

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



ADVANCED PROGRAMMING (CO2039)

ASSIGNMENT REPORT

ADVANCED PROGRAMMING FINAL REPORT

Lecturer: Trương Tuấn Anh

Student name: Nguyễn Minh Hùng

Student ID: 1952737 - Class: CC01

HO CHI MINH CITY, AUGUST 2021



Contents

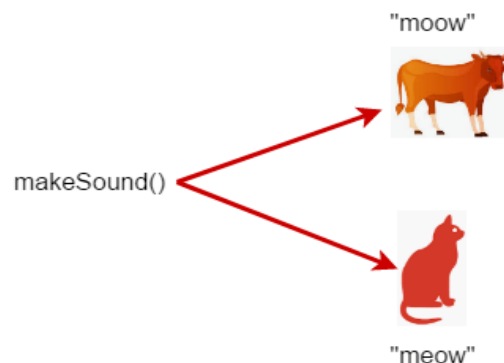
1	Polymorphism in C++	2
1.1	Motivation example	2
1.2	Definition and classification	2
1.3	Compile time Polymorphism and its advantages	3
1.3.1	Function overloading	3
1.3.2	Operator overloading	5
1.4	Run-time Polymorphism and its advantages	6
1.4.1	Function overriding	6
1.4.2	C++ Virtual Function	8
1.5	Summary	10
2	Object Oriented Programming in C++ and Java	11
2.1	Introduction	11
2.2	Comparison in Object Oriented Programming	11
2.2.1	Multiple Inheritance	12
2.2.2	Virtual keyword	15
2.2.3	Overloading	15
2.2.4	Templates in C++ vs Generics in Java	16
2.2.5	Other differences: Pointers, Parameter Passing, Object, Memory Management,	17
2.3	Summary and Conclusion	18
3	Functional Programming VS Object Oriented Programming	20
3.1	Introduction	20
3.1.1	Functional Programming	20
3.1.2	Object Oriented Programming	20
3.2	Comparison	21
3.2.1	Data	21
3.2.2	Model	22
3.2.3	Parallelism	23
3.2.4	Execution Order	24
3.2.5	Iteration	25
3.2.6	Use	25
3.3	Pros and cons	26
3.3.1	Functional Programming(FP)	26
3.3.2	Object Oriented Programming(OOP)	28
3.4	Summary and Conclusion	29

1 Polymorphism in C++

1.1 Motivation example

Suppose that we design a `Animal` system that can manipulate objects of many *different* types, including objects of classes `Cow` and `Cat`. Imagine that each of these classes inherits from the common base class `Animal`, which contains the member function `makeSound()`. Each derived class implements this function in a manner appropriate for that class.

When a cat calls this function, it will produce the "meow" sound. When a cow invokes the same function, it will provide the "moow" sound.



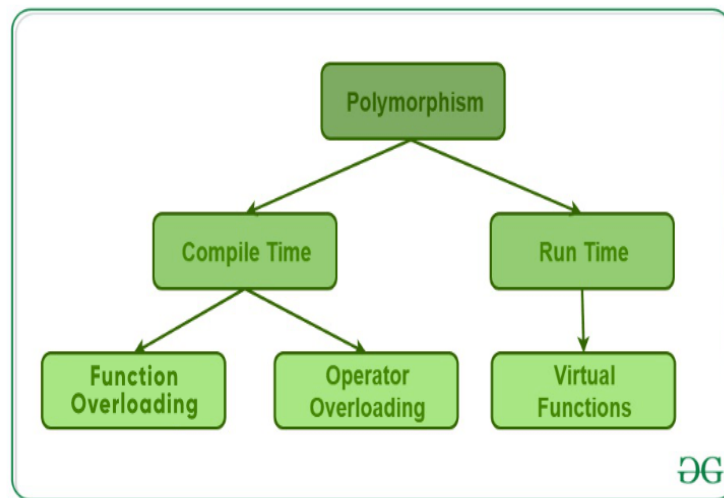
Though we have one function, it behaves differently under different circumstances. The function has many forms; hence, we have achieved **polymorphism**.

1.2 Definition and classification

The word **polymorphism** means having many forms. In simple words, we can define **polymorphism** as the ability of a message to be displayed in more than one form. A real-life example of **polymorphism**, a person at the same time can have different characteristics. Like an animal at the same time is a cow, a cat, a dog. So the same animal possesses different behavior in different situations. This is called **polymorphism**. **polymorphism** is one of the key features of **object-oriented programming**, after **classes** and **inheritance**.

In C++ **polymorphism** is mainly divided into two types:

- Compile time Polymorphism
- Run-time Polymorphism



1.3 Compile time Polymorphism and its advantages

This type of **polymorphism** is achieved by function overloading or operator overloading.

The information is present during compile-time. This means the C++ compiler will select the right function at compile time.

1.3.1 Function overloading

When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

C++ Example code:

```
#include <iostream>
using namespace std;

class myclass{
public:
    // function with 2 int parameter
    void sum(int x, int y)
    {
        cout << "sum_is_" << x + y << endl;
    }

    // function with same name but 2 double parameter
```

```
void sum(double x, double y)
{
    cout << "sum is " << x + y << endl;
}

// function with same name and 1 int parameters
void sum(int x)
{
    cout << "sum is " << x << endl;
}

};

int main(){
    myclass *a = new myclass();
    a->sum(1, 2);
    a->sum(1.5, 1.3);
    a->sum(1);
}
```

Output:

```
sum is 3
sum is 2.8
sum is 1
```

Explanation: In the above example, a single function named `sum()` acts differently in three different situations which is the property of polymorphism. C++ provides programmers the ability to write the same name functions as long as we distinguish it with different types or amount of parameters passing.

Advantages:

- The main advantage of function overloading is that it improves code readability and allows code re-usability (simple name "sum" for all functions with same purpose (calculating sum)).
- The use of function overloading is to save memory space, consistency, and readability.
- It speeds up the execution of the program.
- Code maintenance also becomes easy.
- The function can perform different operations and hence it eliminates the use of different function names for the same kind of operations.

1.3.2 Operator overloading

In Operator Overloading, we define a new meaning for a C++ operator. It also changes how the operator works. For example, we can define the "+" operator to concatenate two strings. We know it as the addition operator for adding numerical values. After our definition, when placed between integers, it will add them. When placed between strings, it will concatenate them.

C++ example code:

```
#include <iostream>
using namespace std;
//vector addition
class vector{
public:
    int x;
    int y;
    vector(int X,int Y){
        this->x = X;
        this->y = Y;
    }
    vector operator+(vector);
};
vector vector::operator+ (vector v){
    vector res(0, 0);
    res.x = this->x + v.x;
    res.y = this->y + v.y;
    return res;
}

int main(){
    vector A(1,2);
    vector B(3,4);
    vector C = A + B;
    cout << "Vector C after addition: ("<<C.x<<","<<C.y<<")\n";
}
```

Output:

```
Vector C after addition: (4, 6)
```

Explanation: In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two vectors.

Advantages:

- Operator overloading enables programmers to use notation closer to the target domain. For example we can add two vectors by writing $M1 + M2$ rather than writing $M1.add(M2)$.
- The use of function overloading is to save memory space, consistency, and readability.
- Operator overloading provides similar syntactic support of built-in types to user-defined types.
- Operator overloading makes the program easier to understand.

1.4 Run-time Polymorphism and its advantages

This happens when an object's method is invoked/called during runtime rather than during compile time. **Runtime polymorphism** is achieved through **function overriding**. The function to be called/invoked is established during runtime.

In C++, there is a special function call **virtual function** which is another way of implementing run-time polymorphism.

1.4.1 Function overriding

Function overriding occurs when a function of the base class is given a new definition in a derived class. At that time, we can say the base function has been overridden.

C++ example code:

```
#include <iostream>
using namespace std;

class Animal{
public:
    void makeSound(){
        cout << "Animal_sound_\n";
    }
};

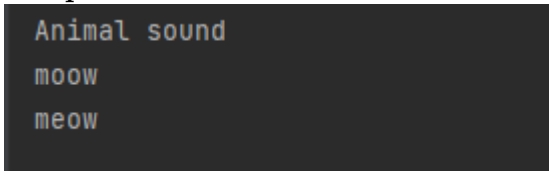
class Cow:public Animal{
```

```
public:
    void makeSound(){
        cout << "moow\n";
    }
};

class Cat:public Animal{
public:
    void makeSound(){
        cout << "meow\n";
    }
};

int main(){
    Animal a = Animal();
    Cow cow = Cow();
    Cat cat = Cat();
    a.makeSound();
    cow.makeSound();
    cat.makeSound();
}
```

Output:



```
Animal sound
moow
meow
```

Explanation: In the above example, the function `makeSound()` in the base class (`Animal`) has been overridden in both class `Cow` and `Cat`. Therefore when we create an object belong to any class, we can call its overridden function (`makeSound()`).

Advantage:

- The child class having same function of parent class, can even have its independent implementation of that function.
- Helps in saving the memory space.
- Maintain consistency, and readability of our code.
- Helps in making code re-usability easy.
- The child class can access function of parent class as well.

1.4.2 C++ Virtual Function

A virtual function is another way of implementing run-time polymorphism in C++. It is a special function defined in a base class and redefined in the derived class. To declare a virtual function, you should use the `virtual` keyword. The keyword should precede the declaration of the function in the base class.

If a virtual function class is inherited, the `virtual` class redefines the virtual function to suit its needs.

C++ example code

```
#include <iostream>
using namespace std;

class Animal{
public:
    virtual void makeSound(){
        cout << "Animal_\n";
    }
};

class Cow:public Animal{
public:
    void makeSound(){
        cout << "moow_\n";
    }
};

class Cat:public Animal{
public:
    void makeSound(){
        cout << "meow_\n";
    }
};

int main(){
    Animal *a = new Animal();
    Cat cat = Cat();
    a = &cat;
    a->makeSound();
}
```

Output



```
meow
```

Explanation: In the above code, we have overridden the function `makeSound()`, then we initialize a pointer `point` to a `Animal` object and a `Cat` object. With this pointer, we can refer to all the different classes' objects by taking their addresses. If we don't use the `virtual` keyword, when the base class pointer contains the derived class address, the object always executes the base class function. The problem is resolved when we use `virtual` function.

Advantages:

- With `virtual` functions, the type of the object, not the type of the handle used to invoke the member function, determines which version of a virtual function to invoke.
- The program can determine dynamically (i.e., at runtime) which derived class function to use, based on the type of the object to which the base-class pointer points at any given time. This is **polymorphic behavior**.
- Reusability of code.
- Small program length.
- Virtual functions and dynamic binding enable polymorphic programming as an alternative to switch logic programming.

1.5 Summary

According to the advantages we have given above for each type of **Polymorphism** in C++, the **key advantages of Polymorphic Programming in C++**:

- Polymorphism enables you to deal in **generalities** and let the execution-time environment concern itself with the specifics. You can direct a variety of objects to behave in manners appropriate to those objects without even knowing their types—as long as those objects belong to the same inheritance hierarchy and are being accessed off a common base-class pointer or a common base-class reference.
- Polymorphism promotes **extensibility**: Software written to invoke polymorphic behavior is written independently of the specific types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.
- Same interface could be used for creating methods with different implementations.
- Polymorphic programming can eliminate the need for switch logic. By using the polymorphism mechanism to perform the equivalent logic, you can avoid the kinds of errors typically associated with switch logic. ⇒ **simplified appearance**.
- The use of Polymorphism can save memory space, consistency, and readability.
- It also improves code read-ability and allows code re-usability.
- Execution time can be speed up (in compile time Polymorphism).

2 Object Oriented Programming in C++ and Java

2.1 Introduction

C++ and Java both are object-oriented programming languages. Yet, both languages differ from each other in many ways.

- C++ is derived from C and has the features of both procedural and object-oriented programming languages. C++ was designed for **both** application and System development, but mainly for system development.
- Java is built upon a virtual machine which is very **secure and highly portable** in nature. The Java foundation class libraries provide for windowing and graphical user interface programming, network communications, and multimedia facilities. Java was **mainly** designed for **application programming** and has a functionality of an interpreter for printing systems which was later developed into network computing.

2.2 Comparison in Object Oriented Programming

Both C++ and Java are [Object-Oriented Programming](#), hence they both have four main OOP concepts, that is: **Abstraction, Encapsulation, Inheritance and Polymorphism**.

Moreover, they also have similar syntax (because Java is strongly influenced by C, C++), operators, primitive data types (`int`, `float`, `char`, `double`), keywords(`break`, `continue`, `return`, ... and both executions start from the main function.

However, dive deep into each concepts of OOP, these two languages has there own ways to design the program. A reason for this is due to their different design goals in industry and real-life applications. As I said above C++ is designed for system programming and Java is designed for application programming.

The different goals in the development of C++ and Java resulted in different principles and design trade-offs between the languages. The differences are as follows:

2.2.1 Multiple Inheritance

We all know that **Inheritance** is an important concept in OOP. **Multiple Inheritance** is a feature of object oriented concept, where a class can inherit properties of more than one parent class.

However, **while C++ supports various types of inheritances including single and multiple inheritances, Java supports only single inheritance.**

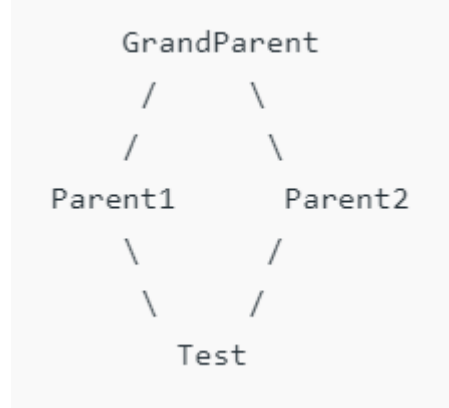
```
// in C++
class GrandParent{
public:
    void func(){cout << "GP\n";}
};
class Parent1: public GrandParent{
public:
    void func(){cout << "P1\n";}
};
class Parent2: public GrandParent{
public:
    void func(){cout << "P2\n";}
};
class Test:public Parent1, Parent2{};
/* -> Compilation Successful !!! */

// in Java
class GrandParent{
    void func(){System.out.println("GP");}
}
// First Parent class
class Parent1 extends GrandParent{
    void func()
    {
        System.out.println("P1");
    }
}
// Second Parent Class
class Parent2 extends GrandParent
{
    void func()
    {
        System.out.println("P2");
    }
}
```

```
}  
// Error : Test is inheriting from multiple  
// classes  
public class Test extends Parent1, Parent2 {}  
/* -> Compilation Error */
```

The question is : **Why Java doesn't support Multiple Inheritance?**

The problem occurs when there exist methods with same signature (func) in the super classes (**Parent1** and **Parent2**). On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority. This problem is nearly similar as the **Diamond problem**.



Therefore, in order to avoid such complications Java does not support multiple inheritance of classes. Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex
⇒ **Simplicity**.

⇒ **Solution:**

Java 8 supports default methods where **interfaces** can provide default implementation of methods. And a class can implement two or more interfaces.

Java Interfaces

```
interface PI1  
{  
    default void show()  
    {  
        System.out.println("Default_PI1");  
    }  
}  
interface PI2  
{
```

```
default void show()
{
    System.out.println("Default_PI2");
}

class TestClass implements PI1, PI2
{
    public void show()
    {
        PI1.super.show();
        PI2.super.show();
    }
}
```

In C++, though we can compile our code, we still stuck into the problem and the compiler still cannot determine which class method to be called.

⇒ The **solution** to this problem is ‘**virtual**’ keyword. We make the classes ‘Parent1’ and ‘Parent2’ as virtual base classes to avoid two copies of ‘GrandParent’ in ‘Test’ class.

Virtual keyword

```
#include <iostream>
using namespace std;
class GrandParent{
public:
    void func(){cout << "GP\n";}
};
class Parent1: virtual public GrandParent{
public:
    void func(){cout << "P1\n";}
};
class Parent2: virtual public GrandParent{
public:
    void func(){cout << "P2\n";}
};
class Test:public Parent1, Parent2{
public:
    void func(){
        cout << "Test\n";
    }
}
```

```
};  
  
int main(){  
    Test t = Test();  
    t.func();  
}
```

2.2.2 Virtual keyword

In C++, as a part of dynamic polymorphism, **the virtual keyword is used** with a function to indicate the function that can be overridden in the derived class. This way we can achieve **polymorphism**. By default, class member methods are **non-virtual** (cannot be overridden). We can make the method **virtual** by using the keyword. ⇒ **complicated and not easy**.

In Java, **the virtual keyword is absent**. However, in Java, all non-static methods by default can be overridden. Or in simple terms, all non-static methods in Java are **virtual** by default. ⇒ **Simple and easy for user**.

2.2.3 Overloading

In C++, **both methods and operators can be overloaded**. This is **static polymorphism**.

In Java, **only method overloading is allowed. It does not allow operator overloading**.

Regarding to method overloading (function overloading), there is **no difference between 2 languages**. However, the given question is:
Why Java doesn't support operator overloading ?

As we have known that operator overloading in C++ is kinda complex and may lead to programming error. The **Java designers** wanted to prevent people from using operators in a confusing manner, but it was not worth it. Therefore, it's **just a choice** made by its creators who wanted to keep the language **more simple**.
⇒ The functionality of operator overloading can be achieved in Java by using **method overloading** in Java in a simple, error-free and clear manner.

2.2.4 Templates in C++ vs Generics in Java

While building large-scale projects, we need the code to be compatible with any kind of data which is provided to it. Here what we meant is to make the code you write be **generic** to any kind of data provided to the program regardless of its data type. This is where **Generics** in Java and the similar in C++ named **Template** comes in handy. While both have **similar functionalities**, but they **differ in a few places**. Take for instance the following:

```
#include <iostream>
using namespace std;

template<class T>
class test{
    T value;
public:
    test(T t):value(t){};
    T getValue(){return value;}
};

int main(){
    test<double> t = test<double>(1.5);
    cout << t.getValue();
}
```

Output: 1.5

In Java "templates" are called generics. They are enclosed with two <> signs. Translating the above code into Java would look as follows:

```
public class test<T> {
    private T t;
    public test(T t) {
        this.t = t;
    }
    public T getT() {
        return t;
    }
    public static void main(String[] args) {
        test<Integer> t = new test<Integer>(1);
        System.out.println(t.getT());
    }
}
```

```
}  
}
```

Output: 1

Although the two different pieces of code may look quite similar, they are a different thing in both languages.

Java generics simply offer *compile-time safety* and *eliminate the need for casts*. They are implemented directly by the Java compiler as a front-end conversion, also known as **type erasure**. The compiler basically just erases all generic specifications (between the angle brackets) and inserts casts where necessary. Moreover it keeps track of the generics internally such that all instantiations will use the same underlying generic class at compile/run time. It is therefore somehow a process of code translation or rewriting.

A **C++ template** gets reproduced and re-compiled entirely whenever a template is instantiated with a new class.

The **main difference** is that **Java generics are encapsulated**. The errors are flagged when they occur and not later when the corresponding classes are used/instantiated.

2.2.5 Other differences: Pointers, Parameter Passing, Object, Memory Management, ...

Pointers:

- C++ is all about pointers. As seen in tutorials earlier, C++ has strong support for pointers and we can do a lot of useful programming using pointers.
- Initially, Java was completely without pointers but later versions started providing limited support for pointers. We cannot use pointers in Java as leisurely as we can use in C++.

Parameter passing:

- C++ supports both Pass by Value and pass by reference.
- Java supports only Pass by Value technique.

Memory management

- Memory management in C++ is manual. We need to allocate/deallocate memory or object manually using the new/delete operators.

- In Java the memory management is system-controlled. Object \Rightarrow automatically with garbage collection.

2.3 Summary and Conclusion

According to the discussion above, I summarize some of the key differences between C++ Vs Java.

Features	C++	Java
Abstraction	Yes	Yes
Encapsulation	Yes	Yes
Inheritance	Yes	Yes
Polymorphism	Yes	Yes
Single Inheritance	Yes	Yes
Multiple Inheritance	Yes, Although there are problems arising from multiple inheritances, C++ uses the virtual keyword to resolve the problems.	No, Effects of multiple inheritance can be achieved using the interfaces in Java.
Methods overloading	Yes	Yes
Operator overloading	Yes	No
Virtual keyword	Yes, the method is non-virtual by default	No, the method is virtual by default
Generics programming	Compile-time templates. Allows for Turing complete meta-programming.	Generics are used to achieve basic type-parametrization, but they do not translate from source code to byte code due to the use of type erasure by the compiler.
Pointers	It strongly supports Pointer.	It supports limited support for pointers.
Parameter Passing	C++ supports both Pass by Value and pass by reference.	Java supports only Pass by Value technique.
Memory Management	Memory management in C++ is manual. We need to allocate/deallocate memory manually using the new/delete operators.	In Java the memory management is system-controlled.

Conclusion:

- C++ and Java are both **object-oriented programming languages**. In addition, C++ is a procedural language as well. There are some features like **inheritance, polymorphism, pointers, memory management, etc.** in which both the languages completely **differ** with one another.
- The **differences** between the programming languages C++ and Java can be traced to their heritage, as they have **different design goals**.
- C++ was designed for both systems and applications programming (i.e. infrastructure programming), extending the procedural programming language C, which was designed for **efficient execution**. Hence, It has many characteristics for programmers to go closeness to hardware and modify (multiple inheritance, operator overloading, virtual, pointers, references, memory management,..), however It would be **complex and difficult** for beginner. Sometimes, some features like pointers can lead to programming errors if we

misused it. Moreover, code implementation of programs can be longer and not easy to read. C++ depends lots on programmers on better optimization, performance, speed,... So C++ is strongly used for large system programming, operating systems (UNIX), high-speed gaming applications, etc.

- **Java** is a general-purpose, concurrent, class-based, object-oriented programming language that is designed to **minimize implementation dependencies**. The easier syntax of Java, elimination of some features in OOP (multiple inheritance, virtual, operator overloading,...), automatic garbage collection, strongly supportive library, lack of pointers, templates,... make **Java** a favourite, simple and easy to use for developers who want to design mobile, web applications.

Remark: The power of **C++ and Java** totally depends on what application we are developing. The best way is to evaluate beforehand the pros and cons of both the languages and verify their uniqueness for the application that we are developing and then conclude which is the best.

3 Functional Programming VS Object Oriented Programming

3.1 Introduction

Firstly, **what is a programming paradigm** ?

A **programming paradigm** is a style, or “way, ” of programming. **Programming paradigms** differ from one another based on the **features and the style** they support. There are several features that determine a programming paradigm such as *modularity, objects, interrupts or events, control flow* etc. Every **programming paradigm** has its own advantage so, it better to know where to use it before actually using it.

3.1.1 Functional Programming

Functional Programming is *style* of programming in which the basic method of computation is the application of functions to *arguments*. A functional language is one that supports and encourages the functional style, in particular the Haskell programming language (in this course).

Functional programming is a **declarative programming** paradigm where programs are created by applying sequential functions rather than statements. Each function takes in an input value and returns a consistent output value *without altering or being affected by the program state*. Consider the following Haskell example:

```
> sum [1,2,3,4,5]
15
```

Given the same arguments (list), we always get the same output regardless of program state.

Functional programming provides the advantages like *efficiency, lazy evaluation, nested functions, bug-free code, parallel programming*.

3.1.2 Object Oriented Programming

Object oriented programming is a programming paradigm in which you program using *objects* to represent things you are programming about (sometimes real world things). These *objects* could be data structures. The *objects* hold data about them in attributes. The attributes in the objects are manipulated through methods or functions that are given to the *object*.

For instance, we might have a **Person** object that represents all of the data a person would have: *weight, height, skin color, hair color, hair length, and so*

on. Those would be the *attributes*. Then the person object would also have things that it can do such as: *pick box up*, *put box down*, *eat*, *sleep*, etc. These would be the *functions* that play with the data the object stores. Consider a C++ example for class Person:

```
class Person{
    double weight;
    Color skin;
    double height;\
public:
    void pick_box_up();
    void put_box_down();
    void eat();
    void sleep();
};
```

3.2 Comparison

3.2.1 Data

Functional Programming uses **immutable data** (cannot be changed after creation).

Meanwhile, **Object Oriented Programming** uses **mutable data** as objects.

For instance, in Haskell, **Immutability** is a confusing concept. It is the property of all Haskell expressions that, once they are assigned to a name, they **CANNOT** change their value. To most programmers, this seems like a completely crazy idea. How can you program at all when you cannot change the value of an expression? To start, let's consider this example of how mutability works in C++:

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    string a = "test";
    cout << a << endl;
    reverse(a.begin(), a.end());
    cout << a << endl;
}
```

Output:

```
test  
tset
```

We see now when we call the reverse function on a, the value of a actually **changes**. Let's see what happens in Haskell:

```
>> let a = [1,2,3]  
>> reverse a  
[3,2,1]  
>> a  
[1,2,3]
```

This time, the value of a did not change. Calling reverse seems to have produced the output we expected, but it had no impact on the original variable. To understand what it is happening, let's first look at the type signature for reverse:

```
reverse :: [a] -> [a]
```

We see reverse takes one argument, a list. It then outputs a list of the same type. It's a pure function, so it cannot change the input list! Instead, it constructs a completely new list with the same elements as the input, only in reverse order. This may seem strange, but it has huge implications on our ability to reason about code.

3.2.2 Model

- **Functional programming** does follow a *declarative programming model*.
- **Object-oriented programming** does follow an *imperative programming model*

Programming languages are divided into different *paradigms*.

- Programs written in traditional languages like C++ are **imperative programs** that contain instructions to mutate state. Variables in such languages point to memory locations and programmers can modify the contents of variables using assignments. An **imperative program** contains commands that describe how to solve a particular class of problems by describing in detail the steps that are necessary to find a solution (or implements algorithms in explicit steps).

- **By contrast, declarative programming** is a programming paradigm—a style of building the structure and elements of computer programs—that expresses the **logic of a computation without describing its control flow**.

For instance, in `Haskell`, all functions are defined and declared without side effects, there is nothing **imperative** about defining things in terms of other things because we can always replace something with its definition (as long as the definition is pure). A lot of `Haskell` code is considered declarative for this reason. We simply define things as (pure) functions of other things.

We can see that there is no 'if' statement in `Haskell`, only an if expression "if a then b else c" (**conditional expressions**).

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
           if n == 0 then 0 else 1
```

The `signum` function defined in `Haskell` with expression of computation without any commands.

In **imperative programming**, we implement the code to solve the problem by many lines of instructions to illustrate for the algorithms. Consider `C++` code for `signum`:

```
int signum(int x){
    int result;
    if(x < 0) res = -1;
    else if (x == 0) res = 0
    else res = 1;
    return res;
}
```

3.2.3 Parallelism

- **Functional Programming** is *parallel programming* supported.
- **OOP** does *not* support strongly *parallel programming*.

Because modern computers have multiple cores per CPU, and often multiple CPUs, it's more important than ever to design programs in such a way that they can take advantage of this *parallel* processing power.

In **Haskell**, **GHC** includes support for running **Haskell** programs in parallel on symmetric, *shared-memory multi-processor (SMP)*. By default **GHC** runs your program on one processor; if you want it to run in parallel you must link your program with the `-threaded`, and run it with the RTS `-N` option. The runtime will schedule the running **Haskell** threads among the available OS threads, running as many in parallel as you specified with the `-N` RTS option.

```
>>ghc --make -threaded test.hs  
>>test +RTS -N3
```

The above commands compile a parallel Haskell program by specifying the `-threaded` extra flag and use three real threads to execute the `test` program.

In **object-oriented programming (OOP)** languages, the ability to encapsulate software concerns of the dominant decomposition in objects is the key to reaching *high modularity and loss of complexity in large scale designs*. However, **distributed-memory parallelism** tends to *break modularity, encapsulation, and the functional independence of objects*, since **parallel** computations **cannot** be encapsulated in individual objects, which reside in a single address space.

3.2.4 Execution Order

- The statements in **Functional programming paradigm** **does not** need to follow a particular order while execution.
- The statements in **OOP** paradigm **need** to follow a order i.e., bottom up approach while execution.

In **functional programming** we avoid **side effects** in most of your code. Without side effects the order of evaluation really doesn't matter. Particularly, **Haskell**, which is a **purely functional lazy evaluation language** can do the exact same thing using *monads*. Then your function to produce the next value is dependant on some value returned on the previous. The chain makes the order of execution a steady path.

In contrast, in some **Object Oriented Language**, the program has to be execute in a particular order.

3.2.5 Iteration

- In **Functional Programming**, we use recursion to iterate.
- In **OOP**, we use both loops and recursion to iterate.

What's the first thing we remember about **functional programming**? Oh right, it focuses on using **pure functions** and doesn't like **side effects**. Let's take a look at this loop in **C++** below:

```
int calSum(int n){  
    int sum = 0;  
    for(int i = 0; i < n; i++) sum += i;  
    return sum;  
}
```

Let's look at the loop, the loop unfortunately is **not side-effect-free**. In fact, *loops rely on side effects to control flow*. Something has to change for you to exit your loop. **A loop is not pure !**.

So, we can use **recursion** to resolve this:

```
calSum :: Int -> Int  
calSum n = if n == 0 then 0 else n + calSum(n - 1)
```

3.2.6 Use

- **Functional programming** is used only when there are **few** things with **more** operations.
- **Object-oriented programming** is used when there are **many** things with **few** operations.

The most important building block that defines **object-oriented languages** is the object itself. Without **objects**, we can't talk about object-orientation.

In short, **objects focus on data. Operations only come after.**

In **functional programming**, functions are the **first-class-citizens**. For example, they can be passed as a parameter for other functions, or we can store them in variables.

In general, **functional programming focuses on operations rather than data.**

3.3 Pros and cons

3.3.1 Functional Programming(FP)

So far, we have had to deal with functional programming by creating Haskell code using functional approach. Now, we can look at the **advantages and disadvantages** of the functional approach, such as the following:

Advantages:

- **Does not need to follow a particular order.** The order of execution doesn't matter since it is handled by the system to compute the value we have given rather than the one defined by programmer. In other words, the **declarative of the expressions** will become **unique**. Because functional programs have an approach toward mathematical concepts, the system will be designed with the notation as close as possible to the mathematical way of concept. This makes FP suitable for mathematical evaluation.
- Variables can be replaced by their value since the evaluation of expression can be done any time. The functional code is then **more mathematically traceable** because the program is allowed to be manipulated or transformed by substituting equals with equals. **This feature is called referential transparency.**
- **Immutability makes the functional code free of side effects** (making debugging and testing easier). A shared variable, which is an example of a side effect, is a serious obstacle for creating parallel code and result in non-deterministic execution. By removing the side effect, we can have a **good coding approach.**
- **The power of lazy evaluation will make the program run faster** because it only provides what we really required for the queries result. Consider this Haskell example:

```
func :: Int -> Int -> Int
func x y = x + 1

>>func 3 (3+4)
4
```

In the example above there is no point in evaluating $3 + 4$, because it's not used. The style of only evaluating what's needed is called **lazy evaluation**, while the opposite (evaluating immediately) is called **strict evaluation**.

- Due to the nature of **pure functions** to avoid changing variables or any data outside it, implementing **concurrency, parallelism** becomes efficacious.
- It's style treats functions as values and passes the same to other functions as parameters. It **enhances the comprehension and readability of the code.**

Disadvantages:

- **Immutable values combined with recursion** might lead to a **reduction in performance.**
- Compared to **imperative programming**, **much garbage will be generated in functional programming** due to the concept of immutability, which needs more variables to handle specific assignments. Because we cannot control the garbage collection, the performance will decrease as well.
- Lack of using loops maybe a challenging task when we can only use recursion (difficult).
- For beginners, it is difficult to understand. So it is **not a beginner-friendly paradigm approach for new programmers.**
- **Requires a large memory space.** As it does not have state, you need to create new objects every time to perform actions.

3.3.2 Object Oriented Programming(OOP)

So far, in this course, we had experienced and known how to write and implement C++ code using object oriented approach. Now, we can look at the advantages and disadvantages of this, such as the following:

Advantages:

- **Re-usability**, “Write once and use it multiple times” you can achieve this by using *class*. Objects can also be reused within an across applications.
- **Modularity**, **Object-oriented programming** is modular, as it provides separation of duties in object-based program development.
- **Extensibility**, objects can be extended to include new attributes and behaviors.
⇒ Three factors – **modularity, extensibility, and reusability** – **object-oriented programming** provides **improved software-development productivity** over traditional procedure-based programming techniques.
- **Easy Maintenance**, It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones. Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes.
⇒ **Improved software maintainability**.
- **Redundancy**, **Inheritance** is the good feature for data redundancy. If you need a same functionality in multiple class you can write a common class for the same functionality and inherit that class to sub class.
- **Security**, using data hiding and abstraction only necessary data will be provided thus maintains the security of data.

These features make software development **improve a lot on productivity, maintainability, faster, lower cost development, higher-quality software**.

Disadvantages:

- **Larger program size**, Object-oriented programs typically involve more lines of code than procedural programs.
- **Slower programs**, Object-oriented programs are typically slower than procedure-based programs, as they typically require more instructions to be executed.
- **Not suitable for all types of problems**, There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

- **Difficulty for beginners**, the thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. Such as inheritance and polymorphism, can be challenging to comprehend initially.

3.4 Summary and Conclusion

The following table is to summarize the key differences of Functional Programming and OOP:

BASIS FOR COMPARISON	Functional Programming	OOP
Definition	Functional programming emphasizes an evaluation of functions.	Object-oriented programming based on a concept of objects.
Data	Functional programming uses immutable data.	Object-oriented uses mutable data.
Model	Functional programming does follow a declarative programming model.	Object-oriented programming does follow an imperative programming model.
Support	Parallel programming supported by Functional Programming.	Object-oriented programming does not support parallel programming.
Execution	In Functional programming, the statements can be executed in any order.	In OOPs, the statements should be executed in a particular order.
Iteration	In Functional programming, recursion is used for iterative data.	In OOPs, loops are used for iterative data.
Use	Functional programming is used only when there are few things with more operations.	Object-oriented programming is used when there are many things with few operations.

Conclusion:

- **Functional Programming and Object-oriented programming** both are different concepts of programming language. Both Functional Programmings vs OOP languages aim to **provide bug-free code**, which can be easily understandable, well-coded, managed and fast development.

- **Functional programming and object-oriented programming** uses different method for storing and manipulating the data. In **functional programming**, data **cannot be stored in objects**, and it can only be **transformed by creating functions**. In **object-oriented programming**, data is stored in objects.
- **Object-oriented languages** are **good** when you have a *fixed set of operations on things*, and as your code evolves, you primarily *add new things*. This can be accomplished by adding new classes which implement existing methods, and the existing classes are left alone.
- **Functional languages** are **good** when you have *a fixed set of things*, and as your code evolves, you primarily *add new operations on existing things*. This can be accomplished by adding new functions which compute with existing data types, and the existing functions are left alone.
- In **Object-oriented programming**, it is really hard to maintain objects while increasing the levels of inheritance. It also breaks the principle of encapsulation and not fully modular even. In **functional programming**, it requires always a new object to execute functions and it takes a lot of memory for executing the applications.

Remark: Finally, to conclude, each has their own advantages and disadvantages, it is always up to the programmers or developers to choose the programming language concept that makes their development productive and easy depending on what purposes and design aims. We cannot say which of them is better.