OOPs

1 Introduction

Class

It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Object

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Object take up space in memory and have an associated address in memory.

Encapsulation

Encapsulation is defined as binding together the data and the functions that manipulate them.

Encapsulation also leads to data abstraction or hiding. As using encapsulation also hides the data.

Abstraction

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

- **Using Classes**: The class helps us to group data members and member functions using available access specifiers.
- **Using Header files**: consider the *pow()* method present in *math.h* header file.

Polymorphism

Polymorphism is the ability of data to be processed in more than one form. It allows the performance of the same task in various ways. It includes function and operator overloading.

Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance.

Inheritance supports the concept of “reusability”, i.e. when we want to create a new class, we can derive it from an existing class.

2 Constructor

Constructors are special member functions which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.

2.1 Default constructor

It takes no arguments. Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2.2 Parameterized constructor

It is used to initialize the various data elements of different objects with different values when they are created. Constructor overloading is possible.

Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case.

2.3 Copy constructor

A copy constructor is a member function which initializes an object using another object of the same class.

In C++, a Copy Constructor may be called in following cases:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- When the compiler generates a temporary object.

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection..etc.

Default constructor does only shallow copy. Deep copy is possible only with user defined copy constructor. In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.

When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources.

A copy constructor is called when an object is passed by value. So if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value.

3 Virtual Function

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. Virtual functions are so useful that later languages like Java keep all methods as virtual by default.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

- A class is abstract if it has at least one pure virtual function.
- We can have pointers and references of abstract class type.
- If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
- An abstract class can have constructors.
- It is not possible to create the object of a class which contains a pure virtual method.

Pure virtual destructors are legal in standard C++ and one of the most important things to remember is that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor. The reason is because destructors (unlike other functions) are not actually 'overridden', rather they are always called in the reverse order of the class derivation i.e a derived class' destructor will be invoked followed by the base class destructor. The compiler and linker enforce the existence of a function body for pure virtual destructors.

4 Abstract Class

An **interface** describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class. The C++ interfaces are implemented using **abstract classes**.

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0".

The purpose of an abstract class is to provide an appropriate base class from which other classes can inherit.

- Abstract classes cannot be used to instantiate objects and serves only as an interface.
- If a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions.

Classes that can be used to instantiate objects are called **concrete classes**.

5 Multiple Inheritance and Diamond Problem

Java does not allow multiple inheritance.

The diamond Problem

6 Dangling, Void, Null and Wild Pointers

A pointer is a variable that stores the memory address of an object. Pointers are used extensively in both C and C++ for three main purposes:

- to allocate new objects on the heap,
- to pass functions to other functions
- to iterate over elements in arrays or other data structures.

Modern C++ provides smart pointers for allocating objects, iterators for traversing data structures, and lambda expressions for passing functions.

Dangling pointer

Dangling pointer is a pointer pointing to a memory location that has been deleted or freed.

Example. They are created by:

- De-allocation of memory without appropriate handling of pointer reference.
- Pointer pointing to a variable outside the scope of the variable.
- Pointer pointing to a local variable in a function call becomes a dangling reference outside the scope of function.

Void pointer

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. The type of data that it points to can be any.

- Void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer. Example: **((int*) ptr))* deferences a type-casted void pointer *ptr*.
- **Pointer arithmetic is not possible** on them due to lack of concrete value and thus size.

Null pointer

Null Pointer is a pointer which is not assigned any address and is pointing to nothing. It stores a defined value that is defined by the environment to not be a valid address for any member or object. An uninitialized pointer, however, stores an undefined value.

Wild pointer

A pointer which has not been initialized to anything (not even NULL) is known as **wild pointer**. The pointer may be initialized to a non-NULL garbage value that may not be a valid address. Example.

7 Smart Pointer

Pointers are used for accessing the resources which are external to the program like heap memory. The problem with heap memory is that you must deallocate it after use. Forgetting the deallocation of objects and pointers causes memory leak.

- Smart pointer is a class template that is declared on the stack and initialized by using a raw pointer that points to a heap-allocated object.
- The smart pointer is responsible to delete the memory that the raw pointer specifies. Its destructor contains the deletion call and is invoked when the smart pointer goes outside its scope.
- The smart pointer class overloads * and → operators to return encapsulated raw pointer.
- Languages like Java, .Net Framework provide a garbage collection mechanism. In C++ 11, smart pointers are introduced that automatically manage memory and deallocate objects when not in use.

auto_ptr

The auto_ptr template class describes an object that stores a pointer to a single allocated object that ensures that the object to which it points gets destroyed automatically when control leaves a scope.

unique_ptr

A unique_ptr explicitly prevents copying of its contained pointer (as would happen with normal assignment), but the std::move function can be used to transfer ownership of the contained pointer to another unique_ptr. A unique_ptr cannot be copied because its copy constructor and assignment operators are explicitly deleted.

shared_ptr

It maintains reference counting ownership of its contained pointer in cooperation with all copies of the shared_ptr whose count can be accessed using the *use_count()* method. An object referenced by the contained raw pointer will be destroyed when and only when all copies of the shared_ptr have been destroyed.

Its stores 2 pointers and takes more memory. Unique and weak pointers store only pointer. However, they may have multiple functions defined in class that they can invoke.

weak_ptr

It similar to shared_ptr but does not maintain a reference counter. It is created as a copy of a shared_ptr. The existence or destruction of weak_ptr copies of a shared_ptr have no effect on the shared_ptr or its other copies. After all copies of a shared_ptr have been destroyed, all weak_ptr copies become empty. The pointer will not have a strong hold on the object to avoid **deadlocks**.

Because the implementation of shared_ptr uses reference counting, circular references are potentially a problem. A circular shared_ptr chain can be broken by changing the code so that one of the references is a weak_ptr.

8 Function Overloading

The functions that cannot be overloaded are:

- If they only differ in return datatype.
- Declarations with the same name and the name parameter-type-list.
- Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types.
  *int fun(int *ptr);*
  int fun(int ptr[]);
- Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the *const* and *volatile* type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. Example
- Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.
  void h(int ());
  void h(int (*)());
- Two parameter declarations that differ only in their default arguments are equivalent.
  void h(int x);
  void h(int x=1;);

9 Name Mangling

Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language. Type names may also be mangled. Name mangling is commonly used to facilitate the overloading feature and visibility within different scopes.

- Name mangling is not desirable when linking C modules with libraries or object files compiled with a C++ compiler. To prevent the C++ compiler from mangling the name of a function, you can apply the extern "C" linkage specifier to the declarations.
- The extern "C" linkage specifier can also be used to prevent mangling of functions that are defined in C++ so that they can be called from C.
- In multiple levels of nested extern declarations, the innermost extern specification prevails.
- It occurs for all the members, even those that are not overloaded. However, it is used only if an instance of it is used. Otherwise, compiler will optimize it and not perform it.

The *main()* function is not mangled as it is the entry-point of the program and the OS needs to identify this. For this reason, the *main()* function cannot be overloaded.Video Example.

10 Inline functions

All the member functions defined inside the class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

Through inline function, the class's data members can be accessed.

Macro

It is also called preprocessors directive. The macros are defined by the #define keyword. Before the program compilation, the preprocessor replaces the macros with its definition.

11 Friend function

A friend class can access private and protected members of other class in which it is declared as friend. Friendship is not inherited.

12 Static Function and Keywords

They are called by using the class name.

Static members do not formally belong to an object; they exist before the object was created, when the program is compiled. In runtime, when you instantiate a class, only instance members are created. Each object has a copy of its instance variables and they all share the single copy of static members.

Static Variables in a function

When a variable is declared as static, space for it gets allocated for the lifetime of the program. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call.

The scope of the variable, however, remains the same. Only the lifetime is extended.

Static variables in a class

As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables in a class are shared by the objects. They are initialized as

int GfG :: i=0;

- There can not be multiple copies of same static variables for different objects.
- The static variables can not be initialized using constructors.

Static Class Objects

Just like variables, objects also when declared as static have a scope till the lifetime of program.

Static functions in a class

Just like the static data members or static variables inside the class, static member functions also does not depend on object of class.

- We are allowed to invoke a static member function, similar to static variables, using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator.
- Static member functions are allowed to access only the static data members or other static member functions.

13 Object Slicing

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

- Object slicing doesn't occur when pointers or references to objects are passed as function arguments since a pointer or reference of any type takes same amount of memory.
- Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

14 Templates

The idea is to pass data type as a parameter so that we don't need to write the same code for different data types. Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. Example.

- Multiple arguments can be passed to the templates and default parameters can be used too.
- Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations.
- Each instance of a template contains its own static variable.
- We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, they must be const.

Using templates, we can write programs that do computation at compile time, such programs are called template metaprograms. Example.

Template specialization

It is possible in C++ to get a special behavior for a particular data type. This is called template specialization. Example.

When we write any template based function or class, compiler creates a copy of it whenever it sees that being used for a new data type(s). If a specialized version is present, compiler first checks with it. It does so by checking with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

15 Memory Allocation in C++

Text segment

A **text segment** ,also known as a **code segment** or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

- As a memory region, a text segment may be placed below the heap and stack in order to prevent heaps and stack overflows from overwriting it.
- It is usually shareable among many processes and only one copy is kept in memory.
- It is often read-only to prevent a process from modifying its instructions.

Initialized data segment

Data segment contains the global and static variables initialized by the programmer. It can be classified into read-only and read-write area.

```
static int i;
...
...
i=1;
```
How is this assigned in the memory layout?

Uninitialized data

Data in the **bss segment** is initialized by the kernel to arithmetic 0 before the program starts executing. It starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

Heap

Heap is the segment where dynamic memory allocation usually takes place. The heap area begins at the end of the BSS segment and grows to larger addresses from there.

- The name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate.
- It is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size. If a programmer does not handle this memory well, memory leaks can occur.
- The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Stack

The stack area traditionally adjoined the heap area and grew the opposite direction and exhausted when the two pointers met. With virtual memory, they may be placed anywhere, but grow opposite directions.

The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated.

- A "stack pointer" register tracks the top of the stack and is adjusted for every "pushed" value.

- The set of values pushed for one function call is termed a "stack frame". A stack frame consists, at minimum, of a return address.
- Memory allocation and deallocation of stack variables happens using some predefined routines in compiler. Programmer does not have to worry about it.

Stack is where automatic variables are stored along with information that is saved each time a function is called. On a function call, its return address and information about the caller function is saved on the stack. The function then allocates memory for its own temporary and automatic variables.

Each time a recursive function calls itself, a new stack frame is pushed on to the stack so the variables dont interfere.

16 Namespace

Using namespaces, we can create two variables or member functions having the same name. They allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names. Example.

- It is possible to create two namespace blocks having the same name. The second namespace block is nothing but actually the continuation of the first namespace. However, we can not use (or redeclare) a variable/ function defined in one namespace in the other namespace block.
- We can access namespace members using scope resolution or the *'using'* directive. It is also possible to create a namespace in one file and access it in another by importing the file.
- It possible to have namespaces nested. Its resolution is hierarchical.
- You can use an alias name for your namespace name, for ease of use.

Unnamed Namespace

In unnamed namespaces, name of the namespace in not mentioned in the declaration of namespace. The name of the namespace is uniquely generated by the compiler.

- They are directly usable in the same program and are used for declaring unique identifiers.
- Unnamed namespaces are the replacement for the static declaration of variables.

17 Access Modifiers

Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. The default the access modifier for the members will be **Private**.

- **Public** The data members and member functions declared public can be accessed by other classes too using the access operator(.).
- **Protected** The class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.
- **Private** They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the **friend** functions are allowed to access the private data members of a class.

18 Scope resolution

The scope of an identifier for a variable, reference or method is the portion of the program in which the identifier can be referenced.

- Methods and instance variables of a class have class scope. Class scope begins at the opening left brace of the class definition and terminates at the closing right brace, of the class definition. Class scope enables methods of a class to blockquoteectly invoke all methods defined in that same class and to blockquoteectly access all instance variables defined in the class.
- Identifiers declared inside a block have block scope. Block scope begins at the identifier's declaration and ends at the terminating right brace of the block. Local variables of a method have block scope as do method parameters, which are also local variables of the method.

To define a member function of a class outside the class definition we have to use the scope resolution :: operator along with class name and function name.

19 Typecasting

In type casting (**narrowing conversion**), a data type is converted into another data type by a programmer using casting operator. In type conversion (**widening conversion**), a data type is converted by a compiler.

- Upcasting can never fail. But if you have a group of different objects (Animals) and want to downcast them all to a another object (Cat), then there's a chance, that some of these objects (Dogs) are not compatible, and process fails, by throwing an exception. Example.

static cast

This is the simplest type of cast which can be used. It is a compile time cast. Example.

dynamic cast

This is similar to static cast but is executed at runtime instead. It is primarily used to downcast from a base class pointer to derived class pointer. If the casting has been completed, it returns the pointer to it, else return NULL. Example1, Example2.

When we try to dynamically cast a base pointer pointing to a base class into a Derived pointer, it will fail, because the conversion can't be made.

const cast

It is used to cast away the constness of variables. Example.

- It can be used to change non-const class members inside a const member function. Declaring a function const, makes all its data members const.
- It can be used to pass const data to a function that doesn't receive const.

- It is undefined to modify a value which has been declared const. The casting is applied to constant pointers to non-const variables.
- It'safer than simple cast as the casting won't happen if the type of cast is not same as original object.
- It can also be used to cast away volatile attribute.

reinterpret cast

It is used to convert one pointer of another pointer of any type, no matter either the class is related to each other or not. It does not check if the pointer type and data pointed by the pointer is same or not. Example.

- It is a very special and dangerous type of casting operator. And is suggested to use it using proper data type i.e., (pointer data type should be same as original data type).
- It can typecast any pointer to any other data type.
- It is used when we want to work with bits.
- Boolean value will be converted into integer value.

20 RAII

Resource Acquisition Is Initialization or RAII binds the life cycle of a resource that must be acquired before use to the lifetime of an object. Another name for this technique is Scope-Bound Resource Management.

The main goal of this idiom is to ensure that resource acquisition occurs at the same time that the object is initialized, so that all resources for the object are created and made ready in one line of code. In practical terms, the main principle of RAII is to give ownership of any heap-allocated resource—for example, dynamically-allocated memory or system object handles—to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated cleanup code.

- encapsulate each resource into a class, where the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done, and the destructor releases the resource and never throws exceptions.
- always use the resource via an instance of a RAII-class that either has automatic storage duration or temporary lifetime itself, or has lifetime that is bounded by the lifetime of an automatic or temporary object

21 To-add

21.1 Dynamic binding

In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.