DAY 02

Ad-hoc polymorphism, operator overloading and Orthodox Canonical class form

Orthodox Canonical class form :

- ### Default constructor :

Default constructor is a constructor that is called automatically by the compiler when an object is created, if no other constructor is defined for that class. The default constructor has no parameters and initializes all non-static data members of the class with default values.

- ### Copy constructor :

Copy constructor is a special constructor that is used to create a new object as a copy of an existing object of the same class. The copy constructor takes a reference to an object of the same class as its argument, and creates a new object that is a copy of the original object.

```
class MyClass {
public:
   MyClass(const MyClass& other) {
      member1 = other.member1;
      member2 = other.member2;
   }
private:
   int member1;
   std::string member2;
};
```

```
MyClass obj1;               // ... initialize obj1 ...
MyClass obj2 = obj1;        // Copy constructor called automatically
MyClass obj3(obj1);         // Copy constructor called automatically
```

Copy constructors take the argument as a reference to an object of the same class in order to avoid unnecessary copying of the object being passed.

By taking the argument as a const reference, the copy constructor ensures that the original object is not modified during the copy process. This is particularly useful when working with large or complex objects, as it can help to avoid unnecessary copies and improve performance.

- ### Copy assignment operator :

The copy assignment operator in C++ is used to assign one object of a class to another object of the same class. It is invoked when the assignment operator (=) is used with two objects of the same class. The copy assignment operator is responsible for copying the data members of the source object to the target object.

```
class MyClass {
public:
MyClass& operator=(const MyClass& other) {
      member1 = other.member1;
      member2 = other.member2;
      return *this;

   }
private:
   int member1;
   std::string member2;
};
```

```
MyClass obj1;                    // default constructor
MyClass obj2;                    // default constructor
obj2 = obj1;                     // copy assignment operator

MyClass obj1;                    // default constructor
MyClass obj2;                    // default constructor
MyClass obj3;                    // default constructor
obj3 = obj2 = obj1;              // chained copy assignment operator
```

Returning a reference and *this from copy assignment to the target object from the copy assignment operator, we can ensure that the operator can be used in chained assignments .

- ### Destructor :

Destructor is a special member function that is called automatically when an object is destroyed. The purpose of a destructor is to release any resources that were acquired by the object during its lifetime, such as memory allocated using new, file handles, network connections, etc.

A destructor has the same name as the class, but with a tilde (~) character before it.

Ad-hoc polymorphism :

In ad-hoc polymorphism, a single function or operator can have multiple implementations, each designed to work with a specific set of argument types or numbers. The compiler determines which implementation to use based on the type and number of arguments passed to the function or operator.

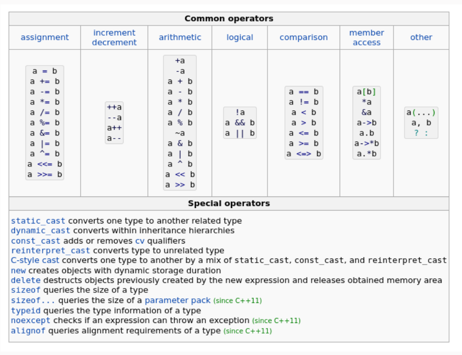
```
// function to add two integers
int add(int a, int b) {
   return a + b;
}

// function to concatenate two strings
std::string add(std::string a, std::string b) {
   return a + b;
}
```

```
int main() {
   int result1 = add(1, 2);
   std::string result2 = add("Hello, ", "world!");
   std::cout << result1 << std::endl;
   std::cout << result2 << std::endl;
   return 0;
}
```

Operator overload :

- Operator overloading enables you to write function members that enable the basic operators to be applied to class objects (source: Beginning C++). To do this, you write a function that redefines each operator that you want to use with your class .

- Groups operators in C++
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Bitwise operators



Common operators

assignment	increment decrement	arithmetic	logical	comparison	member access	other

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
				`a <=> b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						`a(...)`
						`a, b`
						`? :`

Special operators

`static_cast` converts one type to another related type
`dynamic_cast` converts within inheritance hierarchies
`const_cast` adds or removes cv qualifiers
`reinterpret_cast` converts type to unrelated type
C-style cast converts one type to another by a mix of `static_cast`, `const_cast`, and `reinterpret_cast`
`new` creates objects with dynamic storage duration
`delete` destructs objects previously created by the new expression and releases obtained memory area
`sizeof` queries the size of a type
`sizeof...` queries the size of a parameter pack (since C++11)
`typeid` queries the type information of a type
`noexcept` checks if an expression can throw an exception (since C++11)
`alignof` queries alignment requirements of a type (since C++11)

increment & decrement

```
class Counter {
   private:
      int count;

   public:
      Counter(int c = 0) : count(c) {}

      Counter operator++() {
         ++count;
         return *this;
      }

      Counter operator++(int) {
         Counter temp = *this;
         ++count;
         return temp;
      }
```

Arithmetic operators

```
class Number {
public:
   int value;

   Number(int value) {
      this->value = value;
   }

   Number operator+(const Number& other) {
      return Number(this->value + other.value);
   }
```

Assignment operators

```
class Person {
   private:
      string name;
      int age;

   public:
      Person(string n = "", int a = 0) {
         name = n;
         age = a;
      }

      // Getters for private member variables
      string getName() const {
         return name;
      }

      int getAge() const {
         return age;
      }
};

// Overloaded stream insertion operator as a non-member function
ostream& operator<<(ostream& os, const Person& p) {
   os << "Name: " << p.getName() << ", Age: " << p.getAge();
   return os;
}
```

Comparison operators

```
class Point {
   private:
      int x, y;

   public:
      Point(int x = 0, int y = 0) : x(x), y(y) {}

      bool operator==(const Point& other) const {
         return (x == other.x && y == other.y);
      }
};
```