ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

ADVANCED PROGRAMING (CO2039)

FINAL REPORT

RESEARCH ON OBJECT-ORIENTED PROGRAMMING AND FUNCTIONAL PROGRAMMING

GVHD: Trương Tuấn Anh
SVTH: Trần Nguyễn Phước Nhân
MSSV: 1952893

Tp. Hồ Chí Minh, Tháng 8/2021

Mục lục

TOPIC I: OBJECT-ORIENTED PROGRAMING AND ITS PROPERTIES

1. **What is Object-Oriented Programing (OOP)?**

- Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.
- OOP works based on the objects that developers want to handle rather than the logic to handle them. The approach is well-fitted for programs that are big, complex, frequently updated or maintained.

2. **What is the structure of OOP?**

- Classes are user-defined data type that acts as the blueprints for some objects, attributes, and methods.
- Objects are instances of class created with defined data. It can be related to real-world objects or just an abstract entity. Whenever a class is initialized, the description is the only object that is defined.
- Methods are functions that are defined inside a class to describe behaviors of an object. Programmers use methods for reusability or keeping functionality encapsulated inside object.
- Attributes are defined in the class template to represent the state of an object.

3. **What are the principles of OOP?**

- **Encapsulation:** This principle by which we hide information that we don't want the users to see and show users what we want them to see instead.
- **Abstraction:** This principle is used to hide implementation details. It hides and handles complex tasks from users. Users will know what it can does but does not need to know how it was implemented.
- **Inheritance:** This principle is to rewrite the exact same logic in the base class to the derived class that requires that logic. For example, vehicles are the base class, then cars would be the derived class which inherits some basic properties of vehicles like fuel, tires, engines, …
- **Polymorphism:** This principle means "many shapes". In OOP, it means that the same entity (function or object) behaves differently in different scenarios. For example, "walking" in living creatures may vary from no legs, one legged, two-legs, … in which functions differently in different situations.

4. **What are the key advantages of Polymorphism in C++?**

Polymorphism in C++ is mainly divided into two types:

+ **Compile time Polymorphism:** Achieved through functions overloading or operator overloading.
  - **Function Overloading:** When there exist multiple functions with the same name but different parameters then those function are said to be ***overloaded***. Function can be overloaded by change in numbers of arguments or change in type of arguments.
  - **Operator Overloading:** When we need to assign an operator for specific tasks of our defined data type, we use operator overloading to handle that task, minimizing the time for getting and setting new data, minimize the space required for that task.
+ **Runtime Polymorphism:** Achieved through function overriding.

- **Function Overriding:** This occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Example code for function overloading and operator overloading:

```
#include <iostream>
using namespace std;
struct Complex {
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   void Display( ) {   cout << re << ", " << im << endl; }
private:
   double re, im;
};
// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}
// function overloading
void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}
```

Example code for function overriding:

```
#include <iostream>
using namespace std;
class Base {
  public:
   void print() {
      cout << "Base Function" << endl;
   }
};
class Derived : public Base {
  public:
   void print() {
      cout << "Derived Function" << endl;
   }
};
```

In OOP, polymorphism is defined as how to carry out different processing steps by a function having the same messages. Polymorphism treats objects of related classes in a generic manner.

- Advantages of Polymorphism in C++:
  + It allows programmers to reuse, evaluate and execute the program, modules, forms written once. In certain aspects, they can be repeated.
  + You may use the odd variable name to stock variables of different types of data, such as Int, Float, etc.).
  + Polymorphism tends to reduce the pairing of multiple functionalities.
  + Method overloading can be extended to builders that allow multiple ways of initializing class objects. It helps you to identify several builders for managing various forms of initializations.
  + Method overriding functions along with inheritance without the need for re-compilation to allow code reuse of existing groups.

Example code of Polymorphism in C++:

Assume we have an interface of List Data Type, name IList, which share common properties such as: ***add, remove, search***. And also, SLinkedList for Singly Linked List derived from IList. IList has pure virtual functions that can be seen below. And SLinkedList has its detailed definition of each method in SLinkedList.h.

```
#ifndef ILIST_H
#define ILIST_H
using namespace std;
template<class T>
class IList{
public:
    virtual void    add(T e)=0; // for adding at the end
    virtual void    add(int index, T e)=0; // for adding at specific index
    virtual T       removeAt(int index)=0; //remove at specific index
    virtual bool    removeItem(T item)=0; // remove item
    virtual T&      get(int index)=0; // search for item
    virtual bool    empty()=0;
    virtual int     size()=0;
    virtual void    clear()=0;
};
```

IList.h

```
#include IList.h
template <class T>
class SLinkedList : public IList<T>
{
….. //some implementation
public:
    void add(T e);
    void add(int index, T e);
    T removeAt(int index);
    bool removeItem(T item, void (*removeItemData)(T) = 0);
    bool empty();
    int size();
    void clear();
    T &get(int index);
}
template <class T>
void SLinkedList<T>::add(T e)
{
    Node *node = new Node(e, tail);
    tail->next->next = node;
    tail->next = node;
    count += 1;
}
template <class T>
T &SLinkedList<T>::get(int index)
{
    Node *prev = head;
    int cnt = -1;
    while (cnt < index - 1)
    {
        prev = prev->next;
        cnt++;
    }
    return prev->next->data;
}
```

```
template <class T>
T SLinkedList<T>::removeAt(int index){
Node *prev = head;
    Node *cur = head->next;
    int idx = 0;
    while (idx < index){
        prev = prev->next;
        cur = cur->next;
        idx++;
    }
    prev->next = cur->next;
    cur->next = 0;
    count--;
    T backup = cur->data;
    delete cur;
    if (index == count - 1)
        tail->next = prev;
    return backup;
}
template <class T>
void SLinkedList<T>::add(int index, T e){
    if ((index < 0) || (index > count))
        throw std::out_of_range("The index is out of range!");
    Node *prev = head;
    int ite = -1;
    while (ite < index - 1) {
        prev = prev->next;
        ite += 1;
    }
    Node *current = prev->next;
    prev->next = newNode;
    newNode->next = current;
    if (index == count)
        tail->next = newNode;
    count += 1;
}
…. Some other implementations
```

SLinkedList.h

TOPIC II: OBJECT-ORIENTED PROGRAMING BETWEEN JAVA AND C++

C++ is a both procedural and object-oriented programming language while Java is the object-oriented programming language.

We can outline the comparison by below table:

Properties	C++	Java
Object-oriented?	Is a procedural and object-oriented language	Is a completely object-oriented language
Access specifiers	Public, protected, private. Default is private if not declared. Protected: cannot be accessed outside unless it is derived class or accessed by friend class. Public: available for everyone. Private: cannot be access outside unless friend class.	Public, protected, private and default. The default is "default". Default: can access anything within class. Protected: can access inside, can be accessed outside if it is derived class. Private: only access inside class. Public: available for everyone
Root hierarchy	C++ there is no such root hierarchy. C++ supports both procedural and object-oriented programming; therefore, it is called a hybrid language.	Java is a pure object-oriented programming language. That's. Why It follows single root hierarchy.
Operator overloading	C++ allow operator overloading. E.g.: << >> + - * == -= += *, …	Java does not allow operator overloading
Multiple inheritance	C++ supports. One class can be inherited from many class	Java does not support
Interface	C++ does not have interface	Java supports interface to control the behavior of classes
Leaf class	C++ does not have	Java has leaf class which has "final" flag for a class, then no other classes can inherit from it.
Friend class	C++ supports friend class	Java does not support friend class
Organization inside class	C++ can group attributes or methods with the same access specifier. C++ can declare prototypes inside .h files and implementation in .cpp files	Java require declare the access specifier first unless it will set to "default" Java declare and define methods inside one file .java
Call back to base class	<class_name>::method()	super.method()
Call base class constructor	Not allowed	Allowed to call constructor of class in another constructor with this(parameter)
Inheritance type	Private, protected, public	Only public
Pure virtual methods	Has virtual pure method with syntax: return type name_method() = 0	Not exist
Virtual class	Has virtual class and its syntax is "virtual"	Has virtual class, its syntax is"abstract"

Virtual methods	Syntax is “virtual”	Syntax is “abstract”
Multithreaded support	No	Yes, has syntax “synchronized”
Main function	Main does not belong to any class; it is a static function	Main is inside a class because Java is an OOP language
Memory management / Destructor/ Object management	C++ has destructor, but programmers must free the allocated segments before termination	Java does not have destructor, but it has the Garbage Collection (GC), whenever an object is not in use, it will retain the memory.
Pointers	C++ supports pointers	Only limited support for pointers
Parameter passing	Call by value, reference	Call by value
Structure	Supports structure	Does not support structure

To conclude, both programming languages have its pros and cons. We cannot conclude which language is the better because depends on the situation we can use specific language for specific task for efficiency. Java is better at portability, while C++ us platform dependent. C++ is nearer to computer code, so it is stricter than Java in memory management.

TOPIC III: OBJECT-ORIENTED PROGRAMING vs FUNCTIONAL PROGRAMING

FUNCTIONAL PROGRAMMING	OBJECT-ORIENTED PROGRAMMING
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

Example code of Immutable and Mutable data in Haskell and C++ in terms of implementing QuickSort, respectively. Also Haskell implementation avoids side effects:

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

```
void quickSort(int arr[], int left, int right) {
  int i = left, j = right, tmp;
  int pivot = arr[abs((left + right) / 2)];
  while (i <= j) {
    while (arr[i] < pivot) i++;
    while (arr[j] > pivot) j--;
    if (i <= j) {
      tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
      i++; j--;
    }
  }
  if (left < j) quickSort(arr, left, j);
  if (i < right) quickSort(arr, i, right);
}
```

Example code of factorial in Haskell and C++, respectively, to show recursion difference:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

```
template <int N>
struct Fac{
  static int const value= N * Fac<N-1>::value;
};
template <>
struct Fac<0>{
  static int const value = 1;
};
```

Example code of list manipulation in Haskell and C++:

```
mySum []     = 0
mySum (x:xs) = x + mySum xs
```

```
template<int ...>
struct mySum;
template<>
struct mySum<>{
  static const int value= 0;
};
template<int head, int ... tail>
struct mySum<head,tail...>{
  static const int value= head + mySum<tail...>::value;
};
int sum= mySum<1,2,3,4,5>::value;  // 15
```

Example code of pattern matching in Haskell and C++:

```
mult n 0 = 0
mult n 1 = n
mult n m = (mult n (m - 1)) + n
```

```
template <int N, int M>
struct Mult{
static const int value= Mult<N, M-1>::value + N;
};
template <int N>
struct Mult<N, 1> {
static const int value= N;
};
template <int N>
struct Mult<N, 0> {
static const int value= 0;
};
```

For pros, functional programming provides the advantages like efficiency, nested functions, free from side-effects, parallel programming in ease, lazy evaluation. The function is teared down to smaller parts and each part execute tasks. The function can be easily invoked and reused at any points. It allows for very modular and clean code that all works together in harmony.

Also, OOP provides the ability to encapsulate data from outsiders. OOP helps us model real-world design at ease. This allows for a good transition from requirements to code that works like the customer or user wants it to.

Cons of functional programming…. it really takes a different mindset to approach your code from a functional standpoint. It's easy to think in object-oriented terms because it is similar to how the object being modeled happens in the real world. Functional programming is all about data manipulation. Converting a real-world scenario to just data can take some extra thinking.

Similarly, there are a few problems with object-oriented programing. Firstly, it is known to be not as reusable. Because some of your functions depend on the class that is using them, it is hard to use some functions with another class. It is also known to be typically less efficient and more complex to deal with. Plenty of times, some object-oriented designs are made to model large architectures and can be extremely complicated.

In conclusion:

Object-oriented languages are good when we have **a fixed set of operations on things**, and as out code evolves, we 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 we have **a fixed set of things**, and as our code evolves, we 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.

Both Functional programming and object-oriented programming uses a 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. The object-oriented programming is widely used by the programmers and successful also.

In Object-oriented programming, it is 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 always requires a new object to execute functions and it takes a lot of memory for executing the applications.

Finally, to conclude, it is always up to the programmers or developers to choose the programming language concept that makes their development productive and easy.