CSC2040 Data Structures, Algorithms and Programming Languages

Practical 4

Wednesday 1st November 2017

C++ allows us to define class types so that they behave as if they were primitive types. By this we mean that we overload the operator for a user defined class so that you can define the meaning of that operator for a particular class. You will have already seen operator overloading in Practical 3 where two relational operators were overloaded in the `Person` class namely == and != (see below)

```
bool Person::operator==(const Person& per) const {
return ID_number == per.ID_number;
}
```
and

```
bool Person::operator!=(const Person& per) const {
return !(*this == per);
}
```

Program 1

Overload the operators <= and >= in the `Person` class. A Person is less than or equal to another person if their ID_number is less than or equal to each other. A Person is greater or equal to another if their ID_number is greater than or equal to each other. (Hint: you may use the `stoi` function to convert `string` to `int` for making the comparisons). Test these new functions.

Program 2

In the C++ Lecture 3 notes, on pages 2-3, we defined a Stack class. The declaration and code are shown overleaf.

Create a `testStack.cpp` file and type in the following code:

```
void f(Stack &a)
{
   Stack b = a;
}


int main()
{
   Stack s(2);
   s.push(4);
   s.push(13);     f(s);
cout << s.pop() << endl;
return 0; }
```

Run the above test program. Record what value you expect the program outputs and what is actually produced. What happens when the program finishes execution? Make sure you understand what causes the problem.

Create a *copy constructor* for this class to fix the above problem. Then run the same test again.

The Stack class for Program 2:

Header

```
class Stack {
public:
    // constructor
    Stack(int size);
    // destructor
    ~Stack();

    // public members (data and functions)
    void push(int i);
    int pop();

private:
    // private members (data and functions)
    int stck_size;
    int* stck;
    int tos;
};
```

Source code

```
Stack::Stack(int size)
{
    stck_size = size;
    stck = new int[stck_size];
    tos = 0;
}
Stack::~Stack()
{
    delete[] stck;
}
void Stack::push(int i)
{
    if (tos == stck_size) {
        cout << "stack overflow." << endl;
        return;
    }
    stck[tos++] = i;
}
int Stack::pop()
{
    if (tos == 0) {
        cout << "stack underflow." << endl;
        return 0;
    }
    return stck[--tos];
}
```

Program 3

The following question is a modified version of the Shape Class Hierarchy question provided in the text book (pages 220-224). In this question, we will practice the use of protected access specifier, class inheritance and virtual functions as a means of implementing polymorphism - "one interface, multiple methods". You first create an abstract base class to include virtual functions (which may just be functional prototypes) that calculate the area and perimeter of a geometric shape with given data, reads the data from the keyboard and outputs the calculation results. Then you create derived classes in which you redefine these functions, if needed, for specific shapes: Rectangle, Circle and Right Triangle, one derived class for a specific shape. For whichever shape to be calculated, you always use the baseclass pointer, pointing to a derived object of the shape to be calculated, to apply these virtual functions. Therefore your different calculations effectively have just one interface.

The base class containing virtual functions may have the form:

```
class Shape { public:
 virtual void compute_area() = 0;        // a pure virtual function
virtual void compute_perimeter() = 0;   // a pure virtual function
virtual void read_shape_data() = 0;     // a pure virtual function
```

```
virtual void print_result() {          // a virtual function  cout
<< "The area is " << area << endl;
           << "The perimeter is " << perim << endl;
 }
protected:       // protected access specifier
double area, perim;
};
```

An example implementation for the derived class for the Rectangle shape:

```
class Rectangle : public Shape
{ public:
 void compute_area() { area = width * height; }  void
compute_perimeter() { perim = 2 * width + 2 * height; }  void
read_shape_data() {    cout << "Enter width of the rectangle:
";    cin >> width;    cout << "Enter height of the
rectangle: ";    cin >> height;
 }
private:
 int width, height;
};
```

Why can we access the base-class variables `area` and `perim` inside the derived class `Rectangle`?

> These two variables in base-class using *protected which* can be access in the derived class.

As the above example illustrates, the base-class virtual function `void print_result()` is not redefined by the derived class `Rectangle`. Then which version of this function will be used when an object of `Rectangle` calls this function?

> The version in the base-class.

Following the above example, complete the implementations for the Circle and Right Triangle classes.

Program 4

Write a `testShape.cpp` program, which uses the following statements to read data, then to calculate the shape area and perimeter, and finally to output the results, for any of the three shapes for calculation (hence we have achieved the so-called polymorphism).

```
shape_ptr->read_shape_data(); shape_ptr-
>compute_area(); shape_ptr->compute_perimeter();
shape_ptr->print_result();
```

where `shape_ptr` is a base-class pointer, pointing to a derived object of a specific shape to be calculated. Your program should first declare this base-class pointer, and then take the user's selection from the keyboard for which shape (Rectangle, Circle, Right Triangle) to calculate. Based on the user's selection, you allocate an object of the selected derived shape class using new and point `shape_ptr` to this derived object. Then you start to perform the above operations to accomplish the required calculation.