CHALMERS LINDHOMEN
Institutionen för data- och elektroteknik

Uno.Holmer@chl.chalmers.se

Lösningsförslag till tentamen* *Preliminär*

Kursnamn	**Objektorienterad programutveckling**
Tentamensdatum	**2002-01-18**
Program	**DAI 2**
Läsår	**2001/2002, lp I/II**
Examinator	**Uno Holmer**

* se även `www.chl.chalmers.se/~holmer/` där finns det mesta av kursmaterialet.

Uppgift 1 (10 p)

Ingen lösning ges. Se kurslitteraturen.

Uppgift 2 (6+1+3 p)

a) 6p

```
template <class A,class B,class Functor>
B accumulate(const Vector<A> & v, Functor func, const B & unit)
{
    B result = unit;
    for ( int i = v.size() - 1; i >= 0; i-- )
        result = func( v[ i ], result );
    return result;
}
```

b) 1p

```
bool all( const Vector<bool> & v ) {
    return accumulate( v, and, true );
}
```

c) 3p

```
bool oddAcc( int x, bool acc ) {
    return x % 2 == 1 || acc;
}

bool someOdd( const Vector<int> & v ) {
    return accumulate( v, oddAcc, false );
}
```

Uppgift 3 (12 p)

```
template <class Key, class Value>
class AssociationList : private Vector< Pair<Key,Value> > {
public:
    typedef Vector< Pair<Key,Value> > PairVector;

    explicit AssociationList( int size = 0 )
        : PairVector(size) {}
    Value & operator[]( const Key & key );
    const Value & operator[]( const Key & key ) const;
    void insert( const Key & x, const Value & val );
    using PairVector::empty;
    using PairVector::size;
};
```

```
template <class Key, class Value>
Value &
AssociationList<Key,Value>::operator[]( const Key & key ) {
    for ( int i = 0; i < size(); i++ )
        if ( PairVector::operator[]( i ).first == key )
            return PairVector::operator[]( i ).second; // found
    // not found
    throw "AssociationList::key not found";
}


template <class Key, class Value>
const Value &
AssociationList<Key,Value>::operator[]( const Key & key )const {
    for ( int i = 0; i < size(); i++ )
        if ( PairVector::operator[]( i ).first == key )
            return PairVector::operator[]( i ).second; // found
    // not found
    throw "AssociationList::key not found";
}

template <class Key, class Value>
void AssociationList<Key,Value>::insert( const Key & k,
                                         const Value & v )
{
    for ( int i = 0; i < size(); i++ )
        if ( PairVector::operator[]( i ).first == k ) {
            // don't insert duplicate keys, update value instead
            PairVector::operator[]( i ).second = v;
            return;
        }
    // not found, insert!
    push_back( Pair<Key,Value>( k, v ) );
}
```

Uppgift 4 (6+6 p)

a) 6p

Basklasspekaren p har statisk typ Base*. När den pekar på subklassobjektet S har den dynamisk typ Sub*:

`a) Base::F 20`	F är ej virtuell, p:s statiska typ bestämmer valet av funktion
`b) Sub::G 40`	G är virtuell och omdefinierad i Sub, p:s dynamiska typ bestämmer
`c) Base::H 20`	H ärvs eftersom den ej omdefinieras i sub
`d) Sub::G 40`	G är virtuell och omdefinierad i Sub, p:s dynamiska typ bestämmer

När p pekar på basklassobjektet B har den dynamisk typ Base*:

`e) Base::G 5`	basklassens version av G, p:s dynamiska typ bestämmer

När B tilldelas S skärs basklassdelen ut (X). Eftersom subklasskonstruktorn tidigare initierade basklassdelen (X) i S till 20 blir det detta värde som tilldelas till B.

`f) Base::G 20`	basklassens version av G, p:s dynamiska typ bestämmer

b) 6p

Diagram 3 är rätt svar eftersom ett A-objekt förfogar över C- och D-objekt och för att kunna hantera dem polymorft behöver känna till typen B. A har ingen vare sig aggregat- eller associationsrelation till något B-objekt, B-klassen är ju abstrakt, varför de andra tre alternativen faller.

Uppgift 5 (16 p)

```
class DOSTurtle : public Turtle {
public:
    DOSTurtle() : myColor(WHITE), myPenSize(1) {}
    // Redefined
    void Jump( double Length );
    void Crawl( double Length );
    // New
    void SetColor( Color C ) { myColor = C; }
    void SetPenSize( int Sz ) { myPenSize = Sz; }
private:
    Color myColor;
    int myPenSize;
};

void DOSTurtle::Jump( double Length ) {
    Turtle::Jump( Length );
    moveto( (int)GetX(), (int)GetY() );
}

void DOSTurtle::Crawl( double Length ) {
    Turtle::Jump( Length );
    setcolor( myColor );
    setlinestyle( SOLID_LINE, SOLID_FILL, myPenSize );
    lineto( (int)GetX(), (int)GetY() );
}
```

Kommentarer: 1. Attributen behövs om flera sköldpaddsobjekt skall hanteras i samma applikation eftersom varje objekt då själv måste hålla reda på vilken färg m.m. det skall ritas med. Om endast ett objekt skall användas kan attributen utelämnas och setcolor och setlinestyle anropas direkt eftersom dessa inställningar finns kvar tills de ändras. 2. Jump måste inte definieras om i subklassen, anropet av moveto kan alternativt placeras först i Crawl. Fördelen med att definiera om Jump som ovan är att det sparar en mängd anrop av moveto; I de flesta tillämpningar anropas Crawl betydligt oftare än Jump.