OOPS

→ Class is a user defined data type where we group the data and actions performed on the data.

→ public, private, protected } → Access Specifiers.

→ member functions in a class are called method.

→ class is like a blueprint, and object is a runtime variable of the class

[try #include<string> and string name; var]

→ method declared in class and is implemented outside using Scope Resolution operator - Ex: void Node::create(){};

→ constructor has same name as the

class and has no return type and is executed when object is created. * Should be in public access specifier *

→ Constructor overloading is where we have default constructors plus constrs that take few parameters as input

* when you've overloaded constr. default constr. is not implicit, object must be passed with input.

* If overloaded constr. already has inputs given in func. braces section i.e.,

Human(string name = "abc", int age = "0")

we can initialize obj without passing any.

→ constructor & destructor can be called outside class. delete <class_name> will call the destructor. ex: ~Human();

→ age = new int (* pointer to int)

→ destructor is to free all resources used in class.

→ static variable in a class is global to all the objects of that class; i.e., shared b/w all objects.

→ In static methods, we can only use static variables of that class.

→ If we want a function that isn't memb-er of the class to access its private section, we define that function as "friend" and pass the object of the class. A prototype of this function say:

```
void display (Human manu) {
                    }
```

friend void display (Human manu);

has to be there in the class

→ we can even have a friend class. mention friend class <class_name>; in

the class whom you need to access.

→ Inheritance is that we use the features, member functions of one class in another class.

Ex: BASE CLASS

```
class A {
public: int a;
    void seta(int b){a=b};
};
```

SUB CLASS

```
Class B :public A {
public: int c;
    void setc(int b){c=b};
};
```

→ `B b; b.seta(4); b.setc(5);` //

⇒ The members in protected access specifier of a class are only accessible by a sub class i.e., by only the class that inherits the class.

⇒ If No access specifier is mentioned, then private: is the default one.

⇒ In public inheritance, protected of base is protected of sub & public is public.

⇒ In protected inheritance, public & protected members acts as protected members of Sub class.

⇒ In private inheritance, public & protected members acts as private members of sub class.

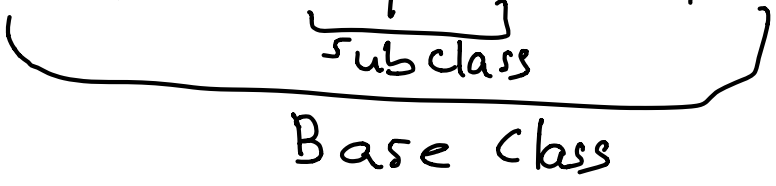
⇒ To change the access level of members of base class in Sub class we specify it as <class_name> :: <member>;

Ex: say name in Person class is protected & student class private inh-erits Person. In public: of student say → Person :: name; or Person :: func; to change access level in Student.

⇒ In Inheritance, the constructor of

base class is called first and the destructor of subclass is called first.

i.e., | C<BC> | C<SC> | D<SC> | D<BC> |



⇒ Multiple Inheritance. Ex: class Father,
class Mother;

class child : public Father, public Mother { };

To pass the values of child class constr. to that of father and mother,

Constr. of child:

```
child (int x, string color): Father(x),
                             Mother(color)
{
    cout<<" child constr"<<endl;
}
```

⇒ We can override the methods of baseclass in subclass i.e., we change statements inside the method without

statements without changing return type or name.

Ex: class A {
public:
void intr() {
cout << "A";
};

class B : public A {
public:
void intr {
cout << "B";
};

B b; b.intr(); ⇒ "B"

To access overriden method, say

B b; b.A::intr(); ⇒ "A"

or in class B : public A {
public:
void intr {
cout << "B";
A::intr();
}

⇒ "BA";

⇒ this → age = age if void setAge(age) { this → age = age; }

⇒ whosthis(A a) { a.intr(); }

B b; b.intr(); whosthis(b)

"B" "A"

⇒ Polymorphism

→ When there is a hierarchy of classes and they are related by inheritance, call to a member function will call a different function to be executed depending on the type of object.

Ex1

```
class A {
public:
virtual void i()
    {cout<<"A"; }
};

class B : public A {
public:
    void i()
    { cout<<"B"; }
};

class C : public A {
public:
    void i()
    { cout<<"C"; }
};
```

```
whosthis (A a){
    a.i();
}
```

if virtual void i() in A, B

```
B b; whosthis(b); ⇒ A
C c; whosthis(c); ⇒ A.
```

"C" ← even if `class C : public B { }`

↳ "C" is output.

i.e., even if sub class acts as base class for others the virtual function holds for overridden ones too.

*) here if class C doesn't override i() func then its base class's i() i.e., B's i() is called.

Pure Virtual function

⟹ if

```
class A {
public:
    virtual void i() = 0;
};
```

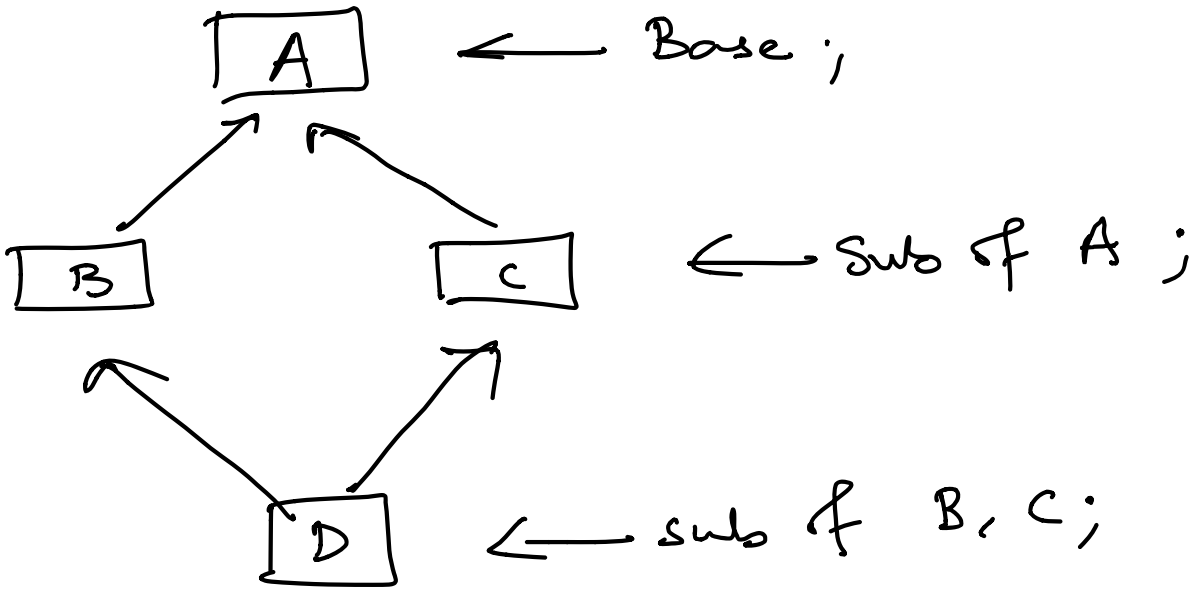
```
void A::i() {
cout << "A";
}
```

Now, all the classes that inherit A, must override i() function otherwise its an error.

⇝ Abstract class : a class that has one/more pure virtual funcs, we can't create an object of

the type abstract class i.e., it can only act as a base for derived classes whose objects we can create i.e; from above we can't say A a;

Diamond Problem:



when we try to access A's members / func from D there arises ambiguity, compiler doesn't know from which instance it has to access i.e; from B or C.

solve for diamond prob:

Ex:

```
class A {
public:
   int a;
   void w() { cout << "base" << endl; }
};
class B : public A { };
 class C : public A { };
 class D : public B, public C { };

int main()
{
    D d;
    d.w();
}
```

// error ||→ to solve add virtual to B & C before access specifiers.

i.e.,

```
class B : virtual public A { };
class C : virtual public A { };
D d;
d.w(); ⇒ base
```

→ here if A, B, C, D have constructors

then D d; calls →

A's constr.
B, constr.
C's, constr.
D's constr.

if no virtual
A's constr.
B's constr.

→ **Nested class**: class inside a class.

i.e, say class Person needs address

so we define class Address in the class person.

to use address we can say

Person anil; anil.addr.country
.street
.hno | etc

also → Person :: Address addr;

addr. - - - ✓

→ **Local classes** → are classes inside say a function and can't be accessed by any except for the function.

⇒ Operator Overloading

1) Ex: 'f'. class Marks { int imark;
int emark;

Marks operator+(Marks m) {
Marks temp;
temp.imark = imark+m.imark;
temp.emark = emark+m.emark;
return temp;
};

Marks m1(10,20);
m2(20,40);
Marks m3 = m1+m2;
⇒ m3 → (30,60);

i.e., we overload operator to work upon classes we defined.

*) Here as operator overloading is done in the class itself only one parameter is passed It, can be done outside by passing two (with it as friend func in class);

Ex:
Marks Marks::operator-(Marks m) {
return temp;
}

In Marks m3 = m1 + m2; → calls operator & m2 is passed

2) overloading shorthand operators

i.e, +=, -=, *=, /=

Ex: ⇒ (In Marks class) ⇐

```
void operator+=(int bonusmark){
    imark = imark + bonusmark
}
friend void operator-=(Marks &curObj,
                       int redmark);
```

outside class.

```
void operator-=(Marks &CurObj, int redmark){
    curObj.imark -= redmark;
}
```

⇒ In main ⇒

```
Marks m1(10,30);
m1 += 20;
m1 -= 20;
```

3) overloading ++ & -- operators

i) prefix (++/--)

Ex:

```
class Marks{
```

```
    int mark;
    public:
        Marks (int m) {
            mark = m;
        }
        void print() { cout<<mark; }

    void operator++() {
            mark += 1;
        }
        friend void operator --(Marks &);
};
void operator -- ( Marks &m) {
    m.mark -= 1;
}
int main()
{   Marks m(60);
    ++m; ++m; --m;
}    ↓    ↓    ↓
     61   62   61
```

ii) Suffix (++ / --)

```
In Class: Marks operator++(int) {
```

```
        Marks duplicate (*this);
        mark += 1;
        return duplicate;
    }
friend Marks operator--(Marks&, int);
```

Outside class:

```
Marks operator--(Marks& m, int)
{
    Marks duplicate(m);
    m.mark -= 1;
    return duplicate;
}
```

In Main:

```
Marks sai(68);
sai.print(); ⟶ 68
(sai++).print(); ⟶ 68
sai.print(); ⟶ 69
(sai--).print(); ⟶ 69
sai.print(); ⟶ 68
```

4) overload special operators

⟹ [], (), →

rules: 1) they must be non-static memb. funcs

2) we can't use frnd funcs to overload these operators.

i) Ex:
```
class Marks()
{ int sub[3];
  pub: Marks(int s1,s2,s3){
       s[0,1,2] = s1,s2,s3;
  }
  int operator[] (int pos){
     return sub[pos];
  }
};
```

[] → subscript Operator

In Main: Marks m(10,11,12);

// cout<< m[0]<<m[1]<<m[2];

(ii) → () [call operator]

Ex:
```
class Mark{ int mark;
  pub: Marks(int m)
       { mark=m; }
  Mark operator()(int mk){
     mark = mk;
     return *this;
  }
```

};

In main: Marks m(10);
m.print() ⟶ 10
m(12);
m.print() ⟶ 12

iii) ⟶ (Arrow operator)

Mark* operator →() {
return this;
}

In main: Marks sai(12);
sai.print() ⟶ 12
sai→print() ⟶ 12

Rules of operator overloading:

→ we can't alter the precedence of the operator
→ can't change no. of operands the operator works on.
→ ?, ::, .*, . } can't be overloaded.

→ Except for "=" operator, all overloaded operators are inherited in derived / sub class of base class.

Exception Handling [EH]

→ A response to an exceptional circumstance during runtime is an exception.

→ #include <exception> has needed 'EH'ers.

Ex:

```
int main()
{
    int a=10, b=0;
    int c;
    try {
     if (b==0)
        throw "divide by zero error";
     c= a/b;
     cout << c<<endl;
```

```
} catch (const char *e){
    cout << "exception occured" <<endl<<e;
  }
  return 0;
}
```

→ #include <stdexcept> has all standard exception handlers.

Ex: std: bad_alloc

std: bad_cast

std: logic_failure
- std: domain_error
- std: invalid_argument
- std: length_error
- std: out_of_range

std: bad_typeid

std: bad_exception

std: runtime_error
- std: overflow_error
- std: range_error
- std: underflow_error

→ Runtime Error

to throw catch we passed an object.

↓

Ex:
```
try {
  if (b == 0)
```

```
        throw runtime_error("div by 0 err");
    c = a/b;
    } catch (runtime_error &err) {
     cout << "exception" << endl;
     cout <<  err.what();
    }
```

⟹ Multiple catch blocks

```
try {
    throw "exception"; // throw 20;
  } catch (const char *ex){
     cout << ex ;
  } catch (int error) {
     cout << error;
  } catch ( runtime_error &e) {
     cout << e.what() << endl;
  }
```

throw runtime_error("a");

*** to catch all exceptions in one block

```
try {
```

```
throw ____________;
}
catch(...){
  cout<<"exception"<<endl;
}
```

→ catches any type of error.

~> We can write known catch's and include catch(...) at last to handle any unknown exceptions.

⇒ Throwing Exceptions from Functions

Ex:
```
void test() throw(int, char, runtime_error){
  throw 20;
}
int main()
{ try{
    test();
```

```
}
catch (...){
    cout<<"Exception";
}
}
```

⟹ Nested try catch | Re throw Except's

```
try{
    try{
        throw "char exc";
    } catch (const char *e){
        cout<<"char in inner"<<e;
        //  throw;
    }
catch (...){
    cout<<"exception";
}
```

gives

char in inner
char exc

if we add throw

→ char in inner
char exec
exception

⇒ Creating custom Exception Class

→ here we need to include<exception>

→ prev work can be done by <stdexcept>

Ex:

```
#include<iostream>
#include exception>

class Overspeed : public exception {
  int speed;
  public:
      const char * what(){
        return " speed Kills \n";
      }
      void getSpeed() { cout << speed; }
     void setSpeed(int s){ speed = s; }
};
class Car{ int speed;
   public:   Car() { speed=0; }
    void accelerate() {
        while (true)
      { speed++;
```

```
        if (speed >= 100)
        { Overspeed s;
          s.setspeed (speed);
          throw s; }
        }
    };

int main()
{   Car audi;
    try {
      audi.accelerate();
    } catch ( Overspeed s ) {
      cout << s.what();
      s.getspeed ();
    }
  return 0;
}
```

Copy Constructor

Ex: class Person{

```
public: string * name;
        int age;

    Person(string name, int age){
    this -> name = new string(name);
    this -> age = age;
    }

   Person (const Person &p){
     cout<<" copy constr";
     name = new string (*p.name);
     age = p.age;
   }
   void setAgeName(string name,int age)
  {    *(this -> name) = name;
       this -> age = age;
  }
};
int main()
{
    Person A("sai", 24);
    Person B = A;
    A.setAgeName("saiki", 25);
```

print(A); → Saiki, 25

print(B); → Sai, 24

→ If we don't use the copy constructor, print(B) gives Saiki, 24

i.e., as it is dynamically allocated, both point to same address. So, if we don't want B's name to change, then we need to explicitly write a copy constructor that allocates new memory to B's name.

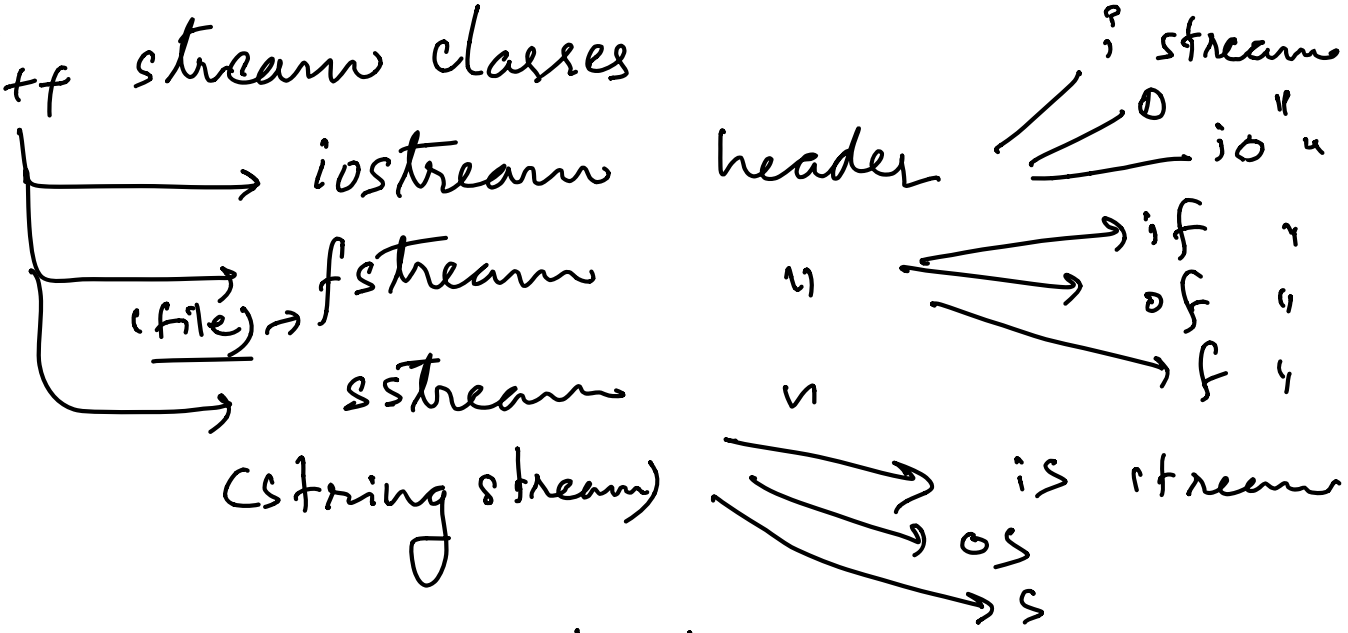
Stream is a logical device that

produces/consumes information. Like a buffer where we store information. From that buffer we read to the program / device.

iostream → input/output stream.

Types of streams ← text
← binary.

C++ stream classes



→ endl → end to line
ends → end to string (null at last).

→ Try string class in C++.

i.e, say string data("hi");

data. ⟷ STL

→ string data;

→ getline (cin, data);

→ cout << data

Ex1 ← Macros

```
#define ABS(number) ((number) < 0 ? -(number) : (number))
```

⇒ if endif elif else in Macros

Ex1:

```
#define W 1
#define L 2
#define M 3

#define OS W

int main(){

#if OS == W
   ...

#elif OS == L
   ...

#else
   ...

#endif
```

} conditional compilation Macros.

#endif

→ #ifdef W (i.e., if W is defined in Macros)

#endif

→ #ifndef S (i.e., if S is not declared)

#endif

→ #undef (to undefine any pre-defined Macros)

Pre-defined Macros in C++

__LINE__ → gives current line.

2 underscore #line 1000 before __LINE__ makes it output 1000

#line 1000 "new.txt" before

__LINE__ } makes 1000
__FILE__ } → new.txt

→ generally outputs curr file name

__DATE__ → curr date

__TIME__ → curr time

__STDC__ → standard c++ code

__cplusplus → confirms cpp with 6 digit code.

Generic Programming

→ A generic function defines a general set of operations that'll be applied to various Types of data

template func:

```
template < typename T > / < class T >
  T sum (T x, T y)
{ cout << x + y; }
```

we can make specific thing generic as of requirement

→ Func with multiple Generic types

Ex: template < typename T1, typename T2 >

void display (T1 x, T2 y)

{ cout << x << y ; }

⇓

Ex: void display (int x, int y)

{ cout << x << y ; }

if we pass display (10,20)
the normal func overrides the
generic function

can be written as.

template < > void display<int, int > (int x, int y)

{ ——— }

(useful in large codes to identify)

Generic-func Overloading

→ Generic Class:

template < class T >

```
class Myclass{
    T a;
    T b;
  public:
    Myclass(T x, T y)
    {   a=x;
        b=y;
    }
};

int main()
{   Myclass<int> intobj(10,20);
    Myclass<string> stringobj("hae","krishna");
}
```

→ Generic class with more generic types.

```
template<class T1, class T2>
class (Myclass{
    T1 a;
    T2 b;
  public
```

```
};
int main() {
  Myclass<int, string> m1;
}
```

we can also pass like

```
template <class T1 = string, class T2 = int>
```

then in main() → we say

```
Myclass< > ob1 ("sai", 18);
```

also ⇒ `Myclass<float> ob2 (22-36, 55);`

↓ T1 is modified.

```
Myclass<int, storing> ob3 (18, "sai");
```

if `temp < class T1, class T2 = int >`

`My class < string > ob4` ✓ works.

if `temp < class T1 = string, class T2 >`

`My class < int > ob5` ✗ fails

as T1 is modified first.

Namespaces

Ex:

main.cpp

```
#include<iostream>
using namespace std;
#in "file_one.cpp"
#in "file_two.cpp"
main()
{
  one:: display();
  two:: display();
}
```

file_one.cpp

```
namespace one{
 int x=20;
 void A()
 {cout<<x;
 }
}
```

file_two.cpp

```
namespace two{
 int x = 10;
 void A()
 { cout<<x;
 }
}
```

or

```
main()
{ using namespace one;
  display();
}
```

```
using ns one;
   "   "  two;
display();
```

X → conflict.

→ Members of namespace are not directly available.

→ We can use nested namespaces.

Ex:
```
ns one {
    int i;
    ns two {
        int j;
    }
}
```

ns
↳ namespace

one::two::j = 10; ✓

→ Unnamed / Anonymous Ns's

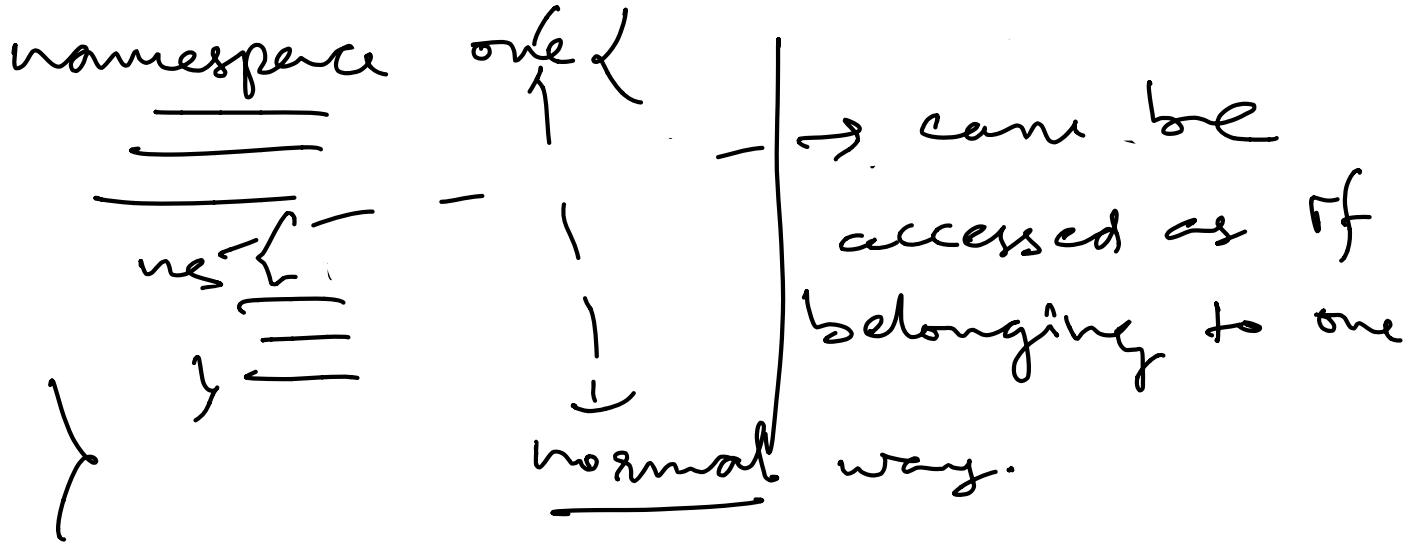
```
namespace {
  int x;
  ____
  ____
  ____
  ____
}
int main()
{
    x = 25;
    ===
}
```

can be defined as many times as possible

→ contiguous in a file.

→ dis-contiguous across files i.e., in diff files they're unrelated.

#nested_unnamed_ns-s;



→ Namespace Aliases

namespace _____ {

namespace ____ {

}

}

namespace one = _____;

i.e, we can use one:: ___ instead of _____. (long)

for nested namespace, we say,

namespace two = one:: _____;

→ Inline nested namespace

Ex:

```
namespace AppVersion {
    namespace version1 {
        display();
    }
    namespace v2 {
    }
    inline namespace v3 {
    }
}

int main()
{
    AppVersion::display();
}
```

AppVersion::version1
::display();

→ display from v3;

i.e; by inline it acts as if its part of perent namespace.

→ Can be useful when usrent version of the App changes i.e., if V4 comes then we say inline to that;

→ And inline should appear at the first instance of that version

i.e., if we have continuation,

```
namespace V3()
{  void deraP()
   {
   }
}
```

we need to put inline only at 1st instance & at all instances if we want explicitly.

Defining Classes in Separate Files

To include say a person.cpp's classes

in main.cpp we need a header person.h

Ex: Person.h

```
#ifndef PERSON_H
#define PERSON_H
```

} → if h defn is included avoid it

```
class Person
{ public:
        Person();
        void whatyougot();
    protected:
    private:
};
```

} ↗

i.e; person.h has a [Prototype / template] of person.cpp & latter is implementation.

Person.cpp

```
#include<iostream>
#include "Person.h"
```

```
Person::Person()
{
std::cout<<" Person class constr";
}
  void Person::whatyougot()
{std::cout<<"; got a million";
}
```

main.cpp

```
#include<iostream>
#include<Person.h>
using namespace std;
int main()
{   Person saiki;
    saiki.whatyougot();
    return 0;
}
```

→ output: Person class constr. ; got a million.