EE2 -12

SOFTWARE ENGINEERING 2: OBJECT ORIENTED SOFTWARE ENGINEERING

1. This is a general question about C++ and Object Oriented Software Engineering.

   a) Explain the concept of *type* in Object Oriented Software Engineering and how it relates to inheritance.

[6]

[bookwork]

In OOP the type of objects is related to the operations that can be performed/called on these objects, as declared in the classes of which the objects are instances.

In this sense inheritance is an "is-a" relationship, implying that the subclass inherits all the operations that are defined in the parent class and is thus expressing a subtype.

   b) Explain what is an abstract class and what is the role of abstract classes in Object Oriented Software Engineering. Illustrate your answer with an example in C++ code.

[7]

[bookwork, programming]

In general an abstract class has at least one "pure virtual" or "abstract" member function, i.e. a member function which is declared but is not implemented. For this reason the class cannot be instantiated and it is meant to be used as an inheritance base for other classes which will then implement the pure virtual member functions in ways that are meaningful for their subtype. For instance consider the following example of a class representing a generic "device with impedance":

```
class ImpedanceDevice{
    public:
        virtual std::complex<double> get_impedance(double omega) = 0;
    ...
};
```

Since there isn't a meaningful way to compute the impedance of such a generic device, the member function is not implemented and it is declared as pure virtual. However other classes (e.g. Resistor, Capacitor etc) may inherit from this class and the relationship of inheritance will express that indeed computing the impedance is a meaningful operation on these subtypes (which will also be in condition to provide a meaningful specific implementation).

c) Explain what the *initialization list* is. Illustrate your answer with an example in C++ code.

[7]

[bookwork, programming]

The initalization list is used in constructors in order to initialize member data using their copy constructors. For instance if we have a class Triangle whose representation is based on three objects of type point:

```
class Triangle {
    public:
        Triangle(const point& ip1, const point& ip2, const point& ip3) :
            p1(ip1), p2(ip2), p3(ip3) {};
        ...

   private:
        point p1;
        point p2;
        point p3;
};
```

d) Explain why the use of the initialization list is important.

[6]

[bookwork]

The initialization list is important mostly for two reasons: constructing an object directly using the copy constructor is more efficient than constructing it using the default constructor and then overwriting it using the assignment operator, moreover for some types the assignment operator is not available (or meaningful).

e) Explain what friend functions are. Illustrate your answer with an example in C++ code.

[7]

[bookwork, programming]

Friend functions are global functions (i.e. not member functions) which are given access to the private member data of a certain class. For instance:

```
class SomeClass {
    public:
    friend void somef(SomeClass& sc);
    ...
    private:

    int data;
```

```
};

void somef(SomeClass& sc){
    sc.data = 0;
    // can access private member data directly
    // because declared as friend function
    // of the class
}

// somewhere else e.g. the main:
SomeClass sc1;
somef(sc1);
```

f) Explain what operator overloading is. Illustrate your answer with an example in C++ code.

[7]

[bookwork, programming]

C++ defines some operators (e.g. +, <<, etc) which are actually short-hands for function calls (e.g. respectively operator+(...), operator<<(...), etc) whose arguments are passed as operands; these functions (sometimes member functions, sometimes global functions) can be overloaded like all other functions. For instance:

```
class SomeClass {
    public:
    friend std::ostream& operator<<(std::ostream& out, const SomeClass& sc);
    ...
    private:

    int data;
};

std::ostream& operator<<(std::ostream& out, const SomeClass& sc){
    out << sc.data;
    return out;
}

// somewhere else e.g. the main:
SomeClass sc;
std::cout << sc << std::endl;
```

2. This question deals with C++ templates and the Standard Template Library.

a) Write a function that, given in input two arguments, returns their sum if they are of a numerical type and their concatenation if they are of type `std::string`.

[5]

[bookwork, programming]

```
template<typename T>
T plusf(T a, T b){
    return a + b;
}
```

b) Consider v of type `std::vector<int>`. Explain the difference between:

```
int a = v[5];
```

and

```
int a = v.at(5);
```

[3]

[bookwork]

In the former case there is no check on the vector bounds. In the latter case a check is performed and an exception is thrown if the index is out of bounds.

c) Consider the following code, where v is of type `std::vector<int>`:

```
int a = v[5];
```

Write an equivalent line using iterators.

[3]

[bookwork, programming]

```
int a = *(v.begin() + 5);
```

d) Consider l of type `std::list<int>` containing at least 10 elements. Write a code fragment printing the second element.

[7]

[bookwork, programming]

```
std::list<int>::iterator it;
it = l.begin();
it++;
cout << *it << endl;
```

e) Mention an example of use case in which `std::list` has advantages in terms of efficiency over `std::vector`.

[3]

[bookwork]

For instance adding one element at the beginning of a `std::list` containing other elements is more efficient than inserting an element at the beginning of a `std::vector` containing other elements.

f) Write a function which prints the content of an `std::vector` (passed by const reference). The function must be able to accept `std::vector` arguments containing elements of any (printable) type. The implementation must use iterators.

[9]

[bookwork, programming]

```
template <typename T>
void printvector(const std::vector<T>& v){
    typename std::vector<T>::const_iterator it;
    for(it = v.begin(); it != v.end(); ++it){
        std::cout << *it << std::endl;
    }
}
```

3. This question deals with C++ exceptions.

a) Consider the following code:

```
double safesqrt(double n){
    if(n < 0){
        return -1;
    }
    return sqrt(n);
}
```

Write a variation using exceptions and explaining why it might be a better choice.

[7]

[bookwork, programming]

This function performs a check on the input, which might be useful. However it works with error codes: in order to use this feature the return value should be checked in turn by the caller in order to determine whether it is a regular result or an error code, and this might be obstrusive. The function could instead use exceptions like this:

```
double safesqrt(double n){
    if (n < 0){
        throw invalid_argument("no real result for sqrt of negative number");
    }

    return sqrt(n);
}
```

And a call to this function would be included somewhere in a try-catch block.

b) Consider the Resource Acquisition Is Initialization idiom.

i) Explain the idiom in words.

[7]

[bookwork]

In the RAII idiom a resource is acquired by the constructor of an associated object and it is released by its destructor. The paradigm works well with exceptions because the destructor of objects instantiated in a function is still called even if the function throws an exception and its execution is therefore interrupted.

ii) Provide an example in C++ code of a class that can be used for a RAII implementation including:

- Suitable declaration of member functions and member data.

[3]

- Implementation of constructor.

[7]

- Implementation of destructor.

[5]

- Implementation of a member function to check the status.

[1]

[bookwork, programming]

```
class Lockdb {

    public:

    Lockdb();
    ~Lockdb();
    bool get_status();

    private:

    bool status;
};

Lockdb::Lockdb() {
```

```
        ifstream infile(".lockdb");
        if(infile.is_open()) {
            status = false;
            //resource is busy
        }
        else{
            ofstream ofile(".lockdb");
            status= true;
            // locked the resource
        }
    }

    Lockdb::~Lockdb() {
        if(status) {
            remove(".lockdb");
        }
    }

    bool Lockdb::get_status() {
        return status;
    }
```