**Day-01:**

# OS concept:

* printf() - library function
* write()- system call - kernel dependent
* driver - h/w dependent

***OS:***

* Resource management
* Interface b/w user & resources (User -> Application -> UI/GUI (Shell) -> Kernel -> Resources)
* Kernel + Utilities (GUI/UI) + Application

***Kernel:***

* collection of several components (several programs)
* components ex: memory management, process management, file management etc..
* system call + data structure + headers + file system + device driver
* Interface b/w user & resources
* User -> Application -> UI/GUI (Shell) -> Kernel -> Resources (H/w)

***Layers:***

Diagram

Description automatically generated

***System structure:***

Program Process

Set of instructions Set of instructions

Resides in Hard disk Reside in Memory

Passive entity Active entity

Does not contain resources consumes various resources

***Process:***

Diagram

Description automatically generated

# Compilation & Build process

* Pre-processor
* Compiler
* Assembler
* Linker

# Structure:

|  |  |
| --- | --- |
| **Structure in C** | **Structure in CPP** |
| Only data | Both data & function |
| Access specifier not used | Access specifier |
| Size of empty structure 0 | Size of empty structure is 1 |

# Limitation of C:

* Data is not secured
* Data can be manipulated by another function
* Not preventing unauthorized access
* Cannot compare two user defined type
* No object code reusability (structure features cannot be added without modifying structure)

|  |  |
| --- | --- |
| **C** | **C++** |
| Procedure oriented | Object |
| Importance: function | Importance: function + data |

# C++ features:

* Encapsulation (Binding Data & functions)
* Re-usability
  + Inheritance (overriding)
  + Containership
  + Templates
* Polymorphism
  + Static polymorphism (compiler differentiate x\*y & \*p, Cpp same function name)
  + Dynamic or run time (virtual functions)

***C++ Access Specifier:***

* private, public, protected

***Object:***

* real world entity which has characteristics (state, behaviour etc..)

***Class:***

* Blueprint of an object
* Defines the characteristics & behaviour of object
* Contains
  + Data (to represents state/identity)
  + Functions (member function or method)

***Encapsulation:***

Diagram

Description automatically generated

# Function overloading: (Polymorphism):

* Define multiple functions with same name
* Rules of function overloading:
  + Number of arguments
  + Types of arguments
  + Order of passing arguments
* One can overload the function to any number if no ambiguity for compiler

# Types of function:

* Non-member functions
* Member function

***Non-member function:***

* function which is defined outside a class

***Member function:***

* function defined within a class

***Static Member:***

* Data that does not needs to be changed for all objects shall be define as static member.
* Bank name, Address, IFSC code will be same for all customers of bank branch.
* Its common for all objects and doesn’t contribute to size of class (size of class does not include size of static member).
* To initialize static data:
  + Data\_type class name::var\_name = value;
  + int PLOT::citycode = 915;
* Initialization must be global space (outside main() or any other function) as it is static variable.

***Static Method:***

* A method which accesses only static data member.
* Ex: PLOT::get\_citycode
* Static method can be called before object creation.
* Non static members are not allowed to access inside static function.
* Static method cannot access non-static member.
* But Non-static method can access both static & non-static members.

***Day-02:***

***CONST Method:***

* Read only method
* Which can read the data but cannot modify the data member

***CONST Data:***

* Same as c programming.

***Mutable Data:***

# Constructor:

* A special method
* It will have the same name as class name
* To initialize data member of an object constructor is used.
* Constructor are called implicitly after the creation of an object.
* Constructor may or may not take arguments.
* Constructor does not have any return type.
* Constructor are called only once during object creation. (P5 = P2, it will not call constructor).
* Constructor can be overload.

**Types of constructors:**

1. Default constructor
2. Parameterized constructor
3. Copy constructor

**Default constructor:**

* It does not take any arguments.
* PLOT()
* {
  + Initialized data member
  + Acquire some resources (memory, device, file open…)
* }
* Default constructor only added when you haven’t defined any other constructor
* For ex: if you have defined parametrized constructor and if you create objects without passing arguments, compiler will give the building error.

# Destructor:

* Are implicitly called when object lose its scope
* It will have the same name as that of class, prefixed by ~
* Ex: ~PLOT()
* {
  + Release the acquired resources.
* }
* Compiler would add destructor for every class and programmer can override the existing destructor.

# “this” Pointer:

* “this” is pointer to an object, which holds the address of the object through which that method is called.
* For ex: PLOT P1;
* “this” pointer holds the address of the P1.
* All method can access this pointer except “static” function as its called outside and it does not need access to non-static member.
* “this” is a local variable declared within the object.

# Pointer type:

* Non constant pointer to non-constant data = int \*p = &x;
* Non constant pointer to constant data = > int \*p = “VOLVO”;
* Constant pointer to non-constant data = int \* const p = x;
* Constant pointer to constant data = const int \* const p = x;

# References:

* Reference is same as constant pointer to non-constant data (int \*const P).

|  |  |
| --- | --- |
| * Int x=10; * Int \*const p = x; * \*p = 50; * Printf(“%d\n”, \*p); | * Int x=10; * Int &refx = x; * Refx = 50; * Printf(“%d\n”,refx); |

* It’s an alias name to variable.
* Both reference variable and original variable share the same memory location.
* We can’t have reference to constant variable.
* We can have reference for pointer also
* For ex:
* Int \*p=1000;
* int \*& refx = \*p;
  + printf(“%d\n”,\*refx)

***Adv. Of references:***

* efficient way of data transfer from one function to another function.

# Operator overloading (compile time polymorphism):

* see code
* cout << “hello”;
* cout is object, << is insertion operator, “hello” is string
* we can interpret it as :
* cout.operator<<(“hello”) //for operator
* operator<<(cout,”hello”) // if its non-membered operator

# Friend Function:

* A class can allow non-member function to access its private member.
* for ex:
* class abc
* {
* }
* Class def
* {
  + Friend class abc;
* }
* So all the members/methods of def class can be accessed by abc object.
* One class can consider another class as friend class.

# Explicit keyword:

* It is a keyword to restrict compiler to create a nameless object creation.
* For ex: class PLOT p1, p2;
* P2 = p1 + 50; => p2 = p1 + namlessobject(50);
* Compiler treat 50 as nameless object and revoke PLOT 1 argument constructor and initialize its first member with 50 value.

# Assignment Operator:

* It will implicitly called when object is created
* But if you define it (override it) it will not call default one.

***DAY-03:***

# Process:

* Each process contains Process control block
* Process contains information of
* Each process has same size of PCB.
* When process created, a structure is created to maintain the PCB.
* Kernel use link list to create the new process
* For creating process kernel use dynamic memory allocation to avoid memory wastage(in case of fixed memory).

Shape, rectangle

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

# Dynamic Memory allocation in C++:

* In C we are using: malloc, calloc & realloc
* In c++ new is used to allocate dynamic memory.

|  |  |
| --- | --- |
| * **C (malloc)** | * **C++ (new)** |
| * Int \*p; | * Int \*p |
| * P = (int \*) malloc(sizeof(int) ); | * P = new int; |
| * Size needs to be specified | * No need |
| * Requires typecasting | * Not require |
| * It is function | * It is an operator |
| * It just assigned memory and not creating object, so no constructor call. | * Its assign memory & creating object as well, so calling default constructor and initialize value |
| * Malloc returns “null” on failure | * New throws exception on failure |

* We can pass the arguments while assigning the dynamic memory to call the parametrized constructor
* Even we declare explicit constructor, it will call the default constructor as its allocating the memory and creating object. Explicit will be useful for nameless object only.
* If we have static member and we try to allocate dynamic memory it will not allocate memory for static member as its stored in data section and all objects share the same memory location for static member.

Dynamic Memory Allocation:

Diagram

Description automatically generated

# Releasing dynamically allocated memory:

|  |  |
| --- | --- |
| Free | Delete |
| Function | Operatoe |
| Free is not calling destructor | Delete will call destructor |
|  |  |

# Size of structure/class:

* When we declared structure,
* Class plot
* {

Int wid;

Int len;

Double area;

Void get() {

}

* }
* Compiler maintain below table to allocate size in memory.

|  |  |  |
| --- | --- | --- |
| * Member | * Type | * Offset |
| * Wid | * Int | * 0 |
| * Len | * Int | * 4 |
| * Area | * Double | * 8 |
| * Total size = 16 | | |

* When we create object: plot p1;
* It allocates 16-byte memory size in memory.

# Reusability (Inheritance):

* We can use the base class properties to inherited class.

***Types of inheritance:***

* Public Inheritance
* Private Inheritance
* Protected Inheritance

**Protected:**

* When we define member with protected type its only allowed to access in derived class.

***Inheritance levels:***

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hybrid Inheritance

# ***Virtual Function:***

* Every class has virtual table which holds the address of virtual pointer.
* When we defined a virtual function in class and create an object of that class its contain extra size (4 byte) which holds the address of the virtual function and called as virtual pointer (VPTR):

|  |  |  |
| --- | --- | --- |
| * VPTR | * Data-1 | * Data-2 |

* Virtual table is structure type maintain by compiler and it is a collection of all virtual table.
* Virtual pointer holds the address of virtual table which contain all virtual function.
* If we have multiple derived class and we are not sure which derived class object is going to select during run time, we have to use base class pointer which can holds the address of different derived class objects.
* When we call the method of derived class through base class pointer and if we haven’t defined same method as virtual function than it will holds only data member at memory location (No VPTR), so it is going to call the method of base class only.
* In reverse, when we call the method of derived class through base class and if we have defined same method as virtual function, compiler holds first byte as VPTR which will holds the address of virtual table (collection of virtual function) from where we get the address of base class function.

Diagram

Description automatically generated with low confidence

***Virtual keyword uses:***

1. To define virtual function.
2. Pure virtual function => To avoid object creation.
3. Virtual destructor => To call the base class destructor when we delete the derived class object to avoid memory leaking.
4. Multi-level Inheritance => To access the base class member through sub child object.

***Pure virtual function:***

* It is used to avoid object creation for that function.
* Pure virtual function needs to be override in base class
* If we call pure virtual function through derived class object without creating override function it will give compile time error.

***Abstract class:***

* The class that contains pure virtual function as method is called Abstract class
* To avoid creation of object.
* We must Inherited that Abstract class.

***Upcasting:***

* Base class pointer holding derived class object is called as upcasting.
* Derived class pointer can not hold the address of base class object.

***Virtual Destructor:***

* To call the base class destructor when we delete the derived class object to which we have assigned dynamic memory using new operator
* To avoid memory leaking: if we delete the derived class object using ‘delete’ keyword it will only call the derived class destructor and will not call the base class destructor so it will lead to memory leaking.

***DAY-05:***

***Multi-level Inheritance:***

* To access the base class member through sub child object which is not override in derived class.
* For ex: class Nexon: virtual public Tata
* If we haven’t mention virtual than we can not access the base class (Tata) member/method.

Diagram

Description automatically generated

# Override Keyword:

* Its use to inform compiler about overriding existing method(feature) not new method(feature).
* If we by mistake change the prototype of override function, compiler will generate the compilation error.
* For ex: base class has virtual function: virtual void fun()
* And we try to override it in derived class : void fun(int) override
* It will generate error.
* This feature is to avoid mistake
* To use the “override” key word base class function must have virtual function and it has to be override in derived class and must have same prototype in derived class.

***NOTES for virtual:***

***Destructor:***

* Any resource/memory we acquired in constructor must needs to release in destructor to avoid memory leaking.
* Destructor cannot be override.
* We can not modify the destructor by changing arguments or something else modification.
* Destructor is using “this” pointer to release resources/memory.

# Type casting Mechanism:

* Casting – forced type conversion

***C style typecasting:***

* A = (type)b;
* Int I = 65;
* Float a;
* A = (float) I \*5;

***Drawbacks of C type casting:***

* Tracing is difficult.
* All type cast looks different.

# Casting in C++:

1. Static\_cast => conversion take place at compile time.
2. Const\_cast =>
3. Dynamic\_cast => run time conversion take place
4. A
5. “static\_cast”:
   * For ex: int I = 50;

I = static\_cast <long>(i);

1. “const\_cast”:
   * For ex: const int a =100;
   * Int \*ptr =(int\*) &a; // one way
   * Ptr = const\_cast <int \*> (&a); // cpp use, we can search it.
2. ***“dynamic\_cast”:***

* N = dynamic\_cast <nexon \*> (p);
* This will compare address of derived class (for ex: nexon) with the address p contain
* If p is holding the address of “nexon” class it assign that address to “n” else it will return “null”.
* Base class must have the virtual function.
* As compiler is comparing the passed address (address p holds ) with virtual table address (its unique for all class), virtual function is needed.
* P hold the address of object of any derived class derived from same base class;

1. ***“reinterprrte\_cast”: -***

* Int \*iopo;
* Iopo = 1000; // not allowed in cpp
* Iopo = (int\*) 1000; // one way
* Iopo = reinterpret\_cast <int \*> (1000);

# Error Handling:

***Types of Error:***

* Syntax
* Linker
* Run time
* Pre-processor

Runtime error Handling:

* Run time error in C++ are …

Exception:

* Some runtime error that can be caught through some appropriate method
* A mechanism to handle error.

# Exception Handling:

* Try => block
* Catch => block
* Throw => statement

***Try:***

* It must be followed by catch block else it will give compile time error.
* Try{
* }

***Catch:***

* We can catch the exception for other than int, double, char or object using catch(…) function.

**Day-05:**

# ***Shallow copy:***

* It will create copy of the existing object and that can create exception if we have allocated dynamic memory in the constructor.
* So it is copying the same address value in newly created object.
* For ex: PLOT p4(p3);
* When class has pointer as data member, the built-in copy constructor and overloaded assignment operator method copies the pointer and not the data to which it is pointing hence, if one object modifies the data, it is reflected in another object.
* So when we use “delete” operator to release memory in destructor it will throw run time error as for both object its trying to release same allocated memory.
* So here we have allotted dynamic memory for member of p4 and same address is copied into p3 as well, so both object member are pointing to same memory location.
* So now when object loose its scope its call destructor where we are deleting the dynamically allotted memory for the last created object (p3)
* And after that p4 destructor is also called where again complier trying to release the dynamically created memory, but that is already released by p3 so we are getting exception.
* To avoid this scenario we must use “deep copy” : overloading copy constructor.

Diagram

Description automatically generated

# Deep copy:

* It a method to overcome shallow copy runtime error.
* In this we are using overloaded copy constructor to allocate different memory for new object creation using assignment operator or PLOT p4(p3).
* It is useful for the scenario where we are using overload assignment operator or creating new object by copying from existing one and we are allotting dynamic memory in constructor.

Diagram

Description automatically generated

# Namespace:

* It is a collection of classes.
* Scope resolution (::) is used to use the object of particular namespace class.
* We can us :: to access global variable inside block when same name variable is defined in that block.

# Fille system:

* Data blocks: stores data.
* Inode blocks: its store permission, file system information etc…
* Super blocks: How many inode & data block are there in system.
* Boot block: booting information.

1. Ofstream => output file stream class.
2. Ifstream => input file stream class.

Diagram

Description automatically generated

# Threads:

A picture containing shape

Description automatically generated

# Templates:

* Templates are used to write generic programs.
* Generic to data type.
* Writing a code independent of data type.

***Two types of templates:***

1. Class template
2. Function template.

***Function Template:***

* Writing a generic function.
* By seeing template keyword, during compilation time, compiler will create overloading function definition based on function call (function insertion) defined in the program.
* For ex: Template <calss T>
* T fun(T x, T y)
* {
  + T r;
  + R = x+y;
  + Return R;
* }
* Here if we have call fun with fun(int, int) & fun(float, float) compiler will create two overload function based on function call.
* If float fun(float, float) {} (specialized function) is already define by user compiler will not create another overloading function with float arguments during compile time so while executing it will call only specialized function and not the template function.

***Class Template:***

* Template <typename T>
* Class abc
* {
  + T wid;
  + T len;

Public:

Void set(T w, T l) { }

Void get(){ }

* }
* To write generic class.