

סדנת תכנות ב- C++

67317

מועד א'.

מרצה: מוטי פריימן

03.02.06

הנחיות:

- משך הבחינה: שעתיים.
- יש לענות על כל שאלות הבחינה.
- ניתן להשתמש בכל חומר עזר כתוב. אין להשתמש בכלי חישוב מכל סוג שהוא.
- כתיבת התשובות תבצע אך ורק על דפי הבחינה. ניתן להעזר במחברות כטייטה, אולם הם לא יבדקו כלל.
- שאלות סגורות: יש להקיף בעיגול את האות בתחילת השורה בה נמצאת התשובה הנכונה (קיימת תשובה נכונה יחידה בכל שאלה סגורה).
- שאלות הדורשות כתיבה: כתבו רק בשורות המיועדות לכתיבה, התשובה תיחשב תקינה במידה של חריגה בשל גודל כתב או סגנון כתיבה.
- יש להקפיד על רישום ברור של מס' ת.ז.

שאלות:

1.

נתונה התכנית הבאה:

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

class X
{
public:
    int *_px;
    X( int init ) { _px = new int; *_px = init; }
    ~X() { delete _px; }
};

void print( X x )
{
    cout<<*(x._px)<<endl;
}

int main()
{
    X x1(15);
    print( x1 );
    X x2(3);
    x2= x1;
    return 0;
}
```

א. (%4) תארו את ניהול הזכרון בתכנית? מוקצה אובייקט בשם x1 על המחסנית. המכיל בתוכו מצביע לאיזור בערימה. עם הקריאה ל print האובייקט מועתק, ברמה הרדודה, כך שבמחסנית של print נוצר אובייקט בשם x המכיל מצביע לאותו מקום בזכרון הערימה שהוצבע על ידי x1. עם שחרור המחסנית של print ישוחרר איזור הזכרון בערימה שהוצבע על ידי x. כשהמחסנית של ה main תשתחרר יהיה נסיון נוסף לשחרור הזכרון באותו מקום – double free.

מצב דומה קורה לגבי X2. נוצר במחסנית של ה main כאשר הוא מכיל מצביע למקום בערימה, לאחר מכן מתבצע אופרטור השוואה דיפולטיבי, המשווה ברמה הרדודה. בתוצאה מכך ההצבעה לאיזור בערימה ש x2 הצביע עליו נאבדת, ומקבלים במקומה הצבעה לאיזור שכבר שוחרר, ומוצבע על ידי x1, כך שיש גם דליפת זכרון של החלק שהוקצה על ידי x2 וגם עוד נסיון לשחרור של איזור בזכרון שכבר שוחרר.

ב. (%5) תקנו את כל הבאגים בתכנית, כך שהתכנית תרוץ באופן היעיל והבטוח ביותר? שינוי של ה prototype של print:

```
void print( const X& x )
```

אופרטור השמה:

```
X& operator=( const X& other) { *_px = *(other._px); return *this;}
```

2.

נתונה התכנית הבאה:

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

class C
{
public:
    C(const C& c) : _s(c._s) {}
    C() : _s("") {}

    string _s;
};
```

```

class C2 : public C
{
public:
    C2( const C2 & c2 ) : _i(c2._i) {}
    C2() : _i(0) {}
    int _i;
};

int main()
{
    C2 c2;
    c2._s= "hello";
    c2._i= 42;
    C2 c3(c2);
    cout << c3._s << " " << c3._i << endl;
}

```

א. (3%) מה יהיה הפלט של התוכנית?

42

ב. (4%) תקן את התוכנית כך ש-c3 יהיה העתק מושלם של c2.

נתקן את ה copy cons

```

C2( const C2 & c2 ) : C(c2) , _i(c2._i) {}

```

3. (8%)

נתונה התכנית הבאה:

```

#include <iostream>
using namespace std;

class MyInt
{
private:
    int _a;
public:
    MyInt () { _a = 0;}
    MyInt (int i) { _a = i;}
    ~MyInt() { /*empty*/ }
};

int main()
{

```

```

MyInt x(5);
MyInt y(3);
cout<<x+y+x<<endl;
}

```

ממשו את האופרטורים החסרים (כולל ה prototype שלהם), כך שהתכנית תעבוד היטב. מותר אך לא הכרחי להוסיף עוד מתודות למחלקה MyInt.

```

MyInt operator+ (const MyInt &other) { MyInt t (_a + other._a); return t;}

```

```

friend ostream& operator<< (ostream &os, const MyInt& a) { os<<a._a<<endl; return os;}

```

הערה: ניתן היה לממש גם רק אופרטור cast ל int ואז החיבור וההדפסה היו של int.

4. (6%)

נתונה המחלקה הבאה:

```

class A
{
protected:
    int *_arr;
    int *_length;
public:
    A (): _arr (new int [1]), _length (new int (1)){}
    A (int length) : _arr (new int [length]), _length (new int (length)){}
};

```

הוסיפו prototype ומימוש של destructor למחלקה:

```

virtual ~A ()
{
    delete _length;
    delete [] _arr;
}

```

5. (5%)

מה יהיה הפלט של התכנית הבאה:

```

#include <iostream>
using namespace std;

```

```

class A
{
public:
    A () { cout<<"A cons, ";}
    ~A () { cout<<"A dest, ";}
};

```

```

class B: public A
{
public:
    B () { cout<<"B cons, ";}
}

```

```

    ~B () { cout<<"B dest, ";}
};

int main()
{
    B b;
}

```

הקיפו בעיגול את הספרה שליד התשובה הנכונה:

1. A cons, B cons, A dest, B dest,
2. B cons, A cons, A dest, B dest,
3. B cons, A cons, B dest, A dest,
4. A cons, B cons, B dest, A dest,

.4

6 (6%)

תקנו את התוכנית הבאה ללא שינוי של main?

```

class Base
{
    int *_base;
public:
    Base (int i) : _base (new int(i)){}
    ~Base () { delete _base;}
};

class Der : public Base
{
    int *_der;
public:
    Der (int i) : Base (i), _der (new int (i)){}
    ~Der () {delete _der;}
};

int main()
{
    Base * b = new Der (5);
    delete b;
}

```

נחליף את ה dest של Base להיות:

```
virtual ~Base () { delete _base;}
```

7

בהתייחס לתכנית הבאה:

```

template <typename T>
class A
{
    T* _pT;
}

```

```

public:
    A (T val): _pT (new T (val)){}
    ~A () {delete _pT;}
};

int main()
{
    int i = 3;
    A <int> a1 (i);
    A <int*> a2 (&i);
    A <A<int> > a3 (a1);
}

```

א. (3%) מהו מספר המחלקות השונות בקוד של התכנית:3.

ב. (3%) מהו הטיפוס של `_pT` בכל מופע של `A` (רשמו את הקוד של המופע ולידו את הטיפוס המתאים של `_pT`).

```

A <int> : int *
A <int*> : int**
A <A<int> > : A <int>*

```

8.

נתון קטע הקוד הבא:

```

#include <iostream>
using namespace std;

int main ()
{
    int a = 3;
    int b = 5;
    const int& c = a;

    int *p1 = &c;           //line no. 1
    const int *p1 = &c;     //line no. 2
    int *const p1 = &c;     //line no. 3

    cout<<"*p1= "<<*p1<<endl;
    a = b;
    cout<<"*p1= "<<*p1<<endl;
}

```

א. (3%) כתבו את מספרי השורות שאינן עוברות קומפילציה מבין השורות הממוספרות:

ב. (4%) בהנחה שהשורות שאינן חוקיות נמחקו, פלט התכנית שהתקבל היה:

```
*p1= 3  
*p1= 5
```

הסבירו בקצרה כיצד יתכן שקבלנו שני פלטים שונים עבור הדפסה של משתנה מסוג `const`?
הגישה דרך `a` אינה מוגנת. `Const` מממש הגנה רק בגישה דרך התוית המוגדרת `const` ולא הגנה על האיזור הפיזי בזכרון, ולכן ניתן לשנות את התוכן של האיזור בזכרון דרך `a` למרות שאי אפשר לשנותו דרך `c` או `*p1`.

9. (8%)

ממשו (כולל כתיבת `prototype`) פונקציית `template` גלובלית `add_vec` אשר תחבר 2 `input iterators` ותכתוב את התוצאה ל-`output iterator` ותחזיר אותו. ה-`main` הבא צריך לפעול כראוי (שימוש לא נכון בפונקציה כמו וקטורים לא באותו הגודל יכול לגרום להתנהגות לא מוגדרת):

```
#include <iostream>  
#include <vector>  
using namespace std;  
  
int main() {  
  
    int in_arr[]={1,3};  
    const int N= sizeof(in_arr)/sizeof(int);  
  
    vector<int> vec;  
    for (int i=0; i<N; ++i) vec.push_back(i);  
  
    int* out_arr= new int[N];  
  
    vec_add(in_arr,in_arr+N,vec.begin(),out_arr);  
  
    for (int i=0; i<N; ++i) cout << out_arr[i] << endl;  
    cout << endl;  
  
}
```

הפלט הצפוי:

```
1  
4
```

```
template<typename In1, typename In2, typename Out>  
Out vec_add(In1 first1, In1 last1,  
            In2 first2, Out res)
```

```

{
    while (first1 != last1)
    {
        (*res++)= (*first1++) + (*first2++);
    }
    return res;
}

```

10.

א. (5%) שכתבו את ה macro הבא כפונקציית template (מותר להניח כי האופרטור > מוגדר על הטיפוסים הנשלחים לפונקציה):

```
#define Max(a,b) ((a)>(b)?(a):(b))
```

```

template <typename T>
inline T Max (T t1, T t2)
{
    if (t1>t2)
        return t1;
    else
        return t2;
}

```

ב. (3%) תארו מקרה של שימוש ב Max כך ששימוש ב macro יצור תוצאה לא צפויה, אולם שימוש בפונקציית template יתן תוצאה נכונה?

למשל שימוש ב

```
Max(a++, b++)
```

ג. (3%) אין יתרון מבחינת זמן ריצה לשימוש ב macro במידה והפונקציה Max מומשה באופן אופטימלי. הסבירו מדוע?

אם פונקציית ה template הוגדרה כ inline והקומפיילר אכן מימש אותה כך, הרי שאין כאן

overhead של קריאה לפונקציה, בדומה לשימוש ב macro, אולם הקומפיילר מספק שירות של

בדיקת טיפוסים וכד'.

11.(6%)

מה יהיה הפלט של התכנית הבאה:


```

#include <iostream>
using namespace std;

class A
{
public:
    A () {cout<<"A cons"<<endl;}
    virtual ~A () {cout<<"A dest"<<endl;}
    virtual void fooA () = 0;
};

class B
{
public:
    B() {cout<<"B cons"<<endl;}
    virtual ~B () {cout<<"B dest"<<endl;}
    void fooB () {std::cout<<"B::fooB"<<endl;}
};

class C: public B, public A
{
public:
    C () {cout<<"C cons"<<endl;}
    virtual ~C () {cout<<"C dest"<<endl;}
    void fooA () {std::cout<<"C::fooA"<<endl;}
    void fooB () {std::cout<<"C::fooB"<<endl;}
};

int main ()
{
    C c;
    c.fooA ();
    c.fooB ();
}
B cons
A cons
C cons
C::fooA
C::fooB
C dest
A dest
B dest

```

12. (5%)

הקיפו בעיגול את התשובה הנכונה:

מנגנון `dynamic_cast` ב `C++` מאפשר:

1. המרה של אובייקטים יורשים למחלקת האב שלהם.
2. המרה של אובייקטים מכל סוג שהוא לאובייקטים מכל סוג שהוא.
3. המרה של מצביעים ומשתני `reference` לטיפוסים פולימורפיים, למצביעים ומשתני

reference לטיפוסים היורשים מהם.

4. המרה של משתני const למשתנים שאינם const.

3.

13. (5%)

הקיפו בעיגול את התשובה הנכונה:

מנגנון ה inline:

1. מאפשר שיפור בזמן ריצה, תוך שמירת עקרון המבניות.
2. הוא בעצם שימוש ב pre-processor המוכר משפת C.
3. אינו מאפשר בדיקת התאמה של types בשונה מקריאה לפונקציה רגילה.
4. מכריח את הקומפיילר לפרוש את הפונקציה הנקראת בקוד עצמו, ובכך תורם לשיפור זמן הריצה.

1. (4) אינה נכונה, כי הגדרת פונקציה כ inline היא רק בגדר המלצה לקומפיילר).

14.

א. (5%) התכנית הבאה אינה מתקמפלת:

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

class A
{
    int *_p;
public:
    A(int i): _p (new int (i)){}
    void print () {cout<<"*_p = "<<*_p<<". "<<endl;}
    virtual ~A () {delete _p;}
};

int main()
{
    const A a (5);
    a.print();
}
```

תקנו את הטעון תיקון כך שהתכנית תתקמפל והפלט שלה יהיה:

*_p = 5.

יש לתקן את הגדרת הפונקציה print:

```
void print () const {cout<<"*_p = "<<*_p<<". "<<endl;}
```

גם הורדת הגדרת ה const מהאובייקט a תתקבל כתשובה נכונה.

ב. (4%) האם התכנית הבאה מתקמפלת?

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

```
class A
{
    int *_p;
public:
    A(const int& i): _p (new int (i)){}
    void setA (const int& i) const {*_p = i;}
    virtual ~A () {delete _p;}
};
int main()
{
    const A a (5);
    a.setA (3);
}
```

כן / לא

כן.

בהצלחה!