Practice Final 2020

Advanced Programming Concepts with C++ (University of Ottawa)

Practice Final Exam

CSI2372 Advanced Programming Concepts with C++

FAL 2020

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) ;
```

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;

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) {}
};
```

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];
        }
        // ...
};
```

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 ;
        }
}
```

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;
}
```

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);
}
```

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;

}
```

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;
}
```

PART B : PROGRAMMING QUESTIONS:

The `Bit_array` class below allows to manipulate bit arrays (arrays in which each element can only take one of the two values 0 or 1). The size of an array (the number of bits) will be defined when it is created (by an argument passed to its constructor). We declare the following operators:
• + =, such that t + = n sets the bit of rank n of array t to 1;
• - =, such that t- = n sets the bit of rank n of array t to 0;
• [], such that the expression t[i] provides the value of the bit of rank i of the t array (we will not provide, here, to be able to use this operator on the left of an assignment, as in t[i] = ...);
• ++, such that t ++ sets all bits of t to 1;
• --, such that t-- sets all bits of t to 0;
• <<, such that stream << t sends the contents of t to the indicated stream, in the form:
<* bit1, bit2, ... bitn *>

The *main* function of the `Bit_array` class, along with the result provided by its execution are given at the end of the declarations to help clarify the functionality of the operators and the class.

- Give the definition of the functions of the `Bit_array` class.

```
/* Bit_array class */
#include <iostream>
using namespace std ;

class Bit_array {
        int nbits ;             // current number of array bits
        int ncar ;              // number of required characters
        char * adb ;            // address of the location containing the bits
public :
        Bit_array (int = 16) ;          // constructor
        Bit_array (bit_array &) ;       // copy constructor
        ~Bit_array () ;                 // destructor
        // binary operators
        Bit_array & operator = (Bit_array &) ;      // assignment operator
        int operator [] (int) ;                     // bit value
        void operator += (int) ;                    // activation of a bit
        void operator -= (int) ;                    // deactivation of a bit
        // sending to flot
        friend ostream & operator << (ostream &, Bit_array &) ;
        // unary operators
        void operator ++ () ;           // setting to 1
        void operator -- () ;           // setting to 0
        void operator ~ () ;            // complement to 1
} ;

int main () {
        Bit_array t1(34) ;
        cout << "t1 = " << t1 << "\n" ;
        t1 +=3 ; t1 += 0 ; t1 +=8 ; t1 += 15 ; t1 += 33 ;
        cout << "t1 = " << t1 << "\n" ;
        t1-- ;
```

```
        cout << "t1 = " << t1 << "\n" ;
        t1++ ;
        cout << "t1 = " << t1 << "\n" ;
        t1 -= 0 ; t1 -= 3 ; t1 -= 8 ; t1 -= 15 ; t1 -= 33 ;
        cout << "t1 = " << t1 << "\n" ;
        cout << "t1 = " << t1 << "\n" ;
        Bit_array t2 (11), t3 (17) ;
        cout << "t2 = " << t2 << "\n" ;
        t2 = t3 = t1 ;
        cout << "t3 = " << t3 << "\n" ;
}

/* OUTPUT*/
    t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 *>
    t1 = <* 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 *>
    t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 *>
    t1 = <* 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 *>
    t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 *>
    t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 *>
    t2 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 *>
    t3 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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" ; }
} ;
```

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 ***
```

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&);
};
```