OOPS

Structural programming language –

→ Jump statements are not included
→ programs runs as a structure and line by line code is executed

- selection statement
- sequence statements
- iteration statements

↓
machine independent
(since, it is so it takes time to convert to machine code)

Procedural programming

- Divided into number of functions that performs all the tasks
- Top down approach.
  ↳ Break problems into small parts → and then break them also ...

Advantages

- → Each part becomes less complex
- → Parts of problems can be reusable

C, FORTAN, COBOL

Disadvantages of POP

- → No data hiding
- → Global variables are used (more prone to bugs).
- → not based on real world programming

Object Oriented Paradigm / Programming

→ data is treated as critical element
→ Data is tied to the functions and cannot be manipulated by external functions / members.
→ uses bottom up approach
↳ parts are specified more clearly and then they are joined to form larger solutions

C++, C#, python

→ Have access specifiers — public, private and protected
→ Data and function can be easily added.

Objects

- Real world identity
- abstract data type created by programmer
-

Classes

- set of data and functions
- blueprint of object to be created
- user-defined data type

Data abstraction

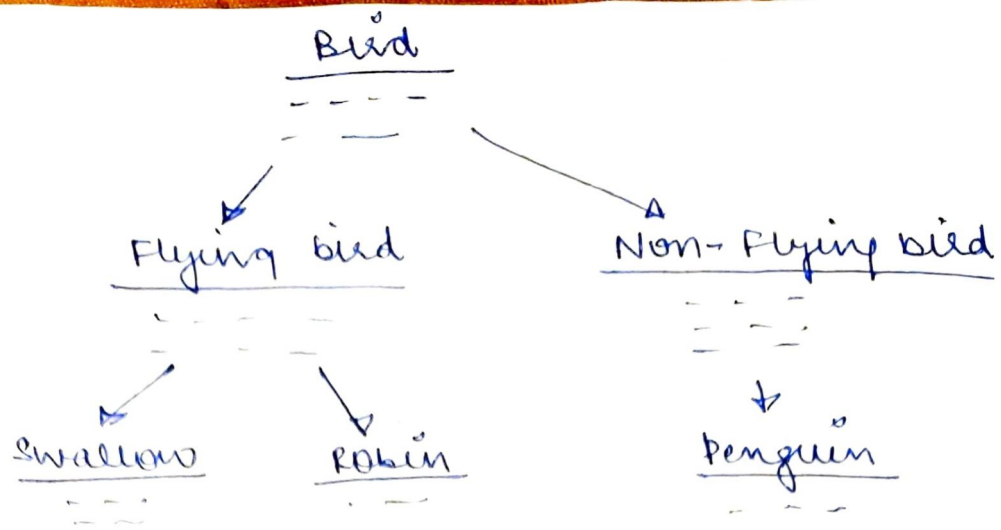
- providing only necessary information / detail to the end user
- since classes use the concept of data abstraction, hence called ADT

Encapsulation

Wrapping up of data and function in a single ~~unit~~ unit is called a encapsulation.
Data is not directly accessible to outside functions, any modification of data can be done by member functions.

Inheritance

↳ object of one class can acquire the properties of another class.
↳ Reusability
↳ heirarchal system type
↳ adding additional features to the class ~~to~~ without modifying it.



Swallow will have attributes of Flying bird and bird both.

Polymorphism

↳ many forms.

↳ If a same function possess different behaviour in different situations, it is called polymorphism

Two types

- compile time
  - Function overloading
  - operator overloading
- Run time
  - virtual functions

Dynamic Binding

Binding → converting identifier (variables, func) into addresses.

Dynamic binding → compiler adds the code and address at runtime

(Detail in ~~ag~~ aage wale pages) :D

Benefit of OOP's

- Inheritance (adding additional features with existing classes)
- Data hiding
- Reusability
- Code maintainance
- Polymorphism

C++

- developed by Bjarne Stroustrup
- Superset of C

C++ = C + classes + inheritance + polymer...

#include is a way of including standard or user defined file in a program.

The process of importing such outside file in our program is called as file inclusion.

- User defined file
  - ~~#include <t:~~
  - #include "abc.c"
- Header file
  - #include <iostream>.

- function declaration are defind in .h (header)
- defination of those func are in .lib (library files)

Namespace

In C++, instead of directly writing func declaration in header file we create a namespace in that

namespace name_of_ns {

keyword ↑

- - - - - - - -
- - - - - - - -

}.

- If in a program we are including two header files and both contains a function of same name, it will lead to contradiction.

To avoid name conflict, we uses namespace.

- contains for identifier
- put names of its member in different space

```
namespace deekshaspace {
    int x;
    // declarations
}.
```

↳ Always have global scope.

we ~~tot~~ can also use <u>alias name</u>.

```
namespace ds = deekshaspace;
```

<u>Accessing member of namespace</u>

```
deekshaspace :: x = 5
```

Now cout and cin ~~of~~ operators are the part of std namespace, to use them we have to use

```
std :: cout <<
std :: cin >>
```

If we used <u>using namespace std</u>, then std has global scope, we don't need to access every function using std ::

⊙ << >> Extraction or get from operator

<u>Structure of C++ program</u>

<u>include files</u>

<u>class declaration</u>

<u>member func. def.</u>

Main function program.

Keywords – Reserved identifiers and cannot be used as variable names.

Identifier – name of variable, function, classes etc created by programmer

C and C++ has a difference on limit of length of names

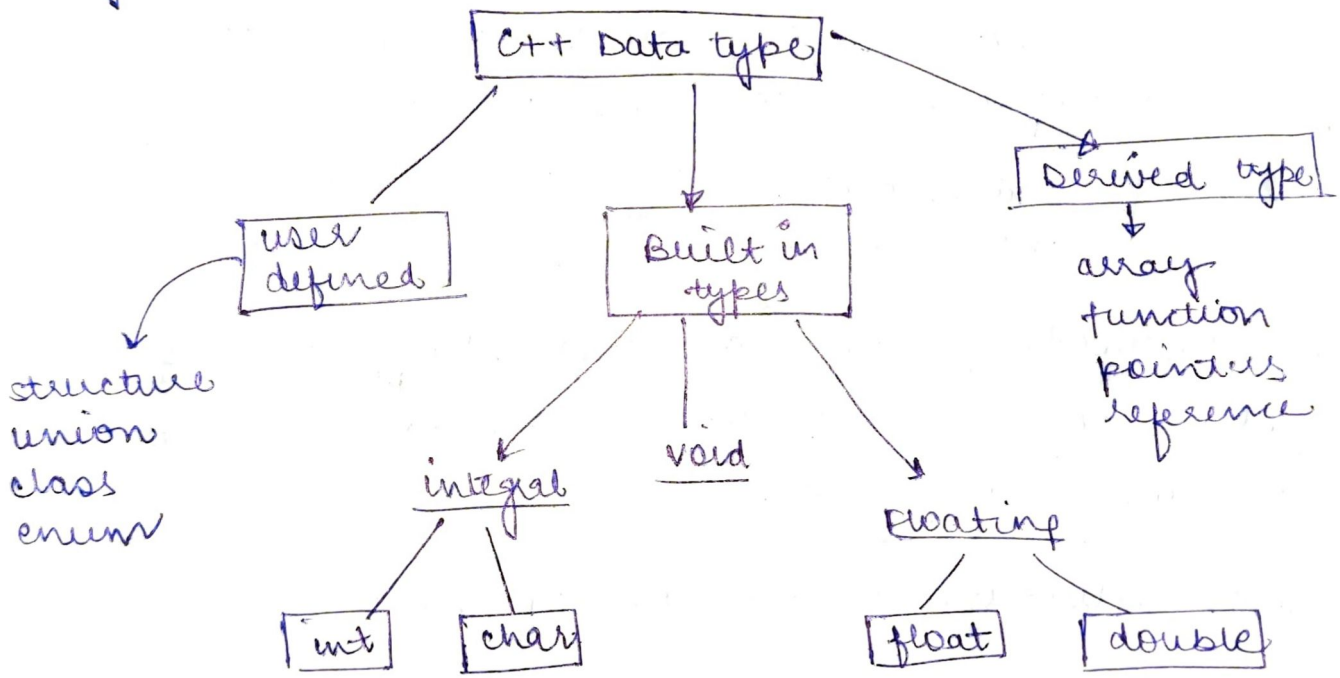
- C → recognise only 32 characters
- C++ → no limit

wchar_t → represent character that require more memory and cannot fit in regular char.

→ must start with L

```
wchar_t = L'A';
```

Data types



Use of void

1) assigning return-type of function, when not returning something
2) generic pointer

Generic pointer

pointer of type void void *gp;

- it can hold the address of any basic data type and can be typecasted to any type
- void pointer cannot be dereferenced.
- If we want to dereference.

```
cout << *(int*) gp      // runs fine
```

User Defined Data types

(a) structure or classes

(b) enumerated data type

- attaching names to numbers
  - enum month { jan, feb, march }.
  - month m1;

Internally compiler treats them as number.

month m1 = june ; //
month m1 = 5 error
month m1 = (month) 5 ; //

We can also do this -

int x = red → color type enum

By default numbering start with zero

enum colour { red, blue, green }

int x = red ;

x = 0

enum color { red = 5, blue, green }

int x = blue // x = 6

In C++, we can create enums without tag names

```
enum {off, on}
    int switch1 = off;
    int switch2 = on;
```

Derived data types

1. arrays ✓
2. functions ✓
3. pointers

store the address of another variable

```
int *p = &a;
```

- const pointer

```
char * const ptr1 = "abc";
```

(cannot modify of of ptr1

- pointer to const

content cannot be changed.

```
char const *ptr2 = &p;
```

Type casting

In C char are stored as ints, but C++ char is not promoted to size of int.

Implicit type casting

bool → char → int → unsigned int → long → long long → double → long double.

```
bool x = false
int y = 10;
y = y + x;
```

↑ converted to int

```
int y = 10
char x = 'a'
y = x + y;
```

Explicit type casting

int x = (type) y ;

int x = type(var)

Reference variable

provide an alias for variable

```
float total = 100
float & sum = total
```

```
sum = 0
{
  total = 0
```

total = total + 10

↳ leads to sum = 110
total = 110.

Difference between: fun1(int &a) & fun2(int *a).

- fun1: Change to a will be reflected in caller
- fun2: change to a will not be reflected, by *a will be reflected

---

Memory management operators

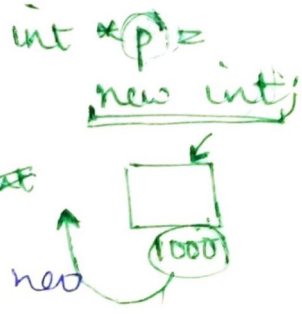
C uses malloc() and calloc() // free()

C++ uses new and delete to dynamically allocate memory

created using new and deleted using delete.

pointer-variable = new data-type

Returns address of data block dynamically created.



we can create a data block also with new

~~int *p = new int [10];~~

int *p = new int [10];

delete a data block q: → delete q[];

* If sufficient memory is not available for allocation, then p return null..

→ In case of failure new returns an exception but malloc() returns NULL.

Difference between malloc() and new

① Syntax

- new invokes object constructor
- malloc() is a function, and needs size of operator to allocate required memory also it returns void pointer which needs to be typecasted

```
ptr = (int*) malloc(sizeof(int)* len);
```

② new can be overloaded, malloc cannot

③ If sufficient memory is not available new throws exception which malloc returns NULL.

bad_alloc

Function prototyping

Defines function interface by return type, number of arguments, type of arguments etc.

```
return_type fun_name(argument list)
```

•) If function body is defined below the main()

•) compiler use this template to ensure that proper args are passed

Call By Reference

passing parameters by reference, any change to the variables in the function will reflect in the called function.

```
swap (int &a, int &b) {
    - - -
}
```

Return by Reference

```
int &max (int &x, int &y) {
    if (x > y) return x;
    return y;
}
```

← Function returns the reference to x or y.

```
max(a,b) = -1
```

assigns -1 to a, if a is greater / vice-versa

Inline Functions

- ~~Some~~ Functions are used to save memory space, but if a function is too small and called very frequently, it will cost in the execution time, because everytime a function is called, it is pushed to stack, copy variables are formed, etc...

* If the function are too small, we can use inline keyword.
During compilation the compiler replaces the function call with the function body everywhere to reduce the execution & calling overhead.

- we just need to add a keyword before function to make it inline

```
inline double cubl(double a) {
    return a*a*a;
}
```

* inline keyword is a request, as compiler may ignore it and execute it like a normal function if function body is too long / complicated.

Inline functions wont work:-

1) there is a loop, switch, goto exist.
2) function contain static variable
3) function is recursive.

Default Arguments

- when argument is not specified during func call.

```
func(int a, int b=10)
fun(10);
```

Constant Argument

function cannot modify the value of argument

```
int fun(const char *p)
```

- significant in call by reference & pointers

Function overloading

same thing for different purpose.

```
int add(int a, int b)
int add(int a, int b, int c)
int add(double a, double b).
```

- First matches the prototype having same number of arguments and then calls app. function for execution.

Difference between C and C++ structures.

1. C structure cannot have member functions but C++ structures can
2. we cannot directly initialize data member in C, but can do in C++
3. we need to use struct to declare a struct variable. In C++ we can directly use structure name
4. C structures cannot have static members.
5. C++ structures can have ~~structure~~ constructor

Classes

way of binding data with appropriate function together

1) class declaration
2) class function defination

```
class item {
    int number;
    int cost;
public;
    void getdata(int a, int b)
};
```

6. The only difference in C++ class and structure is in structure by default members are public whereas in class by default entity is private

Objects

creating object → classname objectname;

accessing member func & variables → ObjectName.func-name();

Defining members function

① Outside class

return-type class-name :: fun-name (args) {

- - -

}

② inside class

Direct declaration ✓

making outside class function declaration + inline

inline return-type class-name :: fun_name (args) {

- - - - -

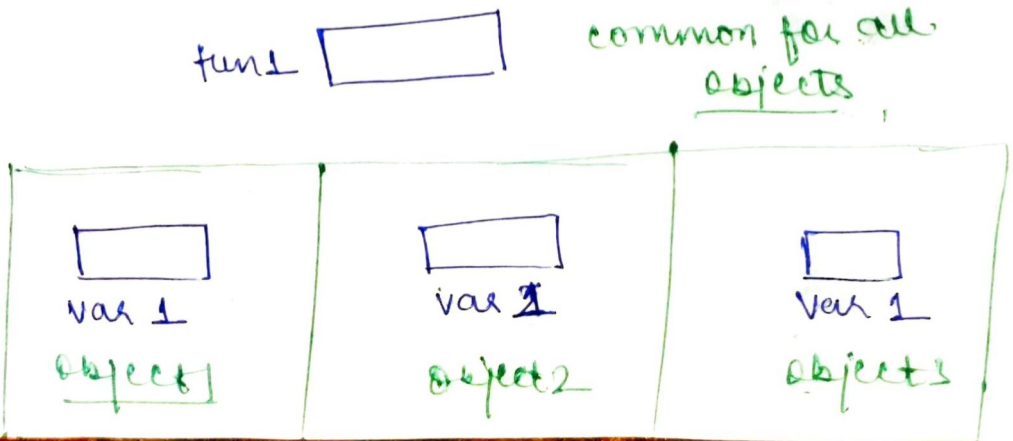
}.

private member functions

can be invoked by ~~et~~ functions of the same class only.

Memory allocation for objects

separate data variables are created for all objects of the class but they use the same member function



Static data member

1) Initialised to zero when first object is created
2) only one copy of that member is created & used by all the objects

- used to maintain common values to the entire class

```
int item :: count;
```

Scope must be defined outside the class, because they are stored separately than as a part of object.

They are called as **class variables**

→ associated with class rather than object.

```
class item {
    static int count;
    - - -
    - -
};
int item :: count = 10;
```

← Initialized to a value of 10.

static member function

- can only access other static (functions or variables) of a class

Accessing static function

```
classname :: fun_name();
```

Friend Functions

- It is not in the scope of class to which it is declared as friend
- It can be invoked like a normal function, without objects
- It cannot access member variables directly and has to use an object name and dot membership operator.

It is defined inside the class using friend keywords

```
class calc {
    int a;
    int b;
    public:
        friend float mean(calc c);
};

float mean(calc c) {
    return (c.a + c.b)/2;
}.
```

function of class X and friend to Y

```
class Y {
    friend int X::fun1();
}
```

Friend class

```
class Z {
    friend class X;      // All members and variables of X are friend to Z
}.
```

Const member Functions

They cannot modify the data variables of the class

Pointer to members

```
int A ::* p = &A :: m
```

- `int` → type of pointer
- `A ::*` → class name
- `p` → pointer var name
- `&A :: m` → member of class whose address is stored in p

`int * ip = &m` ✗ because, m is associated with a class.

`A ::*` → pointer to member of class A

`&A :: m` → address of 'm' member of class A

```
cout << a.*ip
cout << a.m
```
} same — a is object of A

Also we can

```
A *ap = &a;
cout << ap->*ip
cout << ap->m
```
ap = pointer to object a

} — display m.

Local class

class can be defined and inside a function or a block.

local classes can use global variables with scope resolution operator (::)

Restrictions

1) cannot have static data
2) member func. must be defined inside class.
3) Enclosing func cannot access class data.

Constructors

Special member function whose task is to initialise the data members of the class.

- Invoked whenever an object of its class is created.
- No return type
- Name same as class name.

```
class Integer {
    public:
        Integer() {
        }
};
```

```
class Integer {
    public:
        Integer();
};

Integer :: Integer() {
    - - -
};
```

By default constructor is public, if constructor is private, we can instantiate using friend class

```
class A {
    private:
        A() {
          cout << "a";
        }
        friend class B
};

int main() {
    B b;
    return 0
}
```

```
class B {
    public:
        B() {
          A aa;
          cout << "b";
        }
};
```

Output

a
b.

Default constructor

→ that accepts no arguments. A :: A();

If we define no constructor, compiler supplies default constructor

- must be declared in public section
- invoked automatically on object creation
- No return type, not even void
- cannot be inherited, though derived class can call the base class constructor
- constructor cannot be virtual
- We cannot refer to their address.

Parameterized constructors

constructor take arguments

```
class integer {.
        integer (int x, int y);
};.
integer :: integer (int x, int y) {
        ========
        ========
};.
```

~~Objection~~

Objects can be created in two ways

① integer int 1 = integer (0, 100); Explicit

② integer int 1 (0, 100) Implicit

The parameter of a constructor can be of any type ~~exp~~ except that of class to which it belongs.

```
A (A)    // error.
```

copy constructor

```
class A {
    public:
        A(AA);
};.
A I1(10); , AI2(I).
```

reference to its own class as parameter (copy initialization)

assigning value of one object I1 ~~to~~ to I2

Overloaded / multiple constructor

```
class integer {
        integer (int a);
        integer (int m, int n);
};
```

Ambiguity

```
A :: A (int a = 0);
A :: A ();
```

while object creation `A a;` ← It becomes 'ambiguous which const.. to invoke.

Dynamic construction of object

```
Integer * A = new integer ();
```

↑ dynamically allocated

-) Allocating right amount of memory at time of object creation

Destructors

```
~ destructor() { - }
```

- destructor never takes argument nor it does not return any value
- invoked implicitly by the compiler by the compiler upon exit from the program.

Deallocating matrix

```
matrix :: ~ matrix () {
        for (int i = 0 ; i < dl; i++)
          delete p[i];
        delete p;   }
```

Object Overloading and Type Conversions

We cannot overload all the C++ operators.

- Class member access operator (., .*).
- Scope resolution operator (::)
- Size operator (sizeof)
- Conditional operator (?:) (ternary).

```
return-type className :: operator op(args){
        ----
        ----
}
```

Operator functions must be either friend function or member function

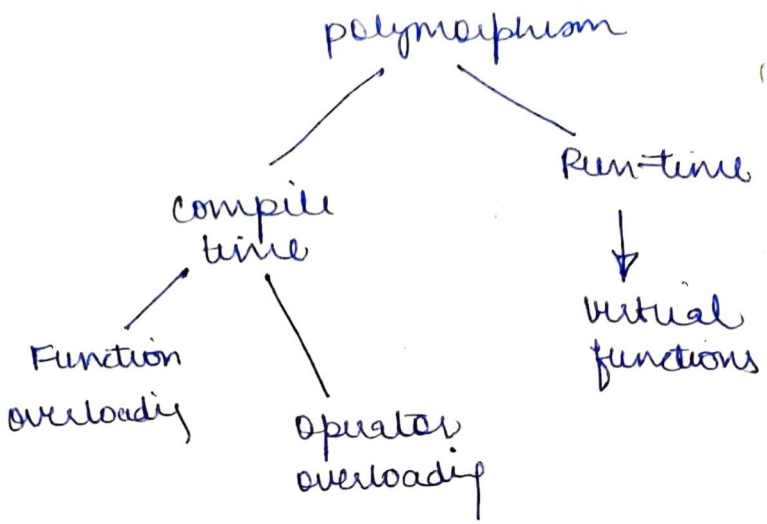
Compile-time polymorphism in which the operator is overloaded to provide special meaning to user-defined datatype.

- can be applied on existing operators
- unary
  - accepts on argument in member func.
  - 1 argument in friend function
- binary
  - accepts one arg on member function
    x op y ⇒ x.operator op(y).
  - two args in friend function
    operator op(x, y).

```
class A{
  int num = 8
  void operator ++(){
       num += 2;
  }
}
```

```
A aa;
aa++;
aa.num = 10
```

Polymorphism (Run-time)



* In function & operator overloading, app. func. is selected for invoking by matching args.
* this information is known to compiles at compile time and thus compiler is able to select appropriate function for a particular call at compile time itself
* This is called early binding or static binding
* when the appt. function is selected at the run-time, this is called as late binding / dynamic binding. or run time polymorphism

Pointers to Functions

- the pointer to function is known as callback function.
- we can use func. pointer to refer to a function
- Declaration

```
data_type (* function_name)();
int (*num_fun)(int x);

void add (int i, int j) {
    cout << i+j;
}
```

↳ ~~void (*add)(int, int) ptr;~~

~~Ptr(2,3) ⇒ 5~~

~~int (*num-fn~~

void (*fun_ptr)(int, int) ; ~~ptr~~ = &add.

fun_ptr(2,3) ⇒ 5
add(2,3) ⇒ 5

Pointer to objects

```
item x;
item *ptr;
  ptr = &x;
```

```
class item {
    int m, n;
    fun() {
    }
};
```

```
ptr->m;
ptr->n;
ptr->fun();
(*ptr).m;
```

*ptr = alias of x

We call also dynamically create objects

```
item *it_ptr = new item;        (single object)
item *it_arr = new item[10];    (objects array)
```

This pointer

'this' a keyword that represent an object that invokes a member function.

```
class ABC {
    int a;
};
```

to access a inside any member function we can use a = ___ directly or this -> a = ___

this points to the invoking object

this is pointing to the object only.

EX

Suppose we want to return greater of 2 objects

```
person & person :: greater(person &x) {
    if (x.age > age)
        return x;
    else
        return *this;
};
```

```
max = A.greater(B).
```

Pointer to Derived class

pointer of the base class are type compatible with pointer objects of derived class, hence same pointer variable can point to base object/derived object

B → Base class
D → Derived class

```
B *bptr;
B b;
D d;
bptr = &d;
```

using bptr we can only access those props of derived class which are inherited from base class.

Hence if both of them has a function of same name, bptr will always access base class function

Problem

```
class B {
    show() {
        ——— print "base";
    }
};
class D : public B {
    show() {
        ——— print 'derived';
};
```

main){

B * bptr;

B b;

D d;

bptr = &d

pt bptr → show() ⟶ base

bptr = &b

bptr → show() ⟶ base

((D*) bptr) → show() ⟶ derived

↳ shows that it can be typecast but it cannot access derived members directly

}.

Virtual Function

We have just seen, the compiler links the function based on the type of pointer, hence everytime function from base class is called.

To achieve run-time polymorphism

- The function in the base class is declared as virtual using keyword virtual preceding its normal declaration
- When a function is made virtual, C++ determines which function to use at run time base on the type of object pointed to by the base pointer, rather than the type of pointer

```
class B {
    virtual void show(){
        cout<<"base";
};
```

```
class D : public B {
    void show(){
        cout<<"Derived";
};
```

B *bptr = &b

bptr → show() → base

bptr = &d

bptr → show() → derived

Rules for virtual functions

- Virtual function must be member of some class
- cannot be static
- can be friend of another class.
- we cannot have virtual constructor, but we have virtual destructor

Pure virtual Functions

- if the function is redefined in derived class then base class function rarely does any task,

The 'do-nothing' function may be defined as

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

- Class containing virtual function (pure) cannot be used to create objects of its own and serve to provide some traits to derived class
  - Also called abstract base classes.
- Also each derived class must have that function as redeclared as pure virtual function.