CS 304

Ch #19:

→ Stream Insertion Operator

In C++ "<<" is used to display the output on the screen.

```
Int i = 1;    cout << i;
```

→ Stream Exertion Operator

In C++ " >> " is used to get data from input stream, again default input stream is from console or from file or network.

```
int i;        cin >> i;
```

→ Overloading Stream Insertion & Exertion Operator :

```
class Test
{   // Extraction Operator Overloading
  Public:
    int age ; string name ;
  friend istream &operator >>(istream &input,
      Test &i)
    {  input >> i.name >> i.age ; return input;}
// Insertion
Operator  friend ostream &operator <<(ostream &output,
overloading
      Test & o)
    {  output << o.name << o.age ; return output;}
};
main( ){  Test T1 ; cin >> T1 ; cout << T1 ;
      getch( ); return 0;}
```

CS 304

Ch# 20:

→ Subscript [] operator

For accessing the value of different indexes, we must be overloading the subscript [] operator. This operator tells us the size of array.

→ Overloading Subscript [] Operator :

```
class Test
{   Private:
      int arr[10];
    Public:
    Test() { for(int i=0; i<=9 ; i++)
                { arr[i] = i+2; } }
    int &operator [](int index)
    {
      if( index > 9 )
        cout << "\n\n Index Out of Bound";
      else
        return arr[index]; }
};
main() {
      Test T;
      cout << "\n\n T[0] -> " << T[0];
      cout << "\n\n T[5] -> " << T[5];
      getch();
      return 0;
    }
```

→ OverLoading Function ( ) Operator:

- Must be a Member Function.
- Any number of Parameter can be specified.
- Any return type can be specified.
- Operator ( ) can perform any generic operation.

```
Class Test
{   Private:
        int age ; string name ;
    Public:
    Test ( ) {  age = 0  , name = " " ;}
    Test (int a , string n) { age = a , name = n;}
    Test operator ( ) (int a, string n)
    {
        Test obj;
        obj.age = a  , obj.name = n ; return obj ;}
    void display ( )
    {
        cout << "\n\n Name: " << name ;
        cout << "\t Age: " << age ;
    }
};
  main ( )
{
   Test T1 (22 , "Ali"), T2 ;   T1.display ( );
   T2 = T1 ( 13 , "Ahmad");   T2.display ( );
    getch ( );
    return 0 ;
}
```

```
Class Test
{
    Private:
        int a , b ;
    Public:
    Test ( ) { a = 0 , b = 0 ;}
    Test (int a , int b){ this -> a = a ;
                          this -> b = b ;}
    // Negation operator,
    Test operator -( )
    { int a = -a ; int b = -b ;
    void display ( )
    {
```

CS 304

Chapter No 23:

→ Accessing base class member functions in derived class:

Public methods of base class can directly be accessed in its derived class.

→ "Protected" access specifier:

This ensure that function in base class is accessible in derived class of this base class and NOT outside the class.

→ Disadvantages of protected Members:

Breaks encapsulation :-

"A class data members and Functions should be encapsulated in the class itself." So, we can say that Protected members breaks the principal of Encapsulation.

→ "Is A" Relationship:

Generally we can say that,

"Drived object IS A kind of Base object."

Static Type:

The type that is used to declare a reference or pointer is called its static type.

1. In Person *pPtr = 0;

   Static type of pPtr is Person*

2. Student s

   Static type of s is student.

→ Implicit & Explicit IS A Relation:

If you convert the object of base class into the any object of base class. This is called implicit. (Same class k dject ko same class ma convert krna)

If you convert the object of child class into the object of base class. This is called explicit.

→ Code For Implicit

```
class Base
{
    Public:
        Void getName( )
        { cout << "\n\n My Name is Ali";
        }
};
class Drived: Public Base
{
  Public:
      void getRollNo( )
```

```
{ cout << "\n\n My Roll No. is BT-234"; }
};
main ()
{
  Base * p , obj1 ;
  Drived *d , obj2;
  p = &obj1 ; // Implicit Type Casting
  d = &obj2 ; // Implicit Type Casting
  p -> getName ( );
  d -> getRollNo ( );
  getch ( );
  return 0;
}
```

→ Code For Explicit

```
Class Base {
        Public :
            Void getName( )
        { cout << "\n\n My Name is Ali" ; }
            vitrtual void getMarks( ) = 0;
};
class Drived : Public Base
{
    Public :
      void getRollNo ( )
    { cout                    }

    void getMarks ( )
    {
      cout<< "\n\n Drived class Function"};
};
  main( )
{
  Base * p;  Drived d;  p = &d; // Explicit typ Casting

  p-> getNam( );      p-> getMarks( );

  getch( );   return 0;
}
```

Lecture No 24:

→ Copy Constructor:

Copy constructor is a member function of a class that is used to create an object of that class by copying values from an already created object.

→ Shallow Copy:

Compiler by default uses shallow copy which simply copies all values of already existing object to newly created object.

e.g;

Line 1 : student sobj1("Ali", "CS");

Line 2 : student sobj2 = sobj1 ; [shallow copy]

→ Deep Copy:

A deep copy is a copying process where a new independent copy of an object is created, including all of its attributes and sub objects.

- Independent -copy
- Recursive copying
- New memory allocation.

→ Assignment Operator

In the case of inheritance when we assign one derived class object to another derive class object compiler uses assignment operator.

- Derived class also contain implict base class part In case of compiler generated assignment operator by itself.

- In case ever programmer has to call assignment operator of base class explicit In case of user-defined assignment operator.

→ Type Conversion:

C++ is strongly typed language. So, we need to convert one type in other type before using it.

This Conversion from one type to another type is called Casting.

→ Casting

Casting can be done in two ways,

1- Implicit Casting Conversion:

Conversions that compiler can perform automatically using build in casting operations without casting request from programmer.

2. Expliciting Casting Conversion:

Conversion in which programmer requests the compiler to convert one type to another.

→ Casting Operators

The C++ draft standard includes the following four casting operators,

1. static_cast ----> downcasting
2. const_cast ---->
3. dynamic_cast ----> (used for) Upcasting
4. reinterpret_cast ....> downcasting

In Perspective of inheritance casting can be of two kinds,

a) Upcasting (casting from derived to base class)
b) Down casting (casting from base to derived class)

→ Code for Static_cast

```
#include<stdlib.h> ,<conio.h> , <iostream>
class Base
{
   Public:
      void myFunction ()
   { cout << "\n\n Base class Function";
   }
};
```

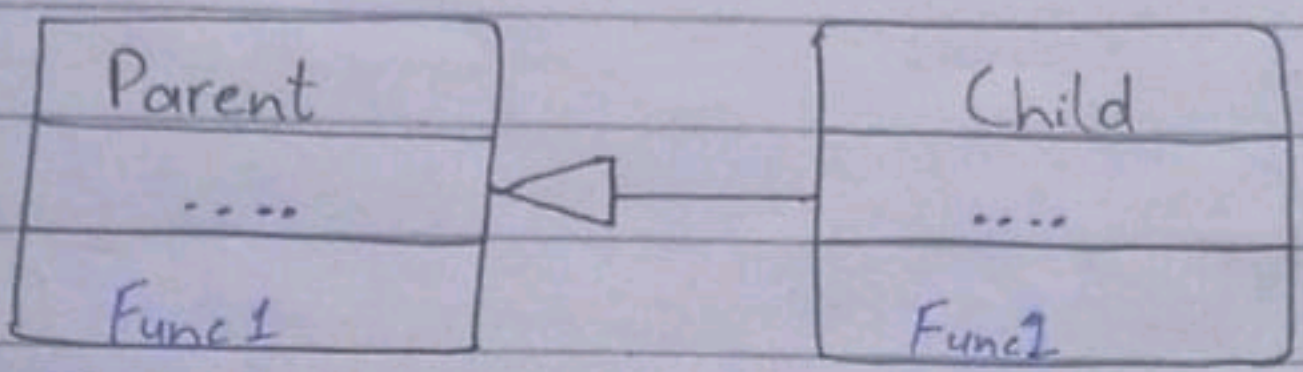
```
class Derived: public Base
{
    int b;
public:
    void myFunction()
    { cout << "\n\n Drived class Function";
    }
};
main()
{
    Base *b = new Derived;
    b -> myFunction();
    Drived *d = static_cast <Drived*> (b);
    d -> myFunction();
    getch();
    return 0;
}
```

```
// dynamic_cast
{
Drived *d   = new Drived;
d -> myFunction();
Base *b = dynamic_cast < Base*> (d);
b -> myFunction()
    getch();
    return 0;
}
```

Lecture No 25:

→ Overriding

To override a function the derived class simply provids a function with the same signature (Prototype) as that of its base class.



→ Function Overriding Vs Overloading:

Function overloading is done within the scope of one class. whereas
Function overriding is done in scope of parent and child classes (inheritance).

- Two functions in same class with same name but different parameter & return type. - - - -.....-> Overloading

- Two functions in different class (base & derived) with same name, parameter & return type.
----> Overriding

Lecture No 26:

→ Base Initialization

- The child can only call constructor of its direct base class to perform its initialization using its constructor initialization list.

- The child cannot call the constructor of any of its indirect base classes.

Example Code given below:

```
#include <iostream>
#include <conio.h>
Usingnamespace std;
class GrandParent
{
    int gp;
    Public:
        GrandParent ( ): gp (0)
        {
            cout << "\n\n gp: " << gp;
        }
};
class Parent : Public GrandParent { } ;
{
    int p;
    Public:
        Parent ( ) : GranParent ( ) , p (1o)
        {
            cout << "\n\n p: " << p;
        }
};
```

```
class Child : public Parent
{
    int c;
    Public :
        Child ( ) : Parent ( ) , c (20)
    {
        cout << "\n\n c : " << c ;
    }
};
  main ( )
{
   Child obj;
   getch ( );
   return 0 ;
}
```

→ Types of Inheritance

1- Public

2- Protected

3- Private

we use keywords (public, protected, private).

1- Public Inheritance :

class Child : public Parent { . . . . };

2- Protected Inheritance :

class Child : protected Parent { . . . . };

3- Private : If the user does't define the type of inheritance then the default type is "private".

Lecture No 27:

→ Code For Public Inheritance :

```
class Base
{
    private: int a;
    protected: int b;
    Public:
        void getData ()
        {
            a = 10;
            b = 20;
            cout << "\n\n a: " << a << "\t b: " << b;
        }
};
class Drived : public Base
{
    Public:
        void getData ()
        {
            b = 25;
            Base:: getData ();
        }
};
main ()
{
    Drived d;
    d.getData ();
    getch ();
    return 0;
}
```

Member Access:	
Base	Drived
Public	Public
Protected	Protected
Private	Hidden

Outsid the Child class, accessed only Public data member.

→ Code For Protected Inheritance :

```
class Base
{
    private : int a ;
    protected : int b ;
    public :
        void getData ( )
        {
            a = 10 ;   b = 20 ;
            cout << "\n\n a: " << a << "\tb: " << b ;
        }
};
class Drived : protected Base
{
    public :
        void getData ( )
        {
            Base :: getData ( ) ;
        }
};
main ( )
{
    Drived d ;
    d. getData ( ) ;
    d . Base :: getData ( ) ; // Error
    getch ( ) ;
    return 0 ;
}
```

In case of Protected inheritance public & protected data members can be accessed in only the Child class. not outside the child class. → if you try, // error occured

Lecture No 28:

→ Polymorphism Revisited:

In OO model, Polymorphism means that different objects can behave in different ways. for the same message (stimulus).

In Inheritance relationship:

In case of shape hierarchy (line, Circle, Triangle and so on.....) our program will keep working correctly if we add more shapes, draw method of appropriate "shape class" should automatically be called, that is benefit of Object Oriented Programing.

To achieve this type of functionality we have the concept of Virtual Function.

"We make those functions virtual in base class which will be implemented by drived classes according to their requirements."

- Keyword "virtual" before the function header
- Static binding means that target function ~~Staty~~ for a call is selected at compile time.
- Dynamic binding ..... target function for a call is selected at runtime.

→ Code for Virtual Function :

```
class shape
{
  Public:
  virtual void draw ()
       { cout << "\n\n Draw Function of
         shape Class"; }
};
Class Line : public shape
{
     Public :
       void draw ( )
     { cout << "\n\n Draw Function of Line Class";}
};
Class Circle : Public shape
{    Public:
       void draw ( )
     {    cout<< "\n\n Circle Class"; }
};
Class Triangle : public shape
{      "      "      "      "      };

main  ( )
{
   shape *s ;
   s = new Line ;
   s ⟶ draw ( );
                         s = new Circle;
                         s ⟶ draw ( );
                         getch ();
                         return 0;
                         };
```

Lecture No 29:

→ Concrete Classes

Concrete classes Implements a concrete concept they can be instantiated they may inherit from an abstract class or another concrete class.

→ Abstract Classes

In C++, we can make a class abstract by making its function(s) pure virtual. Conversely, a class with no pure virtual function is a concrete class (which object can be instantiated).

→ Pure Virtual Functions

Virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise the derived class will also become an abstract class.

```
virtual void draw() =0;
```

Lecture No 31.

→ Multiple Inheritance

A class in C++ can inherit from more than one classes. like Phone class can inherit from Transmitter or Reciever.

The derived class inherits data members and functions from all the base classes.

Object of derived class can perform all the tasks that an object of base class can perform.

* The pointer of one base class can't be used to call the function of another base class.

Problems in Multiple Inheritance :

- If more than one base class have a function with same signature then the child will have two copies of that function.

⇒ Calling such functions will result in ambiguity (making confussion)

→ Virtual Destructor

- The destructor is called according to static type of any class pointer.
- We will call destructor using delete operator the destructor of base class Shape will be called as static type of array is Shape. This will destroy the base class object only, derived class object will not be destroyed.

```
class Shape {
    ....
    public :
    virtual ~ Shape ( ) {
    cout << "\n\n Shape Destructor "; }
    };

main ( )
{ Shape * pShape = new Recrangle ;
  delete pShape ;
  return 0 ;
}
```

★ Base class destructor will run ~~after~~ after the derived class destructor.

```
Class Quad : public Shape
```

```
class Rectangle : public Shape
{
    public :
    ....
};
```

→ V Table

We see compiler keeps track of virtual functions and call them correctly according to nature of the object w.r.t which they are called, Compiler builds a virtual function table (vTable) for each class having virtual functions.

→ Dynamic Dispatch
(Dynamic Binding)

In case of "virtual", for "non-virtual" functions, Compiler just generates code to call the function.

In case of Virtual functions, compiler generates code to

- access the object
- access the associated vTable
- call the appropriate function

In Conclusion:

Virtual Functions should be added in code :

- Memory overhead due to V-Tables
- Processing overhead due to extra pointer manipulation.

→ Code for Multiple inheritance :

```
class A
{
  int a;
  Public :
     A( )
  {
     a = 50;
  }
   void getvalue ( ){
   cout << "\n\n Value of a: " << a;
   }
};
 class B
{
   int b ;
   public :
      B ( ){
   b = 100; }
      void getvalue ( ) {
      cout << "\n\n Value of b: " << b;
      }
};
class C :public A ,Public B { };
main ( )
{
   C obj;
   obj. A:: getvalue ( );
   obj. B:: getvalue ( );
   getch ( );
   return 0 ;
}
```

Lecture No 32:

→ Generic Programming

Generic programming reffers to programs containing generic abstractions, then we instantiate that generic program abstraction (function, class) for a particular data type, such abstractions can work with many different types of data.

Advantages:

a. Reusability: Code can work for all data types.
b. Writability: Code takes lesser time to write.
c. Maintainability: Code is easy to maintain.

→ Templates

In C++ generic programming is done using templates.

Two kinds:

a. **Function Templates** (in case we want to write general function like printArray)

```
template <class T>    // <typeName T>
```

b. **Class Templates** (in case we want to write general class like Array class)

```
void funName (T x);
```

Lecture No 33.

→ Multiple Type Arguments

To convert different types into one another (like char to int, int to float or int to char etc).

So, the concept of templates can be used here as well to write general function to convert one type to another, in this case we need two type arguments or multiple type arguments.

→ Code for Multiple Argument

```
#include <iostream>
using namespace std;
template < class K, class H >
 void Print (K a, H b, K c)
{
    Cout << "\n\n Ans= " << a+b+c ;
}

main () {
   Print (10, 4.5, 2) ;
   getch ();
   return 0 ;
}
```

→ Overloading Vs Template

- '+' operator is overloaded for different types of operands

- A single function template can calculate sum of array of many types.

→ Types of containers

Supporting two operations,

- Increment operator (++) as we are incrementing values in this container.

- Dereference operator (*) as we are getting value from container for comparison by defeferencing.

Lecture No 34:

→ Generic Algorithms

We can apply the same concept of Generic Algorithms to class templates and develop a generic Vector class that will work for all built-in types.

Class Templates

A type in which different types of classes converted.

- template < class T > class Xyz { --- };
- template < typename T > class Xyz { --- };

```
#include <iostream>
#include <conio.h>
using namespace std;
template <class T>
class myClass
{
    private:
        T a, b;
    Public:
        myclass (T, T);
        T sum ();
};
```

```
template <class T>
myClass::myclass(T num1, T num2)
{
        a = num1;
        b = num2;
}
 template <class T>
T myClass<T> :: Sum()
{
      return a+b;
}
 main()
{
   myClass<int> intNum(10,20);
   myClass<float> floatNum(5.6,2.7);
  Cout << "\n\n 10+20= " << intNum.sum();
  Cout << "\n\n 5.6+2.7= " << floatNum.sum();
   getch ();
   return 0;
}
```

Lecture no 40:

Cursors

A cursor is a pointer that is declared outside the container/aggregate object.
Aggregate object provides methods that helps a cursor to traverse the elements

- T * first ( )
- T * beyond ( )
- T * next (T*)

Benefit: The main benefit is that we are using external pointer so we can do now any kind of traversal in any way.

Disadvantage: We can't use cursors in place of pointers for all containers.

Iterators

Iterators which are that traverse a container without exposing its internal representation.

- They are not external pointers but internal members of containers.

- Generic Iterator is implemented using templates.

- Generic Iterator Provides us functionality to create any container object using templates and operator overloading

Advantages:

1- More than one traversal can be pending on a single container.

2- Allow to change the traversal strategy without changing the aggregate object.

5- They contribute towards data abstraction by emulating pointers.

Lecture No 41:

Standard Template Library :

STL is a designed to operate efficiently accross many different applications by using different built-in header file.

Three key of components

- Containers
- Iterators
- Algorithms

★ STL promotes reuse, it saves our development time and cost. There is no error due to their use.

STL Containers:

Container is an object that contains a collection of data elements.

1- Sequence Containers · Vector, Deque, list

2- Associative Containers · Fast retrival

3- Container Adapters · Stack (LIFO), queue (FIFO), Priority

Associative Containers

- Set ... No duplicate value bidirectional
- multiset Allow duplicate value "
- map ... No duplicate keys "
- multimap ... Allow duplicate key "

Functions for First-class Containers:

Sequence & Associative container is First-class.

1- begin( ) return the first element of the container.

2- end ( ) refers to the next position beyond the last element.

3- rbegin ( ) refers to the last element

4- rend ( ) refers to the position before the first element

5- erase (iterator) Removes an element

6- erase (iterator, iterator) " " with range

7- clear ( ) erases all elements from containers.

42.5.Sequence Containers

Container Type	Iterator Type	Reason
vector	random access	(as we can access any element of vector using its index so we can use random access iterator)
deque	random access	(in deque we can add elements only in front and back however we can access any element of deque using its index so we can use random access Iterator)
list	bidirectional	(in list we can move in both directions in sequence, however cannot access an element at specific index randomly so we can use bidirectional iterator with list)

In question container type is given and ask that's Iterator type....

Most Important Short / Long question

42.6.Associative Containers

In associative containers we save values based on keys, and we cannot access elements randomly based on indexes as elements are not stored at contiguous memory locations, however, we can traverse them in both directions so, we can use bidirectional iterators with them.

Container Type	Iterator Type	
-- set	--bidirectional	
-- multiset	-- bidirectional	
-- map	-- bidirectional	
-- multimap	-- bidirectional	

42.7.Container Adapters

Container adapters are made with special restrictions, most important restriction is that they don't allow free traversal of their elements so we CANNOT use iterators with them as given below,

Container Type	Iterator Type	
-- stack	-- (none)	
-- queue	-- (none)	
-- priority_queue	-- (none)	

Lecture No 42:

Iterator Categories

1- Input Iterators:

By using this iterators, we can read an element.

Only we can move in forward direction one element at a time.

Operator used

*P , P1 = P2 , P1 == P2 , P1 != P2 , P->

2- Output Iterators:

Same as Input Iterators.

*P , P1 = P2

3- Forward Iterators:

They have both input and output Iterators capability.

They can bookmark a position in the container.

They support operations of both input & output iterators.

4- Bidirectional Iterators:

They have all the capabilities of forward Iterators. They can be moved in backward direction. As a result they support multipass algorithms.

--P , P-- (Pre & Post decrement operator)

5- Random Access:

They have all the capabilities of bidirectional Iterators. They can directly access any element of container.

They support following operators.

p + i	Pointing at p + i
p - i	" " p - 1
p += i	increment iterator p by i position
p -= i	decrement "
p[i]	Returns a reference of element at p+i
p1 < p2	True if p1 is before p2 in the container
p1 <= p2	T if p1 is before p2 or p1 is equal to p2
p1 > p2	True if p1 is after p2
p1 >= p2	T if p1 is after p2 or p1 is equal to p2

Algorithms:

Standard Template ~~Library~~ Library (STL).

→ STL include 70 standard algorithms.

→ These algorithms may use Iterators to manipulate containers.

→ STL algorithms also work for ordinary pointers and data structures.

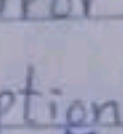
→ A multi-pass algorithm for example, requires bidirectional Iterator(s) at least.

Lecture No 43:

Error Handling Techniques

We use the following techniques for error handling.

1- Abnormal termination
2- Graceful termination
3- Return the illegal value
4- Return error code from a function
5- Exp Exception handling



- It is a "much elegant" solution as compared to other error handling mechanisms.
- It enables separation of main logic and error handling code.

Process:

- Logical code writes in try block.
- If error occured so, we use throws an object for return an error.
- Catch blocks follow try block to catch the thrown object.

Error Handling

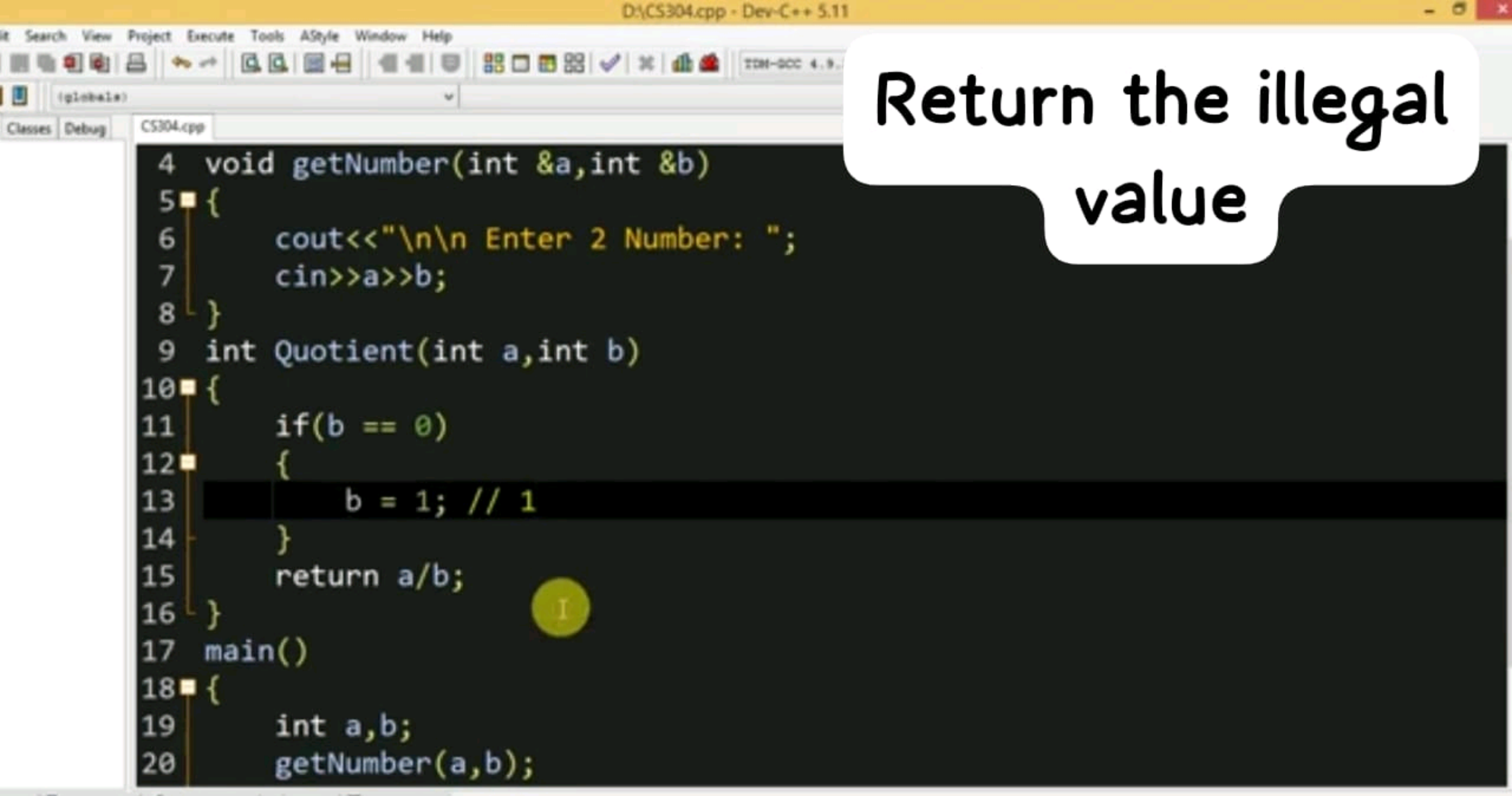
- Issues in Error Handling:
- Programmer sometimes has to change the design to incorporate error handling
- Programmer has to check the return type of the function to know whether an error has occurred
- Programmer of calling function can ignore the return value
- The result of the function might contain illegal value, this may cause a system crash later
- Program's Complexity Increases
  The error handling code increases the complexity of the code
  - Error handling code is mixed with program logic
  - The code becomes less readable
  - Difficult to modify

Return error code

```
 9  bool Quotient(int a,int b)
10  {
11      if(b == 0)
12      {
13          return true;
14      }
15      return false;
16  }
17  main()
18  {
19      int a,b;
20      getNumber(a,b);
21      if(Quotient(a,b))
22      {
23          cout<<"\n\n Denominator Can't Be Zero";
24          exit(1);
25      }
```

Return error code

```
4  void getNumber(int &a,int &b)
5  {
6      cout<<"\n\n Enter 2 Number: ";
7      cin>>a>>b;
8  }
9  Quotient(int a,int b)
10 {
11     if(b == 0)
12     {
13         return true;
14     }
15     return false;
16 }
17 main()
18 {
19     int a,b;
20     getNumber(a,b);
```


D:\CS304.cpp - Dev-C++ 5.11
Search View Project Execute Tools AStyle Window Help
(globals)
Classes Debug
CS304.cpp
Return the illegal value
4 void getNumber(int &a,int &b)
5 {
6 cout<<"\n\n Enter 2 Number: ";
7 cin>>a>>b;
8 }
9 int Quotient(int a,int b)
10 {
11 if(b == 0)
12 {
13 b = 1; // 1
14 }
15 return a/b;
16 }
17 main()
18 {
19 int a,b;
20 getNumber(a,b);

Graceful termination

D:\CS304.cpp - Dev-C++ 5.11

Search View Project Execute Tools AStyle Window Help

CS304.cpp

```
4  void getNumber(int &a,int &b)
5  {
6      cout<<"\n\n Enter 2 Number: ";
7      cin>>a>>b;
8  }
9  int Quotient(int a,int b)
10 {
11     if(b == 0)
12     {
13         cout<<"\n\n Denominator Can't Be Zero";
14         exit(0);
15     }
16     return a/b;
17 }
18 main()
19 {
20     int a,b;
```

Code for Abnormal termination

```
3  using namespace std;
4  void getNumber(int &a,int &b)
5  {
6      cout<<"\n\n Enter 2 Number: ";
7      cin>>a>>b;
8  }
9  int Quotient(int a,int b)
10 {
11     return a/b;
12 }
13 main()
14 {
15     int a,b;
16     getNumber(a,b);
17     cout<<"\n\n Answer of "<<a<<" / "<<b<<" : "<<Quotient(a,b);
18     getch();
19     return 0;
```

When you try 10/0 error occurred. Program terminated abnormally

وَاَطِيْعُوا اللّٰهَ وَالرَّسُوْلَ لَعَلَّكُمْ تُرْحَمُوْنَ

اور اللہ کی اور رسول (صلی اللہ علیہ وآلہ وسلم) کی اطاعت کرو تاکہ تم پر رحم کیا جائے

[القرآن، 3:132]