13/11/23

⊕ Constructor Inheritance. (Not proper name for it)

A
↑
B

A(int)
A(int,int)
A(const A &)

B
{ B(): A(23)
{
}

: HexaRinge (movie)

B(const B &ref): A(ref)
{
}

(Generalized)
A f()
↑
B f()
↑
C k()
↑
D f()

D d1;
C * cp = &d1;
cp -> f(); (B f() runs)

: DSA (Female Teacher)
(Madiha Khalid)

Person
↑
Employee
↑ ↑
Teacher

15/11/23

Commission -> Rate per hour
↓
calculate Pay()

↑

Base Plus Commission -> Rate per hour + Salary.

```
int salary;
calculatePay()
{
    return SalaryOfEmployee::CalculatePay+Salary;
}
```

(*) B obj;

B x = move(obj)

A ← A(A&&)

A(const A&&)

B

: If copy constructor is present only then copy works, default move deleted. If only move constructor is present then move works default copy constructor deleted. (Same in copy constructor)

(*) A A(A&&) ←

A(const A&)

B B(B&& r) (If this call)

B(const B& r) (If this call)

(*) Copy can receives both permanent and temporary object.

int & x = 10; ✗

const int & y = 10; ✓

~~int && x = 10;~~ ✓

(It not access extra things in D)

(*) C *p = new D;

p -> f();

delete p; (C, B, A)

destroys

A f

B f

C K

D f

(*) Polymorphism -> Pointer

Static (p->f()) B f() runs

Dynamic (p->f()) latest C f() runs

(p->f()) ↓ Polymorphic Behaviour

(*) Quiz Related to delete, p->f().

(*) int a; A

A(A && r) : A(const A &&r)

A (const A &r)

A(int i=10) : a1(i)

int b; B

B(int i=5) : b1(i)

{}

B (const B & x) : A(move(r))

{}

8/11/23

Mids Solution

Q:2

"C:\\B\\T\\abc.txt"

".\\T\\abc.txt"

Q3: (i) 101:909
(ii) 101:909

⇒ Test cases (Function Works)

⊙ Inheritance:

```
class Address
{
    String ads;
    String city;
    String country;
    ⋮
};
```

```
class Person
{ String name;
  Address ads;
  String bGroup;
};
```

```
Class Person
{ ⋮
    Person()
    { name = nullptr;
      ads = nullptr;
      bGroup = nullptr;
    }
  Void getPerson(String $n,
                 Address $aD, String $
                 b.G)
```

∴ We have to write setter, getter, assignment operater,

more assignment for it

(*) class Teacher

```
{ Person p;
  double salary;
  string specialization;
}
```

(*) A ——◆ B : B has A

A ——◇ B

→ Whole Part (Aggregation / Composition

→ A ↑ B is-A

is-A

Inheritance

(*) class Teacher : public Person

```
{ // Person p;
  double salary;
  String specialization;
}
int main ()
{ Teacher t;
  t.setName ();
}
```

: Inherited

Person ↑ is-A Teacher

∴ It runs from person

(*) Writing public is not recommended (meaningful)

Class Teacher : ~~public~~ extends Person

(*) Example Dummy Structure with struct A & B.

(*)
```
struct B : public A
{ int z;
  Set z(int);
  get z();
  g();
}
```

```
structA
{ int x;
  int y;
  Void Setx(int);
  int getx();
  }
  void f();
}
```

```
int main()
{ B o1;
  (all features of A class become part of B)
  o1. setx(10);
}
```

x
y
.....
z

First parent Members then child class

P(Base)

C(Derived)

: No wrapper required

(*) Base class is strong.

(*) B can access A but A cannot access B

(*) Functions are not copied in B of A but data members are copied.

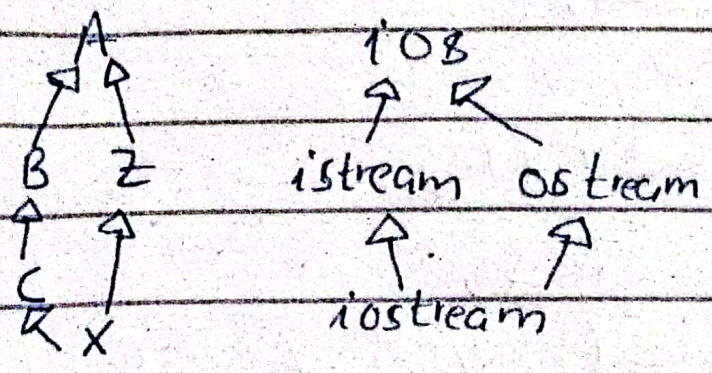
```
struct B
{ A obj;
  int z;
  ;
}
```

B O2

obj	x
	y
z	

For Accessing O2.obj ....
(Not required for inheritance)

(*)



(*) Function Hiding if both class have same name function. Derive class function runs over Base class. If parameter changes then also error in derive and not checks for Base class.

```
struct B : public A
{
    f(int);
}
```

B O1. f(); (Error)

```
struct A
{ ;
  f();
  z()
  { f();
  }
```

(*) A * p = &o1 (only point till A)

p → ....

if B * p = o1; (All Accessible)

p → ....

(*) void z()
{ f();
}
Access f() of A

(*) o1.setx(13);
o1. f(23).
o1 .f(); //o1.A::f()
o1:z(); Access f() of A struct
(This is Dangerous)(Function Hiding)

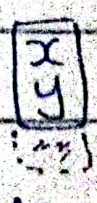
(*) class Person::public Address
{ string name;
Address hAds;
Address wAds;
string bGroup;
}

Flops here because it can point only one time

(multi level)

Address
↑
Person
↑
Teacher / A
(multiple)

(*) A o1;
B* p = &o1;
p→z = 90

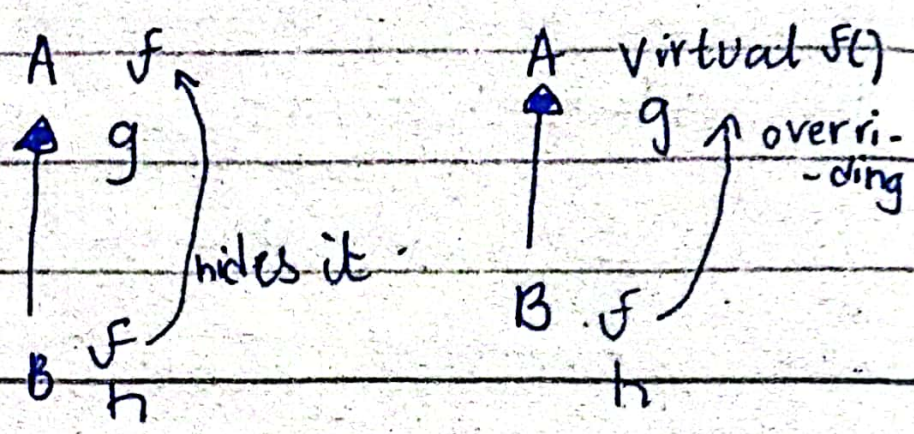


Accessing garbage

Ground & window (Example)

22/11/23

* Polymorphism:



* Use "virtual" keyword.

```
: B obj;
  A *p = &obj;
  p->f();          (B f())
```

* Overloading
  Hiding
  Overriding → (If virtual then override)

(*) A x;

B obj;

A* p = &obj;

$\boxed{p \to f()}$; → This statement act as polymorphic.

Static (Grammar Check over it) (Static type. a)

Dynamic

1- p type a
2- f() accessible
3- No Grammar error

if virtual marked then other steps occurred (stops and move to VT) (Pointer lines executes)

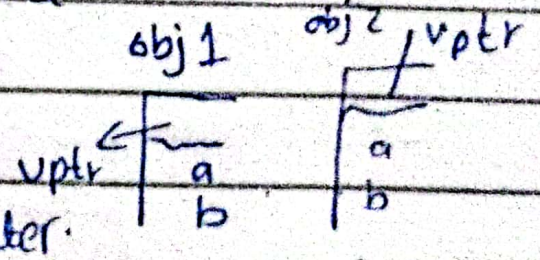
: Compiler can't find whom it is pointing

(*) Mark virtual in the base class. It behaves as polymorphic.

(*) If virtual present in class, then in that class box, some place at first will be allocated.

>vptr (4 bytes or may 8 bytes)

B obj1, obj2;



"vptr" is virtual pointer.

: V Table (Array of function pointers)

(*) Function pointer:

void f(int, double)
{
}

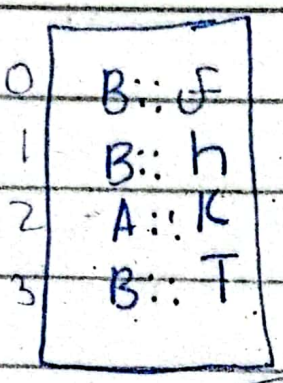
→ void (*p)(int, double) = f;

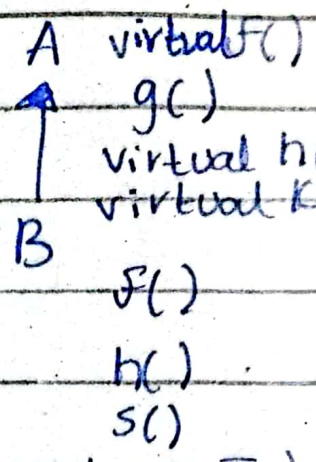
p(2,3).

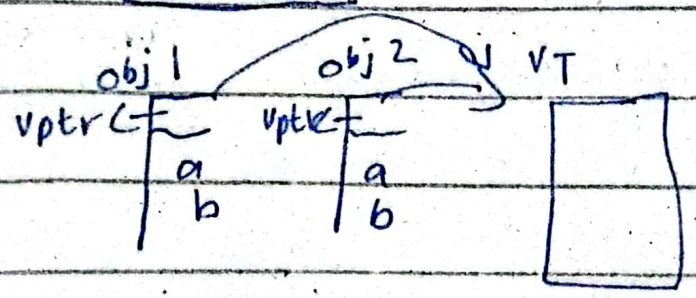
cout << f;
(address of function)

(*) Virtual Table: (if any virtual is present in class)

VT. (by dereferencing we can reach at table)







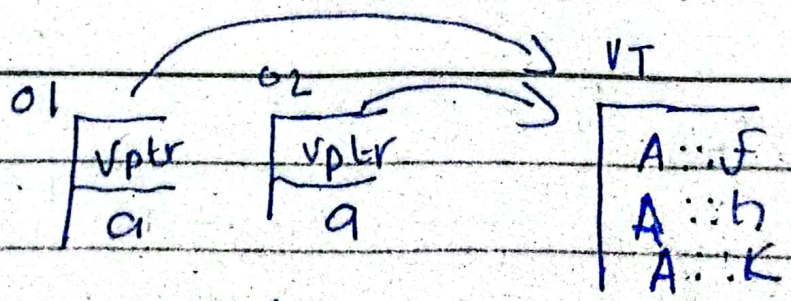
→ A *p = &obj;

p → h();

(Accessing virtual Table) ← (*(int *)p)[1] → Memory reached

(It will points to 1 of VT)

(*) A



A *q = &o1;

q → h();

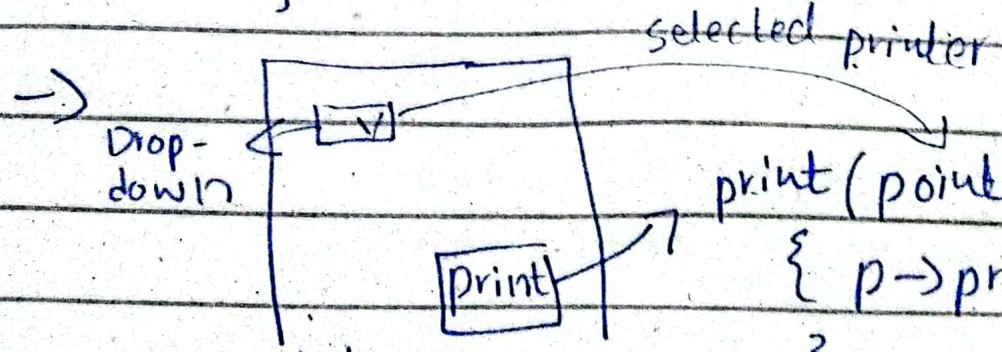
(*) Windows and OS : Programming by Contract. (They have to follow rules) Know other devices (printer)

(*)

```
Class Printer
{ public:
        virtual void print()
            {
            }
};

Class ABC: public Printer
{ public:
        void print()
        {
        }
};

Class XX2: -- Printer
{ :
}
```



printer **list → (array of list)

ABC
XXZ
....
...

Many more can be added

: polymorphic alternate if-else (we have to add more thing)

* Windows is suppose structured language not object oriented. (So how is it working and accept new devices)

* API (Application Program Interface)
  ↓
  Communicating layer of OS with applications.

* If we wanted to add something, then we can change the existing functions by using if else. (Manipulate derive class through Base).

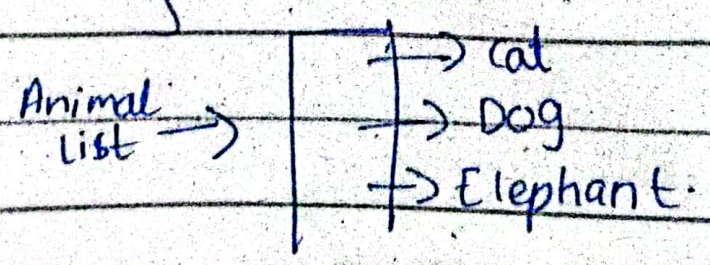
* Animal
  virtual(speak())

  Dog speak() | Cat speak() | Elephant speak()

: Runtime Binding polymorphism

: Compile/ static polymorphism

```
speak ( Animal *a)
{ a -> speak();
}
```



(Abstract class)

* Animal (base)(No existing) & Cat, dog, elephant (object are mades)(Reality of Animal types) (What we can do in base of Animal speak() { ?.... }).

* The function is virtual in Base and in Derive if signature of that same function is different then syntax error.

A virtual void f()

B void f(); ✓
float f(); ✗

: override (Not override if up one is virtual otherwise override).

A virtual f(int)

: Signature & prototype matches

(If not present) run this

B void f(int) override

: Function itself a prototype.

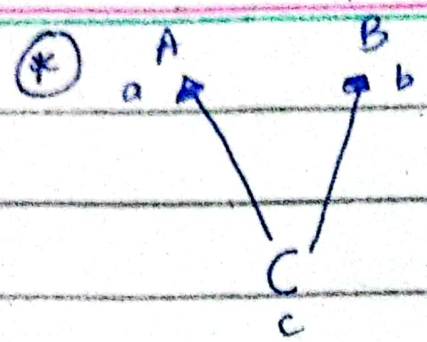
virtual void f(int)
(Hides it)

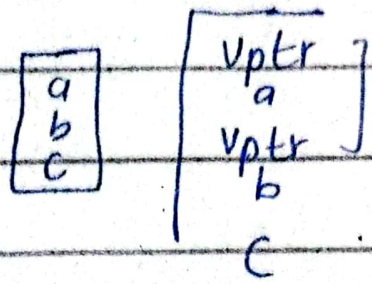
: Read till page 5

* Do a coding of Accounts Questions only.
Read others.

: using for class (No public, private)
: using for function (matters)

(*)





: a & b not known to each other.

27/11/23

(*)

A f g

B x y

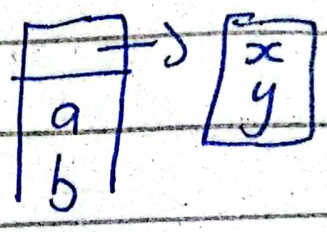
a, b

: A & B different class + signature match

B o1;

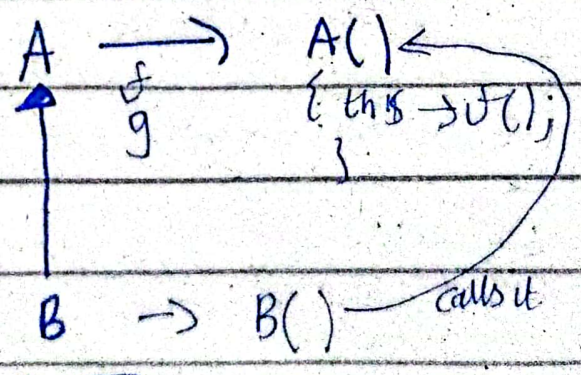
A * p = (A*) &o1;

p → f()



(*) Through base we can manipulate derive class object. (Polymorphism)

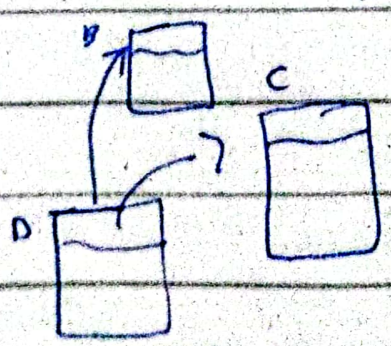
(*)



: Vptr value changes.

: First Base initializes and it is latest implication.

(*)



(till it initializes and it can run).

(By changing vptr value).

(*) a A

↑

b B

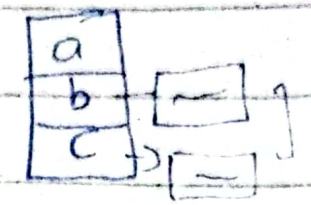
↑

c C

A *p = new C;

delete p;

(A deleted (only its destructor call))



(*) Global functions are not virtual but data members are virtual (friend).

(*) Animal, Printer] Abstract classes

Pure Abstract class/Interface

(Some virtual functions are provided and some function not known what to provide) (Empty Body)

(Animal virtual speak()=0; (Every derive should define it).)

(A
virtual void f() =0;
void g().
{}
virtual void x()
{}

: A class object not made at any cost.

: Pure virtual function is implemented (Not overrides)

: Interface (Nothing of its own self) (Window example). (TV)

(*) class Interface
{ virtual void on()=0;
virtual void off()=0;
}

(Whom use interface design their own functions)

→ Interface is always public.

: interface SWITCH
{ all thing public
}

class Fan: public implements SWITCH
{ }

(In other languages)

: In C++ Keyword interface is changed to class.

* By progamming by contract allows the derive to define their own functions must, otherwise no class would be formed.

* Destructor can also be pure virtual. It allows derive class to define their own destructor (explicitly).

*

```
f(A *p)
{
    B q(*(B*)p);
}
```

Copying any class (Eg: B)

A
↑
B
↑
C

*

```
A virtual A* clone()
{ return new A(*this);
}
```

B → `A* clone()`
{ return new B(*this);
}

(This acts as virtual constructor).

C

→ `f(A*p)`
{ `A* x = p->clone();`
}

: clone, fresh

calls the type of object received.

* Multiple Inheritance, Virtual Inheritance in next class.
* Through Derive class, Base class virtual cannot be accessed.
* Array class (For int, double, float coding same but works for it)
Template Designing (Polymorphism).

29/11/23

* TypeCast Operator:

static_cast<int *>(a)
cast object

dynamic_cast < > ( )

const_cast < > ( )

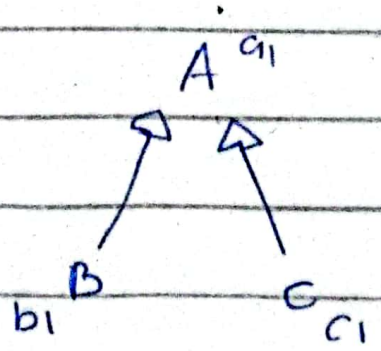
reinterpret_cast < > ( ) ← : (type) a

* upcast ↑ base (upcasting) (Always allowed)
downcast derive

```
B b1;
A a1;
A *p = &b1;
B *q = &b1;
```

p → error not known whom p is pointing

(*) Object slicing.



: Typecasting causes some member to copy (Object slicing)

B * x = dynamic_cast<B*>(p);
(if B object then x stores)
otherwise B *x stores null.

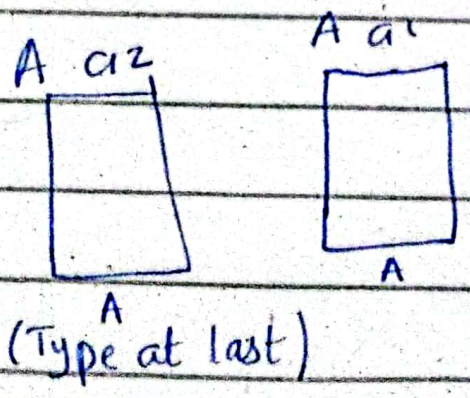
```
if (x)
  { B object;
  }
else
  { other;
  }
int a;
```

: RTTI (Runtime Type Identify) Identification

(*) cout typeid(a).name();

: output (int)(for windows) (i)(for Mac

(*)



: dynamic cast only works with polymorphic class. (RTTI works for polymorphic behaviour)

(*) RTTI enabled then dynamic cast work. Dynamic cast is important others are done by C*.

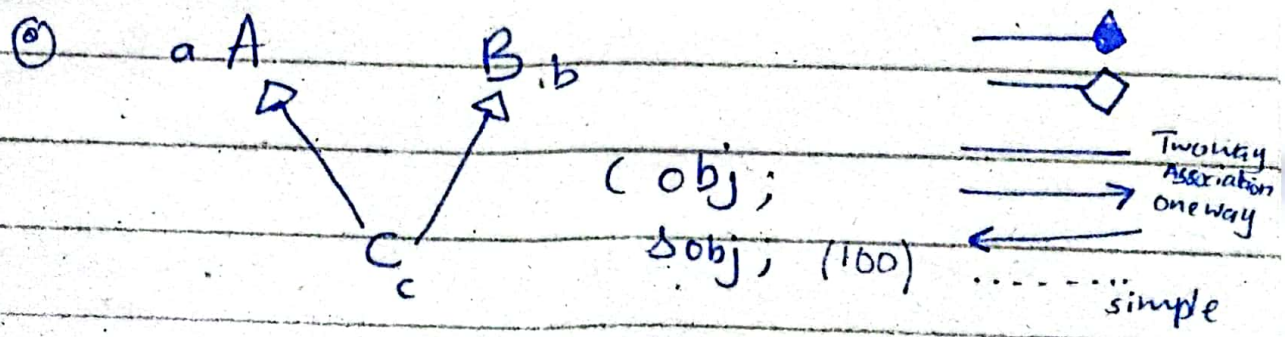
(C++.com
(website)
(Typecast operators)

(*) Static cast used for non-polymorphic class. (Downcasting & upcasting is possible) Const cast used to add or remove const members.

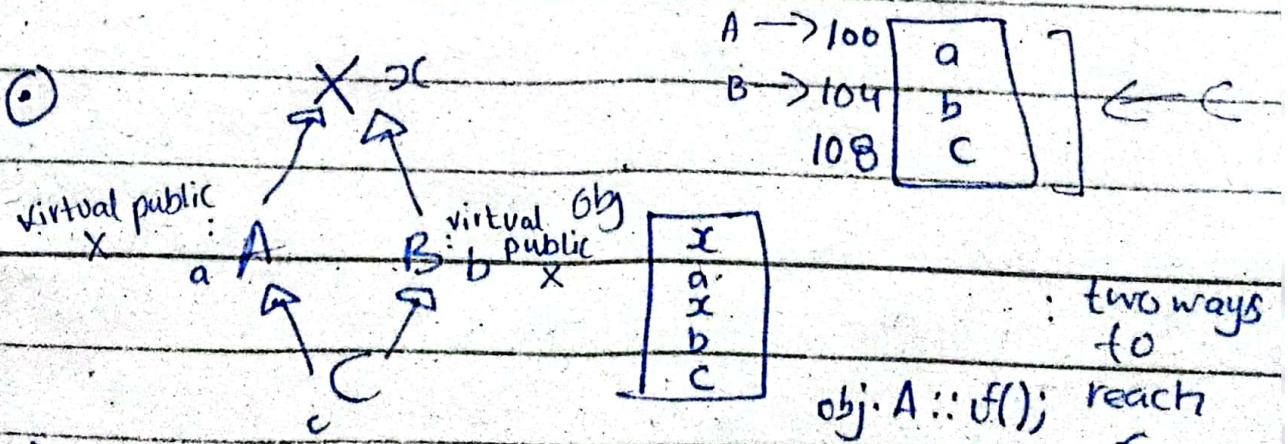
(*) Reinterpret cast used to convert one class pointer to other any how they are unrelated.

int x = (int) 9.7; x = 9;

(*) Multiple Inheritance:



(obj;
&obj; (100)



: two ways to reach x

obj.A::f();
B::f();

→ One copy of X will be made in C. (Now C decide to run constructor of X according to A and B.

: virtual inherit. X class (virtual Inheritance)

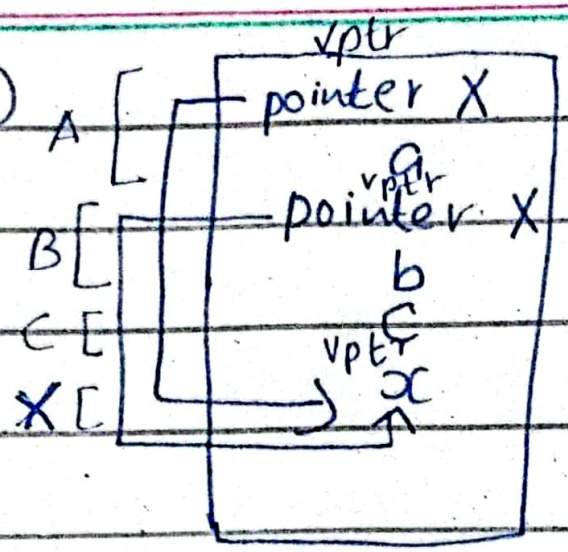
→ C : public A, public B

C() : A(), B(10), X()

: first object virtual constructor runs.

Delegated constructor (not works.

(*)

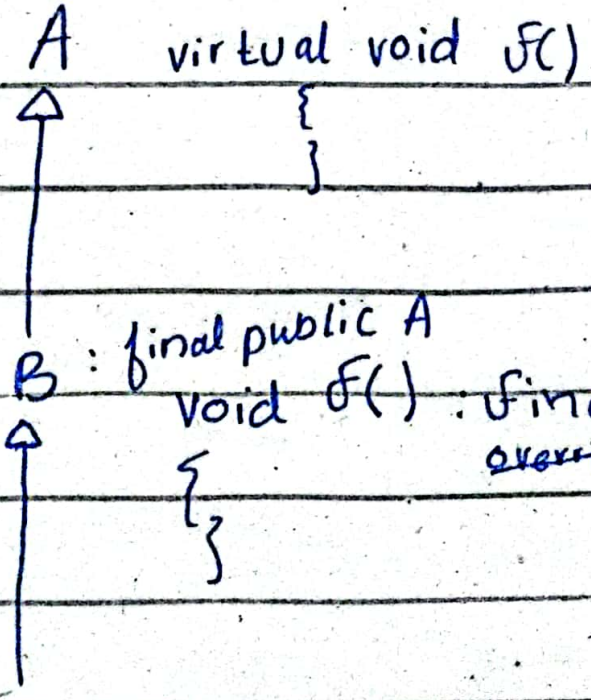


: X * p=&obj;
(not works)

: The class who is virtually inherited runs constructor of it.

(X) if X has Base class Y then Y constructor runs.

(*) final function (cleared everything).



: not inherited by anyone more

final form of function
(Not more override to other classes)

(*) Template (By Single Line it changes)

(*) function template (Logic not changes by changing data type)

↓

Templated function.

→ template <typename _ >

↓

Any Letter capital

template <typename T > : x y z

→ It should start from specific place.

→ Template function not runs but type of data receiving, that runs.

(*) template <typename T> : Its works till range

```
T getMax(T a, T b)
{
    return a>b ? a : b;
}
```

: operator should be define in the data type class

: Proper Syntax:

getMax<int>(3,4) : 4

--- <int>(4.6, 3) : 4

getMax(3, 4.6) x Not works (confusing)

int float/double

(*) char *x = "Hello"

constant

(Not assigns to pointer) (But compiler can run)

: constant memory

: If want to change error occur in old compiler

Use word "const" before "char *".

(*) Function specialization:

for string or char *

template <>

: special function has more priority

const char* getMax(const char *>

(const char *x, const char *y) different

(Same definition but same signature)

(*) Sorting and Searching (For different types)

(*) Limitation (Declaration and Definition at same file).

(otherwise Linker error)

* Template show polymorphic behaviour.

* template <typename T>
  T getMax (T, T); → Prototype

* void myswap (T&a, T&b)
  { T temp = a;
    a = b;
    b = temp;
  }

: Generic

: <> (Angular Brackets)

* Array <int> a;
  Array <float> b, c;
  b = c ; ✓
  b = a ; ✗
  a = b ; ✗

(May or may not "=" define in b or a;)

* class Array <int> ;
  (compiler runs this)

: In header file, all work should be done.

for Array copy constructor;
Array<T> :: T (const Array <T> &)

* template <T, R = int>
  void f(T x)
  { R a;
  }
  f (30);
  f <int, float> (30);
  (explicitly)

: Default type should/can be assigned in declaration.

(*) Works for primitive types and not works, or may work for user defined.

(*) For macros, type changes before compilation. (Not works for primitive and user-defined types).

(*) For 2-D array:

```
Array < Array<int> > a;
```

class Array<int>

```
class Array <Array<int>> {int * data;
    { Array<int> *data; }
    }

int main()
    { Array < Array <int> > a;
        a.resize (3);
        a[0].resisize (4);      : 4 cols
        a[1].resize (2);        : 2 cols
        a[2].resize (4);        : 4 cols
    for (int i=0; i< a.getSize(); i=i+1)
    {   for (int j=0; j< a[i].getSize();
                                i=i+1)
```

(*) Matrix

```
{ int r, c;
    Array<Array<T>> data;
}
```
(composition)

: Automatic loop (Range loop)

(*) # include (initializer_list)

```
int main()
{   auto x=86.7;
```

: auto (decides type) (double)

(*) Range loop: Array<T> x

```
For ( int val : x )
```

auto

↓ container name

```
{ cout<< val;
}
```

int

initializer_list <float> x = {40, 5, 6, 7, 8};

Array <int> x = {3, 9, 5, 6, 7}

(Further Accessing and changing in next class).

(*) Array and Matrix class should be made (till next class).

11/12/23

initializer-list <int>

↑ ↑auto x = {10, 15, 23}

(variable any lists)

Array < Array <int> > a

(Array to pointer object).

(*) initializer_list Know its size.

```
Array <Array <int> > a;
a.resize(3);
for(int i=0; i<a.getSize(); i=i+1)
{   a[i].resize(3);
}
```

(Insertor remove) : Container (collection of objects) : Iterate (It is used for processing of items in container). (Index is iterator)

(*) Beginer :

begin (Returns first location address) end

: begin const int * begin() const (Read/write)

```
int main()
{ initializer_list <int> a = {10,20,30,40};
  const int *p = a.begin();
  cout << *p ;                    : 10
  p++;
  cout<<*p ;                      20
  while (p != a.end())
    { cout<<*p<<' ';
      p++;
    }
```

(*) Range loop always works in straight manner.

```
; const *p= a.end();
  p--;
  while (p! = a.begin())
  {
    cout <<*p<<' ';
    p--;
  }
```

: 40, 30, 20

(*) It will not work with user-defined classes. We have to make begin and end functions.

(*)

```
Array (intialize<T> ; x={}) : Array()
  { if(x.size()==0)
      { return;
      }
    size = x.size();
    data = new T[size];
  }
```

Range loop

```
Array<Array<int>> a={{1,2,3}, {1,2}}
initializer_list<int> a={};
```

(Adds this) (Removes all constructors except copy).

(*) rbegin (reverse begin) . (Not included in Quiz or Exam)
rend (reverse end)

(*) class Set
{ Array <T> data;
// int capacity;
int NOE;

(*) Array <int> a;
Array <floats> b;
b = a; X (Not related to each other)
Array <ints> *p = &b; X (Not possible)

: template < typename T >
class X : public Array<T>
{
receives matrix, string etc...

: template < typename T >
Array
{ template <typename R>
friend void f();
}

: ostream & operator << (ostream & const Array <T> & x)
{ for(——)

It should be defined friend so it can access Array

(*) static is made one time but for Array int, double, float more copies are made.

(*) Array <int> *p;
(Object is made) (Also object is made with alias).

(*) Separating file:
template + specialization.

: Next class Review file handling
: 180 RB

Template void f<int>; (Global functions instantiation)
template class Test<int>; (class instantiation)
(At last of cpp file).
of all data types

: Make new cpp file:
For test:
→ Test_Implementation.
(adds Test.cpp file int)

(*) In namespace/block we cannot call functions of multiple same classes. If we declare functions of classes out it block then it can be called. Otherwise not to be called. It is not advisable to do this method of making copy of multiple same classes.

: cplusplus

13/12/23

* Text mode, Binary Mode.

File Handling:

int a = 12 , 123 (ASCII store)
2 bytes 3 bytes

ifs << "123"
4 bytes (Always).

* Binary Mode/ Raw Mode:
→ Always 4 bit (Any datatype)

int main() : cerr, clog cin, cout
{ ofstream ofs ("test.txt" ios::binary);

* ios.base (flags setting)
ios.state (good, bad, foul etc..) → (Functions in other file).
Openmode (out, in, app, binary)

* Stream operator:
istream ← cin>>s; ostream ← cout<<s;
ifstream ← ifs>>s ofstream ← ofs<<s;
(Work for both).

* Ostream :: write : int a; (Address, size)
((char *)&a, sizeof(a))

: 75(K)

(*) Binary pattern is copied as it is.

(*) Char array initial size is returned unless of char present in it.

```
char c[10] = "Aslam";
```

✓ size = 10
✗ Not size = 5

(*) ifs.read(char * -, sizeof(-))

```
int num;
ifs.read((char *)&num, sizeof(int));
```

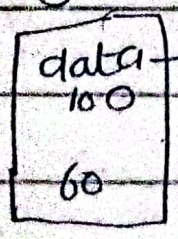
(*) Byte Alignment / Byte padding
(Search it on Google)
(Wikipedia)

(*) Student s;

```
ofstream ofs("student.txt", ios::binary);
```

: In binary form, In file any value is add by rendering & not supported are discarded.

(*) String s;



5 Hello 100

```
ofs.write((char*)&s.rollNO, -);
ofs.write(subject);
ofs.write(marks, subjectCount);
ofs.write(CGPA);
```

: Struct Student { int rollNO; float marks[50]; int subjCount; float CGPA; }

(*) streambuf * rdbuf() const;

ifs.rdbuf();

streambuf * rd (streambuf *sb);

(*) ifs (Test.txt)

ofs (copy.txt)

streambuf * pf = cout.rdbuf(ofs.rdbuf()); : tie (Change ((ofs.rdbuf()));

(*) sstream (Header file) : Internal buffer.

↓

string stream (Arranges the data while reading by itself.) : istring stream (Reading from string object) : ostring (writing in string object)

→ istringstream iss (stringValues);

↑ string stringValues = "145.67 ... -- 80";

float x;

iss >> x; : 145.67 (Ignores special character).

iss >> x; : 80

(*) Exception Handling:

int divide (int n, int d)

{ return n/d; → If zero produced then error aborted

}

f() { g() } g() { h() } h() { }

Proper path

: Error reporting (log use)

* clog (c-logging) , cerr (c-error).

18/12/23

* String and Matrix Library (Finalize it). File Handling in next Lab.

* int & operator [] (int i)

```
{
    cout << "Error";
    exit(0);  -> (If no option left then we use).
}
```

* Exception Handling allows to graceful exit.

: try (Exception thrown)
: catch
: throw

```
: int main()
{ try
  { int a=89;
    int b=0;
    cin>>a>>b;
    if (b!=0)
    { cout << a/(float) b;
    }
    else,
        throw
```

(It not depends on return type)

Anything literal, constant etc...

```
catch( int x)
{ cout<<"int: "<<x;
}
catch (float x)
{ cout<<"float: "<<x;
}
```

: Multiple catch blocks can be made

: try & catch go hand in hand

: After throw not anything executes till catch.

(*) f() g() h() x() y() : back tracing

: Stack Unwind

```
: float divide (int n, int d)
{ if (d==0)
  { cout<<"Error: ";
    throw 10;
  }
  return n/d;
}
```

divide

: Type of throw & catch should match.

(*)

```
try
{ try
  { throw 10;
  }
  catch (float x)
  {
  }
}
catch (int y)
```

: Not matching with throw so passed

: It matches with throw so it is called.

(*) Underflow & Overflow done by other languages and put it into try block. (Java etc.)

(*)
```
int main()
{   int a;
    cin >> a;
    try
    {new int [a]
    }
  catch( bad_alloc &b)
    {   cout << "\n OK";
        b.what();
    }
```

Three functions init constructor = what

(*)
```
class IndexOutofBound
{ string msg;
  public:
    IndexOutofBound()
    {   ... Any Message,
    }
    operator += (___)
    {
    }
  what()
    {return msg;
    }
}
```

(*) Exception Base Class:

(1) Logic error (2) Runtime Eror

: Simple "throw" in catch box only (throwing as it is) (throw;) (Rethrow).

(*)
```
try
  [ a/b ] → x
  ]
catch ( exception &e )
  {   x   (It not accepts that).
  }
```

(*) For Base and Derive class, make catch class in reverse manner.

```
try
  { throw new B();
  }
catch (A*x)
  {
  }
catch (B*y)
  {
  }
```

→ It firstly executes always

(*)
```
class Exception
  {   string msg;
  public:
        Exception()
        {
        }
        operator +=()
        {
        }
  what()
  [ return msg;
  ]
}
```

class OutOfBound:
public Exception
{ using exception::exception
}