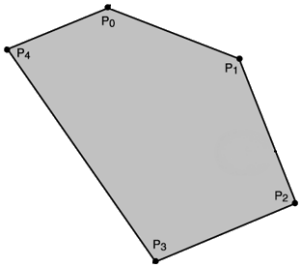
Final Exam - OOP Concepts using C++

Due: 12-08-2021 (Wednesday @ 3:30 p.m.)

Study Guide

Applying OOP Concepts

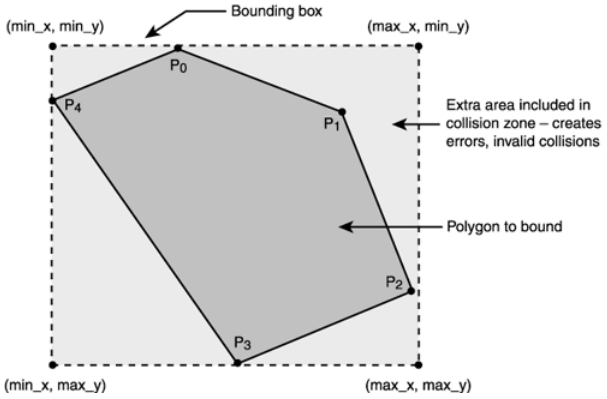
- A polygon is a 2D shape made up of 3 or more sides.



- Each side can also be thought of as a line with a beginning point (x_1,y_1) and an ending point (x_2,y_2).
    - For example, the polygon above has Points: P_0, P_1,P_2,P_3,P_4
    - It also has Lines: (P_0, P_1),(P_1, P_2),(P_2,P_3),(P_3,P_4),(P_4,P_0).
- The length of a line (or distance between two points can be calculated using the following formula:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Remember that:
    - Square root is `sqrt(some number)`
    - Exponentiation is `pow(2,3)` or 2^3.
- A bounding box of a polygon can be found by finding the 4 extreme `x,y` values



General

- You are going to write 3 class definitions
- Each class should have methods to set / get the data members of the class.
- Each class definition should build on the previous class.
- Do not implement any methods.

Applying Many Concepts to One Class

Given:

```
class Rectangle{
    float width;
    float height;
};
```

- Write an appropriate default constructor setting `width` and `height` to zero and an overloaded constructor that would allow the `width` and `height` to be set by the user when an instance of `Rectangle` was created.
- Add the appropriate code to the `Rectangle` class that would allow a `Rectangle` to be printed like so:

```
Rectangle R1(3.4,5.8);

cout<<R1<<endl;

// Would print: {w:3.4,h:5.8}
```

- Write a code snippet that would allow two Rectangles to be added together like so:

```
Rectangle R1(3.6,5.0);
Rectangle R2(2.4,7.0);

R1 = R1 + R2;

cout<<R1<<endl;

// Would print: {w:6.0,h:12.0}
```

- Add a property to the `Rectangle` class that would keep track of how many rectangles have been created during the life of your program.
    - Add one to `count` every time a rectangle is created.
    - Subtract one from `count` when a rectangle is destroyed.
- Don't forget some of the additional things needed when initializing or accessing static data members.
- Write a copy constructor for the rectangle class and overload the assignment operator.

Applying Concepts Again

> Same kind of question, but from a different angle.

You are writing the rudimentary beginnings of a government tracking system. Don't get excited, you're just a peon who does grunt work. For now your employers need you to simply write a class that represents a `location`. They have some general requirements for you and expect you to use all your OOP knowledge in creating this class. You do not have to implement each and every method, only the ones you are asked to. Start with the class definition only, then implement the methods you are asked to outside of the actual definition.

The location holds 3 values: `x,y,z` similar to `(3.4, 6.7, 1000)`

- `x` = x coord
- `y` = y coord
- `z` = altitude rounded to nearest foot

They would like the following capabilities:

- Keep a total count of every location ever created.
- Keep a count of current locations that are in use.

Creating New Locations:

- Create a new empty location with values 0,0,0.
- Create a new location with values passed in.
- Create a new location that is copied from another location.

Example:

```
Location L1;
Location L2(123,45,1234);
Location L3 = L2;
```

Manipulating Existing Locations:

`Set` a location by setting (overwriting) each of the 3 values.

Example:

```
L1.Set(123,23,1000)
L2 = Location(123,23,1000)
L3 = L2
```

`Move` a location by passing in delta values that would be added to the existing values.

Example:

```
Location L1(50,60,1000)
Location L2(10,10,100)
L1.Move(5,5,0); // results in (55,65,1000)
L1 = L1 + L2;   // results in (65,75,1100)
```

Find the `Distance` to another location.

```
Location L1(4.5,5.6,1000)
Location L2(10,10,100)
float distance = L1 / L2;
```

- Find the `Bearing` from one location to another.

```
Location L1(4.5,5.6,1000)
Location L2(10,10,100)
float bearing = L1 -> L2;
```

Displaying a Location

Printing a location would look like: `[x,y,z]` or `[123,456,10]`

Overloading Operators

> Note from editor: I will not now or on the exam ask you to implement a method that I couldn't easily assume you had the background knowledge to get it done. Having said that check out this site: https://www.movable-type.co.uk/scripts/latlong.html. It has all the mathematical functionality you need when dealing with location data. Most importantly, check out the two class (written in javascript) at the bottom of the page. Notice anything about the `Dms` class? (all the static methods). Good real world example of two classes working together.

Implement both bearing and distance operator overloads. Don't actually do the calculations, just get the function skeletons written.

Inheritance

- What's the purpose behind inheritance?
- Is it always the answer? Or does composition and/or aggregation have something to say about that?
- How does inheritance (or composition) help eliminate duplicate code?
- How does inheritance (or composition) make your code more stable or reliable?

Encapsulation

Does the snippet below give us a good example of encapsulation? Explain why either way.

```
struct Fraction {
    int numerator;
    int denominator;

    void set(int n,int d)
    {
        numerator = n;
        denominator = d;
    }

    void print()
    {
        cout<<numerator<<"/"<<denominator<<endl;
    }
};
```

Eval Code

```
class Vehicle {
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};
class FourWheeler {
public:
    FourWheeler() {
        cout << "This is a 4 wheeler Vehicle" << endl;
    }
};
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
{
    Car obj;
    return 0;
}
```

Whats the output?

Core Concepts

- Explain / define abstraction (from an OOP standpoint).
- Explain / define encapsulation (from an OOP standpoint).

- Give multiple examples of both to bolster your answers.

Polymorphism

To add two fractions you need to:

- Find a common denominator by finding the LCM (Least Common Multiple) of the two denominators.
- Change the fractions to have the same denominator and add both terms.
- Reduce the final fraction obtained into its simpler form by dividing both numerator and denominator by the largest common factor.

Assume you have the following class that has all of those listed methods implemented for you.

```
class fraction{
    int numerator;
    int denominator;
    fraction reduce(fraction f);
    int  lca,int b);
public:
    fraction(int n,int d);
    void setumerator(int n);
    void setDenominator(int d);
};
```

Overload the + sign to add two fractions. You can assume your defining your method inline. Assume all the methods above are implemented. Just write the overloaded method ... nothing else.

Protection Mechanisms

- All class members declared as _____ will be available to everyone.
- This access modifier is similar to one of the other access modifiers, the difference is that the class member declared as _____ are inaccessible outside the class but they can be accessed by any subclass (derived class).
- Only member functions or _____ of another class are allowed to access the private data members of this class.
- The _____ data members of this class can be accessed from anywhere in the program using the dot operator.
- The class members declared as _____ can be accessed only by a method inside the class.

Eval Code

Explain whats going on and if its broke, fix it.

```
class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};

int main() {
    base *p;
    derived obj1;
    p = &obj1;

    p->fun_1();
    p->fun_2();
    p->fun_3();
    p->fun_4();
    p->fun_4(5);

    // Are the three lines below valid? State why or why not.
    derived *p1;
    base p2;
    p1 = &p2;
}
```

Copy Constructor

A **copy constructor** is automatically created by the compiler when needed. Based on the partial class definition below determine if a copy constructor will be created or if one needs to be written. If it needs to be written, do it.

```
class NumberContainer{
    nodePtr *head;
    int size;
public:
    NumberContainer(){
        head = null;
        size = 0;
    }
    void push(int);
    int pop();
};
```

Friends

- What are friends for?
- Write a `friend` example when overloading `ostream`.
- Write a `friend` example that does not involve `ostream` or `fstream` and justify why you need to use friend instead of just using inheritance or composition.

Static

- What is the `static` keyword used for?
- Write a function called `countMe` that counts the number of times it has been called.
- Write a class called `NumObjects` that counts the number of objects in existence. You should assume that each object will be created and destroyed before your program ends. So your count should be equal to the number of existing objects.

More Code

- Whats going on here?
- Is this correct (does it work)?
- Add another derived class using Hamster info. Don't memorize the hamster info for the exam. Just remember what you need to do to derive a sub class from an abstract base class.
-

```
class Rodent{
    private: string Order;
    string Family;
    string Subfamily;
public:
    Rodent(){
        Order = "Rodentia";
        Family = "Muridae";
    };
    virtual string getOrder(){ return Order;}
    virtual string getFamily(){ return Family;}
    virtual string getSubfamily()= 0;
};

class Gerbil :private Rodent{
public:
    Gerbil(){
        Subfamily = "Gerbillinae";
    };
    string getSubfamily(){return Subfamily;}
};

int main(){
    Gerbil G;
```

```
    cout<<G.getSubfamily()<<endl;
    return 0;
}
```

- What's the purpose behind inheritance?
- Is it always the answer? Or does composition and/or aggregation have something to say about that?
- How does inheritance (or composition) help eliminate duplicate code?
- How does inheritance (or composition) make your code more stable or reliable?

- Write a simple example in **C++** using class names like A, B, and C ... showing:
  - Simple Inheritance
  - MultiLevel Inheritance
  - Multiple Inheritance
  - Hierarchical Inheritance

- In the snippet below, add a `print` method to `A` that says: "printing in A is super".
- Now **overload** and **override** that method without regard to runtime polymorphism (pointers will never be used) ...

```
class A{
public:
  A(){
    cout<<"construct A"<<endl;
  }
};

class B: public A{
public:
    B(){
      cout<<"construct B"<<endl;
    }
};
```

- If I now told you pointers would be used when accessing your classes. What changes would you need to make and why?

- Be prepared to write examples of and explain method `overloading` **and** method `overriding`.
- Be prepared to write examples of and explain `compile-time` and `run-time polymorphism`.

- How does encapsulation provide security? We talked about hiding or protecting the "implementation" in class, but what does that really mean? Let me show you with code.

```

```

```
// NOT SECURE EXAMPLE of bad ENCAPSULATION
// A little contrived, but not too out of the realm of possibility.
struct Account {
    double balance;

    // local implementation simply reduces the balance
    // and returns a bool for success
    bool withDraw (double amnt){
        if (balance - amnt > 0){
            balance -= amnt;
            return true;
        }
        return false;
    }
};

// Even using "composition" this is not secure.
class BankClient{
    Account account;

    // ...

    bool useATM(double amnt){
        // add to our balance!! Since we have access.
        account.balance += amnt;
        if(account.withDraw(amnt)){
            return true;
        }
        return false
    }

};
```

```
// SECURE ENCAPSULATION EXAMPLE
class Account {
// When balance private! Even derived classes cannot
// gain access to it.
private:
    double balance;
public:
// Balance can only be accessed by public methods
    bool withDraw (double amnt){
        if (balance - amnt > 0){
            balance -= amnt;
            return true;
        }
        return false;
    }
};

// Inheriting Account doesn't give access to private data members
```

```
class BankClient: public Account{
    bool useATM(double amnt){
        balance += amnt; //ERROR!!
        if(withDraw(amnt)){
            return true;
        }
        return false
    }
};
```

- This example shows how that even inheriting from a parent class, that we can lock down (secure) values by keeping them private.
- If we kept `balance` private, suggest how we may still access `balance` in a sub class, but keep it safe.

Given the following class:

```
class Account{
private:
    double account_balance;
protected:
    int account_number;
    string name;
public:
    Account(string n){
        name = n;
        account_balance=0;
        account_number=rand();
    }
};
```

Write 1 or more classes that can access both the `account_number` AND the `account_balance` if possible. If not explain why and make any fixes necessary to `Account`.

What is the output of the following code?

```
class Animals {
public:
    virtual void sound() {
        cout << "This is parent class" << endl;
    }
};

class Dogs : public Animals {
public:
    void sound() {
```

```
        cout << "Dogs bark" << endl;
    }
};

int main() {
    Animals *a;
    Dogs     d;
    a = &d;
    a->sound();
    return 0;
}
```

What can you do to make it print the other statement without changing main? What kind of polymorphism is this? Is this overloading or overriding?

What is the output of the following code:

```
struct A{
  A(){
    cout<<"Running A's constructor"<<endl;
  }
  void print(){
    cout<<"A's print"<<endl;
  }
  void print(string val){
    cout<<"A's overloaded print: "<<val<<endl;
  }

};

struct B: public A{
  B(){
    cout<<"Running B's constructor"<<endl;
  }
  void print(){
    cout<<"B's print"<<endl;
  }
  void print(string val){
    cout<<"B's overloaded print: "<<val<<endl;
  }
};

struct C: public B{
  C(){
    cout<<"Running C's constructor"<<endl;
  }
};
```

```
C c;
c.print()
```

- What is the diamond problem?
- Show bare-bones code examples before virtual inheritance and after.
- Explain with examples showing why there can be ambiguity.

Define

~~You won't necessarily have to write a definition for each one~~, but you will have to know what each one means.

- Abstract Base Class
- Abstraction
- Class
- Class-Variable
- Composition
- Destructor
- Diamond Problem
- Encapsulation
- Friend
- Hierarchical Inheritance
- Inheritance
- Instance-Variable (aka member-variable)
- Interface
- Member-Variable (aka instance-variable)
- Method
- Multilevel-Inheritance
- Multiple-Inheritance
- Object
- Overloading
- Overriding
- Polymorphism
- Private
- Protected
- Public
- Pure Virtual
- Static Member
- Static Method
- Virtual