TENTAMEN: Objektorienterad programutveckling

Läs detta!

- *Uppgifterna är inte ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad.
- Skriv ditt namn och personnummer på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Programkod skall skrivas i C++ och vara indenterad och kommenterad.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklarationer, parameterlistor etc. får ej ändras.
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1

Välj **ett** alternativ för varje fråga! Garderingar ger noll poäng. Inga motiveringar krävs. Varje korrekt svar ger två poäng.

1. Vilket påstående stämmer om MVC-modellen?
   a. den utvecklades av Ivar Jacobson
   b. den är ett ramverk för konstruktion av användargränssnitt
   c. MVC betyder Multiple Vision Control
   d. modellen har ingenting med OOP att göra
   e. inget av ovanstående

2. Vilken diagramtyp används för att beskriva beteendet hos ett objekt
   a. sekvensdiagram
   b. användningsfallsdiagram
   c. samarbetsdiagram
   d. tillståndsdiagram
   e. klassdiagram
   f. inget av ovanstående

3. Ett affärsobjekt modellerar troligen
   a. ett begrepp i problemdomänen
   b. en styrfunktion i ett börsmäklarsystem
   c. en gränssnittskomponent i ett kundordersystem
   d. inget av ovanstående

4. Vilken aktivitet utvärderar användarnyttan hos ett system
   a. testning
   b. verifiering
   c. validering
   d. inget av ovanstående

5. I vilken/vilka versioner av funktionen `test` används `AutoPointer` på ett korrekt sätt?

```
void f( AutoPointer<int> x ) { ... }
void g( AutoPointer<int> x ) { ... }
AutoPointer<int> h( AutoPointer<int> x ) { (*x)++; return x; }

void test() {    // 1
    AutoPointer<int> p(new int(123)), q;
    q = p;
    f( h( p ) );
    g( p );
}
void test() {    // 2
    AutoPointer<int> p(new int(123)), q;
    f( p );
    g( p );
}
```

forts.

```
void test() {    // 3
    AutoPointer<int> p(new int(123)), q;
    q = p;
    f( q );
    g( p );
}
void test() {    // 4
    AutoPointer<int> p(new int(123)), q;
    q = p;
    f( h( p ) );
    g( q );
}
void test() {    // 5
    AutoPointer<int> p(new int(123)), q;
    q = p;
    f( h( q ) );
    g( q );
}
void test() {    // 6
    AutoPointer<int> p(new int(123)), q;
    f( p );
    q = p;
    g( q );
}
```

a. 3 och 5
b. 2 och 6
c. 1 och 4
d. ingen av 1-6
e. en av 1-6
f. fler än två av 1-6
g. inget av ovanstående

(10 p)

Uppgift 2

I lösningen skall en generisk vektorklass användas, se bilagan.
Ofta finns behov av att beräkna ett värde av alla elementen i en vektor, t.ex. summera eller beräkna produkten av elementen i en heltalsvektor. Beräkningen kan göras genom att successivt ackumulera resultatet element för element, i fallet summering genom addition och för produkt multiplikation. Detta beräkningsmönster kan generaliseras och man brukar då prata om en *ackumulatorfunktion*. En sådan funktion beräknar givet en funktion *f*, en vektor *v* och ett s.k. enhetselement *u* uttrycket

$$accumulate(v, f, u) = \begin{cases} u & (n = 0) \\ f(v_0, f(v_1, ..., f(v_{n-1}, u))) & (n > 0) \end{cases}$$

Observera i vilken ordning vektorelementen behandlas, den saknar betydelse för summa och produkt men kan vara väsentlig för andra operationer. Enhetselementet definierar resultatet om vektorn är tom. Det väljs så att det blir neutralt för operationen, d.v.s. så att $f(x,u) = x$. Om t.ex. *f* är addition är enhetselementet 0 och vid multiplikation 1 eftersom x+0=x och x*1=x.
Ex: Om add(x,y) = x+y och v heltalsvektorn [1,2,3,4,5] så beräknar accumulate(v, add, 0) värdet av add(1,add(2,…,add(5,0))) = 1+2+…+5+0 = 15. Analogt: Om mul(x,y) = x*y beräknar accumulate(v, mul, 1) värdet av mul(1,mul(2,…,mul(5,1))) = 1*2*…*5*1 = 120.

a) Definiera `accumulate` som en generisk funktion i C++. Funktionen skall vara generisk med avseende på vektorelementens typ, parameterfunktionen *f*, samt typen för enhetselementet *u* som skall kunna skilja sig från vektorelementtypen.

(6 p)

b) Definiera m.h.a. `accumulate` funktionen `all` som tar en vektor av booleska värden och returnerar `true` om alla vektorelementen är sanna och `false` annars. Till din hjälp finns funktionen[1]

```
bool and( bool x, bool y ) { return x && y; }
```

(1 p)

c) Definiera m.h.a. `accumulate` funktionen `someOdd` som tar en heltalsvektor och returnerar `true` om minst ett av vektorelementen är udda och `false` annars. *Tips:* definiera en lämplig parameterfunktion liknande `and` ovan.

(3 p)

[1] Denna kan även definieras som en funktionsobjektklass men det saknar betydelse för uppgiften eftersom även vanliga funktioner kan skickas som argument till generiska funktioner med funktionstypparametrar.

Uppgift 3

I lösningen skall en generisk vektorklass samt klassen Pair användas, se bilagan.
En associationslista är en struktur som håller reda på en samling nyckel-värdepar, t.ex. kan ett personnamn vara associerat med ett telefonnummer där namnet representeras som en sträng och telefonnumret som ett heltal. Nycklarna är unika i samlingen, d.v.s. inget namn förekommer mer än en gång, men flera personer kan ha samma telefonnummer. Om man har måttliga krav på effektivitet kan en associationslista enkelt implementeras som en vektor av nyckel/värdepar. Uppgiften går ut på att definiera klassen `AssociationList` genom implementeringsarv från klassen `Vector`. Klassen skall vara generisk med avseende på såväl nyckel- som värdetyp så att den blir så generell som möjligt. Följande operationer skall finnas:

```
// AssociationList class interface: support association lists
//
// Key:   must have zero-parameter constructor,
//        operator= and operator==
// Value: must have zero-parameter constructor and operator=
// CONSTRUCTION: with (a) an integer size only
//
// ******************PUBLIC OPERATIONS*********************
// [ ]                        --> Indexing with key
// void insert( key, value ) --> Associate value with key
// bool empty( )              --> Return true if
//                                AssociationList is empty, false ow
// int size( )                --> Return # elements in AssociationList
```

Indexoperatorn skall kasta ett lämpligt undantag om nyckeln (index) ej finns i listan. Operationen `insert` skall kontrollera om nyckeln redan finns i listan och i så fall uppdatera värdet, annars sätts nyckel/värdeparet in, t.ex. sist. Övriga vektoroperationer som ej finns med i listan ovan skall ej vara synliga. Exempel på användning:

```
AssociationList<String,unsigned long> al;
al.insert( "Lisa", 819234 );
al.insert( "Sven", 142365 );
al.insert( "Allan", 978362 );
al[ "Lisa" ] = 776655;          // nytt nummer
cout << al[ ”Sven” ];           // 142365
cout << al[ ”Lisa” ];           // 776655
cout << al.empty();             // 0
cout << al.size();              // 3
cout << al[ ”Eva” ];            // undantag kastas
```

Tips: I definitionen av klassen kan följande användas för att förenkla koden:

```
typedef Vector< Pair<Key,Value> > PairVector;
```

där `Key` och `Value` är klassens generiska typparametrar. Du kan då använda namnet `PairVector` istället för det krångliga typnamnet `Vector< Pair<Key,Value> >`.

(12 p)

Uppgift 4

a) Vad skriver följande program ut? Motivera!

```
class Base {
public:
    Base( int N ) : X(N) {}
    void F() { cout << "Base::F " << X << endl; }
    virtual void G() { cout << "Base::G " << X << endl; }
    virtual void H() { cout << "Base::H " << X << endl; }
    void I() { G(); }
private:
    int X;
};

class Sub : public Base {
public:
    Sub( int N ) : Base(N), Y(2*N) {}
    void F() { cout << "Sub::F " << Y << endl; }
    void G() { cout << "Sub::G " << Y << endl; }
private:
    int Y;
};

void main() {
    Base B(5), *p;
    Sub S(20);
    p = &S;
    p->F();     // a
    p->G();     // b
    p->H();     // c
    p->I();     // d
    p = &B;
    p->G();     // e
    B = S;
    p->G();     // f
}
```

(6 p)

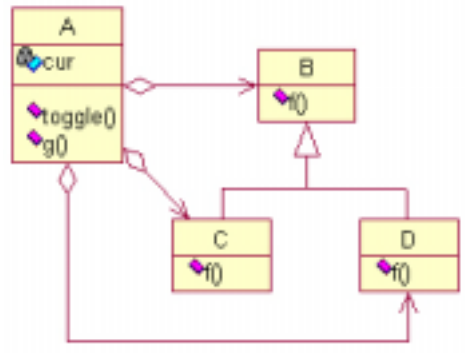
forts.

4 b)
Vilket av klassdiagrammen nedan beskriver modelltekniskt mest korrekt situationen vid polymorfism där ett objekt hanterar andra objekt av olika typ på ett enhetligt sätt? Motivera svaret! En möjlig implementering följer:

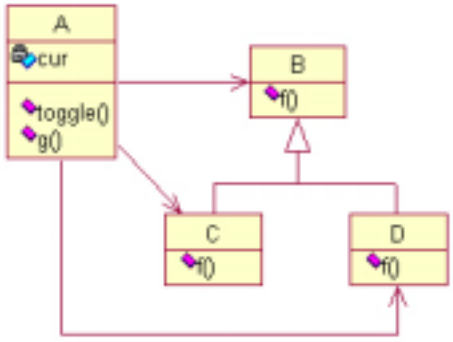
```
class B { public: virtual void f() = 0; };
class C : public B { public: void f(); };
class D : public B { public: void f(); };

class A {
public:
    A() : cur(0) { bp = new B*[2]; bp[0] = new C; bp[1] = new D; }
    ~A() { delete bp[0]; delete bp[1]; delete [] bp; }
    void toggle() { cur = ! cur; }
    void g() const { bp[cur]->f(); }
private:
    B **bp;
    int cur;
};
```

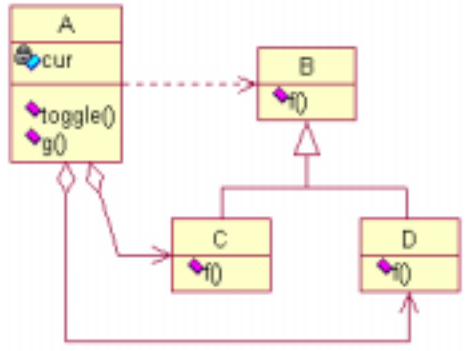
klassdiagram 1



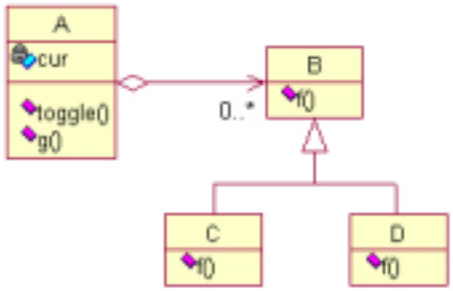
klassdiagram 2



klassdiagram 3



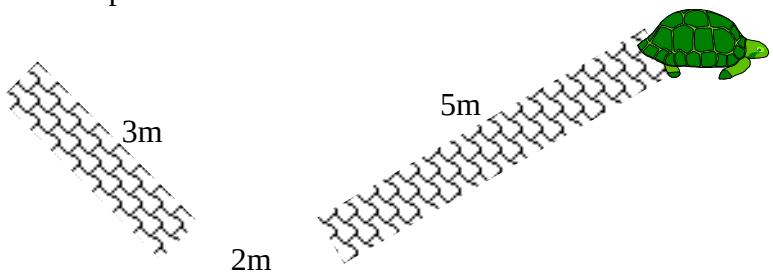
klassdiagram 4



(6 p)

Uppgift 5

Sköldpaddsgrafik (*turtle graphics*) är en i många sammanhang mycket användbar och lättanvänd form av grafikgränssnitt.[2] Namnet har metoden fått från liknelsen med en sköldpadda som rör sig på en sandstrand. Sköldpaddan kan vrida sig i olika riktningar, samt hoppa(!) eller kräla en rak sträcka i sin aktuella riktning. Om sköldpaddan hoppar lämnar den inga spår i sanden, men om den krälar lämnar den ett spår.



Om vi antar att den spänstiga paddan ovan från början var riktad åt höger så har den först vridit sig 45^o medsols, krälat 3m, vridit sig 45^o motsols, hoppat 2m, vridit sig 30^o motsols, krälat 5m, och slutligen vridit sig 30^o medsols.
I datorgrafiksammanhang motsvaras sköldpaddan av en tänkt ritmarkör, hoppen av förflyttningar, och krälandet av räta linjer. Följande basklass utgör en minimal stomme för sköldpaddsgrafik: [3]

```
class Turtle {
public:
    Turtle() : myX(0), myY(0), myDirection(0) { }
    void Turn( double Degrees ) { myDirection += Degrees; }
    virtual void Jump( double Length );
    virtual void Crawl( double Length ) = 0;
    double GetX() const { return myX; }
    double GetY() const { return myY; }
    double GetDir() const { return myDirection; }
private:
    double myX, myY;     //    position
    double myDirection;  // degrees
};

void Turtle::Jump( double Length ) {
    double Angle = M_PI/180.0*myDirection, // to radians
           dX = Length*cos( Angle ),
           dY = Length*sin( Angle );
    myX += dX;
    myY += dY;
}
```

Sekvensen ovan motsvaras av operationerna (medsols = positiv vinkel):

```
T.Turn(45); T.Crawl(3); T.Turn(-45); T.Jump(2); T.Turn(-30);
                  T.Crawl(5); T.Turn(30);
```

[2] Ibland är det enklare att rita på detta sätt än att ange koordinaterna för linjernas start- och slutpunkter. Sköldpaddsgrafik lämpar sig bl.a. för att rita vissa typer av fraktaler, t.ex. Mandelbrot-kurvor och Koch-flingor.

[3] Det är väsentligt att koordinater och vinklar hanteras internt i sköldpaddsobjekten med god noggrannhet (double) för att undvika ackumulering av avrundningsfel vid långa promenader. Vid själva ritningen på skärmen anges dock alltid koordinaterna som heltal.

Av klassdefinitionen framgår inte hur själva ritningen går till. Anledningen är här att man vill separera sköldpaddsobjektets inre tillstånd, d.v.s. position, och riktning, från dess presentation på skärmen. Tanken är att detta senare definieras i subklasser till `Turtle`, beroende på vilken grafikmiljö programmet skall användas i. Man kan t.ex. tänka sig de två subklasserna DOSTurtle och WindowsTurtle. Uppgiften är att definiera DOSTurtle. Subklassen skall definiera om vissa operationer i basklasen på lämpligt sätt, samt innehålla två extra: `SetColor`, och `SetPenSize` för att välja färg och linjebredd. Ritning skall ske med vald färg och linjebredd tills de ändras. För färger kan antas att det finns en typ `Color` med värdena `WHITE, BLACK, RED, BLUE, GREEN, YELLOW`. Linjebredd anges med ett heltal mellan 1 och 10. Ett nyskapat objekt skall som standard ha ritfärgen vit och penntjocklek 1.

Följande grafikfunktioner i DOS är användbara för att lösa uppgiften:

```
moveto( int x, int y);
```

flyttar ritpositionen i grafikfönstret till punkten (x,y).

```
lineto( int x, int y);
```

ritar en rät linje från aktuell ritposition till punkten (x,y), som blir ny aktuell ritposition.

```
setcolor( Color c);
```

ändrar ritfärgen till c.

```
setlinestyle( int style, int pattern, int thickness);
```

Ändrar typ (heldragen streckad, …), mönster och tjocklek för linjer. Använd för enkelhets skull värdena `SOLID_LINE` resp. `SOLID_FILL` för de två första parametrarna.

I DOS koordinatsystem är y-axeln riktad nedåt i fönstret.

Basklassen `Turtle` får ej ändras.

(16 p)

Bilaga

```
// Vector class interface: support bounds-checked arrays
//
// Object: must have zero-parameter constructor and operator=
// CONSTRUCTION: with (a) an integer size only
//
// ******************PUBLIC OPERATIONS*********************
// =                          --> Copying assignment
// [ ]                        --> Indexing with bounds check
// bool empty( )              --> Return true if Vector empty, false ow
// int size( )                --> Return # elements in Vector
// int capacity( )            --> Return capacity of vector
// void resize( int newSize ) --> Change bounds of Vector
// void reserve( int newCapacity ) --> Change Vector capacity
// void push_back( Object x ) --> Add x at end of Vector

template <class Object>
class Vector {
public:
    explicit Vector( int initSize = 0 );
    Vector( const Vector & rhs );
    ~Vector( );
    bool empty( ) const;
    int size( ) const;
    int capacity( ) const;
    Object & operator[]( int index );
    const Vector & operator= ( const Vector & rhs );
    void resize( int newSize );
    void reserve( int newCapacity );
    void push_back( const Object & x );
};


template <class A, class B>
struct Pair {
     Pair() {}
     Pair( const A & f, const B & s ) : first(f), second(s) {}
     A first;
    B second;
};
```