**SEP200 Review**

1. You have been given the class **Shape** as well as a **main()** function.
   1. Create the child classes **Square**, **Circle**, and **Rectangle**. Create the **constructors**, **destructors**, the **Calculate** area function if necessary, and anything else that is required.
   2. Could the function **CalculateArea()** be moved to protected?

#include <iostream>

using namespace std;

class Shape {

double width;

protected:

double radius;

double length;

public:

Shape() { length = width = radius = 0.0; };

Shape(double l, double w, double r) {

length = l;

width = w;

radius = r;

cout << "Shape::Shape" << endl;

};

double CalculateArea() {//could this be moved to protected

return length \* width;

};

~Shape() {

cout << "Shape::~Shape" << endl;

}

};

int main() {

Circle circle(0, 0, 7.0);

Square square(6.0, 0, 0);

Rectangle rectangle(5.0, 4.0, 0);

cout << "The area of the circle is " << circle.CalculateArea() << endl;

cout << "The area of the square is " << square.CalculateArea() << endl;

cout << "The area of the rectangle is " << rectangle.CalculateArea() << endl;

return 0;

}

See <CalculateArea.cpp> for the answer.

1. You have been given the class **MathBase** as well as partially implemented classes **Rectangle** and **Circle**.
   1. Complete the implementation of the classes **Rectangle** and **Circle**.
   2. In **main(),** create two objects of type **Rectangle** and **Circle** which are accessible through the base class **MathBase**. Initialize **Rectangle** with **(5.0,6.0,7.0)** and **Circle** with **(4.0,8.0).**
   3. In **main(),** create for-next loop that prints out the perimeter, area and volume for the objects of type **Rectangle** and **Circle**.
   4. What do you expect to see when **PrintGetArea()** is called both times?

See the code on the next page.

See <Geometry.cpp> for the answer.

#include <iostream>

using namespace std;

class MathBase {

public:

double GetPerimeter() {

cout << "GetPerimeter Error: You have hit the base class" << endl;

return 0.0;

}

virtual double GetArea() {

cout << "GetArea Error: You have hit the base class" << endl;

return 0.0;

}

virtual double GetVolume() = 0;

};

class Rectangle : public MathBase {

double length, width, height;

public:

Rectangle(double l, double w, double h) {

length = l;

width = w;

height = h;

}

**//Add code here**

};

class Circle : public MathBase {

double radius, height;

const double PI = 3.1416;

public:

Circle(double r, double h) {

radius = r;

height = h;

}

**//Add code here**

};

void PrintGetArea(MathBase& shape) {

cout << shape.GetArea() << endl;

}

void PrintGetArea(Rectangle& shape) {

cout << shape.MathBase::GetArea() << endl;

}

int main(void) {

const int NUM = 2;

**//Add code here**

cout << "Testing PrintGetArea()" << endl;

PrintGetArea(\*math[0]);

PrintGetArea((Rectangle&)\*math[1]);

for (int i = 0; i < NUM; ++i) delete math[i];

return 0;

}

1. Create two functions called **division()**.
   1. The first accepts the **numerator** and **denominator** of the same type. It returns the type of both the numerator and denominator.
   2. The second accepts the **numerator** and **denominator** of differing types. It returns the type of the numerator.
   3. Do the arguments to **division()** have to be references?

For a sample run, see the **main()** function.

#include <iostream>

using namespace std;

division() {//Complete the function prototype

//Complete the function

}

division() {//Complete the function prototype

//Complete the function

}

int main(void) {

double a = 10.0;

double b = 4.0;

int c = 3;

double result1, result2;

result1 = division(a, b);

cout << "result1 is " << result1 << endl;

result2 = division(a, c);

cout << "result2 is " << result2 << endl;

return 0;

}

See <Division.cpp> for the solution.

1. Create a class called **Average**.
   1. This class has three variables. The first variable is called **result** and is of a certain type. The second and third variables are called **runningTotal** and **Marks[N]** and are of the same type. The size of the array **N** is also of the same type. **runningTotal** is the running total of all values inside the **Marks** array which has **N** elements. **result** is the running total divided by the number of marks.
   2. The class **Average** will have two functions: **AddMarks()** which copies an array of marks into **Marks[]**, and **GetAverage()** which calculates the running total of all marks in **Marks[]** and returns **result** as **runningTotal/N**.
   3. In **main(),** create an object of type **Average** where **result** is of type double, **running** total and **Marks[]** are of type int, and **N** is 5 and is of type int.

See **main()** for a sample run.

#include <iostream>

using namespace std;

class Average {

public:

AddMarks() {//Complete the function prototype

//Complete the function

}

GetAverage() {//Complete the function prototype

//Complete the function

}

};

int main(void) {

const int size = 5;

int mark[] = { 77,66,77,78,76 };

//Create an object called **a** of type Average where result will be of type double

//runningTotal and Marks[] of type int, and N is 5 and is of type int.

a.AddMarks(mark, size);

double result = a.GetAverage();

cout << "result is " << result << endl;

return 0;

}

See <Average.cpp> for the solution.

1. Create the class **Student** based on the class **Person**. The class **Person** has been fully implemented. The class **Student** has two variables: a **marks** array of type double, and a **numMarks** of type int. Based on the usage of **Student** in **main(),** complete the class **Student**. This includes:
   1. The **constructors**, the **copy constructor**, the **copy assignment**, and the **destructor**.
   2. Getter and setter functions for **name**. A **SetMarks()** function. A **GetAverage()** function.

//Student.cpp - source code for student

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <string.h>

#include <iostream>

class Person {

char\* name;

protected:

void SetName(const char\* \_name) {

if (name != nullptr) delete[] name;

int len = strlen(\_name) + 1;

name = new char[len];

strcpy(name, \_name);

}

char\* GetName() const { return name; }

public:

Person() {

name = nullptr;

}

Person(const char \*\_name){

int len = strlen(\_name) + 1;

name = new char[len];

strcpy(name, \_name);

}

virtual ~Person() {

if (name != nullptr) delete[] name;

}

};

class Student :public Person {//Complete the class

};

int main(void) {

double marks1[] = { 55.0, 66.0, 66.0, 76.0 };

Student student1("Beth", marks1, 4);

double marks2[] = { 77.0, 66.0, 82.0, 86.0, 90.0 };

Student student2("John", marks2, 5);

Student student3(student2);

student3.SetName("Barry");

Student student4;

student4 = student1;

std::cout << student1.GetName() << " has an average of " << student1.GetAverage() << "%" << std::endl;

std::cout << student2.GetName() << " has an average of " << student2.GetAverage() << "%" << std::endl;

std::cout << student3.GetName() << " has an average of " << student3.GetAverage() << "%" << std::endl;

std::cout << student4.GetName() << " has an average of " << student4.GetAverage() << "%" << std::endl;

return 0;

}

See <Student.cpp> for the solution.

1. You have been given the class declaration and function definitions for the class **Player**. You must implement the operators **operator>>**, **operator<<**, **operator>**, and **operator<** according to how they are used in **main()**. Keep in mind that **operator>>** is a friend to the class **Player**.

For the solution, see <Player.h>, <Player.cpp>, and <CardGame.cpp>.

In the file Player.h:

//Player.h - class declaration for a card player

#include <iostream>

class Player {

std::string name;

int numTokens;

int card;

public:

Player();

Player(std::string, int);

void PlayCard();

int GetCard() const;

int GetTokens() const;

void SetTokens(int num);

void DisplayInfo() const;

friend void operator>>(Player& p1, Player& p2);

};

In the file Player.cpp:

//Player.cpp - function definitions for a card player

#include <iostream>

#include "Player.h"

using namespace std;

Player::Player() {

name="";

numTokens=0;

card=0;

}

Player::Player(std::string nm, int tokens) {

name = nm;

numTokens = tokens;

card = 0;

}

void Player::PlayCard() {

if (numTokens > 0) {//21%10 -> 21/10 = 2 remainder 1. 1 is 21%10

card = rand() % 10 + 1;//random number between 1 and 10

}

else {

card = 0;

}

cout << name << " has played a " << card << endl;

}

int Player::GetCard() const {

return card;

}

int Player::GetTokens() const {

return numTokens;

}

void Player::SetTokens(int num) {

numTokens = num;

}

void Player::DisplayInfo() const {

cout << name << " has " << numTokens << " tokens." << endl;

}

In main():

//CardGame.cpp - main function for the card game

#include <iostream>

#include "Player.h"

using namespace std;

int main(void) {

const int NUM = 3;

Player\* player[NUM];

player[0] = new Player("Tony Soprano", 100);

player[1] = new Player("Pauli \"Walnuts\" Gualtieri", 100);

player[2] = new Player("Furio Guinta", 100);

while (player[0]->GetTokens() < 300 && player[1]->GetTokens() < 300 &&

player[2]->GetTokens() < 300) {

for (int i = 0; i < NUM; ++i) {

player[i]->PlayCard();

}

if (\*player[0] > \*player[1] && \*player[0] > \*player[2]) {

\*player[0] << \*player[1];

\*player[0] << \*player[2];

}

else if (\*player[1] > \*player[0] && \*player[1] > \*player[2]) {

\*player[1] << \*player[0];

\*player[1] << \*player[2];

}

else if (\*player[2] > \*player[0] && \*player[2] > \*player[1]) {

\*player[2] << \*player[0];

\*player[2] << \*player[1];

}

else {

cout << "No clear winner" << endl;

}

for (int i = 0; i < NUM; ++i) {

player[i]->DisplayInfo();

}

cout << endl;

}

cout << "Furio steals the tokens from everyone" << endl;

\*player[0] >> \*player[2];

\*player[1] >> \*player[2];

for (int i = 0; i < NUM; ++i) {

player[i]->DisplayInfo();

}

for (int i = 0; i < NUM; ++i) {

delete player[i];

}

return 0;

}

**DEBUGGING**

The following files are debugging exercises. The answers to each are found at the bottom.

1. Debug problem 1: <Debug1.cpp>
2. Debug problem 2: <Debug3a.cpp>
3. Debug problem 3: <Debug3b.cpp>
4. Debug problem 4: <Debug4a.cpp>
5. Debug problem 5: <Debug4b.cpp>
6. Debug problem 6: <Debug5.cpp>

**OPERATOR OVERLOADING**

1. There is one example with overloading the array operator **[]**. See <College.h> and <ArrayOperator.cpp>.