C++

Class Basics

Defining a class

```
class ClassName {

};
```

Access modifiers:

Access Modifiers allows programmer to avhieve **Encapsulation**

Encapsulation is to make concrete object abstract by hiding the internal functionality of the class and only provide interface to interact with the class. This can be achieve my defining access modifiers to specify which methods or variables are accessible to which classes or functions.

1. Public:

public methods are callable by any subclasses and any functions outside of the class

2. Protected:

protected methods are callable by any subclasses but not any functions outside of the class

3. Private:

private methods are only callable by itself, any subclasses and functions outside of the class cannot access it

```
class ClassName {
    public:
        void publicMethod() {};
    protected:
        void protectedMethod() {};
    private:
        void privateMethod() {};
};
```

Class data members:

ALWAYS make data members private

```
class ClassName {
    public:
```

```
        void publicMethod() {};
    protected:
        void protectedMethod() {};
    private:
        int privateVar1;
        int privateVar2;
};
```

Constructor and Destructor

Constructor are called when we initialize a instance of a class

Types of constructors:

1. **Default Constructor**: constructor with no arguments
2. **Parameterized Constructor**: constructor with parameters
3. **Copy Constructor**: constructor called when object of the same type is passed in as parameter. The object data fields are copied by default
4. **Move Constructor**: constructor called when `std::move`(object) used in the constructor argument. The data of the other object is transferred to the current one
5. **Explicit Constructor**: used to avoid implicit call to the constructor.

Destructor: Desctructor is called when we called `delete` on the current instance for dynamically allocated variables or when the stack instance is out of scope

```
class ClassName {
    public:
        // 1. Default Constructor
        ClassName();
        // 2. Parameterized Constructor
        ClassName(int a, int b);
        // 3. copy Constructor
        ClassName(const ClassName& other);
        // 4. move Constructor
        ClassName(ClassName&& other);
        // 5. explicit Constructor
        explicit className(double b);
        // 6. Desctructor
        ~ClassName();
        // MUST ADD virtual keyword when this class will be inherited
        virtual ~ClassName();
};
```

Explanation of 3, 4, 5:

The copy constructor is called in the following scenario:

```
int main() {
    ClassName A;
```

```
    // explicitly copy with ClassName(A),
    // B will copy all data fields of A
    ClassName B(A);
    // implicit copy with = symbol
    // C will copy all data fields of A
    ClassName* C = A;
}
```

The Move constructor is called in the following scenario:

```
int main() {
    ClassName A;
    ClassName B = std::move(A);
}
```

Without the explicit keyword, we can initialize a class by

```
int main() {
    ClassName A = 2.0;
}
```

With the explicit keyword, the above syntax will throw an error, we can only initialize the class with

```
int main() {
    ClassName A(2.0);
}
```

Request and Suppress methods

If we do not define the following functions, the compiler will generate one for us automatically:

1. Default Constructor: If no parameterized constructors are defined, the compiler will generate a default constructor that calls any default constructor of the member variables and left the primitive data uninitialized
2. Copy Constructor: If no copy constructor is defined, the compiler will generate a copy constructor that copies all data field
3. Desctuctor: If no destructor is defined, the compiler will genertate a destructor that destorys all data field

We can specify whether compiler should or should not generate a default methods using `=delete` or `=default` specifier.

```
class ClassName {
    public:
```

```
        // 1. generate a default Constructor
        ClassName() = default;
        // 2. generate a default destructor
        ~ClassName() = default;
        // 3. disallow copy constructor
        ClassName(const ClassName& other) = delete;
};
```

We can also use this property to specify whether an object can be created on heap or on stack\

1. If we don't want the object to be created on heap, use `=delete` on the `new` operator
2. If we don't want the object to be created on stack, use `=delete` on the destructor

```
class NonHeap {
    public:
        void* operator new(std::size_t)= delete;
};

class NonStack {
    public:
        ~NonStack()= delete;
};
```

initializer list:

we can use initializer list to initialize datamembers or parent objects or even call the current classes' constructor

```
class Child : public Parent {
    public:
       Child(): a(1), b(0), Parent(3) {}
    private:
        int a;
        int b;
};
```

Friend Function

If a class specifies that a function is a friend, it allows that function to access its private variables

```
class FeindClass {
    public:

    private:
        int a;
        int b;
```

```
    friend void befriendedFunction(FriendClass& f);
};

void befriendedFunction(FriendClass& f) {
    // this is ok
    std::cout << f.a << f.b << std::endl;
}

int main() {
    FriendClass f;
    befriendedFunction(f);
}
```

Inheritance

Inheritance allows objects to obtain properties of another object without rewriting the same funtionality and allow expanding new funtionality

When B inherits from A, B can

1. Access B's public field
2. Access B's protected field
3. CANNOT access B's private field

When B inherits from A, there are three modes of inheritance:

1. **public** inheritnace: A's public field is still public in B, and A's protected field is still protected in B
2. **protected** inheritnace: A's public field is protected in B, and A's protected field is still protected in B
3. **private** inheritnace: Both A's public and protected fiels are private in B

```
class A {

};

// public inheritance
class B : public A {

};

// protected inheritance
class B : protected A {

};

// private inheritance
class B : private A {

};
```

Function overloading

When a non-virtual function is declared in the base class, the derived class can override it, however, Overriding a function in a derived class hides all the overloads of the same function from the base class.

```
class A {
    public:
        void call() {std::cout << "A";};
        void call(int a) {std::cout << "A: " << a;};
};


class B : public A {
    public:
        void call() {std::cout << "B";};
};

int main() {
    A a;
    B b;
    a.call();    // OK
    a.call(3);       // OK
    b.call();    // OK
    b.call(3);  // NOT ALLOWED, overriding void call() in B hides void
call(int a) in A
}
```

Polymorphism

Polymorphism allows differentiating the behavior of different classes while having the same interface

Virtual Function

virtual functions allows derived classes to reimplement different functionality with the same function prototype

Virtual functions are achieved by creating a virtual method table. When class B inherits from class A which contains virtual functions

1. B will create a virtual method table that copies all methods pointers from A
2. B will add any newly defined method in the current class
3. B will check if any virtual method is overriden in the current class, if so, replace the corresponding entry of the virtual method table to the new definition

virtual function allows us to use a base reference or base pointer to call the virtual method of a derived class.

Pointing a derived class's pointer to its base class object is **NOT** allowed

```
class A {
    public:
```

```
        virtual void print() {std::cout << "I'm A";};
        virtual ~A() = default;
};

class B : public A {
    public:
        virtual void print() {std::cout << "I'm B";};
        virtual ~B() = default;

};

int main() {
    B b;
    A a;
    b.print(); // OUTPUT: I'm B
    a.print(); // OUTPUT: I'm A
    A* bptr = &b;
    bptr->print(); // OUTPUT: I'm B
    B* aptr  = &a; // ERROR: NOT ALLOWED
}
```

ALWAYS make destructor virtual if a class needs to be inherited. This is because when we have a base class pointer that points to a derived class, when we call `delete` on the derived class, if the destructor is not virtual, it will call the base classes' destructor and thus will not properly release the memory of the derived class.

Pure Virtual Function

Pure Virtual Function are virtual functions that has no definition and must be implemented for any class derived from it. A class with at least one pure virtual function is called an abstract class. Abstract classes cannot have instances

```
class Abstract{
    public:
        // define a pure virtual function
        virtual void mustBeOverriden() = 0;
        // do not forget to define a virtual destructor
        virtual ~Abstract() = default;
};

// SillAbstract does not override the pure virtual function in Abstract,
it is still a abstract class
class SillAbstract : public Abstract {
    public:
        void someFunctionality() {};
        virtual ~StillAbstract() = default;
};


// Concrete has overriden the pure virtual function, it can be instantiate
class Concrete : public StillAbstract{
```

```
    public:
        // overriding the pure virtual function
        virtual void mustBeOverriden() {};
        virtual ~Concrete() = default;
};

int main() {
    Concrete c; // OK
    c.mustBeOverriden(); // OK
    Abstract* ab = &c;
    StillAbstract sa = &c;
    ab->mustBeOverriden(); // OK
    sa->someFunctionality(); // OK
    Abstract a; // ERROR, cannot intantiate an abstract class
    StillAbstract s; // ERROR, cannot intantiate an abstract class
}
```

STL

STL contains common extended funtionalities of C++

Algorithm

`std::min` `std::max` and `std::minmax`

These function computes the min or max from a pair of values or a list of values.

```
#include<algorithm>

bool cmp(int a, int b) {
    return a < b;
}
int main() {
    int a = 1;
    int b = 2;
    std::min(a, b); // returns the min of a and b
    std::min(a, b, cmp); // return the min with cmp comparator
    std::min({1,2,3,4,5}); // return the min of elements in a initializer
list
    std::min({1,2,3,4,5}, cmp); // return the min of elements in a
initializer list with cmp comparator


    // similar syntax for max and minmax, notice that
    // std::minmax returns a pair p
    // where p.first is the min
    // and p.second is the max
    std::max(a, b);
    std::minmax(a, b);
}
```

Utility

`std::move`

The function moves the current resource to a new reference, it is usually cheaper than copying

```
#include<utility>

int main() {
    std::string str1 = "abcd";
    std::string str2 = "efgh";
    std::cout << "str1: " << str1 << std::endl;
    std::cout << "str2: " << str2 << "\n\n";
    // OUTPUT:
    // str1: abcd
    // str2: efgh

    // Copying
    str2 = str1;
    std::cout << "After copying" << std::endl;
    std::cout << "str1: " << str1 << std::endl;
    std::cout << "str2: " << str2 << "\n\n";
    // OUTPUT:
    // After copying
    // str1: abcd
    // str2: abcd

    str1 = "abcd";
    str2 = "efgh";

    // Moving
    str2 = std::move(str1);
    std::cout << "After moving" << std::endl;
    std::cout << "str1: " << str1 << std::endl;
    std::cout << "str2: " << str2 << "\n\n";
    // OUTPUT:
    // After moving:
    // str1:
    // str2: abcd
}
```

To allow moving a class, the class must have a move operator defined

```
class Movable {
    public:
        // must not be deleted
        MyData& operator = (MyData&& m) = default;
};
```

`std::pair` and `std::tuple`

Pair stores pair of elements

```
int main() {
    // declare a pair
    std::pair<int, double> p = std::make_pair(5, 3.14);
    // accesing the first element
    p.first;
    std::get<0>(p);
    // accessing the second element
    p.second;
    std::get<1>(p);
}
```

Tuple generalizes pair and can store arbitrary number of elements

```
int main() {
    // declare a tuple
    std::tuple<int, double, char> t = std::make_tuple(5, 3.14, 'C');
    // accesing the n-th element
    std::get<n>(t);
}
```

Containers,

`std::array` (C++11)

array is fixed size and supports random access at O(1)

```
#include <array>
int main() {
    // Initialize an array of size n
    std::array<int, n> arr;
    // Initialize an array with initializer list
    std::array<int, 3> arr = {1,2,3};
    // access the i-th element
    arr[i];
    // retrieve the size of array
    arr.size();

}
```

`std::vector`

vector is a dynamic size array with O(1) random access, O(1) push back and O(1) pop back

```cpp
#include <vector>
int main() {
    // initialize an empty vector
    std::vector<int> v;
    // initialize an vector of size n and default value d
    std::vector<int> v(n, d);
    // initialize with specific elements
    std::vector<int> v{1,2,3};
    // retrieve the size of the vector
    v.size();
    // push an element to the end
    v.push_back(1);
    // pop an element from the back
    v.pop_back();
    // access the n-th in the vevtor
    v[n];
    // insert an element at the i-th position O(n)!!!
    v.insert(v.begin()+i, 3);
    // erase an element at the i-th position O(n)!!!
    v.erase(v.begin()+i);
    // reserve n capacity for an vector
    v.reserve(n);
    // resize the array to size n, all elements after n are deleted
    v.resize(n);
    // remove all elements from the vector
    v.clear();
}
```

std::stack

Stack supports LIFO insertion and deletion in O(1)

```cpp
#include <stack>
int main() {
    // initialize an empty stack
    std::stack<int> s;
    // retrieve the size of the stack
    s.size();
    // push an element to the stack top
    s.push(3);
    // pop an element to from the stack top
    s.pop();
    // retrieve the top element
    s..top();
}
```

std::deque

Deque supports push and pop from the begin or end in O(1) time and random access in O(1)

```
#include <deque>
int main() {
    // initialize an deque
    std::deque<int> d;
    // initialize an deque with size n and default element d
    std::deque<int> d(n, d);
    // retrieve the size of the dque
    d.size();
    // push to the back
    d.push_back(5);
    // pop from the back
    d.pop_back();
    // push to the front
    d.push_front();
    // pop from the front
    d.pop_front();
}
```

std::list

List is doubly linked in C++ that supports O(1) insertion and deletion from both end, but random access takes O(n)

```
#include<list>
int main() {
    // initializing an empty list
    std::list<int> l;
    // initializing an list with size n and default value d
    std::list<int> l(n,d);
    // push an element to the back
    l.push_back(3);
    // pop and element from the back
    l.pop_back();
    // push an element to the front
    l.push_front(3);
    // pop an element from the front
    l.pop_front();
    // remove elements with specific value
    l.remove(2);
    // remove elements with criteria
    l.remove_if(predicate_fn);
    // sort the list
    l.sort();
    // remove duplicate elements in a sorted list
    l.unique();
    // merge sorted lists l1 and l2 to l1
    l1.merge(l2);
}
```

std::priority_queue

Priority Queue implements a heap data structure where the top is the maximum element. The insertion takes O(logn) and deletion takes O(logn). Retrieving the maximum element takes O(1) time

```
#include<queue>
struct mycomp {
    bool operator(const int& lhs, const int&rhs) const {
        return lhs < rhs;
    }
};

int main() {
    // initialize an max heap
    std::priority_queue<int, std::vector<int>, std::less<int>> pq;
    // initialize an min heap
    std::priority_queue<int, std::vector<int>, std::greater<int>> pq;
    // initialize an heap with custom comparison
    std::priority_queue<int, std::vector<int>, mycomp> pq;
    // push to the priority queue
    pq.push(5);
    // look at the top element
    pq.top();
    // pop the top element
    pq.pop();
}
```

std::set, std::multiset

std::map, std::multimap

std::unordered_map, std::unordered_set