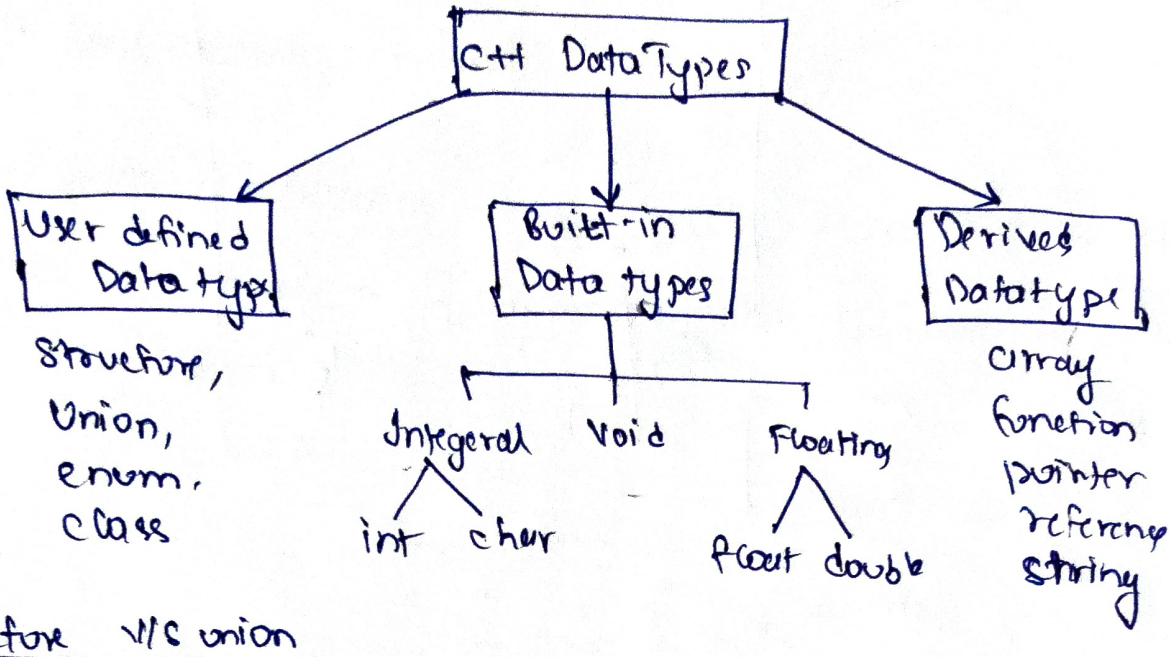
OOPs

- **Data Types**



- **Structure v/s union**

* Size of structure is equal to sum of sizes of individual members. However size of a union is equal to size of largest member of union.

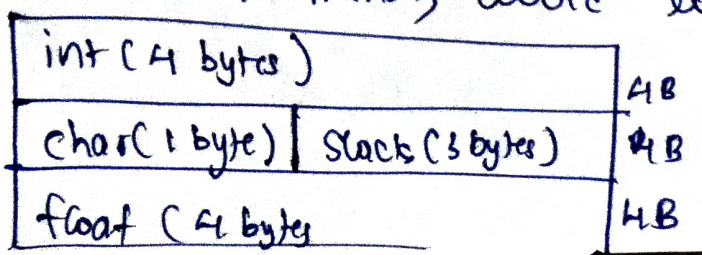
eg:

```
union Result {
    int marks;
    char grade;
    float percent;
};
```

This will occupy 4B in memory as its largest size member element is float

If result was declared as structure, it would occupy 12B on 32-bit system. The memory representation of structure uses concept of word boundary. Size of word boundary is machine dependant. For 32-bit system, word boundary is 4 bytes wide.

So, structure representation in memory would look like



3 byte between char and float is left unoccupied → SLACK BYTES

- Enum

enum keyword automatically enumerate a list of words by assigning values 0, 1, 2 ....

enum shape {circle, square, triangle}

shape becomes new data type name. We can declare new variables of that type.

```
shape box = square      //allowed
shape box =             //NOT ALLOWED
shape box = (shape)     //ALLOWED
```

C++ doesn't allow int value to automatically converted to enum. It should be typecasted as shown above.

Custom enumeration → ~~shape box{circle, ...}~~

enum shape {circle=2, square=1, triangle=4}

For → enum shape {circle=2, square, triangle}, square will be 3 and triangle will be 4

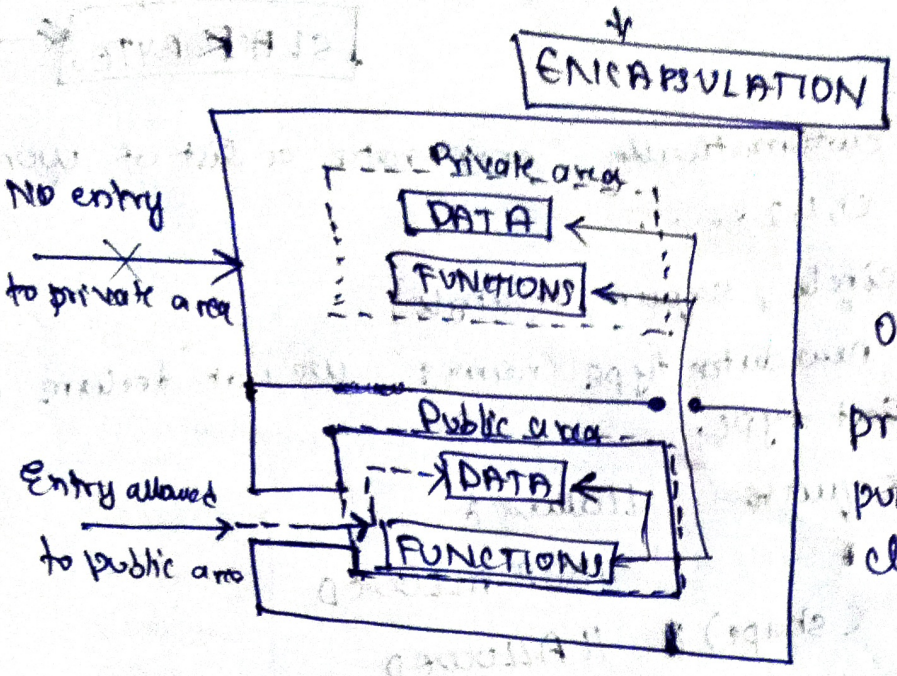
- Limitation of Structure

1) Doesn't support data hiding
2) Memory of structure in → stack
   class in → heap
3) Functions cannot be created in structure
4) By default, member of structure are public while members of class is private.

CLASSES & OBJECTS

* functions & variables inside class → CLASS Members
* private & public → visibility labels — private: public:
* variables → DATA MEMBERS
  functions → MEMBER FUNCTIONS

* Binding of data and functions into a single class-type



* It is a good practice to declare variables of class as private and member functions as private.

Defining Member Functions

Inside the class

When a function is defined inside a class, it is treated as INLINE FUNCTION

Only small functions should be defined inside the class.

Outside the class

```
return-type class-name :: function-name (arguement {
...
}
```

Class name :: → Membership label

However an outside function can also be made inline

```
inline return-type class-name :: function-name (arguments) {
...
}
```

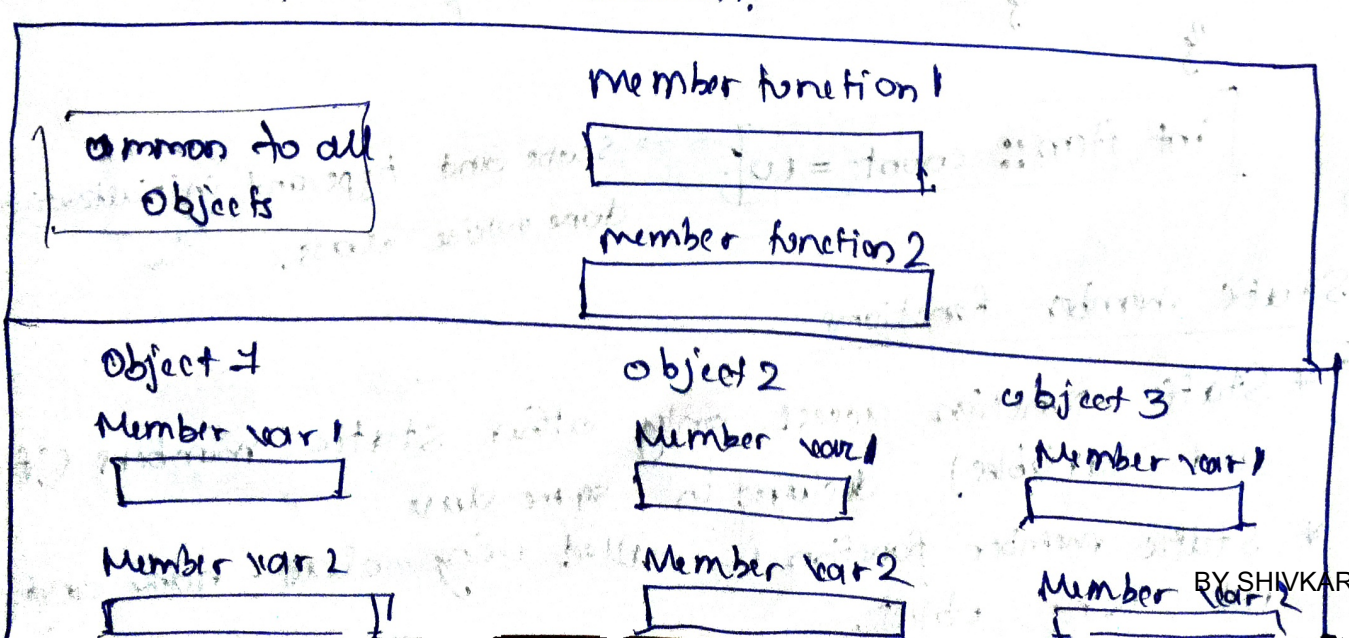
- Private Member function

* Even though it is a general practice to place all the data items in private section and functions in public some situations may require certain functions to be hidden from outside calls. Such a deleting an account in customer class or providing increment to an employee
* Such functions should be put in private section.
* Private functions can be accessed only by other function of the class.

- Memory allocation for objects.

* Memory space for an object is declared when they are declared. and not when class is created
* This statement is only partially true. Memory for member functions are created at the time of class creation. This is because all objects belonging to the class use the same functions, only arguement passed varies.

* Conclusion → Memory for variables is created when object is created and Memory for function is allocated at the time of class creation.



- Static data members

→ Initialized to zero when first object of class is created.
NO OTHER INITIALIZATION IS ALLOWED

→ Only one copy of member is created for entire class and is shared by all objects of the class

→ Visible only within class. But lifetime is entire program.

Eg: Static data member can be used as counter.

NOTE: Type and scope of each static variable must be defined outside the class definition. This is because static data members are stored separately rather than just a part of object.

STATIC VARIABLES ARE NON-INLINE MEMBER FUNCTIONS

Eg:
```
class item {
    static int count;
    int number;
  public:
    void get_data(int a){
        number = a;
        count++;
    }
};
int item :: count = 10;
```
→ scope and type and initialization done outside class.

- Static member functions

* Static function access only other static members (functions and variables) declared in same class
* Static member function is called using class-name and not using object.

obj item;

item.function-name ✗ item::function-name(); ✓

- **Friendly Functions**

* To make an outside function friendly, we simply declare the function as a friend of class.

```
class ABC {
  --
  public:
    ---
    friend void xyz();  // Declaration
};
```

↓

A function can be declared as friend in any no of classes.
A friend function, although not the member of the class has full access to private members of the class.
- Friend function is not member of any class. Hence it cannot be called with an object.
- To access the variables of class, within friend function, it should use object.variable.
- A function can be declared a friend inside either public or private area. It makes no difference.
- Usually it has objects as arguments.

An object when passed by value to friend function, the friend function can only access the private data of the class and cannot modify it.
But when object of a class is passed by reference, it can modify the private data. This is against the principle of OOP. So this is used when its necessary

- **const Member function**

If a member function doesnot alter any data in class, it can be declared as const member.

void mult (int, int) const;

If a const function tries to modify data → ERROR

- **Pointer to members**

- declared using ::*

Eg:

```
class A{
    int m;
public:
    void show ();
};
```

int A::* ptr = &A::m;

A::* → Pointer to member of class A

&A::m → address of m of class A

A a;

now a.m <same as> a.*ptr

- **pointer to object**

A a;

A *ap = &a;

a.m ⟷ (*ap).m ⟷ ap→m

(*ap).*ptr ⟷ ap→*ptr

CONSTRUCTERS & DESTRUCTORS

- **Constructor**

Special member function whose task is to initialize objects of a class. Its name is class_name and doesnot have any return-type.

• **Default Constructor**: A constructor that accepts no parameters. If no constructor is defined, compiler supplies a default constructor.

• constructors
- → Memory address of constructor cannot be fetched.
- → Should be declared in public section
- → Invoked automatically when obj are created.
- → Donot have return types.
- → Not inherited
- → Cannot be virtual
- → They are inline functions

• **Parameterised Constructor**: When parameters are passed to constructor

```
ABC obj = ABC(0,100) → Explicit call
ABC obj(0,100); → Implicit call
```

• **Copy constructor**: A constructor which takes an object of its type as an argument. If not defined, compiler creates a copy constructor on its own.

```
class A {
  public:
    A(A);
};
```
INVALID

Initialization using copy constructor → COPY INITIALIZATION

```
class A {
  public:
    A(A&);
};
```
VALID

It cannot take an object of its own type as a parameter. However, it can ~~take~~ accept a reference to it own class as a parameter.

```
A a1(1,2);
A a2;
a2 = a1
```
→ will call copy constructor

It is important to distinguish between a default constructor A() and a default argument constructor A(int = 0). When both of them are used in a Class, it leads to ambiguity

```
A a; → ambiguity
```

Therefore two such constructor shouldn't be defined.

* So compiler creates default constructor and copy constructor on its own if not defined.

- **const object**
  * `const class-name obj(x,y);`
  * cannot modify x and y.
  * A const object can call only const member function.
  * when a const obj calls non const member function → error.

- **Destructor**
  * Used to destroy objects
  * It is a member function whose name is same as class name precided by a tilde
  * `~class-name(){}`
  * It takes no arguement nor returns any value. It is invoked implicitly by compiler upon exit from program to clean up storage.
  * whenever new is used to allocate memory in constructor, we should use delete to free memory in destructor.

OBJECTS ARE DESTROYED IN REVERSE ORDER OF THEIR CREATION

OPERATOR OVERLOADING

* Operator overloading is not possible for
  1) Class member access operator (. , .*)
  2) Scope resolution operator (::)
  3) sizeof Operator (sizeof)
  4) Conditional operator (?:)

Why? →

1) . , .* → Because it is used to access members of a class. Overloading it would lead to ambiguity in terms of how class members are accessed.

2) :: → Used to access names of different scope. Allowing overloading of this operator will lead to complications in understanding scope of identifiers and could result in ambiguity

3) sizeof → It is a compile-time operator that returns size in bytes. Overloading this will to difficulty to determine size of a data type at compile time.

4) Conditional operator → will increase ambiguity in expression evaluation leading to unexpected results

→ In public part of class

```
return_type operator op (arg list){
  ....
}
```

Inside class

Operator functions must be either member functions (non-static) or friend functions.

```
return_type class-name:: operator op (arglist) {
  ....
}
```

Outside class

Overloading

- using member function
  - * One argument for unary op overloading
  - * two arguments for binary op overloading
- using friend function
  - * No argument for unary op overloading
  - * One argument for binary op overloading

1 Unary operator

```
class space {
    int x, y;
    space(a, b){
        x=a, y=b;
    }
private:
    void operator - ( ) {
        x = -x;
        y = -y;
    }
};
```

using class

```
friend void operator - (space &s);

void operator - (space &s){
    s.x = -s.x;
    s.y = -s.y;
}
```

using friend function

Class

space S(10, 15);

-S → calls operator -

↓

Equivalent to S.operator - ()

friend

operator - (S). ↔ -S

Overloading

- using member function
  - * One argument for unary op overloading
  - * two arguments for binary op overloading
- using friend function
  - * No argument for unary op overloading
  - * One argument for binary op overloading

1 Unary operator

```
class space {
    int x, y;
    space(a, b){
        x=a, y=b;
    }
private:
    void operator - ( ) {
        x = -x;
        y = -y;
    }
};
```

using class

```
friend void operator - (space &s);

void operator - (space &s){
    s.x = -s.x;
    s.y = -s.y;
}
```

using friend function

Class

space S(10, 15);

-S → calls operator -

↓

Equivalent to S.operator - ()

friend

operator - (S) ↔ -S

2 Binary Operators

```
class complex {
    complex operator + (complex c) {
        complex temp;
        temp.x = x + c.x;
        temp.y = y + c.y;
        return temp
    }
};
```

→ c2 is copied

→ refers to x and y of c1

→ returned as new obj and stored in c3

```
complex c1 (10,15);
complex c2 (20,30);
complex c3 = c1 + c2;
```

equivalent to → `c3 = c1.operator+(c2);`

So in op overloading `x + y` → x is used to invoke operator function and y is passed as argument

$$x \text{ op } y \iff x.\text{operator op}(y)$$

~~Using friend function~~ We can avoid creation of temp by calling constructor

```
return complex ((x + c.x), (y + c.y));
```

Using friend function

```
complex operator + (complex a, complex b) {
    return complex ((a.x + b.x), (a.y + b.y));
}
```

C3 = operator + (C2, C3) ⟷ C3 = C1 + C2

	Unary	Binary
Declaration	op x or x op	x op y
Interpretation		
member func	x.operator op()	x.operator op(y)
friend func	operator op(x)	operator op(x, y)

- Overloading of obj with builtin-datatype

Eg ⇒ A = B + 2 where A and B are objects of same class

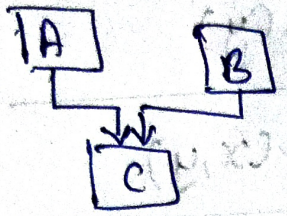
Using member function:
- A = B + 2 → Valid
- A = 2 + B → Not valid → because LHS of op y is B is used to invoke operator function. In A = 2 + B 2 cannot invoke operator function

Both are valid when we use friend function

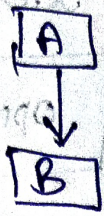
Inheritence

- Deriving a new class from an old class → Inheritence

Old class → Base class
New class → Derived class or sub class



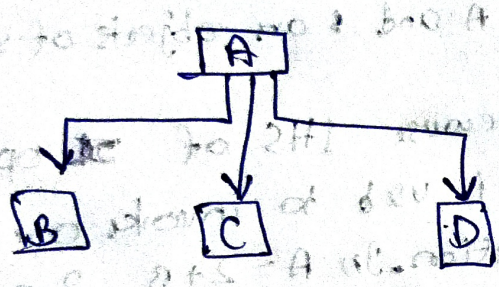
Multiple inheritence
>1 base classes



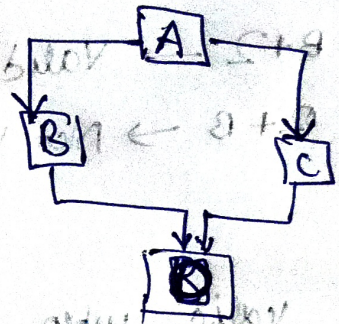
Single inheritence

single base class

Multilevel inheritence



Hierarchical inheritance



Hybrid inheritance

- Defining derived class

class derived-class-name : visibility-mode base-class-nam

3

Visibility modes

1. private → Public members of base class become private members of derived class
2. public → public members of base class become public members of derived class.

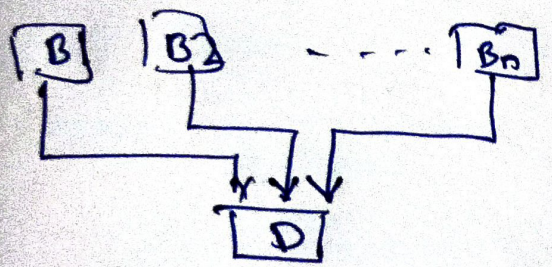
In both cases private members of base class are not inherited.

- **protected** → Member declared as protected is accessable by the member functions within class and CLASS IMMEDIATELY DERIVED FROM IT
  → When protected member is derived in
  1) Public mode → It becomes protected in derived class too. It can also be further inherited.
  2) Private mode → It becomes private in derived class. Not available for further inheritence

3 protected

When a class is derived in protected mode both public and protected members of base class become protected members of derived class → Protedted derivation

- **Multiple inheritence** → class D : visibility B1, visibility B2 ---
  {
  };



Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

```
class M
{
    public:
        void display(void)
        {
            cout << "Class M\n";
        }
};

class N
{
    public:
        void display(void)
        {
            cout << "Class N\n";
        }
};
```

Which **display()** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```
class P : public M, public N
{
    public:
        void display(void)      // overrides display() of M and N
        {
            M :: display(); }
        };
```

We can now use the derived class as follows:

```
int main()
{
        P p;
        p.display();
}
```

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

```
class A
{
    public:
        void display()
        {
            cout << "A\n";
        }
};
class B : public A
{
    public:
        void display()
        {
            cout << "B\n";
        }
};
```

In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to **display()** by **B** type object will invoke function defined in **B** only. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

Example:

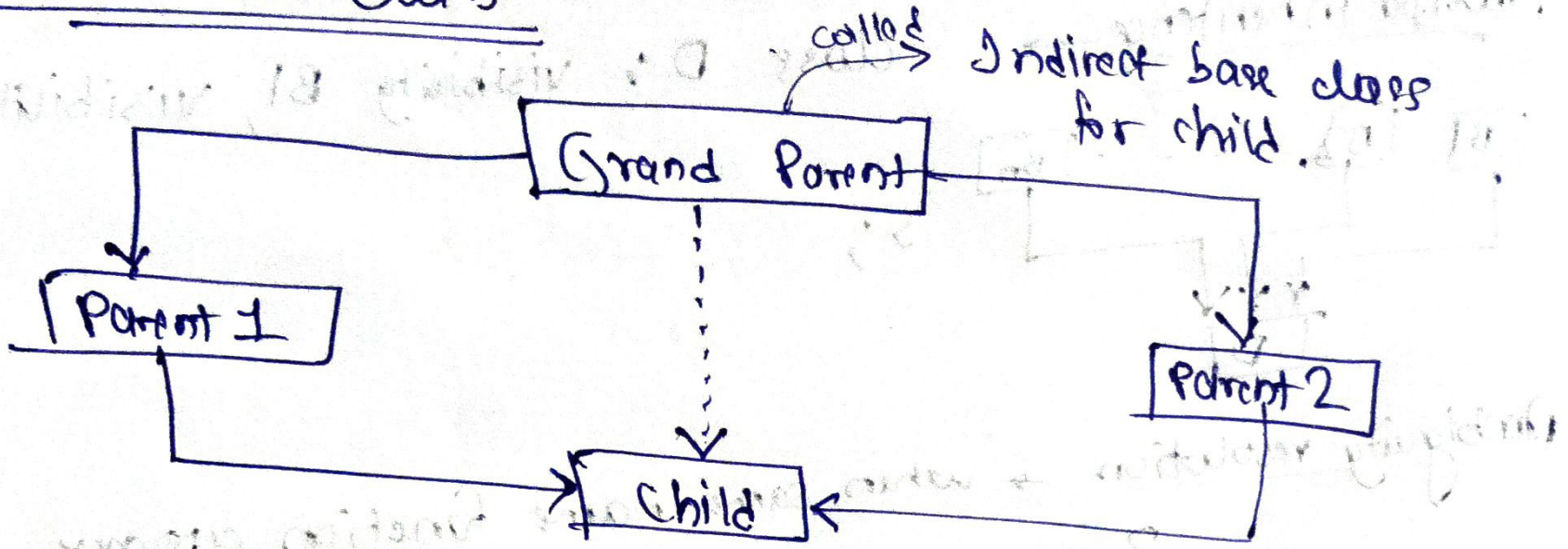
```
int main()
{
    B b;                          // derived class object
    b.display();                  // invokes display() in B
    b.A::display();               // invokes display() in A
    b.B::display();               // invokes display() in B

    return 0;
}
```

This will produce the following output:

```
B
A
B
```

Virtual Base Class



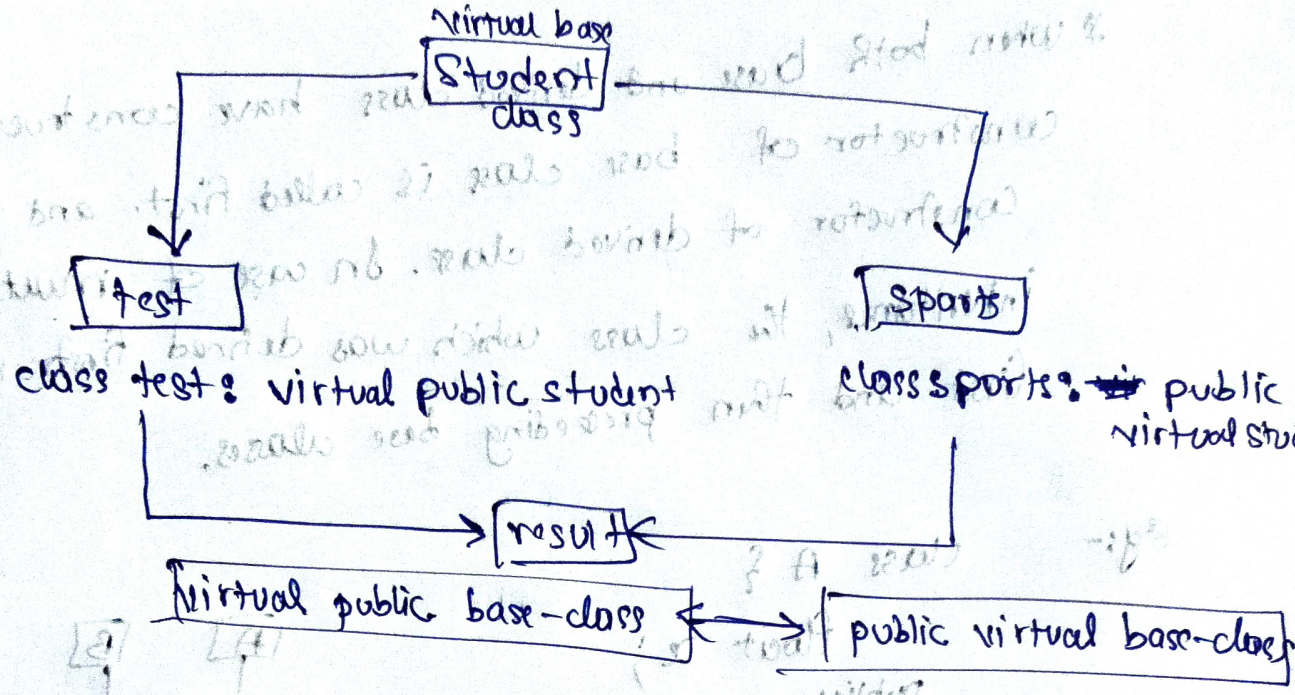
The child inherits the traits of grandparents via two separate paths. It can also inherit directly as shown by the broken line.

VV

Problem: All public and protected members of grandparent are inherited into child twice. first via Parent 1 and again via parent 2. This means child will

have duplicate sets of member inherited from grandparent. This introduces ambiguity and should be avoided.

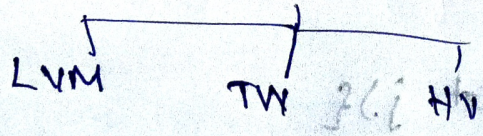
Solution → Make grandparent virtual. By making a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited.



- **Abstract Classes**

  * An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class.

Eg: Abstract base class → Vehicle



LVM – light Motor Vehicle
HMV – Heavy
TW – Two wheeler

Eg: Vehicle class contains common properties like → rpm, milage, speed etc

Abstract class → Must have atleast one pure virtual function

• Constructor in derived class

- If base class has no constructor, derived class need not have a constructor.
- If base class has a constructor with one or more arguements it is mandatory for derived class to have a constructor and pass arguements to base class.
- When both base and derived class have constructors, constructor of base class is called first and then constructor of derived class. In case of multiple inheritance, the class which was defined first is called first and then preceding base classes.

Program 8.7 Constructors in Derived

```
#include <iostream>

using namespace std;

class alpha
{
        int x;
    public:
        alpha(int i)
        {
            x = i;
            cout << "alpha initialized \n";
        }
        void show_x(void)
        { cout << "x = " << x << "\n";}
};

class beta
{
        float y;
    public:
        beta(float j)
        {
            y = j;
            cout << "beta initialized \n";
        }
        void show_y(void)
        { cout << "y = " << y << "\n"; }
};
```

```
class gamma: public beta, public alpha
{
        int m, n;
    public:
        gamma(int a, float b, int c, int d):
            alpha(a), beta(b)
        {
            m = c;
            n = d;
            cout << "gamma initialized \n";
        }
        void show_mn(void)
        {
            cout << "m = " << m << "\n"
                 << "n = " << n << "\n";
        }
};

int main()
{
        gamma g(5, 10.75, 20, 30);
        cout << "\n";
        g.show_x();
        g.show_y();
        g.show_mn();

        return 0;
}
```

The output of Program 8.7 would be:

```
beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30
```

Note ***beta*** *is initialized first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also, note that* ***alpha(a)*** *and* ***beta(b)*** *are function calls. Therefore, the parameters should not include types.*

C++ supports another method of initializing the class objects. This method uses what is known as initialization list in the constructor function. This takes the following form:

• ~~Polymorphism~~

POLYMORPHISM

• **this Pointer**

* this → object that invoked the member function

Eg A a(10);

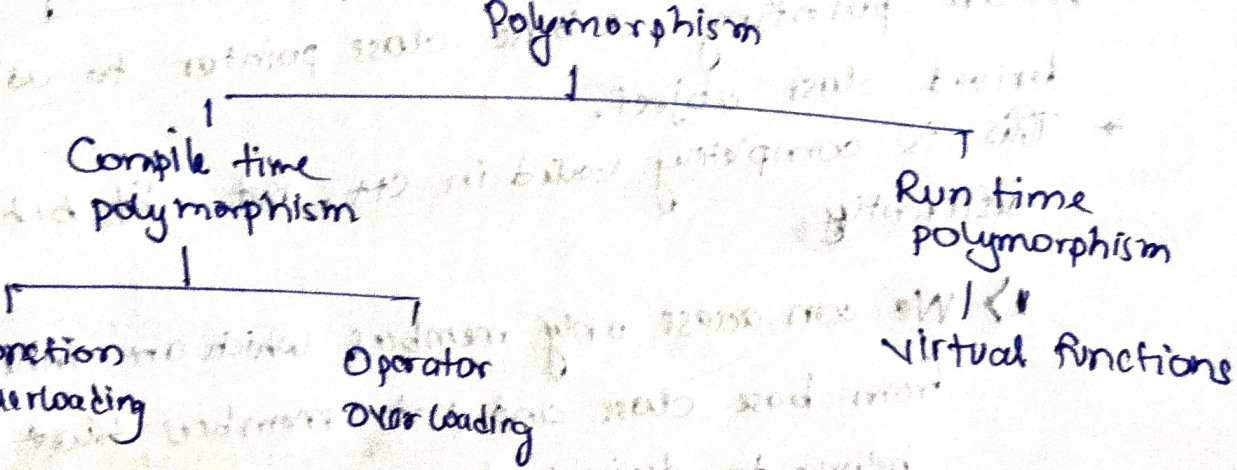
a.display(); → void display(){ this → x }

this refers to obj it was called by

• **Polymorphism**

— one name, multiple forms

— Function overloading, operator overloading



• **Compile-time polymorphism**

Compiler is able to select appropriate function for a particular call at compile time itself. There is no ambiguity at the compile time. This is called

EARLY BINDING | STATIC BINDING | STATIC LINKING

- **Runtime Polymorphism**

Program is able to choose appropriate function when a runtime-object actually makes a call. This is also called as

LATE OR DYNAMIC BINDING

- **Virtual Functions**

I) **Pointer to Derived Class**

Eg:
```
Base *base_class_ptr;
Base obj_base;
Derivedclass obj_derived;
base_class_ptr = &obj_derived;
```

* We are pointing a base class pointer to address of derived class object.
* This is completely valid in C++. But it behaves differently

1) We can access only members which are inherited from base class and not members that originally belong to derived class.

2) This means, say if base and derived class have member function with same name, say display. When base_class_ptr → display will be called, the function of base class will be called and not that of derived class

Why? → Because the pointer of the type ~~Base Mclass~~ Base. So type of pointer decides which method to call. This is done during runtime. ← Run time polymorphism.

(2) Virtual function

We saw that when a base class pointer points to a derived class object and we call a method using base class pointer, it always executes the function of base class.

what if I want to access derived class function using base_class pointer → Make function of base class as virtual.

When a function is made virtual, C++ determines which function to use at runtime based on the type of object the base pointer is pointing to rather than the type of pointer.

virtual void display () { } → in base class

→ This decision is taken during Run-time.

• Pure Virtual function

* It is also called as ~~pure virtual function~~ do-nothing function

virtual void display () = 0;

A pure virtual function is a function defined in base class that has no defination relative to base class.

* Each derived class must define the function or should redeclare it as a pure virtual function, else it will throw an error.

* It is mostly used in abstract classes.

> PURE VIRTUAL FUNCTIONS SHOULD MANDITORILY BE OVERRIDED BY ITS DERIVED CLASSES

- **Virtual constructors and destructor**

WRT a function in base class is declared virtual so that when a base pointer to derived class calls a function, the function of derived class is called instead of base class.

Also, To create an object the constructor of object class must be of same type as the class. But if we declare a constructor of class virtual, it is never called.

This is why > A CONSTRUCTOR CANNOT BE VIRTUAL

* But in C++, virtual destructor is possible. WRT the destructor are called in opposite direction of inheritance hierarchy. When both derived class and base class have destructors, the following happens

```
class A {
  public :
     ~A ( ) {
        // Base class destructor
     }
};

class B : public A {
   public:
      ~B ( ) {
          // Derived class destructor
}  . }
```

```
main ( ) {
     A * ptr = new B( );  → pointer to derived class
     :
     :
     delete ptr
}
```

When we delete memory pointed by ptr, it will trigger the base class destructor only and the destructor of derived class is never called.

This is because. the compiler is looking at the type of pointer and calling its function (as discussed in pointer to derived class)

This can be solved by declaring the base class destructor as virtual.

```
virtual ~A ( ) {
   ...
}
```

Functions that cannot be overloaded in C++

Functions with Different Signatures: You cannot overload functions based solely on their return types. For example, the following is not allowed:

cpp Copy code

```cpp
int foo();
double foo(); // Not allowed
```

Functions that Differ Only in Default Arguments: Overloading based only on different default arguments is not allowed. This is because it would create ambiguity when the function is called without explicitly providing arguments.

cpp Copy code

```cpp
void bar(int x = 0);
void bar(int x = 1); // Not allowed
```

Functions that Differ Only in Const Qualification: Overloading based only on const qualification is not allowed. This could lead to ambiguity when calling the function with a const or non-const object.

cpp Copy code

```cpp
void baz(int x);
void baz(int x) const; // Not allowed
```

Overloading static and non static member function

```cpp
class Test {
   static void fun(int i) {}
   void fun(int i) {}
};
```