שאלה 1

א)

**template**<**class** **T**, **class** **iterator**, **class** **checker**>

std::set<**T**> **makeSet**(**const** **iterator** begin,**const** **iterator** end, **const** **checker** check){

std::set<**T**> set;

**for**(**iterator** divider = begin; divider != end; ++divider){

**if**(check(begin, divider, divider, end)){

set.insert(\*divider);

}

}

**return** set;

}

הסבר: בקריאה ל check, הדיווידר השמאלי מיוחס לסוף האוסף הראשון, והימני כתחילת האוסף השני.

ב)

**#include** <set>

**#include** <iostream>

**class** Checker{

**public**:

**bool** **operator ()**(**int**\* first\_begin, **int**\* first\_end,

**int**\* second\_begin, **int**\* second\_end) **const**{

**if**(first\_begin == second\_begin) **return** **true**;

**for**(**int**\* left\_side\_ptr = first\_begin; left\_side\_ptr != first\_end;

++left\_side\_ptr){

**if**(\*left\_side\_ptr > \*second\_begin) **return** **false**;

}

**return** **true**;

}

};

**template**<**class** **T**, **class** **iterator**, **class** **checker**>

std::set<**T**> **makeSet**(**const** **iterator** begin,**const** **iterator** end, **const** **checker** check){

std::set<**T**> set;

**for**(**iterator** divider = begin; divider != end; ++divider){

**if**(check(begin, divider, divider, end)){

set.insert(\*divider);

}

}

**return** set;

}

**int** **main**(){

**using** **namespace** std;

**int** array2[] = {6,5,4,3,2,1};

**int**\* begin, \* end;

begin = array2;

end = array2 + 6;

Checker checker;

std::set<**int**> int\_set = makeSet<**int**,**int**\*,Checker>(begin, end, checker);

cout<<int\_set.size()<<**endl**;

**return** 0;

}

שאלה 2

א)

**#include** <iostream>

**using** **namespace** std;

**class** A {

**public**:

**virtual** **void** **print**() **const** { cout << "a" << **endl**; }

**virtual** **~A**(){}

};

**class** B : **public** A{

**public**:

**void** **print**() **const** { cout << "b" << **endl**; }

};

**void** **f**(**const** A& a) {

a.print();

}

**int** **main**(){

B b;

f(b);

**return** 0;

}

ב)

**#include** <iostream>

**using** **namespace** std;

**class** A {

**public**:

**virtual** **void** **print**() **const** { cout << "a" << **endl**; }

**virtual** **~A**(){}

};

**class** B : **public** A{

**public**:

**void** **print**() **const** { cout << "b" << **endl**; }

};

**void** **f**(**const** A a) {

a.print();

}

**int** **main**(){

B b;

f(b);

**return** 0;

}

הסבר: מה שהשתנה הוא שכרגע בעת הקריאה ל f נוצר עותק של הארגומנט b. מכיוון שלא הוגדר אופרטור העתקה ולבטח אינו עובר בפולימורפיזם למחלקה היורשת, נקרא ה copy c'tor הדיפולטיבי של מחלקת האב A.  
בדומה למנגנון ה slicing, גם כאן יועתק רק גודל הזיכרון מן המשתנה המתאים לגודל של המחלקה A, וזה קורה משום שהזיכרון מסודר ברצף, קודם החלקים עבור מחלקת האב ואח"כ עבור המחלקה היורשת.

ג)

**#include** <iostream>

**using** **namespace** std;

**class** A {

**public**:

**virtual** **void** **print**() **const** { cout << "a" << **endl**; }

**virtual** **void** **print**() { cout << "c" << **endl**; }

**virtual** **~A**(){}

};

**class** B : **public** A{

**public**:

**void** **print**() **const** { cout << "b" << **endl**; }

};

**void** **f**(A& a) {

a.print();

}

**int** **main**(){

B b;

f(b);

**return** 0;

}

הסבר: החזרת הרפרנס בחתימת הפונקציה f והסרת המילה const. כמו כן הוספת גרסה לא קונסטית לפונקציה print ב A המדפיסה את התו c. מכיוון שהפונקציה f מקבלת כארגומנט רפרנס הרי שלא תבוצע העתקה ולכן לא ייחתך החלק הרלוונטי של a. על כן נגיע לקריאה להדפסה של הקלאס A, כאמור בגרסתה הלא קונסטית.