שאלון + פתרון

0600603652

המסלול האקדמי המכללה למינהל

# המסלול האקדמי המכללה למינהל

# ביה"ס למדעי המחשב

ברקוד נבחן

ת.ז הסטודנט: _____

מספר חדר: _____

מספר נבחן: _____

מספר אסמכתא: _____

**מבחן בקורס: תכנות מונחה עצמים**

**תאריך הבחינה:** 07.08.14

**שנת הלימודים:** תשע"ד, **סמסטר:** ב', **מועד:** ב'

**משך הבחינה:** 3 שעות

| **שם המרצה/ים:** | **שם המתרגל/ים:** |
|---|---|
| אליהו חלסצ'י | יוסי עדי |
| | מור ורד |
| אליאב מנשה | חיים שפיר |

**מבנה הבחינה:** הבחינה מורכבת מחלק אחד.

**מספר השאלות הכולל בבחינה:** 6.

**משקל כל שאלה:** בצמוד לכל שאלה

**הוראות לנבחן:**

- אסור השימוש בכל חומר עזר
- יש לענות בגוף השאלון.
- נדרש להחזיר את השאלון.
- לא מצורף נספח לבחינה
- מחברת טיוטה: כן
- מחברת נפרדת לכל שאלה: לא
-

**בהצלחה!!**

## שאלה 1 – Constructors and virtual methods ( 12 נקודות)

נתונות ההגדרות של המחלקות הבאות :

```
class A{
   A* a;
   public:
   A(){
      cout<<"A"<<endl;
      a=this;
      a->m();   // notice!!
   }
   void call_m(){
      //...
      m();
   }
   virtual void m(){
      cout<<"A::m()"<<endl;
   }
   ~A(){
      cout<<"~A"<<endl;
      m();   // notice!!
   }
};
```

```
class B: public A{
   A a; // notice!!
   A* pa;
   public:
   B(){
      cout<<"B"<<endl;
      pa=this;
      pa->m(); // notic!!
   }
   virtual void m(){
      cout<<"B::m()"<<endl;
   }
   ~B(){
      cout<<"~B"<<endl;
   }
};
```

```
int main()
{
   B b;
   b.call_m();
   return 0;
}
```

מהו הפלט של התוכנית? (12 נק')

תשובה:

```
A
A::m()
A
A::m()
B
B::m()
B::m()
~B
~A
A::m()
~A
A::m()
```

**מפתח בדיקה:**

**כל שורה נקודה**

**שאלה 2: ירושה ופולימורפיזם (28 נקודות)**

*טיפ: קראו את כל השאלה עד הסוף לפני שתתחילו לענות. שימו לב מתי משהו הוא סוג של משהו ומתי משהו מכיל משהו, ומתי זה גם וגם.*

נתונה לנו המחלקה SimpleItem המייצגת חפץ שאינו יכול להכיל חפצים אחרים.

א. הגדר\י את המחלקות הבאות – הצהרת המחלקה וה data members שלהן בלבד:

לדוגמא:

```
class B : public A {
    int x,y;
```

* אם לדעתכם זה נדרש, אז ניתן להוסיף מחלקות נוספות וגם לשנות את הגדרת SimpleItem

a. הגדר\י את המחלקה CreditCard המייצגת כרטיס אשראי. אין צורך ב data members (1 נק')

b. הגדר\י את המחלקה HolderItem המייצגת חפץ המכיל חפצים (4 נק')

c. הגדר\י את המחלקה Bag עבור תיק המכיל חפצים, למשל ארנק או עפרון (1 נק')

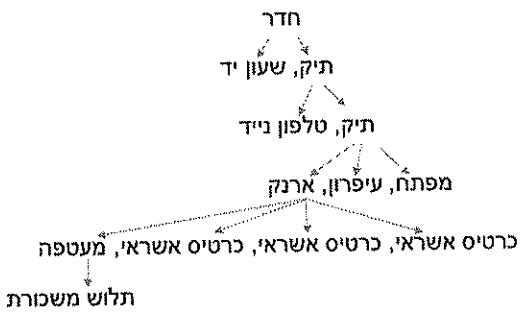
d. הגדר\י את המחלקה Room עבור חדר המכיל חפצים, למשל תיקים או שעונים (2 נק')

ב. ממש\י מתודה במחלקה Room המחשבת כמה חפצים ישנם בחדר. (20 נקודות)

ניתן ואף רצוי להוסיף מתודות עזר במחלקות אחרות.

* אם השתמשתם במערך כלשהו במחלקה כלשהי ניתן להניח שהמחלקה מחזיקה משתנה int size השומר את גודל המערך.

**לדוגמא:** אם בחדר ישנם שני חפצים - שעון יד ותיק ובתוך התיק ישנם גם שני חפצים - תיק קטן יותר וטלפון נייד, ובתוך התיק הקטן יותר ישנם שלושה חפצים – מפתח, עיפרון וארנק, ובתוך הארנק ישנם ארבעה חפצים - שלושה כרטיסי אשראי וגם מעטפה, שבתוכה תלוש משכורת, אז על המתודה להחזיר את הערך 12.




תשובה:

כרטיס אשראי הוא סוג של SimpleItem כי אינו יכול להכיל חפצים אחרים

```
class CreditCard : public SimpleItem {
```

נשים לב ש HolderItem הוא גם חפץ בעצמו וגם יכול להכיל חפצים. לדוגמא תיק הוא חפץ שיכול להכיל עיפרון שהוא חפץ פשוט וגם ארנק שהוא חפץ שמכיל חפצים אחרים (למשל כרטיסי אשראי).

לשם כך עלינו להגדיר תחילה מהו חפץ:

```
class Item {...} // 1 נק'
```

חפץ פשוט וחפץ המכיל חפצים הינם סוג של חפץ:

class SimpleItem : public Item {...}  // 1 נק'

class HolderItem : public Item {...  // 1 נק'

כעת במחלקה HolderItem נוכל להגדיר מערך של חפצים ע"י ה data member:

Item** myItems;  // 1 נק'

נשים לב שכל פוינטר לחפץ במערך הדינאמי של החפצים יכול להיות כל סוג של Item, ובפרט נוכל להצביע על אובייקטים מסוג SimpleItem כמו CreditCard או על אובייקטים מסוג HolderItem כמו תיק, קופסא או ארנק.

נשים לב שהגדרת מערך של SimpleItem לא תספק אותנו כי נרצה להכיל חפץ שגם יכול להכיל בתוכו חפץ. וכן הגדרת מערך של HolderItem תהיה מיותרת כי עבור חלק מהחפצים ברשימה נסתפק ב SimpleItem.

תיק יוגדר פשוט ע"י מחלקה שהיא סוג של HolderItem:

class Bag : public HolderItem {...}

נשים לב שאין צורך ב Data members כי את רשימת החפצים הוא ירש מ HolderItems.

חדר מכיל חפצים אך אינו סוג של חפץ. לכן, החדר לא יורש את Item או את HolderItem המהווה סוג של חפץ... לכן ההגדרה של חדר תהיה:

class Room { // אין ירושה - 1 נקודות

Item** myItems; // 1 נקודות: אפשרות ראשונה:

HolderItem myItems; // או לחילופין אפשרות יפה יותר:

סעיף ב':

נתחיל ממתודות עזר. לא נרצה להוסיף למחלקה Item מתודה וירטואלית שתענה לנו כמה חפצים החפץ הזה מכיל (7- נקודות על טעות זו). זאת מכיוון שלא כל חפץ מכיל חפצים אחרים, רק חפצים מסוג מסוים – HoldeItem מכילים חפצים אחרים ולכן שם נרצה לממש כזו מתודה.

נשים לב שחלק מהחפצים ברשימה הם פשוטים ועליהם להיספר כ 1 ואילו את החפצים האחרים (שאינם פשוטים) יש לשאול כמה חפצים יש להם. לשם כך נידרש להבדיל בין הטיפוסים השונים ואף לבצע המרה נכונה. לכן נממש את המתודה הבאה במחלקה HolderItem כך:

```
int HolderItem :: getNumberOfItems(){
        int count=0;
        for(int i=0;i<size;i++){
                if(typid(*myItems[i])==typeid(SimpleItem))          // 5 נקודות
                        count++;                                    // 1 נקודות
                else { // it is a HolderItem type of object
                        HolderItem* hi = (HolderItem*)myItems[i]; // 5 נקודות
                        count += hi->getNumberOfItems() + 1;        // 2 נקודות
                }
        }
        return count;                                               // 1 נקודות
}
```

כעת נוכל להוסיף את המתודה במחלקה Room שתחשב לנו את מספר החפצים בחדר (6 נקודות). אם הגדרנו את החדר ע"י Items** myItems אז המתודה שלו תיראה בדיוק כמו המתודה לעיל. אם הגדרנו את החדר ע"י HolerItem myItems; אז ניתן פשוט להחזיר את myItems.getNumberOfItems(); ואין לנו כאן קוד כפול ☺. אם טעיתם קודם והחדר ירש את HolderItem אז בסעיף זה מספיק לממש ב HolderItem את המתודה לעיל והחדר פשוט יורש אותה.

## שאלה 3: Operator Overloading – Const (23 נקודות)

נתונה המחלקה הבאה המייצגת תפוח. כל תפוח יכול להחזיר העתק של עצמו ב heap.

```
class Apple{
  //...
  public:
  Apple* clone(){
    // reurns a copy of the apple
    Apple* c=new Apple();
    // copy the apple's contents into c...
    return c;
  }
};
```

כמו כן, נתונה המחלקה הבאה המייצגת שק תפוחים:

```
class AppleBag{
   Apple** myApples;
   int _size;
   void deleteMyApples(){
      for(int i=0;i<_size;i++)
         delete myApples[i];
      delete []myApples;
   }
   public:
   AppleBag(int size){
      myApples=new Apple*[size];
      for(int i=0;i<size;i++)
         myApples[i]=new Apple();
      _size=size;
   }
   AppleBag(){
      myApples=NULL;
      _size=0;
   }
   ~AppleBag(){
      deleteMyApples();
   }
};
```

נתונה פונקציית ה main הבאה:

```
int main() {
   AppleBag ab1(5),ab2(10);
   AppleBag ab3 = ab1+ab2;   // copies all 15 apples to ab3
   AppleBag ab4(3);
   ab4=ab3;    // copies all ab3's apples to ab4
   cout<<ab4<<endl;    // prints: I have 15 apples
   return 0;
}
```

השלימו את הקוד החסר כך שפונקציית ה main תעבוד ללא בעיות. אך שימו לב שה main לא בודקת את כל מקרי הקצה, ואתם נדרשים להקפיד על יעילות, צורת העברת הפרמטרים השונים בדגש על const , ערכי החזרה ושחרור זיכרון בעת הצורך.

תשובה:

נשים לב ש ab1,ab2,ab3 יכולים להיות כולם const.

נשים לב של ab4 יש הקצאת זיכרון של 3 תפוחים שאמורה להימחק לפני שמעתיקים לשם את 15 תפוחי ab3. ונשים לב שמחיקה זו עלולה ליצור לנו באג במקרה של ab3=ab3 לדוגמא.

נשים לב שיש לנו צורך ב CCtor בגלל AppleBag ab3=... וגם בגלל ההחזרה by val של אופרטור+.

נשים לב שיש לנו צורך באופרטור השמה בגלל ab4=ab3 ושפעולתו אינה שונה מזו של ה CCtor ולכן נרצה לחסוך קוד כפול.

נשים לב ש cout<<ab4 לא יכול להיות ממומש ע"י מתודה של AppleBag אלא ע"י אופרטור גלובאלי.

נשים לב שיכולנו גם לכתוב משהו כמו cout<< ab4=ab3 ולכן אופרטור ההשמה לא יחזיר void

ולכן התשובה שלנו תהיה:

```
AppleBag(const AppleBag& ab){   // 1 נקודות
   myApples=NULL;
   *this=ab;   // חיסכון בקוד כפול 1 נקודות
}

// 1          1                    1              1      : נקודות
const AppleBag& operator=(const AppleBag& ab){
   if(&ab!=this){  // 2 נקודות על הבדיקה
      if(myApples!=NULL)
         deleteMyApples();   // 2 נקודות על המחיקה
      _size=ab._size;
      myApples=new Apple*[_size]; // 2 נקודות על העתקת התפוחים
      for(int i=0;i<_size;i++){
         myApples[i]=ab.myApples[i]->clone();
      }
   }
   return *this;   // נקודה 1 על החזרה נכונה
}
```

```
//    1                    1            1          1     : נקודות
AppleBag operator+(const AppleBag& ab) const{
   AppleBag temp;
   temp._size=_size+ab._size;
   temp.myApples=new Apple*[temp._size];
   for(int i=0;i<_size;i++)
      temp.myApples[i]=myApples[i]->clone();
   for(int i=0;i<ab._size;i++)
      temp.myApples[_size+i]=ab.myApples[i]->clone();
   return temp;       // 2 נקודות על מימוש נכון
}

friend ostream& operator<<(ostream& out,const AppleBag& ab);
}; // end of AppleBag

//    1                          1          1            1        : נקודות
ostream& operator<<(ostream& out,const AppleBag& ab){
   out<<"I have "<<ab._size<<" apples";
   return out;
}
```

## שאלה 4 – Templetes (16 נקודות)

א. ממש\י פונקציה כללית בשם count_if העוברת על מבנה נתונים (יהיה אשר יהיה) אשר צוברת את מספר האיברים העונים על תנאי כלשהו (יהיה אשר יהיה) (10 נקודות)

ב. ממש\י object function בשם IsEven שיכול להיות מועבר ל count_if כתנאי הבודק האם האיבר זוגי. (6 נקודות)

תשובה:

כדי שהפונקציה לא תהיה תלויה במבנה נתונים כזה או אחר נצטרך לקבל אובייקטים מסוג Iterator לתחילתו ולסופו של מבנה הנתונים. וכדי שהפונקציה לא תהיה תלויה בתנאי עלינו לקבל object function כפרמטר שמחזיר לנו bool. הפונקציה תיראה כך:

```
template<class iterator, class predicate>          // 2 נקודות
int count_if(iterator begin, iterator end, predicate p){     // 3 נקודות
   int count=0;
   while(begin!=end){      // מימוש נכון 5 נקודות
      if(p(*begin))
         count++;
      begin++;
   }
}
```

ב.

```
class IsEven{
   public:
   template<class T> // 1 נקודות
   bool operator()(const T& t){    // bool (1) operator() (1) const(1) &(1)
      return (t%2==0);      // 1 נקודות
   }
};
```

## שאלה 5 שאלות פתוחות (11 נק')

א. בשאלה 4 מהן ההנחות הסמויות שהנחת על הפרמטרים השונים? (4 נק'

ב. מנה\י 3 יתרונות לשימוש ב templates יחסית לפולימורפיזם (3 נק')

ג. מנה\י 2 יתרונות לפולימורפיזם יחסית ל templates (2 נק')

ד. נכון או לא נכון: אם למשתנה הפרטי שלנו יש setter ו getter פומביים שלא עושים כלום מלבד לאתחל ולהחזיר אותו אז כבר אפשר להפוך את המשתנה הזה לפומבי. (2 נק')

תשובות:

א. =! , *, ++ ל iterator, ו %(int x) ל T

ב. שגיאות יתגלו בזמן קומפילציה וגם הקוד רץ יותר מהר כי זה static binding לעומת ה dynamic binding בפולימורפיזם הקורה בזמן ריצה, קוד מקור קצר יותר כי לא צריך לבנות היררכיית מחלקות.

ג. ניתן לשמור במבנה נתונים אחד משפחה של טיפוסים, הקוד המקומפל לא מתנפח כמספר הקריאות ל template.

ד. כמובן שלא נכון, כי מחר אולי נצטרך להוסיף ניהול. אם המשתנה יהיה פומבי יהיה ניתן לגעת בו בכל מקום בפרויקט ואת הניהול נצטרך להוסיף בכל מקום כזה.

## שאלה 6: ירושה מרובה (10 נקודות)

האם קטע הקוד הבא מתקמפל, אם לא – מדוע?

```
class A{
   public:
   virtual void m(){
      cout<<"A::m"<<endl;
   }
};
class B1: virtual public A{
   public:
   virtual void m(){
      cout<<"B1::m"<<endl;
   }

};
class B2: public A{
};

class D: public B1,B2{
};

int main()
{
   D d;
   d.m();
   return 0;
}
```

תשובה : הקוד לא מתקמפל.

מדוע? מכיוון של D יש שני חלקים של A. לא ניתן לדעת האם לקחת את m מ A של B2 או מ B1.

מפתח בדיקה – 5 נק' על תשובת ה"לא מתקמפל" + 5 נק' על הסבר מדוע.

# בהצלחה