# 1. C++-programmering

## TNM056 – Vetenskaplig visualisering

Patric Ljung*

HT 2001
Version 1.1

## 1 Introduktion

Denna laboration syftar till att ge övning i C++-programmering och insikter om grundläggande delar i språket. Utgångspunkten för övningarna kommer att vara olika problem och principer snarare än en genomgång av vad språket innehåller. Efter genomgången lab skall du känna till och kunna tillämpa följande idéer och koncept:

- Innebörden av konkreta klasser och deras användning. Du skall förstå hur operatorer och överlagring går till samt vad referenser är och hur de används. Hur attributet *const* används och varför.
- Elementära datastrukturer (enkellänkade listor) och rekursivitet.
- Generella basklasser (abstrakta klasser) och hur polymorfism används (virtuella funktioner och arv).
- Innebörden av konceptet objektfabrik (Object Factory) och hur en enkel sådan kan konstrueras.
- Hantering av dynamiskt minne och hur det kan kapslas in i klasser.
- Klassfunktioner och klassvariabler med exempel på användning.

### 1.1 Anvisningar

Genomför denna laboration från början till slut. Svårighetsgraden varierar mellan uppgifterna. För att bli godkänd på laborationen måste samtliga *uppgifter* genomföras. Genomgående i texten finns instruktioner om att lägga till och ändra på saker i koden du arbetar med. Dessa måste följas, annars kan inte dina uppgifter genomföras. Ibland finns inte alla detaljer utskrivna, vilket ibland kommer att ge dig fel vid kompilering. Kolla då upp i man-sidor och info-sidor vad som krävs eller läs på i litteraturen om C++. Även om man jobbat flera år med programvaruutveckling kommer man inte ihåg allt. En minst lika viktig del är att kunna hitta rätt information när man har problem.

*Epost: plg@itn.liu.se

Vi förutsätter att läsaren har viss datorvana och förstår programflöden och datastrukturer. Det är lämpligt att ha en bok i C++ till hands, exempelvis *The C++ Programming Language* av Bjarne Stroustrup. Kurskompendiet C/C++ är även det lämpligt att ha till hands. Det finns flera sajter på Internet med introduktioner och tutorials.

**DevCentral - C++ Tutorials**
http://devcentral.iftech.com/articles/C++/default.php

**Learn C/C++ today**
http://cyberdiem.com/vin/learn.html

**Cprogramming.com – Tutorials: C++ Made Easy**
http://www.cprogramming.com/tutorial.html

**1001 Tutorials: C++ Tutorials**
http://www.1001tutorials.com/cpp/index.shtml

## 1.2 Allmänt om C++

Det mesta som gäller för C gäller även för C++. Man måste hantera minne, allokera och avallokera minnet. Pekare används även i C++ och ställer till också det även där. Men C++ erbjuder ett betydligt rikare språk där man kan ta hand om en hel del genom att bygga in det i klasser. Den viktigaste utökning av C++ (i jämförelse med C) är introduktionen av klasser och stödet för objektorienterad programmering (OOP).

Alla bibliotek som används i C kan användas i C++. Vidare har det tillkommit ytterligare bibliotek som erbjuder objektorienterade versioner av I/O och fundamentala klasser för datastrukturer. C++ är ett relativt ungt språk och standarden blev klubbad 1997. 1998 kom en korrigerad standard och den aktuella standarden har benämningen ISO/IEC 14882:1998 (Information Technology - Programming Languages - C++).

Programspråket C++ skapades ursprungligen av Bjarne Stroustrup under början av 80-talet. Man kan säga att C++ växte fram, det skapades inte i något specifikt projekt eller designades på papper. Språket är i många avseenden lösningar på problem som programmerare ställdes inför. Det finns ingen särskild anledning att tro att C++ kommer att ersättas av vare sig C, Java, C## (C-sharp), eller andra språk som finns idag. Troligtvis kommer det nya språk i framtiden som ersätter C++ (och de flesta andra språk vi använder idag).

# 2 Att programmera i C++

Att programmera i C++ är svårt, ännu svårare än i C. Men man kan också låta bli det knepigaste och använda en del av språket. Det är inte så svårt, och det kan göras säkrare än i C. En av de viktigaste delarna är objektorienteringsparadigmet. Möjligheten att kapsla in funktioner och representation av data i klasser är mest till för att underlätta för oss svagsinta människor som inte kan hålla för mycket i huvudet på en och samma gång. Det är därför bekvämt att koncentrera sig på ett litet delproblem och lösa det (dvs implementera den koden). Sedan kan man gå vidare med nästa. Klasser och objekt hjälper oss alltså att fokusera på en del av problemet och lita på att en annan del fungerar på ett väldefinierat sätt.

Laborationen kommer inte att behandla *streams*, dvs objekten *cout*, *cin* och *cerr* och deras allmänna I/O-kompanjoner (in- och utdatahantering). Vi kommer heller inte att beröra *templates* och virtualla basklasser.

## 2.1 Nu börjar vi

Vi skall ta och kompilera ett gammalt välkänt exempel från C, *Hello World*. Men vi skall fortfarande använda *printf*.

```
//-*- Mode:c++; c-indentation-style:stroustrup; c-basic-offset:4 -*-


#include <stdio.h>

int main(int argc, char **argv)
{
    printf("This is my world!\n");

    return 0;
}
```

I C++ är det viktigt med returtyper. Funktionen *main* skall returnera en *int*, och då måste man explicit ha med *return 0;* eller liknande. Skapa nu filen *hello.cc* och skriv in koden från ovan. Du kan kompilera med följande kommande `g++ -o hello hello.cc`. GNU:s kompilator för C++ heter `g++`, på andra system heter den ofta `CC`. IBM:s kompilator heter exempelvis `xlC`. Det är dock smidigt att lösa det hela med en make-fil. Du kan ladda ner alla delar till labben från webben (*http://www.itn.liu.se/∼plg/code*) antingen fil för fil eller hela paketet *AllFiles.tar*. Titta på innehållet med `tar tvf AllFiles.tar`, tänk på att om du packar upp arkivet efter att modifierat någon fil skriver över denna. Du kan packa upp till ett annat bibliotek för säkerhets skull.

Om du undrar över den första kommentarsraden i filerna så är det bara information till emacs när filen laddas. Emacs får genom denna kommentar information om vilken typ av indentering som önskas och vilken mod som skall användas när man redigerar filen. Vi har satt att indenteringen skall följa Stroustrups stil och ha fyra mellanslag. Vi säger också att filen är en C++-fil, emacs kan dock avgöra det tack vare suffixen *.hh* och *.cc*.

# 3 Konkreta klasser

En konkret klass är små enkla klasser utan virtuella funktioner, dvs ingen polymorfism. De har till uppgift att göra livet lättare, exempelvis hantera datum, id eller vektorer (lämpliga för 3D-grafik). Vi skall skapa en enkel sådan klass för att hantera tredimensionella vektorer. Vi vill att denna klass skall kunna användas tillsammans med OpenGL. Vi vill också att den skall kunna mappas mot en bit av minnet där en array av tal ligger (tre stycken *float* exempelvis). Vidare vill vi kunna utföra de matematiska operationer vi är vana vid att kunna göra med vektorer.

## 3.1 En vektorklass

Låt oss nu börja konstruera denna vektorklass. Vi skapar filen *Vec3f.hh* och skriver in följande:

```
//-*- Mode:c++; c-indentation-style:stroustrup; c-basic-offset:4 -*-


#ifndef _Vec3f_hh_
#define _Vec3f_hh_
```

```
class Vec3f {
public:
    inline Vec3f();
    inline ~Vec3f();

private:
    float  e[3];
};

inline Vec3f::Vec3f()
{
    e[0] = e[1] = e[2] = 0.0;
}

inline Vec3f::~Vec3f()
{
}

#endif // _Vec3f_hh_
```

Vår konstruktor initierar våra element till noll. Destruktorn gör absolut ingenting. Nyckelordet *inline* betyder att programkoden i funktionen skall kopieras in i programmet där funktionen anropas. Denna mekanism finns inte i C och det främsta syftet är att skapa snabbare kod. Funktionsanrop är i regel kostsamma i fråga om CPU-användning.

Tyvärr gör vår lilla klass inte mycket nytta än. Vi behöver kunna skapa ett objekt genom att sätta värden samt ändra på värdena i vektorn.

**Uppgift 1** Lägg till ytterligare en publik konstruktor som tar tre argument *x*, *y* och *z* av typen *float* och initierar vektorn med dessa värden i stället. Skapa också en publik funktion *set* med samma argumentlista och uppgift som den nya konstruktorn.

Nu börjar vi känna behov av att testa denna lilla klass, så det vore väl lämpligt att kunna plocka ut värdena på elementen. Vi behöver därför en funktion *element(int i) const* som ger värdet på element med index *i*. Det avslutande nyckelordet *const* definierar att vi inte kommer att ändra på innehållet i objektet, dvs inga värden kommer att ändras. Vi kommer att ha användning av detta senare.

**Uppgift 2** Lägg till en publik funktion *float element(int i)* som returnerar värdet av element *i*. Testa nu din klass, skriv ett litet program *vectest*. Det ser förmodligen ut på följande sätt (*vectest.cc*):

```
//-*- Mode:c++; c-indentation-style:stroustrup; c-basic-offset:4 -*-


#include "Vec3f.hh"
#include <stdio.h>

int main(int argc, char **argv)
{
    Vec3f      a(1, 2, 3);
    Vec3f      b(0.5, -0.5, 1.7);

    printf("Vector a: (%.2f, %.2f, %.2f)\n",
           a.element(0), a.element(1), a.element(2));
    printf("Vector b: (%.2f, %.2f, %.2f)\n",
           b.element(0), b.element(1), b.element(2));
```

```
    return 0;
}
```

Vad skulle nu hända om någon använda din vektorklass och skickade med ett index $i < 0$ eller $i > 2$ till funktionen *element*? Det skulle innebära en access till minne som inte hanteras. Denna kontroll borde kanske var möjlig att stänga av. Man kan anta att de flesta indexen anges som konstanterna 0, 1 eller 2, som i testprogrammet ovan. Vi kan använda makrot *assert* för en sådan test.

```
assert(i >= 0 && i <= 2);
```

Makrot *assert* kommer att avsluta programmet och skriva ut en text som talar om vilken fil och rad som falerade. För våra små övningar är detta adekvat, men i större system bör man ha litet mer robusta och förfinade metoder än att brutalt avsluta program. Genom att definiera pre-processorsymbolen *NDEBUG* kan detta makro oskadliggöras, som om det inte fanns. I din make-fil lägger du till `-DNDEBUG` till variabeln *CXXFLAGS*.

Det kanske vore på sin plats att göra vektorklassen litet mer lik en vektor. Vi kan använda oss av indexeringsoperatorn []. Vi har redan definierat en funktion som tar fram enskilda element, så steget är minimalt. Vi behöver bara kopiera funktionen *element* och ge den namnet *operator []* i stället. Nu kan vi ändra i vårat testprogram så att utskriftsraden får följande utseende:

```
printf("Vector a: (%.2f, %.2f, %.2f)\n", a[0], a[1], a[2]);
```

Vektorklassen börjar nu bli mer lik en vanlig array, men vi har också introducerat gränskontroll av indexvariabeln. Denna gränskontroll kan enkelt inaktiveras. Men, vi är inte riktigt framme än. Anta att vi vill ändra på det andra elementet enbart, hur gör vi det? Det kan vi inte än. Vi skulle kunna skriva en funktion *setElement(int i, float e)*. Det finns ett enklare sätt som inbegriper referenser. Ändra funktionen *operator []* så att den returnerar en *float &* i stället för bara *float*. Lägg till följande sats i ditt program och kontrollera att det fungerar, skriv ut vektorn *a*.

```
a[1] = 17.0;
```

Det gick förståss inte, kompilatorn gav dig en varning för att du konverterar en konstant till en icke-konstant. Operatorn [] returnerar nu inte längre en *float* utan en referens till en *float*. Denna referens pekar på den adress där elementen ligger lagrat så att innehållet på denna plats kan uppdateras i tilldelningssatsen. Men i och med att funktionen är konstant så är medlemmarna implicit omgjorda till konstanter. Vi kan alltså inte ha *const* efter funktionsnamnet. Lösningen på detta problem är att definiera två funktioner, en funktion som är konstant och bara returnerar en *float* och en funktion som inte är konstant och returnerar en *float &*. Deklarationen av operatorerna blir därför:

```
inline float operator [] (int i) const;
inline float & operator [] (int i);
```

Själva innehållet i funktionen är dock detsamma för båda operatorerna.

## 3.2 Vektoroperationer

Vad har vi då uppnått hitills, inte mycket. Dags att introducera litet vektoraritmetik, vilket innebär fler operatorer. Först behöver vi en tilldelningsoperator, *operator =*. En sådan ser ut som följande:

```
inline Vec3f & Vec3f::operator = (const Vec3f & v)
{
    e[0] = v.e[0];
    e[1] = v.e[1];
    e[2] = v.e[2];

    return *this;
}
```

Denna operator tillhör klassen. I uttrycket *a = b* är det objektet *a* som operatorn verkar på. Objektet *b* blir argumentet till funktionen, dyker alltså upp som referensen *v*. Vidare vill vi kunna skala en vektor genom att multiplicera eller dividera med en skalär.

**Uppgift 3** Implementera operatorerna *a += b* och *a -= b* så att vektorn *b* adderas till respektive subtraheras från vektorn *a*. Implementera också skalning av en vektor, dvs *operator *= (float s)* och *operator /= (float s)*. Vektorargumenten skall vara prefixade med *const*, vi skall inte ändra på de argument vi får:

```
inline Vec3f & operator += (const Vec3f & v);
inline Vec3f & operator -= (const Vec3f & v);
inline Vec3f & operator *= (float s);
inline Vec3f & operator /= (float s);
```

Nu behöver vi skapa ytterligare ett antal operatorer för att kunna utföra vanlig addition och subtraktion av vektorer och vektor–skalär multiplikation och division. Dessa operatorer tillhör dock inte klassen *Vec3f* utan är globala. Vi börjar med att titta på hur *operator +* implementeras:

```
inline Vec3f operator + (const Vec3f & a, const Vec3f & b)
{
    Vec3f s;

    s[0] = a[0] + b[0];
    s[1] = a[1] + b[1];
    s[2] = a[2] + b[2];

    return s;
}
```

Notera att funktionen måste returnera att objekt. Ju större objekten är (med avseende på minnesanspråk) desto mer jobb innebär det. Detta är något man vill undvika i ett komplett och seriöst system. Tyvärr inbegriper det ganska avancerade konstruktioner som vi inte har möjlighet att gå in på. För våra små exempel är detta inget problem. Det kan också tänkas att kompilatorn kan optimera rätt bra tack vare att alla funktioner är deklarerade med *inline*.

**Uppgift 4** Implementera nu resterande operatorer; subtraktion, multiplikation, division samt skalärprodukten. Nedan ser du hur prototyperna ser ut.

```
inline Vec3f operator - (const Vec3f & a, const Vec3f & b);
inline Vec3f operator * (const Vec3f & a, float f);
inline Vec3f operator * (float f, const Vec3f & a);
inline Vec3f operator / (const Vec3f & a, float f);
inline float operator * (const Vec3f & a, const Vec3f & b);
```

Vi kan förståss inte utföra division med en vektor i nämnaren.

**Uppgift 5** Lägg till två publika medlemsfunktioner för att normalisera en vektor och för att beräkna längden på en vektor. Utnyttja de matematiska operationerna du har definierat.

```
inline void Vec3f::normalize();
inline float Vec3f::length() const;
```

Till sist skall vi lägga till en typomvandlingsoperator. Anta nu att våra objekt avser hörnpunkter på polygoner. Det vore därför bekämt att kunna använda ett objekt direkt som ett argument till exempelvis OpenGL-funktionen *glVertex3fv*. Denna funktion tar en pekare till en *float*-array, vilket är precis vad vi har i objektet. Lägg till följande operator till klassen:

```
inline Vec3f::operator const float * () const
{
    return e;
}
```

Typomvandlingsoperatorer har ingen explicit returtyp, det är definierat i själva operatorn.

Tack vare typomvandlingsoperatorn kan vi nu skriva följande stycke kod.

```
Vec3f        a(0.5, 0.7, 1.3);
Vec3f        v(1, 1, 0);
Vec3f        w(0, 1, 1);

v.normalize();
w.normalize();

glBegin(GL_TRIANGLES);
    glVertex3fv(a);
    glVertex3fv(a + v*0.7);
    glVertex3fv(a + w*0.5);
glEnd();
```

### 3.3 Att optimera vektoruttryck i C++

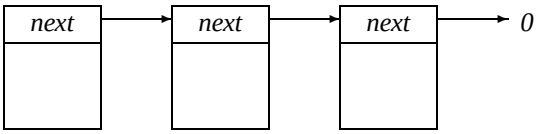
För de hugade finns en intressant artikel av Jim Blinn i IEEE Computer Graphics and Applications, Juli/Augusti 2000. Jag roade mig med att implementera hans idéer fullt ut för en vektorklass, om du vill se på resultatet kan du titta på *Vec3.hh* på adressen *http://www.itn.liu.se/∼plg/code*. Med artikels hjälp bör man kunna förstå hur det fungerar och varför man väljer att göra så.

## 4 Datastrukturer

Det finns ett stort behov av att kunna hålla reda på en mängd av objekt. Beroende på vad man avser att göra med dessa är olika datastrukturer lämpliga. En av de mest elementära typerna är listor som är väldigt begränsad i sin användning. Men det finns många tillämpningar för listor. Vi skall se hur vi kan implementera en lista i C++ genom att länka ihop objekt med varandra till en lång kedja. Varje nod i listan har en pekare till nästa objekt. Den sista noden i listan har sin pekare satt till noll för att markera slut på listan. Se figur 1.

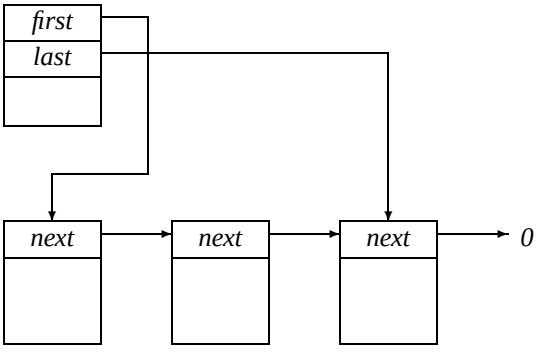
För att hålla ordning på en lista behöver man egentligen bara hålla rätt på en pekare till den första noden i listan. En sådan lösning är dock inte speciellt praktisk. Vad händer

Figur 1: Enkellänkad lista av noder




om man tar bort den första noden i listan? Vad krävs för att lägga till en nod till slutet av listan? Dessa problem löser man med hjälp av ett listobjekt. Detta objekt har en pekare till första och sista noden i listan. Är listan tom är dessa pekare noll (nil eller null). Klassen för detta listobjekt implementerar alla funktioner för att ta bort och lägga till noder i listan. Det enda som nodeobjekten behöver kunna är att återge och sätta pekaren till nästa objekt. Figur 2 visar strukturen för ett listobjekt och noderna i dess lista.

Figur 2: Enkellänkad lista av noder




Många gånger så vill man att hela listan skall tas bort om man tar bort själva listobjektet. I C++ måste detta hanteras av objekten själva. Det enda man behöver göra är att ta bort nästa objekt i destruktorn. I listobjektets destruktor tar man bort det första objektet i listan. Vi pratar här förstås om *delete*. Om en användare tar bort listobjektet kommer följdaktligen även alla noder att tas bort i och med rekursionen. För att skydda sig mot eventuell felaktig kod bör man sätta alla pekare till noll efter att man tagit bort ett objekt med *delete* (som normalt brukar benämnas *operator delete*). Nedanstående brottstycke kod visar principen.

```
Node::~Node() {
    delete mNext;
    mNext = 0;
}
```

När man nu i koden tar bort ett objekt av typen *Node* kommer samtliga objekt som är länkade att tas bort.

**Uppgift 6** Studera innehållet i filerna *GraphicObjectList.hh*, *GraphicObject.hh* samt *GraphicObject.cc* och implementera vad som saknas för att det skall fungera. Testkör genom att kompilera och köra programmet *golist*. Du kan hitta både make-fil och filerna på URL *http://www.itn.liu.se/∼plg/code*. Om allt är OK skall följande utskrift dyka upp. Finns det något "Core dumped" eller "Segmentation fault" är uppgiften inte korrekt löst.

```
main is starting - what happens...
GraphicObject::GraphicObject() - go-0: Created
GraphicObject::GraphicObject() - go-1: Created
...
GraphicObject::GraphicObject() - go-9: Created
GraphicObject::draw() - go-0: Called
GraphicObject::draw() - go-1: Called
...
GraphicObject::draw() - go-9: Called
main is done, expecting a lot of delete now...
GraphicObject::~GraphicObject() - go-0: Deleted
GraphicObject::~GraphicObject() - go-1: Deleted
...
GraphicObject::~GraphicObject() - go-9: Deleted
```

# 5 Basklass, arv och polymorfism (virtuella funktioner)

Klassen *GraphicObject* definierar en "smutsig" abstrakt klass (inte helt ren). En äkta abstrakt klass bör inte ha något data och enbart rent virtuella funktioner (pure virtual). Många gånger är det dock praktiskt att definiera ett defaultbeteende hos funktionerna och att placera data som kommer att användas av flertalet ärvda klasser.

Funktionen *draw* i klassen *GraphicObject* utför inget speciellt, den gör en utskrift för att visa att den har blivit anropad. Poängen är att den skall definieras om i nedärvda klasser, den har därför definierats som virtuell. Notera också att destruktorn också har gjorts virtuell. Varför man gör så skall vi motivera senare. Ladda ner filerna *Cube.hh* och *Cube.cc* och studera klassen *Cube*.

**Uppgift 7** Implementera nu det som saknas för att kunna rita en kub. För varje ny konfiguration (varje gång som *configure* anropas) skall du slumpa fram en färg för objektet. Använd klassen *Vec3f* för att lagra färgen. Varje färgkomponent skall slumpas med funktionen *drand48* enligt formeln $r = 0.8 * drand48() + 0.2$ och motsvarande för $g$ och $b$. Skapa även motsvarande klasser för en kon och en torus. Lägg till de medlemsvariabler som behövs för respektive objekttyp.

Testkör din kod genom att kompilera (`make goviewd`) och köra programmet *goviewd*. Filerna måste heta *Cube.xx*, *Cone.xx* och *Torus.xx* för att make-filen och programmet skall fungera (*.xx* byter du mot *.hh* och *.cc*).

# 6 Dynamiskt minne och OpenGL

I denna del av laborationen skall vi experimentera litet med lagring av data. Vi skall skapa ett grafiskt objekt men vill göra uträkning en gång och lagra resultatet i form av uträknade punkter, normaler och färger. Vi kan formalisera detta i en klass *PointVCN*

där suffixet *VCN* står för vertex, color och normal. Kopiera ner filen *PointVCN.hh* från webservern, delar av filen finns nedan.

```
class PointVCN {
public:
    PointVCN() { }
    PointVCN(const Vec3f & vertex,
             const Vec3f & color,
             const Vec3f & normal) :
        mVertex(vertex), mColor(color), mNormal(normal) { }
    ~PointVCN() { }

    Vec3f & vertex() { return mVertex; }
    Vec3f & color()  { return mColor; }
    Vec3f & normal() { return mNormal; }
    const Vec3f & vertex() const { return mVertex; }
    const Vec3f & color() const  { return mColor; }
    const Vec3f & normal() const { return mNormal; }

protected:
    Vec3f   mVertex;
    Vec3f   mColor;
    Vec3f   mNormal;
};
```

En av de två nya konstruktionerna i språket som vi använt i denna klass är en initieringslista i konstruktorn som tar tre vektorobjekt. En initieringslista krävs för att ange vilka konstruktorer i medlemsobjekt och basklasser som skall användas och vilka värden argumenten har. Den andra detaljen är att funktionerna har definierats direkt i klassdefinitionen. När funktionerna blir så små som i detta fall blir det oftast enklare så. Funktioner som deklareras på detta sätt blir implicit inline, förutsatt att funktionen inte är virtual. Virtuella funktioner kan inte var inline och alla icke-inline funktioner måste placeras i *.cc*-filen.

Normalt sett placerar man en klass per fil som har samma namn som klassen. Har vi en klass som heter *PointVCN* finns det följdaktligen en fil *PointVCN.hh* och oftast också en *PointVCN.cc* (för alla vanliga och virtualla funktioner, mm).

**Uppgift 8** Du skall nu skriva en klass *Cylinder* som definierar en cylinder med radie och höjd. Vidare behöver man veta antal stackar och skivor (motsvarande det som anges till *glutSolidSphere* och *glutSolidCone*). Bygg cylinder som ett antal fyrkanter (quads) i OpenGL. Antalet runt cylindern bestäms av variabeln *slices* och antalet på höjden av *stacks*. Topp och botten skapas av en *triangle fan* i OpenGL.

Du skall slumpa fram två färger (varje färg som för de tidigare objekten) och alternera mellan dessa runt cylindern. På så viss skall den bli randig med tonade övergångar. Varje gång som funktionen *configure* anropas skall du avallokera tidigare använt minne för punkterna och allokera nytt minne så att det räcker för den angivna upplösningen ($slices$ och $stacks$). Totalt antal punkter som behövs är

$$points = slices(stacks + 1) + 2(slices + 1)$$

och skall allokeras i en enda array av typen *PointVCN* med hjälp av *new*, avallokering sker med *delete []* för att markera en array. Det kan vara lämpligt att skriva om defaultkonstruktorn i *Vec3f* så att den inte initierar vektorn till noll. Vi kommer ju ändå att tilldela nya värden till vektorn så det är bara onödigt. Klassen skall inte lagra onödig information utan bara det som krävs för att kunna rita ut cylindern korrekt. Du kan

ta bort kommentaren för inkludering av cylinderklassen samt ta bort *#ifdef–#endif* i funktionen *keyboard* så att cylinderobjekt också kan skapas.
**Obs!** Justera din make-fil *Makefile* så att variabeln *GOVIEWDOBJS* även innehåller filnamnet *Cylinder.o*.

För att förenkla din kod kan du göra en liten inline-funktion *drawPoint* som tar en referens till en *PointVCN* och anropar *glNormal3f*, *glColor3f* och *glVertex3f* i rätt ordning.

# 7 Objektfabriker, klassvariabler (globalt data) och klassfunktioner

En objektfabrik (Object Factory) har till uppgift att skapa objekt av en specifik klass utan att den som vill ha objektet känner till den exakta klassen. Ofta finns önskemålet om att få en klass som hanterar något visst filformat, kommunikationskanal, eller ritar grafiska objekt. Man vill alltså koppla skapandet av en klass med ett namn eller identifierare. För filformat kan man ofta använda extensionen, exempelvis .DXF, .3DS. Något säkrare är att öppna filen och verifiera typen genom att undersöka innehållet i den. De flesta format har en form av id i början, brukar ofta kallas *magic id* eller *magic code*.

Vi skall associera en sträng (ett namn) med våra klasser vi har gjort tidigare i labben. Denna sträng skall bara vara klassnamnet, exempelvis 'Cone' motsvarar klassen *Cone*. För att kunna göra detta behöver vi utnyttja variabler och funktioner som inte är bundna till objekt utan till klassen. Funktionerna tillhör klassen men appliceras inte på ett objekt av klassen utan klassen självt. Variablerna finns endast i en instans per klass. Nedan ser du hur en sådan variabel och funktion deklareras i en header-fil.

```
class TypeNode {
public:
    TypeNode() {
        add(this);
    }
    ~TypeNode() { }

    ... additional stuff
private:
    static TypeNode * mFirst;

    static void add(TypeNode * node) {
        node->next(mFirst);
mFirst = node;
    }
    ... additional stuff
};
```

Variabeln *mFirst* ingår inte i något objekt av klassen *TypeNode* eller någon av dess subklasser. Funktionen *add* arbetar inte på något specifikt objekt, man kan inte använda variabeln *this* i funktionen, den finns helt enkelt inte. Man skulle kunna säga att funktionen är som en vanlig funktion i C. I detta fall är funktionen deklarerad inline. Variabeln *mFirst* är dock inte allokerad ännu. Det måste ske i en *.cc*-fil.

```
#include "TypeNode.hh"
```

```
TypeNode * TypeNode::mFirst = 0;

static TypeNode  aNode;
```

Kodsnutten ovan skapar en instans av klassvariabeln *TypeNode::mFirst* och initierar den till noll. Denna typ av data initieras vid kompileringstillfället och blir en del av det körbara programmet. Objektet *aNode* är ett objekt som har gjorts lokal i den aktuella filen (förmodligen *TypeNode.cc*). I detta fall har bara minnesplats reserverats för objektet, men det är inte initierat. Problemet här är att en konstruktor måste anropas, programkod måste köras. När skall det göras?

Konstruktorer för globala objekt anropas före det att funktionen *main* startas. Motsvarande kommer deras destruktorer att anropas efter att man returnerat från *main*. I vilken ordning globala objekt konstrueras och destrueras mellan olika filer är odefinierat (i alla fall inget man bör förlita sig på). De kommer att anropas sekventiellt i alla fall.

Om vi nu förser våra noder med ytterligare två medlemmar, en sträng för att lagra typnamnet och en för att lagra en pekare till funktion. Denna funktion är också en klassfunktion och dess uppgift är att skapa ett objekt av den klass den är definierad i. För att kunna lagra en pekare till en funktion är det också lämpligt att först definiera en typ som definierar vilken typ av funktion det är fråga om, så att kompilatorn kan kontrollera att argumenten i anropet blir korrekt.

```
class Factory {
public:
    typedef Object * (*NewFunction)(const char * name,
                                    const char * config);

    class TypeNode {
        TypeNode *       mNext;
        const char *     mName;
        NewFunction      mNewFunc;
    public:
        TypeNode(const char * name, NewFunction * func) {
            mNext = 0;
            mName = name;
            mNewFunc = func;
        }
        ... basic stuff

        inline Object * newObject(const char * name,
                                  const char * config) {
            return (*mNewFunc)(name, config);
        }
        ... more stuff

        friend Factory;
    };
};
```

I klassen *Factory* har vi definierat en typ *Factory::NewFunction* som definierar en pekare till en funktion som tar två konstanta strängar som argument och returnerar en pekare till ett objekt av klassen *Object*. När en instans av klassen *TypeNode* skapas skall ett typnamn anges och en pekare till en funktion.

**Uppgift 9** Studera klassen *GraphicObjectFactory*. Du skall nu modifiera dina grafiska objekt så att de automatiskt registrerar sig hos objektfabriken. Det som skall göras

är principiellt:

1. Definiera en klassfunktion *NewObject* som tar två argument.

```
static GraphicObject * newObject(const char * name,
                                 const char * config);
```

2. Lägg till en klassvariabel *mFactoryRegistration* som är av typen *GraphicObject-Factory::TypeNode*.

3. Du måste också definiera och initiera registreringen i din *.cc*-fil. Ange först namnet på klassen som en sträng och sedan funktionen *newObject* för rätt klass.

4. Implementera funktionen *newObject* så att den skapar ett nytt objekt av aktuell klass och konfigurerar denna med hjälp av strängen *config*. Exempelvis kan du använda *sscanf* för att plocka ut parametrarna. Koden nedan visar delar av cylinderklassen.

```
float radius, height;
int stacks, slices;

if (sscanf(config, "%f %f %d %d",
           &radius, &height, &stacks, &slices) != 4) {
    printf("Error: Cylinder::newObject() - "
           "incorrect config information, "
           "expected Radius Height Stacks Slices\n");
    return 0;
}

Cylinder * cyl = new Cylinder(name);
cyl->configure(radius, height, stacks, slices);
```

   Denna princip kan du använda för de andra klasserna likaså.

När du är klar kompilerar du och testkör med programmet *goview*. Visa för labbhandledare och var nöjd.

# 8 Tankar vid labbens slut

Det här blev förstås en hel del jobb. Å andra sidan har vi gått igenom konstruktioner och idéer man faktiskt har nytta av i industrin. Principerna du har stött på här återkommer i många olika skepnader men är i grund och botten ofta samma problem. C++ och objektorientering ger dig möjlighet att lösa dessa problem en gång och förpacka det i ett trevligt klassbibliotek som kan återanvändas om och om igen.