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** svarsalternativ för varje fråga! Varje korrekt svar ger två poäng. Garderingar ger noll poäng. Inga motiveringar krävs. I 1.6 skall dock mer explicit svar ges.

1. Vilken/vilka av följande är en objektorienterad metodik (OOM)?
   a. CBT
   b. Coad/Yourdon
   c. SSA
   d. JSP
   e. samtliga är OOM
   f. ingen är OOM

2. Vilken/vilka av följande aktiviteter har ej en framträdande roll vid objektorienterad design?
   a. indelning av klasser i moduler
   b. klassificering av objekt
   c. konstruktion av algoritmer
   d. beskrivning av objektrelationer
   e. fler än en av a-d
   f. samtliga a-d ingår i designfasen
   g. inget av ovanstående är korrekt

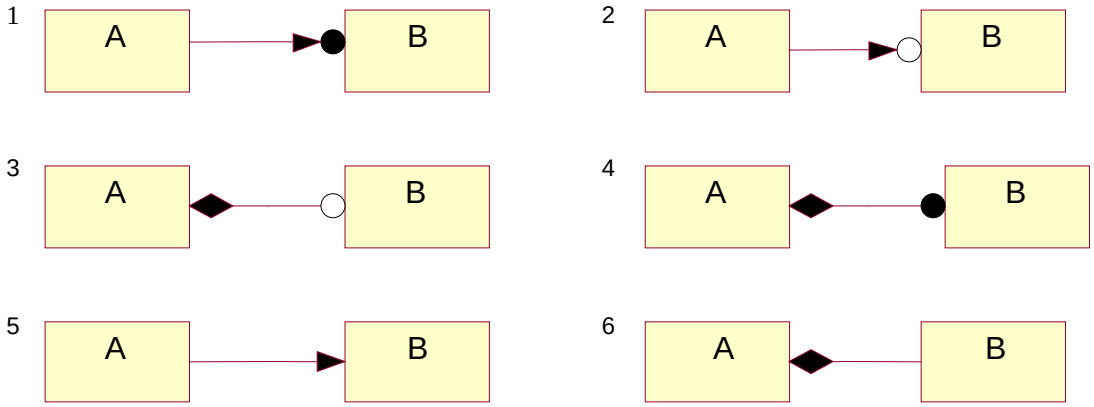
3. Vad menas med ”tvingande protokoll”?
   a. mottagaren måste svara inom viss tid
   b. ett objekt styr ett annat att utföra operationer i en viss ordning
   c. någon måste föra anteckningar på projektmötet
   d. subklasser måste implementera vissa operationer i basklassen
   e. de överenskommelser som gjorts på projektmöten måste följas
   f. fler än två av ovanstående
   g. inget av ovanstående är korrekt

4. Med testning kan man garanterat
   a. hitta alla funktionsfel i systemet
   b. visa att operationernas algoritmer är felfria
   c. visa att det levererade systemet uppfyller kravspecifikationen
   d. inget av ovanstående är korrekt

5. Vilket/vilka påstående karakteriserar vattenfallsmodellen
   a. varje utvecklingsfas upprepas innan nästa fas påbörjas
   b. systemet byggs ut stegvis
   c. varje utvecklingsfas avslutas innan nästa fas påbörjas
   d. de tidiga utvecklingsfaserna genomförs parallellt
   e. alla utvecklingsfaserna upprepas
   f. fler än ett av a-e
   g. inget av ovanstående är korrekt

forts.

6. Para ihop varje kodavsnitt c1 – c3 med det diagram d1 – d6 som passar bäst:




```
c1: class A {
       B *theB;
       ...
    };
```

```
c2: class A {
       B theB;
       ...
    };
```

```
c3: class A {
       B &theB;
       ...
    };
```

(12 p)

Uppgift 2

En vanlig sexsidig tärning kan modelleras med följande klass:

```
class Tarning {
public:
    Tarning();      // initierar genom att kasta tärningen
    void Kasta();   // kastar tärningen
    int Varde();    // returnerar antal ögon på uppsidan
private:
    int _Varde;     // antal ögon på uppsidan
};
```

Normalt är utfallen vid olika tärningskast oberoende av varandra – efter ett kast kan vilket tal som helst komma upp vid nästa kast. En ”minnestärning” beter sig som en vanlig tärning, men den hamnar aldrig med samma sida upp två gånger i följd. Definiera klassen `Minnestarning` genom arv från `Tarning`. Den nya klassen skall erbjuda exakt samma operationer som basklassen – ärvda eller egna, d.v.s. i ett användarprogram skall det räcka att byta ut typnamnet `Tarning` mot `Minnestärning` för at få det nya beteendet.

(10 p)

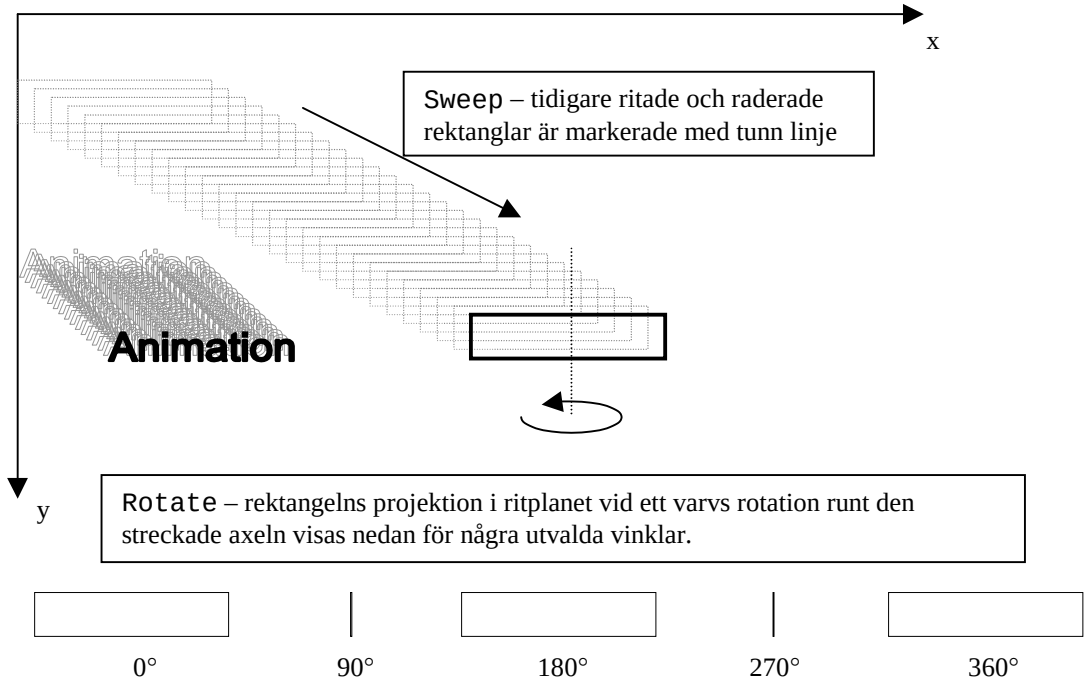
Uppgift 3

Animerad rörelse av en figur i ett ritfönster kan åstadkommas genom att man upprepade gånger ritar och raderar figuren och förändrar den något mellan varje omritning. Om omritningarna sker i tillräckligt små steg (en bildpunkt), och tillräckligt ofta, kan man få ett intryck av en jämn och obruten rörelse. Egenskapen att kunna animeras kan definieras i en klass, oberoende av den ritade figurens exakta utseende. Det senare kan definieras i olika subklasser, t.ex. för text, rektanglar, etc. Objekt av subklasserna ärver då möjligheten att kunna animeras. I en bilaga till tesen finns klassen `BasicAnimation`. Klassen har en operation `Sweep()` som används för att animera en svepande rak rörelse i angiven längd och riktning. Med denna operation kan man låta en figur ”åka” över ritytan. Attributen X och Y håller reda på aktuell position i ritfönstret (obs, koordinatsystemet är uppochnervänt). Det finns även en klass `Text` som kan användas för att animera text med angiven teckenstorlek. Huvudprogrammet ger beteendet i bilden nedan. Den grå ”svansen” illustrerar bara var figuren passerat under animationen och skall ej finnas kvar i ritfönstret. (Alla dimensioner är ungefärliga.)

a) Definiera subklassen `ExtendedAnimation` till `BasicAnimation` som dessutom har operationen `Rotate(int Turns);` `Rotate` skall rotera figuren `Turns` varv runt en vertikal axel i ritytans plan. Figuren betraktas som plan. *Tips:* denna subklass skall ej definiera Draw och Erase och den behöver hålla reda på figurens aktuella vinkelorientering. (7 p)

b) Definiera klassen `Rectangle` av rektangulära figurer som både kan svepas och roteras i ritfönstret (dock ej samtidigt). Definiera klassen genom arv från ovanstående klasser på lämpligt sätt. Rotationsaxeln skall gå vertikalt genom rektangelns mitt. Inför lämpliga dataattribut m.m. Rektanglar ritas med DOS-funktionen `rectangle`(x_1,y_1,x_2,y_2,), där (x_1,y_1) anger det övre vänstra hörnet, och (x_2,y_2) det nedre högra. Rektangelns fram- och ”baksida” har samma färg (välj själv). (7 p)

Avstånd i ritfönstret anges i bildpunkter (pixels) och riktningar i grader, medsols.



Uppgift 4

Betrakta följande program:

```
class B {
public:
    virtual int F() { return 5; }
    int G() { return F(); }
};
class S1 : public B {
public:
    int F() { return 1; }
};
class S2 : public B {
public:
    int G() { return 2; }
};
class S3 : public B {
public:
    int F() { return 3; }
};

void main() {
    B *Table[ 5 ];
    for ( int I = 0; I < 5; I++ )
        switch ( random( 3 ) ) {
            case 0: Table[ I ] = new S1; break;
            case 1: Table[ I ] = new S2; break;
            case 2: Table[ I ] = new S3; break;
        };
    int P = 1;
    for ( int I = 0; I < 5; I++ )
        P = P * Table[ I ]->G();
    cout << P << endl;
}
```

Vilka utskrifter är möjliga resp. omöjliga?

a) 135
b) 450
c) 35

För vart och ett av a-c:

- om utskriften är möjlig: förklara med en noggrann motivering och ett exempel hur beräkningen går till.
- är den omöjlig: förklara varför.

Vilka objektorienterade principer och mekanismer spelar roll för resultatet i a-c?

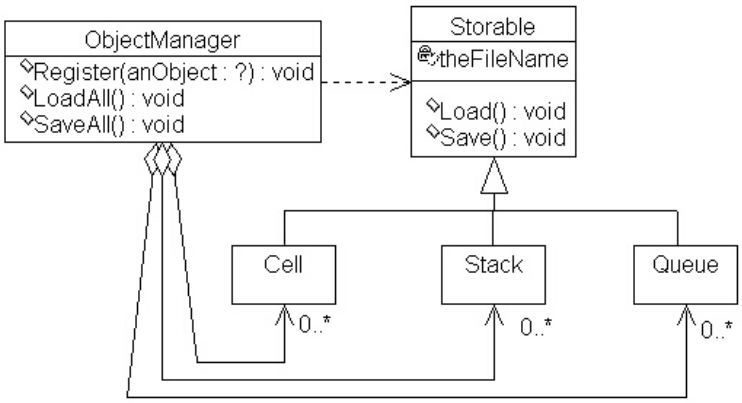
(10 p)

Uppgift 5

Ibland finns behov av att kunna spara datastrukturer som stackar, köer etc. i filer för att kunna arbeta vidare med dem vid en senare programexekvering, eller i ett annat program. Egenskapen att vara lagringsbar kan beskrivas med klassen

```
class Storable {
public:
    Storable( const char *aFileName ) {
        strcpy( theFileName, aFileName );
    }
    virtual void Save() = 0; // Saves object state in file
    virtual bool Load() = 0; // Loads object state from file,
                             // return true on success
protected:
    char theFileName[ 256 ]; // Name of file that stores object state
};
```

Datastrukturer som skall var lagringsbara definieras sedan i förhållande till Storable enligt följande klassdiagram



a) Datastrukturen `Cell` modellerar en enkel heltalsvariabel. Den kan tilldelas med Set och läsas av med Get. Dessutom håller den reda på om dess värde är definierat, i ett nytt objekt är det odefinierat.

```
class Cell {
public:
    ...
    void Set( int aValue );
    int Get();
    ...
private:
   ...
};
```

Klassdefinitionen är långt från fullständig. Komplettera den så att objekt av klassen blir lagringsbara. Definiera samtliga operationer. Operationerna skall uppfylla kraven:

`Get`	Om `Get` anropas för en oinitierad cell kastas ett lämpligt undantag.
`Load`	Om `Load` ej hittar något tal i filen ändras ej cellens tillstånd. `Load` returnerar true om läsningen gick bra, false annars.
`Save`	Om cellens värde är odefinierat ändras ej filen.

Vid alla former av filhanteringsfel kastas lämpligt undantag.

(6 p)

b) Klassen ObjectManager används för att samordna lagring och inläsning av ett antal lagringsbara objekt. Den har tre operationer:

`Register`	Registrera ett nytt lagringsbart objekt i intern datastruktur.
`LoadAll`	Läs in data till alla registrerade objekt från deras resp. filer.
`SaveAll`	Spara alla registrerade objekts tillstånd i deras resp. filer.

Definiera operationerna. Implementera relationen mellan ObjectManager och de lagringsbara objekten och med lämplig datastruktur (se bilaga).

(5 p)

c) Skriv ett program som skapar och registrerar tre cellobjekt hos ett manager-objekt. Det finns tre befintliga filer: `Allan.txt`, `Bellan.txt` och `Cellan.txt` som innehåller tidigare sparade celltillstånd och har skapats av något annat program. Programmet skall sedan ladda data till cellerna från filerna, öka cellernas värden med tal som läses in från tangentbordet, och till sist spara deras nya tillstånd. All filhantering skall ske med hjälp av manager-objektet.

(3 p)

Anm. Stack och kö omnämns bara som exempel på datastrukturer ovan. De skall ej implementeras.

Bilaga till uppgift 3

```
const double PI = atan(1.0)*4.0;
double ToRad( double Deg ) { return Deg*PI/180.0; }
double Sin( double Deg ) { return sin( ToRad( Deg ) ); }
double Cos( double Deg ) { return cos( ToRad( Deg ) ); }

class BasicAnimation {
public:
    BasicAnimation() : X(0), Y(0), StepRate(10) {} // ms
    void JumpTo( int X, int Y ); // Change position without redrawing
    void Sweep( int Length, int Direction ); // Animate a sweeping motion
    virtual void Draw() = 0;     // Draw the figure
    virtual void Erase() = 0;    // Erase it
protected:
    int X, Y;                     // The figure's current position
    const int StepRate;           // Delay between calls to Draw
};

void BasicAnimation::JumpTo( int X, int Y ) {
    this->X = X;
    this->Y = Y;
}

void BasicAnimation::Sweep( int Length, int Direction ) {
    int X0 = X, Y0 = Y;
    for ( int I = 0; I < Length; I++ ) {
        Draw();
        delay( StepRate );
        Erase();
        // get next position
        X = X0 + I*Cos( Direction );
        Y = Y0 + I*Sin( Direction );
    }
    Draw(); // so we don't miss the final position
}

class Text : public BasicAnimation {
public:
    Text( const char *Msg, int Size );
    ~Text() { delete theString; }
    void Draw();
    void Erase();
private:
    char *theString;
    int FontSize;
    void Draw( int Color );
};

Text::Text( const char *Msg, int FSize ) : FontSize(FSize) {
    theString = new char[ strlen(Msg) + 1 ];
    strcpy( theString, Msg );
}

void Text::Draw( int Color ) {
    settextstyle( SANS_SERIF_FONT, HORIZ_DIR, FontSize );
    setcolor( Color );
    outtextxy( X, Y, theString );
}
```

```
void Text::Draw() { Draw( YELLOW ); }
void Text::Erase() { Draw( BLACK ); }

// define these!
class ExtendedAnimation ...
class Rectangle ...

void main() {
    graphinit();

    Text T( "Animation", 10 );
    T.JumpTo( 0, 200 );
    T.Sweep( 100, 45 );         // 100 pixels sweep in dir. SE

    Rectangle R( 100, 20 );     // Width 100, height 20
    R.JumpTo( 0, 50 );
    R.Sweep( 300, 30 );
    R.Rotate( 1 );              // one complete 360 deg. turn

    closegraph();
}
```

```
// List class interface (á la Weiss)
//
// Access is via ListItr class
//
// ******************PUBLIC OPERATIONS*********************
// bool IsEmpty( )        --> Return true if empty; else return false
// bool IsFull( )         --> Return true if full; else return false
// void MakeEmpty( )      --> Remove all items
// ******************ERRORS********************************


// ListItr class interface; maintains "current position"
//
// CONSTRUCTION: with (a) List to which ListItr is permanently
//     bound or
//
// ******************PUBLIC OPERATIONS*********************
// void Insert( Etype X ) --> Insert X after current position
// bool Remove( Etype X ) --> Remove X
// bool RemoveNext()      --> Remove next cell
// bool Find( Etype X )   --> Set current position to view X
// bool IsFound( Etype X )--> Return true if X would be found
// void Zeroth( )         --> Set current position to prior to first
// void First( )          --> Set current position to first
// void operator++        --> Advance (both prefix and postfix)
// bool operator+( )      --> True if at valid position in list
// Etype operator( )      --> Return item in current position




// Queue class interface
//
// Etype: must have zero-parameter constructor and operator=
// CONSTRUCTION: with (a) no initializer;
//     copy construction of Queue objects is DISALLOWED
//
// ******************PUBLIC OPERATIONS*********************
// void Enqueue( Etype X )--> Insert X
// void Dequeue( )        --> Remove least recently inserted item
// Etype Front( )         --> Return least recently inserted item
// int IsEmpty( )         --> Return 1 if empty; else return 0
// int IsFull( )          --> Return 1 if full; else return 0
// ******************ERRORS********************************
// Predefined exception is propagated if new fails
// EXCEPTION is thrown for Front or Dequeue on empty queue
```