Practice Final Exam

CSI2372 Advanced Programming Concepts with C++

PART A : Short Questions :

A.1) Consider the following definitions of pattern of function:

template <class T, class U> void f (T a, U b) { ... } // I
void f (int a, double b) { .....} // II

- With these statements :

int n;
double x;

what is the correct call for the following:

f (n, x) ;

Solution:
II

A.2) Use the function std::*copy* to copy the content of the array int tab[5]= {1,4,2,7,8}; into std::vector<int> v;

Solution:

```
int tab[5]= {1,4,2,7,8};
std::vector<int> v(5);
std::copy(tab,tab+5,v.begin());
```

A.3) Given a class `Point` with class variables *d_x* and *d_y* . Define the necessary operation to convert a `Point` to an integer (the result should be *d_x*d_x+d_y*d_y*).

```
class Point {
  int d_x, d_y;
public:
  Point ( int _x, int _y ) : d_x(_x), d_y(_y) {}
};
```

Solution:

```
class Point {
  int d_x, d_y;
public:
  Point ( int _x, int _y ) : d_x(_x), d_y(_y) {}
  operator int() const { return d_x*d_x+d_y*d_y; }
};
```

A.4) Implement the assignment operator for class A such that it implements a deep copy strategy.

```
class A {
        int* d_a;
        int d_sz;
public:
        A(int sz=0) : d_sz(sz) {d_a = new int[d_sz];}
        // ...
};
```

Solution:

```
class A {
        int* d_a;
        int d_sz;
 public:
        A(int sz=0) : d_sz(sz) { d_a = new int[d_sz];          }
         // …..
        A& operator=( const A& o ) {
                if ( this != &o ) {
                                d_sz = o.d_sz;
                                delete d_a;
                                 d_a = new int[d_sz];
                                for ( int i=0; i<d_sz; ++i ) d_a[i] = o.d_a[i];
                                }
                return *this;
        }
};
```

A.5) An abstract class is a class that

a. must have no data members.
b. must have no member functions.
c. is privately derived from an abstract base class.
d. has one or more pure virtual functions.
e. none of the above.

A.6) For the following, wich exception handler is suitable?

```
#include <iostream>
using namespace std ;
int main() {
        void f() ;
        try{ f() ;}
        catch (int n){ cout << "except int in main : " << n << "\n" ;}
        catch (...){ cout << "exception other than int in main \n" ;}
        cout << "end main\n" ;
}
void f(){
        try{ float x=2.5 ; throw x ;}
        catch (int n){
                cout << "except int in f : " << n << "\n" ;
                throw ;
        }
}
```

Solution :

catch (...){ cout << "exception other than *int* in main \n" ;}

A.7) The following routine prints the elements of type *int* stored in *std::vector* in-order. Change it to print the elements stored in the container in reverse order.

```
void printOrder( vector<int>&  container ) {
  // loop over the elements
  for (vector<int>::iterator iter = container.begin(); iter != container.end(); ++iter ) {
        cout << *iter << ' ';
  }
  cout << endl;
  return;
}
```

Solution:

```
void printReverseOrder( vector<int>&  container ) {

  for ( vector<int>::reverse_iterator iter = container.rbegin(); iter != container.rend(); ++iter ) {
        cout << *iter << ' ';
  }
  cout << endl;
  return;
}
```

A.8) Write a function *printData()* that will print the *lData* vector elements.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
	vector<int> lData;
	lData.push_back(27);
	lData.push_back(0);
	lData.pop_back();
	printData(lData);
}
```

Solution:

```
#include <iostream>
#include <vector>
using namespace std;

void printData(const vector<int>& inData) {
	for (int i = 0; i < inData.size(); ++i) cout << inData[i] << ' ';
	cout << endl;
}


int main() {
	vector<int> lData;
	int lDatum;
	lData.push_back(27);
	lData.push_back(0);
	lData.pop_back();
	printData(lData);
}
```

A.9) Write an independent function *compare* so the following program will print :
sorted list : {(1, 3), (2, 5), (5, 2)}

```
/*Code*/
class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) { }
};
```

```
int main() {
    Point p1(2, 5), p2(5, 2), p3(1, 3);
    list<Point> lis;
    lis.push_back(p1);
    lis.push_back(p2);
    lis.push_back(p3);
    lis.sort(compare);
    list<Point>::iterator itr;
    cout << "sorted list : {";
    for (itr = lis.begin(); itr != lis.end(); itr++) {
        if (itr == lis.begin()) cout << "(" << (*itr).x << ", "
            << (*itr).y << ")";
        else cout << ", (" << (*itr).x << ", " << (*itr).y << ")";
    }

    cout << "}" << endl;

}
```

Solution :

```
/*Code*/
class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) { }
};

bool compare(const Point& p1, const Point& p2) {
    if (p1.x == p2.x)
        return p1.y < p2.y;
    else
        return p1.x < p2.x;
}

int main() {
    Point p1(2, 5), p2(5, 2), p3(1, 3);
    list<Point> lis;
    lis.push_back(p1);
    lis.push_back(p2);
    lis.push_back(p3);
    lis.sort(compare);
    list<Point>::iterator itr;
    cout << "sorted list : {";
    for (itr = lis.begin(); itr != lis.end(); itr++) {
        if (itr == lis.begin()) cout << "(" << (*itr).x << ", "
            << (*itr).y << ")";
        else cout << ", (" << (*itr).x << ", " << (*itr).y << ")";
    }

    cout << "}" << endl;
}
```

A.10) For the following program, which *sort* algorithm call is correct ?

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool funct(int i,int j) { return (i<j); }

struct myClass {
  bool operator() (int i,int j) { return (i<j);}
} myObject;

int main () {
  int ints[] = {32,71,12,45,26,80,53,33};
  vector<int> myVector (ints, ints+8);
  sort(myVector.begin(), myVector.begin()+4);
  sort(myVector.begin()+4, myVector.end(), funct);
  sort(myVector.begin(), myVector.end(), myObject);
  return 0;
}
```

Solution

All are correct

/*code*/

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool funct(int i, int j) { return (i < j); }

struct myClass {
    bool operator() (int i, int j) { return (i < j); }
} myObject;

int main() {
    int ints[] = { 32,71,12,45,26,80,53,33 };
    vector<int> myVector(ints, ints + 8);    // 32 71 12 45 26 80 53 33

    // uses default comparison (operator <):
    sort(myVector.begin(), myVector.begin() + 4);    // 12 32 45 71 26 80 53 33

    // uses funct for comparison
    sort(myVector.begin() + 4, myVector.end(), funct);    // 12 32 45 71 26 33 53 80

    // uses myObjcet
    sort(myVector.begin(), myVector.end(), myObject);    // 12 26 32 33 45 53 71 80

    return 0;
}
```

PART C: PROGRAMMING QUESTIONS:

Considering the following `List` class, which makes it possible to handle "linked lists" in which the nature of the information associated with each "node" of the `List` is not known (by the class).

The *add* function should add, at the beginning of the `List`, an element pointing to the information whose address is provided as an argument (void *). To "explore" the `List`, three functions are provided:

• *first*, which will provide the address of the information associated with the first node in the `List` and which, at the same time, will prepare the process of browsing the `List`;

• *next*, which will provide the address of the information associated with the "next node"; successive next calls should allow you to browse the `List` (without having to call another function);

• *finish*, which will allow you to know if the end of the `List` is reached or not.

C.1) Provide the definition for this `List` class so that it works as requested.

C.2) Consider the given `Point` class.

Create a `List_points` class, derived from both `List` and `Point`, so that it can be used to handle linked lists of points, and in which the associated information is of type `Point`. We must be able to:

• add a `Point` at the beginning of such a `List`;

• have a member function *display* displaying the information associated with each of the points in the `List` of points.

Provide the definition of the `List_points` class.

```
/* structure of a list items*/
struct Element{
        Element * next ;        // pointer to the next element
        void * content ;        // pointer to any object
} ;


/ * List  and Point classes * /
class List{
        Element * beg ;                 // pointer to the first element
public :
        List () ;                       // constructor
        ~List () ;                      // destructor
        void add (void *) ;             // adds an  element at the beginning of the list
        void * first () ;               // position on first element
        void * next () ;                // position on next element
        int finish () ;
} ;
```

```
class Point {
        int x, y ;
public :
        Point (int abs=0, int ord=0) { x=abs ; y=ord ; }
        void display () { cout << " Coordinates: " << x << " " << y << "\n" ; }
} ;
```

Solution:

C.1)

/*code*/

```
#include <stdlib.h>  // for NULL
#include <iostream>
using namespace std;

/*Declaration of List and Point classes */
/* structure of a List items*/
struct Element {
        Element* next;       // pointer to the next element
        void* content;       // pointer to any object
};
class List {
        Element* beg;        // pointer to the first element
        Element* current;    // pointer to current element
public:
        List() {             // constructor
                beg = NULL;
                current = beg; // for protection
        }

        ~List();             // destructor
        void add(void*);     // adds an element at the begining of the List
        /*position on first element*/
        void* first()
        {
                current = beg;
                if (current != NULL) return (current->content);
                else return NULL;
        }

        /*position on next element*/
        void* next() {
                if (current != NULL)
                {
                        current = current->next;
                        if (current != NULL) return (current->content);
                }
                return NULL;
        }

        int finish() { return (current == NULL); }
};
```

```
List::~List()
{
	Element* nex;
	current = beg;
	while (current != NULL)
	{
		nex = current->next;
		delete current;
		current = nex;
	}
}


void List::add(void* thing)
{
	Element* adel = new Element;
	adel->next = beg;
	adel->content = thing;
	beg = adel;
}
```

C.2)

Note that this inheritance leads to introducing, in the `List_points` class, two given members (x and y) having no interest thereafter. The creation of the requested member functions becomes extremely simple.

Indeed, the function of inserting a `Point` at the beginning of the `List` can be the add function of the `List` class: so we don't even need to over-define it. Regarding the function of displaying all the points in the `List` (which we will also call *display*), it will suffice to call:

- the *first*, *next* and *finish* functions of the `List` class for browsing the `List` of points;
- to the *display* function of the `Point` class for displaying a `Point`.

```
class List_points : public List, public Point {
 public :
	List_points () {}
	void display () ;
} ;

void List_points::display (){
	Point * ptr = (Point *) first() ;
	while ( ! finish() ) { ptr->display () ; ptr = (Point *) next() ; }
}
```

PART D: STANDARD LIBRARY PROGRAMMING QUESTIONS:

Exercise D.1.:
Let's consider again the class `Stack_int` of Practice Mid Term Exam 2 (question **B.3)**. Modify the interface of this class using containers and consider only the functionalities (addition, extraction, deletion, full stack test or empty stack test). Change the corresponding test program in this way.

```
#include <iostream>
using namespace std;
class Stack_int {
	int nmax;		// maximum number of elements of the pile
	int nelem;		// current number of elements of the stack
	int* adv;		// pointer on elements
public:
	Stack_int(int = 20);			// constructor
	~Stack_int();				// destructor
	Stack_int(Stack_int&);			// copy constructor
	void operator = (Stack_int&);		// assignment operator
	Stack_int& operator << (int);		// stacking operator
	Stack_int& operator >> (int&);		// unstacking operator (note int &)
	int operator ++ ();			// full stack test operator
	int operator -- ();			// empty stack test operator
};

/* Question a) the constructor */
Stack_int::Stack_int(int n) {
	nmax = n;
	adv = new int[nmax];
	nelem = 0;
}

/* Question b) the destructor */
Stack_int::~Stack_int() {
	delete adv;
}

/* Question c) the copy constructor */
Stack_int::Stack_int(Stack_int& p) {
	nmax = p.nmax; nelem = p.nelem;
	adv = new int[nmax];
	int i;
	for (i = 0; i < nelem; i++)
		adv[i] = p.adv[i];
}

/* Question d) the assignment operator */
void Stack_int::operator = (Stack_int& p) {
	cout << "*** Attempt to allocate between stacks - STOP execution ***\n";
	exit(1);
}
```

```
/* Question e) the stacking operator:*/
Stack_int& Stack_int::operator << (int n) {
	if (nelem < nmax) adv[nelem++] = n;
	return (*this);
}

/* Question f) the unstacking operator */
Stack_int& Stack_int::operator >> (int& n) {
	if (nelem > 0) n = adv[--nelem];
	return (*this);
}

/* Question g) the operator++ to test if the stack is full  */
int Stack_int::operator++ () {
	return (nelem == nmax);
}

/* Question h) the operator-- to test if the stack is empty  */
int Stack_int::operator-- () {
	return (nelem == 0);
}

/* Example of main */

int main() {
	void fct(Stack_int);
	Stack_int pile(40);
	cout << "full : " << ++pile << " empty : " << --pile << "\n";
	pile << 1 << 2 << 3 << 4;
	fct(pile);
	int n, p;
	pile >> n >> p;		// unstack 2 values
	cout << "Top of the stack when fct returns : " << n << " " << p << "\n";
	Stack_int pileb(25);
	pileb = pile;		// assignment attempt
	return 0;
}

void fct(Stack_int pl) {
	cout << "stack top received by fct : ";
	int n, p;
	pl >> n >> p;		// unstack 2 values
	cout << n << " " << p << "\n";
	pl << 12;		// we add one value
}

/*OUTPUT*/
   full : 0 empty : 1
   stack top received by fct : 4 3
   Top of the stack when fct returns : 4 3
    *** Attempt to allocate between stacks - STOP execution ***
```

Solution :

If we are only interested in the functionality (add, extract, delete, test stack full or stack empty) of the `Stack_int` class, we can use the ***stack <int, vector <int>>*** container **adapter (LIFO** type stacks (Last In, First Out (the last element added to the stack will be the first to leave it)). Here, one would expect more simply *stack <int>*. It is a template class, based on a container of a given type (here *vector<int>*) which modifies its interface, both by restricting it and by adapting it to given functionalities, to know here:
- empty stack test (empty function);
- access to the information located at the top of the stack (*top* function): this function does not modify the value of the top;
- deposit of a value on the stack (*push* function);
- deletion of the value located at the top of the stack (*pop* function); note that to truly "pop" a value, you have to make two calls: *top* then *pop*.

This adapter, *vector <int>*, can be based on *vector*, *deque* or *list*. Here we choose *vector*, dictated only by implementation and efficiency details.

It should be noted that the notion of a full stack no longer exists, given the dynamic aspect of this component. Moreover, there is no longer any reason to ban the assignment.

Here is the test program and the *fct* function:

/*code*/

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

int main() {
	void fct(stack<int, vector<int> >);
	stack<int, vector<int> > pile;
	cout << "empty : " << pile.empty() << "\n";	// here, we display a boolean
	pile.push(1); pile.push(2); pile.push(3); pile.push(4);
	fct(pile);
	int n, p;
	n = pile.top(); pile.pop();
	p = pile.top() ; pile.pop() ; // stack 2 values
	cout << "top of the stack when fct returns: " << n << " " << p << "\n";
	stack <int, vector<int> > pileb;
	pileb = pile; // here the assignment works /!!!
}

void fct(stack <int, vector<int> > pl) {
	cout << "top of the stack received by fct : ";
	int n, p
	n = pl.top(); pl.pop(); p = pl.top(); pl.pop(); // we stack 2 values
	cout << n << " " << p << "\n";
	pl.push(12); // we add one
}
```

/*OUTPUT*/

empty : 1
top of the stack received by fct : 4 3
top of the stack when fct returns: 4 3

Exercise D.2.:
Consider the following `Vect` class which makes it possible to represent "dynamic vectors", ie whose dimension may not be known during compilation. More precisely, provision will be made to declare such vectors by an instruction of the form:
Vect t(exp);
in which *exp* denotes any expression (of type integer).
This class has the following operators:
• [] for access to one of the components of the vector, and this both within an expression and to the left of an assignment (but the latter situation should not be allowed on "constant vectors");
• <<, such that flow << v sends the vector v on the indicated flow, in the form:
<integer1, integer2, ..., integer>

- Appropriately over-define the *operator []* so that it allows to access elements of an object of a `Vect` type as one would do with a classical array. We will make sure that there is no risk of an index "overflow".
- Over-define the operator <<.

We will not try to solve the problems posed possibly by the assignment or transmission by value of objects of the `Vect` type.

```
#include <iostream>
#include <vector>
using namespace std;

/* Vect class */

class Vect : public vector<int> {
public:
       Vect(int n) : vector<int>(n) {}     // essential !
       int& operator [](int);
       friend ostream& operator << (ostream&, Vect&);
};
```

Solution :
The *[]operator* is over-defined in the `vector<int>` class, so it provides a more concise writing for direct access to an element. If v is an object of type `vector<int>`, the notation *v[i]* is equivalent to ** (v.begin () + i)*.
However, this *operator []* is not protected against index overflows. The problem can be solved by using, instead, the ***at*** member function which raises a standard `out_of_range` exception on index overflow.

/*code*/

```
#include <iostream>
#include <vector>
using namespace std;
class Vect : public vector<int> {
public:
	Vect(int n) : vector<int>(n) {}	// essential !
	int& operator [](int);
	friend ostream& operator << (ostream&, Vect&);
};

/*Definition of operator [] */
int& Vect :: operator [] (int i) {
	return (*this).at(i);
		/* We could also seek to modify the value of i by : if ( (i<0) ||
		(i>=(*this).size()) ) i=0 ;*/
}

/*Definition of operator << */
ostream& operator<< (ostream& output, Vect& v) {
	output << "<";
	vector<int>::iterator iv;
	for (iv = v.begin(); iv != v.end(); iv++) output << *iv << " ";
	output << ">";
	return output;
}


/*Example of main */
int main() {
	Vect ve(1);
	ve.push_back(2);
	ve.push_back(3);
	ve.push_back(1);
	cout << ve << "\n";
	ve.pop_back();
	cout << ve << "\n";
	return 0;
}


/*OUTPUT*/
<0 2 3 1 >
<0 2 3 >
```