FREEMIND

Date

Page

C++

- It is an extension of language c developed by Bjarne Stroustroup

Datatypes in C++

- Primitive → char, short, int, float, long, double, bool, etc
- Derrived → Array, pointer, etc
- Enumeration → enum
- User defined data types → structure, class, etc

Namespace in C++

- Delarative region that provides scope to identifiers like variables, functions and classes in a group
- Mainly used to organize code and avoid name conflicts especially when integrating multiple libraries

```
namespace name { }
```

Accessing member → namespace_name :: member_name ;

Input an string and print the value

```
int main () {
    String name ;
    Cin >> name ;
    Cout << "name is : " << name << endl
```

ANSI Standard

- Ensures C++ is portable
- Write Your code for Microsoft's compiles will compile without errors using a compiles on Mac, UNIX or Alpha

C++ Compiles

- GNU C/C++ compiler, or from HP or Solaris
- Compiles the source code

C++ basics syntax

- C++ is an collection of objects that communicate via invoking each other's methods

⇒ Object → Instance of a class

⇒ Class → Template/blueprint that describes the behavior/state that object of its type support.

⇒ Methods → Data is manipulated and all the action were executed

#include <iostream> → header file

using namespace std; → tells compiler to used std namespace

int main()
{
}

- Semicolon is the terminator of statement

C++ identifiers

- Is a name used to identify a variable, function, class, module or any other user defined item
- Does not allow punctuation characters like @, $, % etc within of identifiers
- Case sensitive programming
- Can use _ in naming eg:- ab_cb

C++ keywords

- Reserved words in C++, may not be used as constant or variable or any other identifier names

eg:- auto, enum, else, new, this, throw, bool, break, etc

Trigraphs

- Alternative representation of two characters
- Three character sequence that represents a single character and the sequence always starts with two question marks

eg:- ??=# → #

??/ → \

??(→ ^

White space

- Compiler ignores it
- To distinguish new character, lines, commands etc

Data types

(Variables → are nothing but reserved memory locations to store values)

- Stores various data types like character, integer, string etc.

Types are,

1. Primitive data types
   - Built in data types
   - bool, char, int, float, wide char
   - Each has particular memory to store value

   char - 1 byte
   int - 4 byte
   bool - 1 byte
   float - 4 byte

2. Derived data types
   - Data types derived from basic / primitive types
   - array, pointer, reference, function

3. User defined data types
   - Custom data type created by user according to the need
   - class, struct, union, typedef, using

4. Enumeration
   - Constant integer
   - Enum

Void data types

- Represents absence of value
- Used for pointer and functions that do not return any value using keyword void

create ambiguity / run time error and dangling pointers.

operator

(Modifiers) → char

Type def data types

- Can create a new name for existing data type

  eg: typedef int ft feet;

  now feet can be used as an integer to declare other variables

  feet i;

Modifiers

- Keywords used to give extra meaning to already existing data types.
- Added to primitive data types as a prefix to modify their size or range of data they can store.
- Four types are,
  - short → int
  - long → int, double
  - signed → int, char, long
  - Unsigned → int, char, short

eg: long int var1; // 8 bytes

short int var2;

long double, long long int, unsigned int, etc

- short can be only used with int & short decreases size of data type in memory

FREEMIND

Date

Page

Compound data types

- Derived from built in data types
- Arrays → Collection of variables of same data types stored continuously
- References → Aliases of variables that can be used to refer to them instead of their name
- Pointers → Variables that stores memory address of another variables
- Strings → Sequence of characters
- Structure & Union → Used to group data of different types

auto keyword

- Detects the variable type automatically when value is assigned to it.

eg: auto i = 7; // i is treated as int

variable scope

- Region of program

Types are,

- Local variables → inside a function / block
- Formal parameters → In the definition of function parameters
- Global variables → Outside all functions

Literals

Also known as Constants

- Refers to fixed value that cannot be changed
- #define & const keyword #define → preprocessor

FREEMIND

Date

Page

Sequence	Meaning
\n	new line
\t	Horizontal tab
\v	Vertical tab
\b	back space
\r	Carriage return

Access modifiers / Access specifiers

- Used to assign the accessibility to the class members
- 3 types are,
  - Public → Accessible to all
  - Private (default) → Accessible only to member functions inside the class
  - Protected → Can access outside of its class using friend class / derived class

Storage Class

- Defines the scope and lifetime of variables / functions within a program during the run time.
- Types are,
  1. auto storage class
     → Default storage class for all local variables
     → After C++11, auto keyword is changed, no longer uses it to define

eg: int a = 1;
float b = 2.0; etc

→ Memory location → RAM

2. Static storage class

- Property of preserving their value even after they are out of their scope.
- Once initialized, and exists until termination of the program. So no new memory allocated, cause they are not re declared when an function is called multiple times
- Memory location - RAM

eg: int fnc() { static int count = 0;
return count++; }
int main() { cout << fnc() << "\n";
cout << fnc();

3. Register Storage Class

- Compiler tries to store variable in register of microprocessor if a free register is available (if no free register, then stored in RAM)
- Makes the use of register variables to be much faster than that of variables stored in memory during Run time.
- We cannot obtain the address of register variables using pointers

eg: register char b = 'be';
cout << b;

4. Extern Storage class

- Variable is defined elsewhere and not within the same scope where it is used.

- Used to give a reference to of a global variable that is visible to all program files

eg:

def.cpp
int x = 10;

Main.cpp
extern int x;
int main() { cout << x; }
// output is 10.

5. mutable storage class

- Used to allow a particular as data member of a const object to be modified.
- mutable member can be modified by a const member function

eg: class A { int x;
A() : x(4) { }
}
int main { const A obja;
obja.x = 20; // can be modified
}

6. thread local storage class

- Used in thread.
- Each thread access the value as same as a local variable inside the thread.

eg: thread_local int x = 1;
thread T1([]() { x += 2; });
thread T2([]() { x += 5; });

FREEMIND

Date

Page

Forward declarations (usually)

- H is for classes and functions
- The class is predefined (declared) before its use

eg: void sum(); // forward declaration
class A; " "

Operators in C++

- Symbols that operate on values to perform speific mathematical / logical computations
- Types are,

1. Arithmetic operators
   - Mathematical operations on operands
   - +, -, *, /, %, ++, --

2. Relational operators
   - Comparison of the values of two operands
   - ==, >, >=, <, <=, !=

3. Logical operators
   - Combines 2 or more conditions or constraints or to complement the evaluation of original condition
   - Result returns an boolean value
   - &&, ||, !
     - && → true only when both are true and non zero
     - ! → if false/zero

4. Bitwise operators
   - Compiles converts to bit level and then the value is calculated

(Binary And) & → Copies a bit to the evaluated result if it exists in both operands

(Binary OR) | → Copies a bit to the result if it exists in any of operand

(Binary XOR) ^ → " " " present in either of the operands but not both

(Left shift) << → Shifts value to left by no: of bits specified by the right operand

(Right shift) >> → Shifts value to right " " " " " right operand

(One's compliment) ~ → Changes binary digits 1 to 0 and 0 to 1.

& → will multiply bitwise (0,0 → 0
0,1 → 0
1,1 → 1
1,0 → 0)

| → will add (0,0 → 0
0,1 → 1
1,0 → 1
1,1 → 1)

^ → (0,0 → 0
0,1 → 1
1,0 → 1
1,1 → 0

5. Assignment operators
- Used to assign value to a variable
- =, +=, -=, *=, /=

6. Ternary or Conditional operators
- Returns value based on the condition.
- ? :

7. Miscellaneous operators

1. Sizeof operator
- Unary operator used to compute the size of operand or variable in bytes
- sizeof()

2. , Comma operator (,)
3. Address of operator (&) → to find memory address in which variable is stored
4. Dot operator (access members of structure or class objects)
5. Arrow operator (->) → used to access variables of classes or structure through its pointer
6. Casting operator (convert value of one data type to another datatype)

Loops Control Statements

- break - terminates the loop
- continue - skips the remaining code and proceeds to next iteration

- goto statement - provides unconditional jump. Here goto to a labeled statement in the same function

```
eg: LOOP: do { if(..)
              {..
               goto LOOP; }
            }
```

- infinite loop → for(;;)

Recursion
- Function calling itself
- Types are,
  1. Direct recursion
     - Head → call at start of function
     - Tail → call at end of fn
     - True → call at anywhere in fn (multiple calls)
  2. Indirect recursion
     - fn does not call directly from itself, but calls another fn and that fn will calls the recursive need fn
- Recursion can cause stack overflow if it called infinite times.
- During recursive call, copy of the function is created each time and stored in stack memory

FREEMIND

Date

Page

Lambda expressions in C++

- To allow inline functions which can be used for short snippets of code that are not going to be reused.
- So they do not require a name
- Used in STL algorithms as callback functions

eg:
```
int main() {
    auto fnname = [] int x { return x+x; } // lambda
    cout fnname(5);
    return 0;
}
```
o/p → 10

eg: auto fnname = [] () { }

Parameters / Argument passing technique in C++

1. Pass by value → variable value is copied and then passed to fn.
   So the original value remain unchanged

2. Pass by reference → Reference of the variable is passed.
   So the original value will be changed when passed value changes

3. Pass by pointer → Passing raw address of the argument as parameters

eg:
```
int x = 5;          void change (int* a) {
change (&x)             *a = 22; }
```

FREEMIND

Date

Page

Structures

- User defined data type used to store group of items of different data types

struct name { ... };

Union

- User defined data type used to store different types of data in same memory location

union name { ... };

Dynamic Memory Allocation

- Allocating memory at run time of a program and free it after the use.
- Memory is allocated in heap memory
- Used when don't know the size of array, in data structures (linked list, tree), programs that require effective memory management.

- Achieved using 2 operators,

1. new operator
   - Allocate memory of the given size of data type.
   - If sufficient memory available, initializes memory to the default value according to its type and returns the address to this newly allocated memory
   - new datatype;
     - ↳ int *ptr = new int(5);
     - ↳ int *ptr = new int[10];

FREEMIND

Date

Page

- If enough memory is not available in the heap to allocate, throws and exception type std::bad_alloc · If nothrow is used with new operator, it returns a nullptr pointer

```
int *ptr = new(nothrow)int;
  if(!p) {cout<<"failed";
```

2. delete operator

- To release the dynamically allocated memory
  - → delete ptr;
  - → delete[] ptrarr;

Errors for dynamic memory

1. Memory leak
   - ↳ memory is not deallocated after its use · It will remain until the program runs
   - By using smart pointers, it will deallocate memory after its use

2. Dangling pointers
   - ↳ Pointer that points to a memory location after it is deallocated ie, if we used delete after the pointer usage, it will become dangling pointer
   - ↳ Cases are,
     - Deallocation of memory using delete or free
     - Referencing local variable of function after it is executed
     - Variable goes out of scope

FREEMIND

Date

Page

- int * getdata{ int n = 42; return n;}
  int main(){ int *data = getdata();
- It can lead to unpredictable behaviours, crashes, data corruption

→ Using nullptr or NULL assigning, it can be solved

```
delete ptr;
ptr = nullptr
```

→ Using smart pointers

3. Double deletion
   - ↳ delete is called twice on same memory, leads to crash
   - Assign nullptr after delete

4. Wild pointers
   - ↳ when the pointer is not initialized

5. Buffer overflow
   - ↳ data write out of block
     - ie, write more data into a fixed size buffer or array
     - Use size validation

Placement new operator

- normal new operator does allocates memory and constructs an object in allocated memory.
- Placement new seperates these two things
- i) we can pass a preallocated memory and constructs an object in the passed memory
  - new (address)(type) initializes

eg: int x = 10;

int *mem = new (&x) int(100);

- By invoking destructor, it can be deallocated.

- Difference blw new and placement new

new	placement new
• Allocates memory in heap and constructs objects there	• object construction at known address
• not known the address of memory location its pointing to	• Memory location or address that its pointing to is known
• Deallocation using delete operator	• Deallocation using destructor

Inline functions

- The whole line of code inside inline function is inserted or substituted at the point of its call during the compilation
- Optimizes the performance of program by reducing the overhead related to a function call.

inline int fname() { }

- Execution time is less

Inline functions and Macros

Inline function	Macros
• using inline keyword	• using # define
• Have scope and type checking like regular functions	• No scope or type checking. They are replaced by preprocessor
• Arguments are evaluated once	• Evaluated multiple times
• Handled by compiler	• By preprocessor
• Can access private members of a class	• Cannot access private members
• If fn is large, compiler may ignore	• Always substituted into code
• Can be recursive	• Cannot be recursive

- Both are used for faster execution of program by removing the overhead of function calls
- Inline function offers more safety and scoping benefits while macros are simply preprocessor directives.

FREEMIND

Date

Page

Errors in C++

1. Syntax error → while writing C++ code by violating the rules of writing
   eg: missing }

2. Run time error → During execution (run time) after successful compilation
   eg: division by zero

3. Linker error → when linking the different object files with main
   eg: using Main() for main()

4. Logical error → if desired o/p is not obtained

5. Semantic error → when the statements written in the program are not meaningful to the compiler.
   eg: int a,b,c;
   a+b = c; // semantic error

Exception handling in C++

- Using try-catch block
- By throwing exceptions → using throw keyword
  ↳ try {throw val;}
  catch (ExceptionType e) { }

Types of values that can thrown,

1. Built in Types : try {throw 1;} catch (int e) { }

2. Standard exceptions
   - Set of classes that represent different types of common exceptions
   - All these classes are defined inside <stdexcept> header file
   derived from std::exception

   eg: logic error, runtime error, bad alloc, bad cast

```
try {
    vector<int> v = {1,2,3};
    v.at(10); // out of bound element
}
catch (out_of_range e) { }
```

3. Custom exceptions

- We can use different types of catch blocks for a single try block

- Catch by reference and Catch by value

Stack unwinding

- Process of removing function call frames from function call stack at runtime
- ie, It occurs when an exception occurs in a function but it is not handled immediately in that function.

FREEMIND

Date

Page

OOPS

OOPS - Object Oriented Programming

· To implement real world entities like inheritance, hiding, polymorphism etc in programming.

· The basic concepts that act as building blocks of OOPS are,

1. **Class**
   · User defined data type that acts as a blueprint representing a group of objects which share common properties and behaviours.
   These properties are stored as data members and the behaviour is represented by member functions.

2. **Object**
   · Instance of a class
   · It is an identifiable entity with some characteristics and behaviours.

→ When a class is defined, no memory is allocated. But when an object is created, memory is allocated.

The basic pillars are,

3. **Encapsulation**
   · Wrapping up of data and information in a single unit.
   ie, hiding the sensitive data from users

· It is implemented using classes and access specifiers that keeps the data and the manipulating methods enclosed in a single unit.

· To achieve encapsulation, the variables/attributes should be set as private and it can be accessed only through public get and set methods

· Advantages
   → Better control of data
   → Increased security of data
   → Modularity (organizing the code into seperate objects that handle specific tasks)

4. **Abstraction**
   · Displaying only the essential ~~and~~ information and ignoring the details.

   · Real life eg: Driving a car, only man knows how to drive. He didn't know how its working

Types are,
   (1) Data Abstraction → Shows the required info about data and ignores unnecessary details
   (2) Control Abstraction → Shows only required info about implementation

· Abstraction using classes → class can decide which data members should be visible to outside. It is achieved through access specifiers.

- Abstraction using in header files → when we call and predefined methods using .h (libraries), we don't know its inner working. we just get the o/p.

Difference b/w Abstraction and Encapsulation

Abstraction	Encapsulation
• Process of gaining the information	• Process/method to contain the information
• Problems are solved at design level	• Solved at implementation level
• Hiding the unwanted information	• To hide data in a single entity along with a method to protect info from outside
• Implement using abstract class and interfaces	• By using access specifiers
• Hidden using abstract classes and interfaces	• Data hidden using get and set methods
• Objects that helps to perform abstraction are encapsulated	• Need not be abstracted

Abstract class

- Class that designed to be specifically used as a base class.
- An abstract class contains at least one pure virtual function

```
eg: Class Shape { protected:
                    float a;
                  public:
                    void d;
    (Pure virtual fn) virtual float Cal()=0;

Class Square: Public Shape {
    float Cal() {return a*a;}}
int main(){Square s;
    s.Cal();
}
```

Virtual function

- Member function defined in the base class, and that we expect to redefine in derived class
- If we call ~~a base cls~~ a function using the derived class object by pointer of base class, if the function is not virtual, it will call the base class function.
- Virtual function ensures correction function is called for an object.
- Mainly used to achieve Run time Polymorphism
- It is difficult to debug and it is slow as compiler take more time to call

```
eg: class Base {
        Public:
            virtual int fnname() {}
    }
    class derived : Public Base {
        Public:
            int fnname() { }
    }

int main(){
    Base *bptr;
    derived dobj;
    bptr = &dobj
    bptr -> fnname(); // will call derived class fn
    return 0;
}
```

- override identifier is used to avoid mistakes while redefine virtual fn in derived cls.

FREEMIND

Date

Page

VPTR and VTABLE

VPTR → If an object of class is created, then a virtual pointer is inserted as a data member of the class to point to the VTABLE of class.
For each new object created, a new virtual pointer is inserted as a data member of that class.

VTABLE → Irrespective of whether the object is created or not, the class contains as a members, a static array of function pointers called VTABLE.
Cells of this table store the address of each virtual function contained in that class.

Virtual destructor

- A class may have an virtual destructor, but no virtual constructor.
- Deleting a derived class object using base class pointer that has a non virtual destructor will results in undefined behavior. So the base class should be defined with virtual destructor. (The derived class destructor will not call if no

```
eg: Class base { base(){};                Class derived : public base {
                 virtual ~base(){};                derived(){};
               }                                   ~derived(){};
                                                   }

int main(){ derived * d = new derived();
            base * b = d;
            delete b;     // by deleting, it will call both base and derived destructor
            return 0;     // otherwise if not virtual base, derived destructor will not call.
          }
```

FREEMIND

Date

Page

5. Polymorphism

- Means it has many forms with different characteristics.
- Can be applied to functions and operators.

Types of polymorphism,

1. Compile time polymorphism

- Also known as early binding and static polymorphism.
- The compiler determines how the function or operator will work depending on the context.
- This type of polymorphism is achieved by function overloading or operator overloading

(1) Function overloading

- Two or more functions can have the same name, but different parameters.
- Can be overloaded by changing no. of arguments or changing the type of arguments

```
eg: Class Add {                          int main(){
      void sum(int a, int b)               Add obj;
      { return a+b; }                      obj.sum(5,6);
      void sum(double a, double b)         obj.diff(6.6, 6.6);
      { return a+b; }                    }
```

FREEMIND

Date

Page

(6) operator overloading

- Provide operators with special meaning for particular data type without changing its original meaning.
- achieved with operator keyword followed by arithmetic operator
- We can overload '+' operator in a class, so that we can concatenate two strings by just using +.
- Other example classes where arithmetic operators may be overloaded are complex numbers, fractional no:'s, big integers etc.

```
eg: int a;
    float b, c;
    c = a + b;  // + operator is predefined to add variables of
                   built in data type only
```

```
eg: class A { // statements ; }
    int main() {
        A a1, a2, a3;
        a3 = a1 + a2;   // Here class A is user defined data
        return 0; }        type, + cannot be used as
                           compiler generates an error
```

In the above eg, the user has to redefine the meaning of + to add two classes, done by operator overloading.

FREEMIND

Date

Page

Syntax :- name operator + (const name &)

↳ to add two objects and returns final new object

- Most overloaded operators may be defined as ordinary non member functions or as class members functions.

For non members fn of a class, we have to pass two arguments for each operand,

name operator + (const name &, const name &)

Eg for adding 2 objects using + operator,

```
class Box {
  Public:
    int getvol() { return l*b; }
    void setl(int len) { l = len; }
    void setb(int bre) { b = bre; }
    Box operator + (const Box& b)
    { Box box;
      box.l = this->l + b.l;
      box.b = this->b + b.b;
      return box;
    }
  Private:
    int l, b;
}
```

```
int main() {
    Box b1, b2, b3;
    b1.setl(2);
    b2.setl(3);
    b1.setb(2);
    b2.setb(2);
    b3 = b1 + b2;
    return 0;
}
```

O/p :→ (2+3) × (2+2)
= 5 × 4 = 20

FREEMIND

Date

Page

- Operators which cannot be overloaded are,
  - scope resolution ← no syntax to capture
  - Size of ← could provide at run time
  - typeid
  - .(dot) operator
  - *(pointer) operator
  - Ternary / conditional operator (?:)

- Operators that can be overloaded,
  - Arithmetic → +, -, % (binary)
  - Unary Arithmetic → ++, --
  - Assignment operator → =, +=, ...
  - Bitwise → &, <<, >>, ^
  - Dynamic memory allocate and deallocate
  - [], (), ||, !, ==, <, >, >=, etc

Eg for conversion operator (convert one type to others),

```
class Fraction { int n, d;
    Fraction(int ni, int di)
      { n = ni;
        d = di; }

    operator float () const
      { return float(n) / float(d); }
int main () {
    Fraction(3,5);
    float val = f; // o/p is 0.6
    return 0;
```

FREEMIND

Date

Page

2. Run time polymorphism
   - Also known as late binding or dynamic polymorphism
   - function call is resolved at run time
   - Implemented using function overriding with virtual functions.
   - Redefinition of base class function in its derived class with same return type and parameters

eg:

```
class Base {
        virtual void disp() { };
        virtual ~Base() { };
    };
class derived : public Base {
        void disp() { };
    };
int main() {
        Derived dobj;
        Base * bobj;
        bobj = & dobj ;
        bobj -> disp(); // will call derived class function
        return 0;
    }
```

FREEMIND

Date

Page

Difference blw Compile time and Run time polymorphism

Compile time polymorphism	Run time polymorphism
· Static polymorphism	· Dynamic
· Compiler determines which fn or operator to call based on the arguments, types, orders	· Fn call is determined at run time based on actual object type rather than the reference or pointer type
· Fn calls are statically binded	· Fn calls are dynamically binded
· Faster execution rate	· Comparitively slow
· Inheritance is not involved	· Involves inheritance
· Fn, operator overloading	· Function overriding with virtual fn

Difference blw Fn overloading and overriding

Function overloading	Function overriding
· Compile time polymorphism	· Run time
· Fn can be overloaded multiple times as it is resolved at compile time	· Cannot be overridden multiple times as resolved at Run time
· Can be executed without inheritance	· Cannot be executed without inheritance
· They are in same scope	· They are of different scope

FREEMIND

Date

Page

6. Inheritance

· Capability of a class to derive properities and characteristics from another class

3 modes are,

(1) Public inheritance → public members of base class will be public in derived and protected members of base will be protected in derived

(2) Protected inheritance → Both public and protected members of base class will be protected in derived class

(3) Private inheritance → Both public and protected members of base will be private in derived class. When not specify a mode, it will apply private as default.

Types of inheritance,

1. Single inheritance
   · A class is allowed to inherit from only one class.
   ie, one base class is inherited by one derived class only

2. Multiple inheritance
   · A class can inherit from more than one class.
   · ie, one subclass is inherited from more than one base class
   eg: class A {}; class C : public A, public B {}
   class B {};

3. Multilevel inheritance

- A derived class created from another derived class and that derived class can be derived from a base class or any other derived class

There can be any numbers of levels

eg: class A {};
class B : public A {};
class C : public B {};

4. Hierarchical inheritance

- More than one derived class inherited from a single base class.

eg: class A {};
class B : public A {};
class C : Public A {};

5. Hybrid inheritance

- Combining more than one type of inheritance.
ie, combining hierarchical and multiple inheritance will create hybrid.

Friend class

- Can access private and protected members of another class in which it is declared as a friend.
- It is not mutual, ie, if A is friend of B, then B doesn't become friend of A.

```
eg: class Base {
        private :
            int a;
        Protected :
            int b;

        Public :
            Base() { a = 1;
                     b = 0; }

        friend class Der;
    };
int class Der {
    public :
        void display (Base &t)
        { t.a;      // can access private and protected
          t.b; }
    };
int main () {
    Base bas;
    Der de;
    de.display(bas);
    return 0;
}
```

FREEMIND

Date

Page

Friend function

- Can be granted special access to private and protected members of a class.
- They are not member fn of the class but can access and manipulate the private and protected members of the class for they are declared as friend.
- A friend fn can be,

(i) Global function

- friend keyword is used only in the declaration
- Defined globally like fn.

```
eg: class A {
        private :
            int c;
        Protected:
            int d;
        Public :
            A() { c=1;
                  d=2; }
    friend void fnname (A & obj);
    };
void fnname (A & obj)
    {
        obj.c;
        obj.d;
int main() {
    A ob;
    fnname(ob);
    return 0;
    }
```

(ii) Member function of another class as friend function

- Forward declaration of class is needed

```
eg: class A;
    class B { public : void fnname (A & obj); };
    class A { private: int a;
              Protected: int b;
              Public : A() { a=1;
                             b=2; }
              friend void B::fnname (A & );
    };
    void B::fnname (A & obj) { obj.a;
                               obj.b; }
    int main() { A ob;
                 B ob2;
                 ob2.fnname(ob);
                 return 0; }
```

Friend function and virtual function

Friend function	Virtual function
• non members fns have private access to class representation	• Base class fn overridden by derived class
• access private and protected classes	- ensure correct fn is called
• Declared outside class scope	• Declared with in the base class
• friend keyword	• virtual keyword
• Can access members without inheriting class	• Can be friend of other functions

Constructors in C++

- Special methods that are automatically called whenever an object of a class is created.

Types of constructors are,

1. Default constructor

   - It is automatically generated by the compiler if not define one.

2. Parameterized constructor

   - Allows to pass arguments to constructors, helps to initialize object members

eg:
```
class A {
        A(int x) { };
};

int main() {
        A ob(10);
        return 0;
}
```

3. Copy constructor

   - Is a member function that initializes an object using another object of same class
   - Takes a reference to an object of the same class as an argument.

- User defined or explicity copy constructor / create own version

Syntax : classname (const classname &obj) { }

eg: class A {

Public :

int x;

A(){};

A(A& obj) { // copy constructor definition

x = obj.x;

}

};

int main() {

A ob;

ob.x = 10;

A ob2(ob); // creating another object by copying

return 0;

}

- User defined copy constructor is needed, if an object has pointers or any runtime allocation of resources like file handling, network connection, etc, because default constructor allow only shallow copy.

- Deep copy is only possible with user defined copy constructor

Copy constructor is called,

- when an object of class is returned by value
- when an object of class is passed by value as argument
- when an object is constructed based on another object of same class
- when compiler generates a temporary object.

Copy Elision

Compiler prevents the making of extra copies by making use of technique such as NRVO and RVO which results in saving space and better the program complexity.

NRVO - Named Return Value Optimization (instantly create in the called place)

RVO - Return Value optimization (object is constructed in fn call place)

Copy Constructor and Assignment operator

Copy Constructor

- Called when an object is created from existing object, as a copy of existing object
- Creates seperate memory block for new object

Assignment operator

- Called when an already initialized object is assigned a new value from another existing object
- Does not automatically create a seperate memory block.

- It is an overloaded constructor
- classname (const classname & obj){}

- Bitwise operator
- classname o1,o2 ;
  o2 = o1;

Destructors

- Instance member function that is invoked automatically whenever an object is going to be destroyed.
- Uses defined destructors are needed when dynamic allocation is present
- Cannot be declared as static or const.
- Destructor cannot be overloaded
- Private destructor → when want to prevent destruction of an object

Shallow copy

- In shallow copy, an object is created by simply copying the data of all variables of the original object.
- It works well if object are defined in heap memory
- If variables are dynamically allocated in heap memory, then the copied object variable will also reference the same memory location.
- This will create ambiguity / run time error and dangling pointers.
- Assignment operator

Deep copy

- In deep copy, an object is created by copying data of all variables.
- It also allocates similar memory resources with the same value to the object.
- To perform deep copy, we need to explicitly define the copy constructor and assign dynamic memory if required. Also need to dynamically allocate memory to the variables in the other constructors

Shallow copy	Deep copy
• Create a copy of object by copying data of all members variables as it its	• Create an object by copying data of another object along with the values of memory resources that reside outside the object.
• Copies all of the member field values	• By implementing copy constructor
• Creates a copy of dynamically allocated objects	• Does not create, if properly managed
• The two objects are not independant	• Independant as for dynamically allocated memories

Static data members

- Class members declared using keyword static.
- only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- Life time is the entire program.
- There should be an explicit definition for the static member outside the class, otherwise compiler give an error.
- The static members are only declared in the class declaration.

eg:
```
Class A {
    Public:
      static int x;
    };

int A::x = 2;

int main() {

    Cout << A::x << endl;
return 0;
    }
```

Inline definition of static member

- C++ 17 introduced inline definition of static members of type integral or enumeration

Syntax : static inline data-type user-name = value;

- we can access the static member using class name and scope resolution operator. It is used when no objects of class is present in the scope.

'this' pointer

- pointer to the current instance of a class.
- Used to refer to the object within its own member function

Scope resolution and this pointer

Scope resolution → Accessing static or class members

this pointer → Accessing object members when there is a local variable with same name.

Nested class

- A class declared in another enclosing class.
- The members of an enclosing class have no special access to members of a nested class

eg:
```
class Enclosing {
    Private :
        int x;

    class Nested {
        int y;
        void NestedFun (Enclosing *e)
        {
        cout << e->x;
        }
    };

};

int main()
{
}
```

Structure and class

class	Structure
• Members of class are private by default	• Members of structure are public by default.
• class keyword	• struct keyword
• Data abstraction and inheritance	• Used for grouping of different datatypes
• class name { members fn; };	struct name { :: };

⇒ • Inheritance is for code reusability
⇒ Polymorphism makes a language able to perform different task at different instance
⇒ Encapsulation makes data abstraction (privacy to data)

• Not possible to create a virtual constructor and virtual copy constructor

Pure virtual function and Abstract classes

- Sometimes implementation of all functions cannot be provided in the base class since we don't know the implementation. Such a class is called an abstract class.
- We cannot create object of the abstract classes
- If an class having at least one pure virtual function, it is called abstract class
- A pure virtual function (abstract function) is a virtual function for which we can have the implementation, but we must override the function in derived class, otherwise the derived class will become abstract class.
- A pure virtual function is declared by assigning 0 in the declaration.
- Pure virtual function is implemented by classes that are derived from an abstract class.

```
eg:- Class Base { int x;
                  Public:
  pure virtual ← virtual void show() = 0;
                 int getx() {return x;}
               };

Class Derived : public Base {
                 void show() { cout << "..." ;} // implementation of pure virtual function
               }

int main() {
  Derived d;
  d.show();
}
```

Pure virtual destructor

- Can be declared in C++, but has to provide a destructor body.
- Destructors will not be overridden in derived class, but will instead be called in reverse order. So, for a pure virtual destructor, specify a destructor body.

Static function - virtual posibility

- Static member function of a class cannot be virtual.
- Static functions are tied to the class, not to the instance of the class. C++ doesn't have pointers to class, so not possible to invoke a static function virtually.

Run Time Type Information (RTTI)

- Mechanism that expose information about an objects data type at Run time and is available only for classes which have at least one virtual function.

Run time casts

- Upcasting → pointer or reference of derived class object is treated as base class pointer
- Downcasting → when base class pointer or reference is converted to a derived class pointer

Templates in C++

- Allows to write a generic code that can work with any data type
- The idea is to simply pass the parameter so that we don't need to write the same code for different datatypes.

Syntax :- template < typename A, typename B, ... >
entity - definition

- After definition, we can create instance of template for any desired type by passing type as template parameter.

3 types are,

1. Function templates

- Handle different data types in a same function.
- eg: we can write a function that accept any number whether its float, double or int, and gives a max number as o/p

```
eg) template <typename T> T
    Max(Tx, Ty)
    {
      return (x>y) ? x : y;
    }

    int main() {
      cout << Max<int>(3, 7);
      cout << Max<double>(3.0, 7.1);
      cout << Max<char>('a', 'c');
    }
```

2. Class templates

- Are useful when a class defines something that is independent of the data type.
- It is useful for classes like Linked list, binary tree, stack, Queue, Array etc

```
eg: template <typename T> class Base {
    public:
      T x;
      T y;
    Base(T val1, T val2) : x(val1), y(val2) {} //constructor
    void getvalues() {
      cout << x << y;
    }
};

int main() {
    Base <int> intBase(10,20);
    Base <double> dBase(11.0, 20.1);
    intBase.GetValues();
    dBase.getvalues();
    return 0; }
```

```
template <typename T1,
          typename T2>
class Base {
  public:
    T1 x;
    T2 y;
};
int main() {
  Base<int, float>
      (1, 2.1);
```

3. Template variables

- variable that can work with any type specified when the variable is used

⇒ template <typename T> constexp = T(3.14);

eg:

```
template <typename T>
constexpr T val = T(5);

int main () {

    cout << val<float>;

    cout << val <<double>;

return 0;
}
```

⇒ Templates are expanded at compiles time.
This is like macros, difference is that compiler checks type checking before template expansion

⇒ Variadic template → pass any number of parameters to the template

Function overloading and function templates

Function overloading	Function templates
• Allow multiple fn with the same name but with different parameters	• Defines a generic function that can work with any data type
• Each overloaded fn is explicitly defined for specific types	• Defined once and can be used for multiple types
• Seperate fn for each type	• Same template for different types
• Used when to perform same task on diff data type, but with diff implementations	• Used when same fn logic applies to different types

'using' keyword

- Allows to specify the use of a particular namespace
- useful when working with large code bases or libraries
  - → using for namespace (using namespace std;)
  - → using for inheritance (using Base::Base; in derive class)
  - → using for aliasing (alternative name for a data type (eg: using l = long long;)
  - → using for directives (header file (using std::cout;) using std::endl;)

Standard Template Library (STL)

- Is a set of template classes and functions that provides the implementation of common data structures and algorithms such as list, stacks, arrays, sorting, searching etc.

- It also provides the iterators and functors which make it easier to coordinate with algorithms and containers.

- Use almost every data type without repeating the implementation code.

STL components (features) are,

1. Containers

- Data structures used to store objects and data according to the requirement.

- Each container is implemented as a template class that also contain the methods to perform basic operations on it.

- Every STL container is defined inside its own header file

Types of containers are,

(1) Sequence containers

- Stores the data in linear manner
- Also used to implement container adaptors.
- 5 sequence containers are,

(i) Arrays

- Collection of homogeneous objects
- array container is defined for constant size arrays
- include array header [#include <array>]
- Syntax :- array <datatype, size> name = {...}
- eg : array <int, 5> arr = {3,4,5,6,7};
  - Sort operation

(ii) Vectors

- dynamic array that stores a collection of elements of same type in contiguous memory.
- Ability to resize itself when an element is inserted or deleted.
- Syntax → vector <type> v; (v → name)
- eg: - vector <int> v1 = {1,4,3};
  - vector <int> v2(5,9); // vector of 5 elements with default value 9.
- Element insert using vector.insert(...); and at insert at end using pushback (val);
  v.begin() → to find start

(3) Deque

→ provides fast insertion and deletion at both ends.

→ DEQue → Double Ended Queue

→ Special type of queue where insertion and deletion are possible at both ends

→ Syntax :→ deque <type> name ;

→ eg :→ int main() {

deque<int> dq = {1, 2, 3} ;

dq.push_back (4); // insert back

dq.push_front (0); // insert front

auto pos = dq.begin() + 2; // find 3rd position

dq.insert (pos, 11); // insert at 3rd position

}

dq.size() -> find the size for traversing

pop_back() and pop_front() for deleting

(4) List

→ Implements a doubly linked list in which each element contains the address of next and previous element in the list.

→ It stores data in non-contiguous memory, hence providing faster insertion and deletion once the position is known.

→ Syntax :- list <type> name ;

→ Unlike vectors, list do not support random access. To access element at a position, sequentially go through start or end to the position.

→ front() and back() for accessing first & last elements

(5) Forward List

→ Provides the implementation of singly linked list data structure.

→ Stores data in non contiguous memory where each elements points to the next element in the sequence, makes insertion and deletion faster once position of element is known.

→ Syntax :- forward_List <Type> name ;

→ front() and back() for first and last elements

insert element using insert_after() function

(2) Associate containers

- Stores data is some sorted order
- It provides fast search, insert and delete by using balanced trees like Red black trees.
- Stores the elements in a sorted order based on keys

4 types are,

(1) Sets

→ Each element has to be unique because the value of the element identifies it.

→ By default values are stored in ascending order

→ set<type> name ;

(2) Map

→ Stores data in the form of key-value pairs sorted on the basis of keys

→ No two mapped values have the same key

→ By default, stores data in ascending order

→ map <datatype, value-data type> name ;
(key)

map <int, string> m1 = {{1, "Geeks"}, {2, "Hi"}, {3, "fi"}}

m.at(2) → 'insert at key 2'.

for(auto &p : m1)
cout << p.first ;
cout << p.second ;

(3) Multiset

→ Stores multiple elements with same values

→ Collection of elements sorted on the basis of their values but allows multiple copies of values.

→ multiset <type> name

(4) MultiMap

→ Multiple elements have same key value

→ Collection of key-value pairs sorted based on the keys where multiple pairs can have same key.

→ multimap <keytype, value type> name ;

(5) Unordered Associative containers

- Stores data in no particular order
- Unsorted hashed data

4 types are,

(1) Unordered Set

→ Collection of unique elements, stores its elements using hashing.

(2) Unordered Map

→ Stores data in the form of key-value pairs, but stores its elements using hashing.

→ Collection of key-value pairs that are hashed by

(3) Unordered Multiset

↳ Similar to unordered set, but it can store multiple copies of same value.

(4) Unordered MultiMap

→ Stores data in key-value pairs

→ Similar to unordered map, but allows multiple elements to coth same key.

(5) Containes Adaptoos

- Adapt existing containes classes to suit specific needs or requirements
- Poovides a different interface foo other containes

3 types core,

(1) Stack

↳ LIFO (Last in First Out) priniple foo insert and delete data structuore

→ only at one end of stack

→ stack<type> name;

name.push(1);

(2) Queue

↳ FIFO

→ possible to insert element at one end and remove at other end

→ queue<type> nance;

(3) Poiority Queue

↳ type of queue in which there is some poiority assigned to the elements

Based on this poiority, elements are removed from the queue

→ Adapts a containes to provide heap data structure.

→ Higher its value, higher its poiority

→ poiority_queue<type> name;

4. STL Algorithms

- Povide colde range of functions to perfom common operations on data (mainly containers)
- Foo sorting, searching, modifying and manipulating data in containes

2. Iterators

- pointers like object that are used to point to the memory address of STL containers.
- Connect STL algorithms with containers
- Used to loop through the contents of STL containers

- <type>::iterator name;
eg: vector<int>::iterator it
= v.begin();

3 types are,

(1) Input iterator → Read values from a sequence once and only moves forward
→ Dereferencing, Increment, Equality
↓ allows to update values of element
↳ move iterator to next element
↳ comparing

(2) O/p iterators → Used to write values into a sequence once and only move forward
→ Dereferencing, increment

(3) Forward iterators → Combine the features of both i/p and o/p iterators.
→ forward list, unordered map, unordered set

(4) Bidirectional iterators → Move in both directions forward / backward.
→ support all of forward iterators and additionally can move backward.
→ eg:- list, set, map, multimap support bidirectional iterators.

(5) Random Access Iterators → Iterators that can be used to access elements at distance from the element they point to
→ Support all operations of bidirectional iterators and provide random access to elements.
→ vector, array, deque, string

3. Functors

- Are objects that can be treated as though they are a function.
- Used in STL algorithms.
- It overloads the function call operator and allows to use an object like a function

4 types are,

(1) Arithmetic functors → +, -, *, /, %.
(2) Relational functors → ==, !=, >, >=, <, <=
(3) Logical functors → &&, !, ||
(4) Bitwise functors → &, |, ^

Data Structures

- Used to store and organize data.

(1) Vectors

- dynamic array
- size of array cannot be modified, but vector can resize as needed
- vector <type> name;
- name [0], name [1] -> accessing
- adding and removing at end
- support random access

(2) List

- similar to vector, store multiple elements of same type and dynamically grow in size.
- adding and removing at both ends
- does not support random access

(3) Stack → LIFO

(4) Queue → FIFO

(5) Deque → Double Ended Queue

(6) Set → sorted automatically in ascending order
values are unique

Design Patterns

- Helps to create maintainable, flexible and understanding code.
- Is a generic repeatable solution to a frequently occuring problem in software design
- It is a description or model for problem solving that may be applied in a variety of contexts.

Types of design patterns are,

1. Creational Design patterns
2. Structural Design patterns
3. Behavioral Design patterns

1. Creational Design patterns

- Deal with process of object creation, trying to make it more flexible and efficient.
- It makes the system independant and how its object created, composed and represented

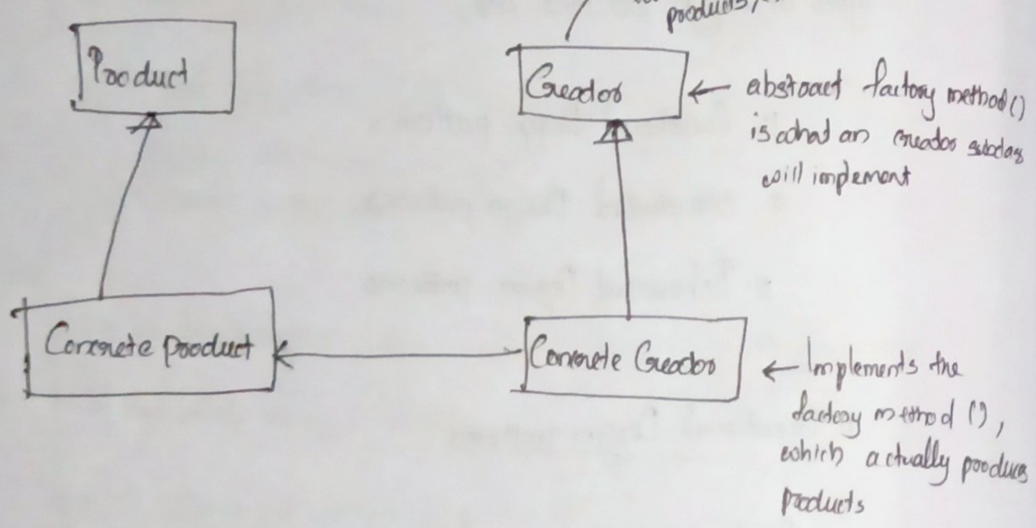
Types are,

(1) Factory method

· It provides an interface for creating object in a superclass while allowing the subclasses to specify the types of objects they create

· This pattern simplifies the object creation process by placing it in a dedicated method, promoting loose coupling btw the object creator and object themselves.

Core Components are,



(1) Creator :- · Is an abstract class or an interface that declares the factory method.

· Typically contains a method that serves as a factory for creating objects

· May also contain other methods that work with the created objects.

(2) Concrete Creator :- · Subclasses of the creator that implement the factory method to create specific type of objects.

· Each concrete creator is responsible for creating a particular product

(3) Product :- · Is the interface or abstract class for the objects that the factory method creates.

· The product defines the common interface for all the objects that the factory method can create.

(4) Concrete Product :- · Concrete Product classes are the actual objects that the factory method creates

· Each concrete product class implements the product interface or extends the product abstract class.

eg :

```
// Abstract product class
class Shape {

    Public :

    virtual void draw() = 0;

    virtual ~ Shape() { }

    } ;
```

```
// concrete product class
class Circle : public Shape {

        Public :
         void draw ()
          {
          cout << " Draw circle " << endl ;
          }
        };

// concrete product class
class Square : public Shape
 {
   Public :
    void draw ()
    {
     cout << " Draw square " << endl;
    }
 };

// Abstract creator class
class shapefactory
 {
  Public :
     virtual Shape* createShape ()= 0;
     virtual ~ shapefactory {}
  };

// concrete creator class
class CircleFactory : public shapefactory
 {
  public :
    Shape*  CreateShape ()
      {
        return new circle();
      }
  };
// concrete Creator class
class SquareFactory : public shapefactory
 {
  Public :
     Shape* CreateShape ()
       {
        return new square();
       }
 }

int main()
{
 cout << " Enter shape (circle / square) " ;
 string shapeType;
 Cin >> shapeType;

ShapeFactory * SF = nullptr;
if ( shapeType = " circle " )
 {
  SF = new CircleFactory();
 }
else
 {
   SF = new SquareFactory();
 }

Shape * shape = SF -> CreateShape();
 shape -> draw ();

delete ShapeFactory;
 Shape factory = nullptr;
delete shape ;
  Shape = nullptr;

return 0;
}
```

(2) Builder Method / Builder pattern

. Defined as creational design pattern that seperates the construction of complex object from its representation, allowing us to create different representations of an object using the same construction process

Components are,

(1) Director :- . Main component of builder pattern responsible for the construction process of the program.
- It cooks with builder to build an object.
- It knowns the actual step required to build an object, but does not know details of how each step is implemented.

(2) Builder :- . Main interface or abstract class that defines the constructions steps required to create an object.

(3) Concrete builder :- • Are the classes that implement the builder interface.
- Each concrete builder is responsible for constructing a particular variant of the object.

(4) Product :- . It is the complex object we want to create
- Product class have methods to access or manipulate these components
- It often has multiple parts or components that are built by the builder.

eg!- @

```
class Pizza  // product class
{
 Public :
void setDough (const std::string& dough)
 {
  this → dough = dough;
 }
void set Topping (const string& Toping)
 {
  this → toping = toping :
 }
private :
  string dough, toping;
Public :
  void displayPizza() const
  {
  cout<< "pizza with dough:" << dough
      << "Toping:" << toping << endl;
  }
};
```

```
// Abstract builder class
class PizzaBuilder
{
Public:
 virtual void buildDough()=0;
 virtual void buildToping()=0;
 virtual Pizza getPizza()const = 0;
};

// concrete builder class for a pizza type
class SpicyPizzaBuilder : public PizzaBuilder
{
 Public:
 void buildDough()
 {
  pizza.SetDough("Pan Dough");
 }
 void buildToping() {
  pizza.SetToping("cheese"); }
 Private:
  Pizza pizza;
};
// concrete builder for another pizza type
class HotPizzaBuilder : public PizzaBuilder
{
 Public:
 Pizza getPizza() const
 {
  return pizza;
 }
 void buildDough()
 {
  pizza.SetDough("Atta Dough");
 }
 void buildToping()
 {
  pizza.setToping("onion");
 }
 private:
  Pizza pizza;
};

// Director class
class cook
{
 Public:
  void makePizza(PizzaBuilder* Pb)
  {
   Pb.buildDough();
   pb.buildToping();
  }
};

int main()
{
 Cook cook;
 HotPizzaBuilder hp;
  cook.makepizza(hp);
 Pizza HotPizza = hp.getPizza();
 hotpizza.displayPizza();
}
```

(3) SingleTon Pattern

- It ensures that only one instance of a class can exist in the entire program.
- ie, if tried to create another instance, it will return already created instance.

It should have,

- private constructor
- Delete the copy constructor
- Private static pointer which points to same class
- Public static method which returns instance of the same class

```
class Singleton
{

Private :

  Singleton()
  {}

~Singleton()
  {}

static Singleton * instance;  // private static instance

Public :

static Singleton& getinstance()  // static method to access instance
{
  if (!instance)
  {
    instance = new Singleton();
  }
  return *instance;
}

Singleton (const Singleton&) = delete ;  // delete copy constructor

Singleton& operator = (const singleton&) = delete ; //delete assignment operator
};

Singleton * singleton :: instance = nullptr ;  // initialize the instance

int main()
{
 Singleton& singleton = Singleton :: getinstance();

  singleton . someoperation in the class();

  return 0;
}
```

(4) Abstract factory method

(5) Prototype method

2. Structural Design patterns

(1) Adapter method

(2) Composite method

(3) Bridge method

(4) Decorator method

(5) Facade method

(6) Proxy method

(7) Flyweight method

3. Behavioral Design patterns

(1) Command method

↳ AT communication

(2) Observer pattern

- Defines a one to many dependancy b/w objects.

ie, if one object (subject) changes its state, its all dependants (observers) are notified and updated automatically.

- This pattern is used to built distributed event handling systems, uses in MVC (Model & View Control).

Key components are,

(1) Subject :- - Is the object that is being observed - It maintains a list of observers and notifies them of state changes

(2) Observers :- - Objects that are interested in the state change of the subject.

- They registers with subject to recieve updates

(3) Concrete Subject :- - Concrete Subject class inherits from the subject interface or class and is responsible for maintaining the state and notifying observers when changes occur.

(4) Concrete observers :- - Implements observer interface or inherit from an observer class.

- They registers themselves with a concrete subject and react to state changes.

eg:-

```
class Observer  // observer interface
{
virtual void update ( int temperature , int humidity )=0 ;
};

class WeatherStation  // subject class
{
private :
    int temperature, humidity ;
    set  Observer *obs[5]
    vector < Observer * > obser ;
Public :
  void registerObserver (Observer * ob)
    {
     ob . push_back (ob) ;
    }
void notifyObservers()
{
for ( Observer * obs : obser)
{ obs -> update (temperature, humidity) ; }
```

```
void setValues ( int temp , int hum)
  {
    temperature = temp ;
    humidity = hum ;
    notifyObservers() ;
  }
};
// concrete observes
class Display : public Observes
{
 Public :
   void update (int temperature , int humidity )
    {
      cout << temperature << " " << humidity << endl ;
    }
};
int main()
{
 WeatherStation ws ;
 Display D1 ; // create display
 Display D2 ;
 ws . registerobserver (&D1) ; //register display
 ws . registerobserver (&D2) ;
 ws . set
}
```

Solid Principles

- 5 essential guidelines that enhance software design, making code more maintainable and scalable.

Principles are,

1. Single Responsibility Principle (SRP)

   ↳ Every class should have a single responsibility or single job or single purpose

2. Open/Closed Principle

   ↳ ~~Class~~ classes, modules, functions etc, should be open for extension, but closed for modification

   ie, should be able to extend class behaviour without modifying it.

3. Liskov's Substitution Principle (LSP)

   ↳ A child class should be usable in place of its parent without any unexpected behaviour

4. Interface Segregation Principle (ISP)

   ↳ Applies to interfaces instead of classes and similar to single responsibility principle

   ↳ Do not force any client to implement an interface isn't relevant to them

5. Dependency Inversion Principle (DIP)

   ↳ High level modules should not depend on low level modules, both should depend on abstractions

   ↳ Suggests that classes should rely on abstractions (interface or abstract classes) rather than concrete implementation

Multithreading

- Technique where a program is divided into smaller units of execution called threads.

- Each threads runs independantly but share resources like memory, allowing tasks to be performed simultaneously.

Syntax :- thread thread_name(callable type)

↳ fn pointer/object

```
void func()
{
  cout << "started" &;
}
int main() {
  thread T(func); // thread that runs function func()
  T.join(); // waiting thread to finish
return 0; }
```

eg: void fun(int n)

```
{
cout <<n;
}
int main()
{
thread T(fun, 4);
T.join();
return 0;
}
```

eg: class Base

```
{
public:
    void f1(int n)
        { cout<<n;}
    static void f2(int n)
        {cout <<n;}
};
int main {
    Base ob;
thread T1(&Base::f1, &ob, 3);
T1.join();
thread T2(&Base::f2, 2);
T2.join();
return 0;
}
```

Problems with multithreading

1. Dead lock → · when two or more threads are blocked forever since they are each waiting for shared resources that the other thread holds.
   · This create a cycle of waiting and none of threads can proceed.

2. Race condition → When two or more threads access shared resource at same time and atleast one of them modifies the resource.

3. Starvation → When a thread is continously unable do access shared resource since other threads keep getting priority.

Thread Synroonization

1. Mutex

· Locks the shared resocorce for a thread and release it after the use of thread.

```
mutex m;
m.lock(); // at start of fn
m.unlock(); // at end of fun
```

2. Critical Section

- Critical section is a segment of a program where shared resources are accessed by multiple processes or threads.

- Critical section has an entry section and exit section

Entry section → Request permission to enter

Exit section → Process releases lock or semaphore allowing other processes to enter the critical section

3. Semaphores

- Synchronization tool used in concurrent programming to manage access to shared resources.

- It is a lock based mechanism

- Semaphore uses a counter to control access

The process Semaphores provides two operations,

(i) wait(P) :- wait operation decrements the value of semaphore

(ii) signal(V) :- increments the value of semaphore

when the value of semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

- When a process performs a wait operation on a semaphore, the operation checks the value of semaphore is >0

- If so, it decrements the value of semaphore and lets the process continue its execution; other wise block the process.

A signal operation on a semaphore activates a process blocked on the semaphore if any or increments the value of semaphore by 1.

- The initial value of semaphore determines how many processes can get past the wait operation.

- Binary Semaphores and Counting Semapbores

wait(P) :- If value >0, the process is allowed to continue and semaphore value decremented by 1. If value is zero, the process is blocked (waits) until semaphore value become >0.

Signal(V) :- After a process is done using shared resource, it performs signal operation. This increments semaphore value by 1, potentially unblocking other waiting processes and allowing them to access the resource.

~~Events~~

A. Condition Variable

- used with mutex
- Mainly used to notify the threads about the state of shared data

Event handling / Event communication

- For handling multithreaded operations
- Creat an event using

```
Handle hEvent = CreateEvent (NULL, TRUE, FALSE, NULL);
```

(TRUE → manual reset; NULL → security attribute; FALSE → initially not signalled; NULL → unnamed event)

: Wait for the Event using

```
DWord dwas = WaitforSingleObject (hEvent, INFINITE);
```

- Another thread signals the event using,

```
SetEvent (hEvent);
```

- Reset the event using

```
ResetEvent (hEvent);
```

- Inter thread communication
- One thread waits, another signals

-> Wait for Multiple Objet -) waits until one or more objects in the array of handles become signaled or until a specified time out.

- Create the thread using,

```
CreateThread (NULL, 0, WorkerThread, hEvent, 0, NULL);
```

(NULL → security; 0 → stack size; WorkerThread → thread fn name; hEvent → Parameter to thread; 0 → run immediately; NULL → don't need thread ID)

Inter process communication

- Allows two or more processes to communicate with each other and share data/ resources

Types are,

(1) Pipe (Anonymous / Named) - unidirectional data channel
(2) Shared memory - memory level sharing
(3) Message queues
(4) Socket communication - over network / local host
(5) Signals - Event communication
(6) Files - shared file for read/write

Shared memory

- Create using CreateFileMapping() memory mapped file or shared memory
- Maps the shared memory into process address space using MapViewOfFile
- OpenFileMapping – allows another process to access the same shared memory
- UnMapViewOfFile
- Two or more process can access a common shared memory address space

COM Servers

Component Object Model (COM) allows to communicate using interfaces, via proxies, etc

- A COM servers (exe)
- A COM client (another process creates and call its interface)

AT Communication

Application Template architecture

- Support two communication,
  1. client to server Communication
  2. Multicase / Event Communication

Proxy communication

- A proxy component (middle man) receives a request from a client and forwards it to the actual target
- TCP proxy, http proxy
- Send command via proxy and get the replay from same path

Smart pointers

- Is wrapper class ~~around~~ over a pointer that acts as a pointer but automatically manages the memory it points to.
- It ensures that memory is properly deallocated when no longer needed
- Smart pointers are implemented as templates so we can create a smart pointer to any type of memory.

Types are,

(1) auto_ptr → auto_ptr <type> name; (automatically manage its life time)

(2) unique_ptr → stores one pointer only at a time. Cannot copy

(3) shared_ptr → more than one pointer can point to same object at a time holds a reference counter

(4) weak_ptr → holds a non owning reference to an object. Similar to shared but no reference counter

Avoid circular dependency created by 2 or more objects pointing to each other using shared ptr

Aggregation & Composition

- Aggregation → one class is a container for objects of another class
  one object uses another, but does not own it. The contained object can exist independantly of parent

- Composition → Parents own and controls lifetime of the contained object. When parent is destroyed, the child also destroyed.

DICOM

Digital Imaging and Communications in Medicine (DICOM) is the standard for communication and management of medical imaging information and related data.

Dicom Tag → Unique value for identifying an attribute
Composed of grouptag (4 digits) and followed by element tag (4 digits)

0010, 0010 - Patients name
0008, 0008 - Image type

- Factory method used
- Get instance
- Get Info and Set Info
- Attribute value and Attribute name

Binary Tree

- Is a hierarchical data structure where each node has atmost two children.
- Can store any structure (not sorted)

Basic operations are,

(1) Create a node
(2) Insert nodes manually
(3) Traverse the tree
  - → inorder (Left, Root, Right)
  - → Preorder (Root, Left, Right)
  - → Postorder (Left, Right, Root)

- A binary tree node contains, Data, Pointer to left child, pointer to right child

Binary Search Tree (BST)

- Is a type of binary tree data structure in which each node contains a unique key and satisfies a specific ordering property
- All node in the left subtree of node contains value strictly less than node value
- All none in the right of node contains value greater than node's value.

Inserting a value in binary search tree is by,

(1) Initialize the current node with root node

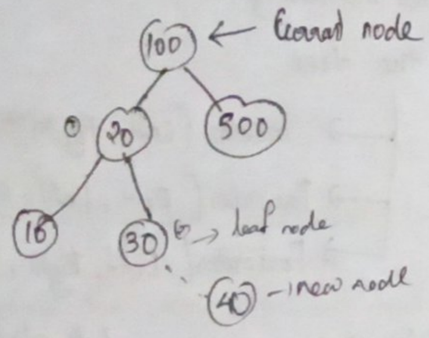
(2) Compare the value with root node.

(3) Move left if value is less than or equal to current node value

(4) Move right if value is greater than current node value

(5) Repeat (3) and (4) until reaching the leaf node

(6) Attach the new key as left or right child based on comparison with leaf node.



if 40 is to be inserted,

Hashing

- For storing and retrieving data in faster as it performs optimal searches.
- A hash function takes an input (message) and returns a fixed size string of characters
- Used to map data of arbitrary size to fixed size values called ~~class~~ hash values / hash codes or hashes.
- Used in unordered map, unordered set

- A hash function takes an i/p string or integer and returns a fixed size integer (hash code)

```
class hashfn .                    // with std STL example
{
 public :
    static int hfn ( string st)
    {
     int hash_value = 0;
     for( int i=0; i < st.length() ; i++)
     {
       hash_value += static_cast <int>(
     }
     return hash_value % 101 ;  // return hash value modulus of some
                                   large prime number
    }
};

int main()
{
 string str = "Hey" ;
 int hval = hashfn :: hfn (str);
 return 0;
}
```

- It is difficult to generate the original i/p from hash value

Binary Heap

Is a binary tree with following properties.

- Its a complete tree (all levels are filled completely except the last level and last level has all keys as left as possible).

This property of binary heap makes them suitable to ~~seen~~ store in array.

- Min Binary heap - key at the root must be minimum among all keys present in binary heap
- Max Binary heap - similar to min heap, mainly ~~is~~ implemented using an array

Binary ~~hap~~ heap can be implemented by,

- Create a binary heap class that has a private array to store elements of heap and private method for maintain heap property.
- Class should have constructor for initialize heap and a method to add new elements to heap. Add method should insert element at end of array.
  If the value of new element is larger (for max heap) or smaller (for min heap) than value of its parent, Then two elements should be swapped. The process is repeated until heap property satisfied.
- Class should also have a method for removing root element from heap.
  same process as add method, difference is smaller (for max heap) and larger (for min heap)
- Also method for checking heap is empty

SFINAE

- Substitution Failure Is Not an Error
- It means that compiler should not fail to compile a program just because it cannot substitute a template parameter.
- It is a way of allowing a compiler to decide which template to use in a given context.

Pipe Communication

- Pipe is a medium between 2 or more related or interrelated processes.
- It can be either with in one process or a communication blw child and the parent processes.
- Communication is achieved by ~~eee~~ one process writing into the pipe and other reading from the pipe.
- To achieve pipe system call, create two files, one to write and other to read
- It is a one way communication
- It has two descriptors, Descriptor pipedes[0] for reading and Descriptor pipedes[1] for writing

Named pipe

- For unrelated process communication
- ie, to execute client program from one terminal and works at server program at other terminal.
- Two way communication is achieved in named pipe.
  ie, communication blw server and client, also at same time also.
- It is FIFO

Process for creating a named pipe,

- Create 2 process, fifo server and fifo client
- In seve
  - Create a named pipe
  - open the named pipe for read/write purpose
  - Waits infinitely for message from client
  - If message received from client is not End, then prints the message. If message is End, close the fifo and ends the process. If not end, reverses the string and send back to client

- In client
  - Opens the named pipe for write purpose
  - Accepts the string from user
  - Checks if user send End or not. Either case, it sends message to server. If not end, wait for (reversed string) message from client and prints
  - The process repeats infinitely until user enters end string.

Dereference

- Accessing the value stored at memory location held by a pointer

eg: `int x=10;`
`int *ptr = &x; // dereferencing (op is 10).`
`*ptr = 100; // changing the value`

Reference and Dereference

Reference

- Is a variable for another variable, which allows to access or modify the original value/variable without using its name directly.
- & is used
- eg: int &x = &y;

Dereference

- Is the process of accessing value stored at memory address held by a pointer.
- * is used
- eg: int *x = &y;

eg: int number = 10;

int *ptr = &number // dereference

int &ref = number // reference

cout << ref; // original value will print

ref = 20; // modifying value

Socket programming

- socket - end point of two way communication b/w 2 programming in network. They are generally hosted on different application ports and allow bidirectional data transfer.
- Socket programming refers to the communication b/w 2 sockets on the network, allows bidirectional communication
- Use socket API to create a connection b/w 2 programs running on the network, one which recieves data by listening to the particular address port and other sends the data

Types of socket are,

(1) Stream socket :- provide reliability and connection based communication, ensuring data sent and receive in correct order, w/o loss or duplication.

(2) Datagram socket :- used for connectionless, unreliable communication faster, but does not guarantee reliable data

Stages for creating server socket,

- Create server socket
- Define the server address

- Binding the server socket
- Listening for connections
- Accepting a client connection
- Receiving data from client
- Closing server socket

Stages of client socket,

- Creating client socket
- Defining server address
- Connecting to the server
- Sending data to server
- closing the client socket

Message Queue

- Linked list of messages stored within the kernal and identified by a message queue identifier and it is used to send and receive messages b/w process in a FIFO manner.

- msgget() → to create a new queue or open an existing queue
- msgrcv() → new messages are added to end of queue

- In message queue, every message has a +ve long integer type field, a non-ve length, and actual bytes of data, all of which are specified to msgsnd().

- Messages are fetched from a queue by msgrcv().

working of IPC using message queue,

- Creating or accessing a message queue
- Sending the message
- Receiving the message
- Deleting the message if no longer needed by msgctl().

Functions of Message queue,

- Message storage → until received, queue stores it
- Ordered communication → delivers in the order they are sent
- Asynchronous communication
- Prioritization
- Error handling

Void pointer

- pointer that has no associated data type with it.
- It can hold address of any type and can be type casted to any type.

eg: int a = 10;
char b = 'x';

void *p = &a; // hold address of a
p = &b; // hold address of b

- It cannot be dereferenced

auto_ptr

auto_ptr<type> name;

eg: auto_ptr<int> ptr1(new int(10));

auto_ptr<int> ptr2 = ptr1; // transfer ownership of pointer to ptr2

cout << *ptr1;

- It takes the ownership of the object it points to, ensuring object is automatically deleted when auto_ptr goes out of scope

Unique_ptr

- Stores one pointer at a time
- Cannot copy unique_ptr, only transfer ownership to another unique_ptr using move method.

eg:
```
class base
{
  int a,b;
  public:
    base (int a, int b)
    { a = a;
      b = b; }
    int area () {return a*b;}
};

int main()
{
  unique_ptr<base> obj1 (new base(10,5));
  cout << obj1->area();
  unique_ptr<base> obj2;
  obj2 = move(obj1);
  obj2->area();
  return 0;
}
```

shared_ptr

- Uses a reference counter using use_count() method

eg: shared_ptr<type> name1;

shared_ptr<type> name2;

~~name2~~

name2 = name1;

cout<<name1.use_count(); // point 2 values as counter (shared no:) is 2

(eg: of unique_ptr can use)

weak_ptr

- Same as shared_ptr, but doesn't have reference counter
- The pointer will not have strong hold on the object

eg: shared_ptr<base> ob1(new base(1,2));

weak_ptr<base> ob2(ob1);

ob2.use_count(); // returns no: of shared_ptr objects