++ standard library for casting to a rvalue reference.

⇒ Internally we can assume it is similar to type cast.

```
static_cast<T&&>(obj T)
```

```
void print(std::vector<int>& v)
{
    cout << "print vector" << v.size() << endl;
    for(auto it = v.begin(); it != v.end(); it++)
        cout << *it << " ";
    cout << endl;
}

template<typename T>
void swap(T& a, T& b)
{
    T tmp = std::move(a);
    a = move(b);
    b = move(tmp);
}
```

```
int main()
{
    std::vector<int> v1;
    vector<int> v2;
    for(auto i=0; i<5; i++)
        v1.push_back(i);
    for(auto i=10; i<15; i++)
        v2.push_back(i);
```

print(v1); ⇒ print vector 5
0 1 2 3 4

print(v2); ⇒ print vector 5
10 11 12 13 14

```
v1=v2;
```

print(v1); ⇒ print vector 5
10 11 12 13 14

print(v2); ⇒ print vector 5
10 11 12 13 14

Elements of v2 are copied in v1 and v2 still exists.

```
//* v1 = std::move(v2)
```

print(v1); ⇒ print vector 5
10 11 12 13 14

print(v2); ⇒ print vector 0

Elements of v2 are moved in v1 and now v2 is empty.

```
*//
    swap(v1,v2);
```

print(v1); ⇒ print vector 5
10 11 12 13 14

print(v2); ⇒ print vector 5
0 1 2 3 4

```
    return 0;
}
```

* Move Constructor

→ In class, in copy constructor we pass l value.

→ We can now also use "move constructor".

```
class A
{
    A(const A&) {}
    A(const A&&) {}
    ...
};
```

e.g:

```
class A
{ public:
    A() { cout << "Default Constructor" << endl; }
    A(const A&) { cout << "Copy Constructor" << endl }
    A(const A&&) { cout << "Move constructor" << endl };
}

int main()
{
    A a;          → Default Constructor
    A b = a;      → Copy constructor.
    A c = move(b); → move constructor

    A d(c);       → copy constructor
    A e(move(d)); → move constructor.

    return 0;
}
```

e.g.:②

```
class A
{
    int *p;
public:
    A()                                   // Default Constructor
    {
        p= new int[100];
        cout << "Default Const." <<endl;
    }

    A(const A& aObj)                      // Copy Constructor
    {
        p= new int[100]
        // copy everything from aObj to *this
        for(auto i=0; i<100; i++)
            p[i] = aObj.p[i];
        cout << "Copy Const" <<endl;
    }

    A(const A&& obj) : p(obj.p)           // move constructor.
    {   cout << "move const" <<endl;
    }
};
```

* Move Assignment Operator

```
class A
{
    A & operator = (const A & ) {}
    A & operator = (const A&&) {}
    ----
};
```

eg:① same as previous (Example on back page) with operator overloading as follow.

```
class A
{
⇒   A& operator = (const A & obj)
    {
        std:: cout << " Copy Assignment " <<endl;
        p = new int [100];
        // copy everything from obj to *this
        for(auto i=0 ; i<100 ; i++)
            p[i] = obj.p[i];
    }

⇒   A& operator = ( A&& obj)
    {   cout << " Move Assignment " <<endl;
        // free the resources of this object.
        delete [] p;
        obj.p = nullptr;
        return * this;
    }
};
```

// Can watch this lecture again.
short video (knowledge center)

Section: Lambda Functions

Syntax:

```
auto someVar = [] (int a, int b) -> int {
    // Body of the function
};
```

auto someVar: We want to create a variable called "someVar" and its type is upto the compiler, auto. someVar will be something like "function pointer".

[]: Capture List: Capture list is sometimes called as "closure." List of the local variables we want to be able to use within the body of Lambda function.

(int a, int b): Parameter list for the Lambda function.

->: It's a optional when the return type is obivious, you don't need to supply this array nor a return type.

int: The return value. If the return type is easy to determine, this is optional along with the arrow thing.

⇒ Shortest Lambda function

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

Code →

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
using namespace std;
```

①

```
int main()
{
    auto sayHelloWorld = []()
    {
        cout << "Hello World" << endl;
    };

    sayHelloWorld();
}
```

②

```
int main()
{
    auto getProduct = [](int a, int b)
    {
        return a*b;
    };

    cout << getProduct(10,20) << endl;
    return 0;
}
```

* Capture List

→ By default, we can't use any outside variable inside the body of Lambda function. But that's exactly what the Capture List is for!

→ We can capture specific variables by supplying their names in a comma separated list in Capture List.

e.g: [myVariable, i, g, someOtherVar]

→ But this will just copy the values, like a parameter "passed by value."

IMP If you want to pass "variable by reference", you need to supply a "&" beside it's name.

e.g: [&myVariable, &i, &g]

e.g: ①

```
int main()
{
    int i=10;
    int g=11;
    auto getSum = [i,g](int a, int b) -> int
    {
        // i=50;   ✗ Fail: As parameter is passed by value
        // g=30;   ✗ so we cannot change the value of
                     outside variable.
        return a+b+i+g;
    };

    cout << getSum(190,10) << endl;      o/p: 221
}
```

②

```
int main()
{
  int i=10;
  int g=11;
  auto getsum = [&i, &g](int a, int b) -> int
  {  i=30;   ✓
     g=50;   ✓   Pass: We can change the value as they are passed by reference.
     return a+b+i+g;
  };
cout<<getsum(190,10)<<endl;   o/p: 280
}
```

③

```
auto getsum = [=](int a, int b) -> int.
{
  // All outside variable will be automatically passed by value in Lambda function.
}
```

i.e. i & g will be accessible & read only.

④

```
auto getsum = [&](int a, int b) -> int
{
  // All values are passed by reference.
}
```

⑤

```
auto getsum = [=, &i](int a, int b) -> int
{
}
```

i will be passed as reference and all other variable except i will be passed as value.

→ So we have full control over which outside variable should be passed as value and which should be passe as reference.

* Lambda and for_each

→ Below is an example of using lambda with "for_each" and a vector.

→ We could easily write a separate function "Add(int i)", but the lambda is much more concise.

→ It is a function defined in the middle of the code, exactly where it's needed.

```
int main()
{
    std::vector<int> arr = {1, 2, 3, 4, 5, 6};
    double total = 0;

    std::for_each(begin(arr), end(arr),
                  [&](int x) {
                      total += x;
                  });                    // Basically passing lambda function as third parameter of for_each loop.

    cout << "Sum is " << total << endl;

    return 0;
}

for_each(arr.begin(), arr.end(), [&total](int x) {total += x;});
```

* Lambda and std::function ().

```
#include <iostream>
#include <functional>
using namespace std;

void performOperation(function<void()> f)
{
    f();
}

int main()
{
    int x=100;
    auto func = [&]() { x++; };
    performOperation(func);
    cout<<"x: "<<x<<endl;
    return 0;
}
```

std::function

② lambda function will be executed here. & x will be incremented.

③ We are incrementing x in performOperation and it seems out of scope for x. but this is advantage of lambda. that we can access x outside as we have passed by reference.

① lambda function is passed as parameter.

⇒ Conclusion

① Lambda do not offers any new functionality, but they allow us to write simple functions in place., in the middle of our code, where they are needed.

② This saves us time because we don't need to define a new class or function external to the code.

Section : Casting in C++

① Const_cast <> ()

→ The expression `count_cast <T> (v)` can be used to change the const or volatile qualifiers of pointers or references.

→ where T must be pointer, reference or pointer to member type.

```
#include <iostream>
using namespace std;
```

1. when the actual referred object/variable is not const.

```
int main()
{
```

Ex:1 Invalid

```
    const int a1 = 10;      // Ideally we should not use cast when
                               variable is "const"
    const int * p = &a1;

    int * p2 = const_cast < int * > (p);
    * p2 = 15;              // Invalid or undefined behavior

    cout << a << endl;      // O/P: 10
    cout << * p2 << endl;   // O/P: 15
```

O/P is 10 & 15 as initially compiler directly stores value 10 for a. and at address of a 15 is stores.

→ There will be no compiler error but this is Invalid behavior.

```
//Ex 2: valid.

    int a2 = 20;                // As here variable a2 is not
                                   const hence this is valid
    const int * p3 = &a2;

    int *p4 = const_cast < int *> (p3);
    * p4 = 50;

    cout << a2 << endl;     O/P: 50
    cout << * p4 << endl;   O/P: 50

    return 0;
}
```

2. When we need to call some 3'rd party library where it is taking variable/object as non-const but not changing that.

```
Void thirdPartyLibrary (int *x)
{
    int k=10;
    cout << k + *(x);
}

int main()
{
    const int x=20;
    const int *px= &x;
    thirdPartyLibrary ( const_cast <int*>(px));
}
```

→ Here third Party Library API needs data in non const form but we have it in const form. so we convert it in non const form and pass.

→ Provided that API just use the data and don't try to modify. If It tries to modify then we will get undefined behaviour.

//Bottom Line

NEVER USE THIS

1. Use only when you have to
2. Use only when the actual referred object/variable is not const
3. above example case.

② Static_cast <> ()

1. It performs implicit conversion between types.

```
int main()
{
    float f = 3.5;
    int a;

    a = f;
    a = static_cast<int>(f);

    return 0;
}
```

} This both are doing same conversion.

⇒ In implicit conversion (a=f), if in the large code if some errors comes and we have to find where we have done ~~int~~ conversion. then it is very hard to find this implicit c-type conversion.

→ But if we use explicity static_cast conversion then with keyword static_cast we can find where we done conversions.

2. Use static_cast when conversion between types is provided through "conversion operator" or "conversion constructor."

```
class int
{
    int x;
public:
    Int (int x=0) : x{x}
    {
        cout << " conversion constructor " << endl;
    }

    Operator string()
    {
        cout << " conversion operator " << endl;
        return to_string(x);
    }
};
```

Conversion Operator: They are specialized member function that make it possible to directly or automatically convert an object from one data type to another

Conversion Constructor: It is a single parameter constructor that is declared without the function specifier explicitly.

```
int main()
{
    Int obj(3);
    string str1 = obj;
    obj = 20;
    string str2 = static_cast<string>(obj);
    obj = static_cast<Int>(30);
}
```

→ O/P: conversion Operator (conversion is done inside operator overloading)

→ O/P: conversion Constructor
(As it is a single parameter constructor "=" will directly call the constructor while assigning 20 to object.)
only valid for single parameter.

} This is will do same work as above two lines

⇒ But in large code when 3rd person is looking or debugging so he won't understand ~~wrote~~ which type of conversion has been done.
so it is always better to do static_cast instead of normal C++ assignment.

3. static_cast is more restrictive than c-style.

Example: char* to int* is allowed in c-style but not with static_cast.

```
int main()
{
    char c;        // 1 byte data
    int *p = (int*)&c   // 4 byte data
    *p = 5;        // PASS at compile-time but FAIL at run-time
                   (that's why it is dangerous)

    int * p1 = static_cast<int*>(&c)   //FAIL: compile time error
                                       because not compatible
                                       pointer type.

    return 0;
}
```

→ char type acquire 1 byte of memory while int type acquires 4 byte of memory.

→ If we cast char* to int* and then try to modify the value. then we are trying to store 4 byte size data in 1 byte size location.

→ So this is wrong, it will try to acquire next available memory and this can corrupt our memory location

→ So this type of conversion should be restricted and in static_cast it is restricted. It throws compile time error.

4. static_cast avoid cast from derived to private Base pointer
   → It allows to do for public Base pointer.

```
class Base {};
class Derived : private Base {};

int main()
{
    Derived d1;
    Base * bp1 = (Base*)&d1;   // Allowed at compile time
    Base *bp2 = static_cast <Base *>(&d1)  // fail: Not allowed
                                           at compile time.
}
```

5. When to use when not to use
   Use for all upcasts, but never use for confused down cast.

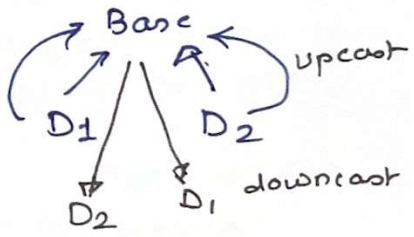
```
class Base {};
class Derived d1 : public Base {};
class Derived d2 : public Base {};

int main()
{
    Derived d1;
    Derived d2;

    Base *bp1 = static_cast <Base *> (&d1);
    Base * bp2 = static_cast < Base*> (&d2);

    Derived *d1p = static_cast <Derived1 *> (bp2);
    Derived *d2p = static_cast <Derived2 *> (bp1);

    return 0;
}
```



Upcasting is fine

Downcasting is not fine

As we are cross down casting and this is confusing.

So we should avoid.

6. static_cast should be preferred when converting to void* OR from void*.

```
int main()
{
    int i=10;
    void *v = static_cast<void*>(&i);
    int *ip = static_cast<int*>(v);
    return 0;
}
```

// BOTTOM Line for static_cast.

1. Use for compatible type conversion, such as float to int.
2. Use for conversion operator and conversion constructors.
3. To avoid unrelated pointer conversion
   e.g: char* to int*.
4. Avoid derived to private base pointer conversion.
5. Use for all up-cast but never use for confused down cast because there are no run time checks performed for static cast conversions.
6. Intentions are more clear in c++ style cast (express your intent better and make code review easier)
7. Finding in code is easy. can find just with keyword static_cast.
8. Error found at compile time.

③ dynamic_cast < > ( )

Syntax: dynamic_cast <new_type> (expression)

1. dynamic_cast is used at run_time to find out correct down-cast
2. As it is used at run-time so it has overhead.
3. Need atleast one virtual function in base class.
4. If the cast is succesful, dynamic cast returns a value of type new-type.
5. If the cast is fails and new-type is a pointer type, it returns a null pointer of that type.
6. If the cast fails and new type is a reference type, it throws an exception that matches a handler of type std::bad_cast.

Code:

```
#include <iostream>
#include <exception>
using namespace std;

class Base {
    virtual void print () { cout << "Base" <<endl;}
};

class Derived1 : public Base {
    void print () { cout << "Derived1" <<endl; }
};

class Derived2 : public Base {
    void print () { cout << "Derived2" <<endl;}
};
```

```
int main()
{
    Derived d1;
    Base * bp = dynamic_cast < Base * > (&d1); // upcast fine

    Derived 2 *dp2 = dynamic_ cast < Derived 2* > (bp);

    if (dp2 = NULL)
        cout << "NULL " <<endl;
    else
        cout << "NOT NULL" <<endl;
```

O/P : NULL

As we have upcasted from d1 to derived1 to base and now we are downcasting from base to derived2. so it is wrong as base pointer contains information of derived1 . not derived 2.

OR if

```
Derived 1 *dp2 = dynamic_cast < Derived1 *) (bp);
    if (dp2 = NULL)
        cout << "NULL" <<endl;
    else
        cout << " NOT NULL <<endl;

    try {
        Derived 1 &d1 = dynamic_cast < Derived 1 & > (d1);
    }
    catch (std:: exception & e) {
        cout << e.what() <<endl;
    }
}
```

O/p : Not null.

O/p : std:: bad_cast.

1→ work only on polymorphic base class (at least one virtual function in base class) because it uses this information to decide about wrong down cast.

2→ It is used for up-cast (D→B) and down cast (B→D) but it is mainly used for correct down cast.

3→ using this cast has run time overhead, because it checks object type at run time using RTTI (Run time Type Information)

4→ If we are sure that we will never cast to wrong object then we should always avoid dynamic_cast and instead use static_cast.

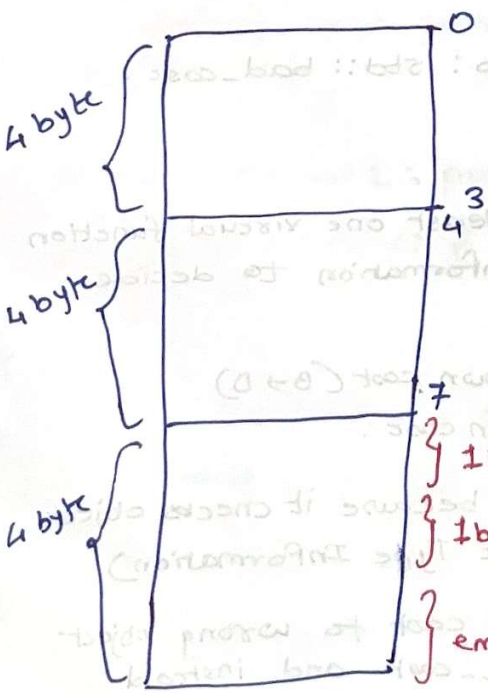
④ reinterpret_cast < > ( )

1. It can perform dangerous conversions because it can typecast any pointer to any other pointer.

```
class Mango {
  public:
    void eatMango() { cout << "eating Mango" << endl; }
};

class Banana {
  public:
    void eatBanana() { cout << "eating Banana" << endl; }
};

int main()
{
    Banana * b = new Banana();
    Mango * m = new Mango();
    Banana *newbanana = reinterpret_cast <Banana*> (m);
    newbanana -> eatBanana();        o/p : eating Banana.
    return 0;
}
```



int *p = reinterpret_cast <int*> (&5);

o/p = 5

p++

o/p = 10

p++

char *c = reinterpret_cast <char*> (p)

o/p = a

reinterpret_cast <bool> (c++)

++c

o/p = 1

2. It is used when you want to work with bits.

```
struct mystruct
{
    int x;
    int y;
    char c;
    bool b;
};

int main()
{
    mystruct s;
    s.x = 5;   s.y = 10;   s.c = 'a';   s.b = true;
    int *p = reinterpret_cast<int*>(&s);
    cout << *p << endl;          o/p = 5

    p++;
    cout << *p << endl;          o/p = 10

    p++;
    // cout << *p << endl;       o/p = some random integer
                                 as compile will try to convert into
                                 int as it is moving 4 byte.
```

commented

```
    char *c = reinterpret_cast<char*>(p);
    cout << *c << endl;          o/p = 'a'
```

It will read only 1 byte memory and will take it as char and print/convert.

```
    cout << *(reinterpret_cast<bool*>(++c)) << endl;
```

o/p: ~~1~~ 1 (true).

```
    return 0;
}
```

// Bottom line

1 → The result of reinterpret_cast can not safely be used for anything other than being cast back to its original type e.g: 2.

2 → We should be very careful when using this cast.

3 → If we use this type of cast then it becomes non-portable product.

i.e. code written in one PC can not be surely use in other PC/system.

String atoi() : Ascii (i) to Int (i) conversion

```
#include <iostream>
#include <string>
using namespace std;

int myAtoi (/* char *s */ string s)
{
    int num = 0;
    int sign = 1;
    int i = 0;

    while (s[i] == ' ')
        i++;

    if (s[i++] == '-')
        sign = -1

    while (s[i] != '\0')      // checking Null or end of string.
    {
        num = num * 10 + s[i] - '0';
        i++;
    }

    return num * sign;
}

int main()
{
    // char str[] = "-1234";
    string s1 = "     -4596";
    int val = myAtoi(s1);
    cout << val << endl;

    return 0;
}
```

commented

// ASCII code for 0 is 48
and for 1 it is 49.
So taking diff to get Int 1.
// s[i] - '0' == 49 - 48 = 1

o/p: -4596