Objektorienterad programmering i C

Denna uppgift riktar sig till dig som anser att du är duktig på att programmera, eller vill ha något att bita i, och som redan förstår grundprinciperna i objektorientering. Uppgiften illustrerar något om vad som försiggår under huven på objektorienterade språk.

Partiell lista över innehållet

- Dynamiska datastrukturer (listor)
- Pekare och pekararitmetik
- Funktionspekare, **`void`**-pekare
- Makron

Introduktion

I ett objektorienterat program kommunicerar objekt med varandra via meddelandesändning. Om *s* är ett sträng-objekt kan man t.ex. be det byta ut sin första bokstav till `A` genom att skicka meddelandets *set* med argumenten 0 och `A`:

```
s.set(0,'A');
```

Syntaxen är avsiktligt lik C (det är nämligen Java ...) – '.'-operatorn avser dock meddelandesändning och inte att man läser en post ur en strukt. En viktig skillnad mellan *metodanropet* `s.set(0,'A')` och *funktionsanropet* `set(s, 0,'A')` är att vid det senare fallet har du som skrev koden bestämt att operationen `set` skall utföras på objektet *s*, medan du i det föregående skickar ett meddelande, `set`, till objektet *s* som bestämmer vilken operation som skall utföras som ett resultat.

Utdelat i denna uppgift finns ett stor blob med C-kod som förenklat implementerar objekt i C.

Klasser och objekt

Klasser har delats upp i två delar: en strukt som innehåller alla instansvariabler (data layout) och en strukt som innehåller pekare till C-funktioner som kan anropas med hjälp av meddelandesändning (vtable).

Ett objekt är en instans av en data layout-strukt med en pekare till en instans av en vtable-strukt. Den första fångar objektets tillstånd, den andra dess beteende.

Arv

Arv implementeras med två byggstenar: data layout-strukten använder sig av en teknik som heter ”prefixing” vilket enkelt uttryckt betyder att en subklass innehåller hela superklassens layout. T.ex.:

```
1  class Point2d { // Javakod
2    int x;
3    int y;
4    void setX(int x) {_this.x = x; }
5    void setY(int y) {_this.y = y; }
6  }
```

kan representeras med strukten

```
1  struct point2d {
2    struct vtable *methods;
3    int x;
4    int y;
5  }
```

Klassen `Point3d` ärver av `Point2d`

```
1  class Point3d extends Point2d { // Javakod
2    int z;
3    void setZ(int z) {_this.z = z; }
4  }
```

och med prefixing får vi:

```
1  struct point3d {
2    struct vtable *methods;
3    int x;
4    int y;
5    int z;
6  }
```

Motsvarande teknik används för vtables. En vtable för `Point2d` kan vara denna array, där ↑`foo` avser en pekare till funktionen `foo`:

$$[\uparrow \texttt{setX}, \uparrow \texttt{setY}]$$

Och för `Point3d`:

$$[\uparrow \texttt{setX}, \uparrow \texttt{setY}, \uparrow \texttt{setZ}]$$

Hela tanken med prefixing är att layouten hos subklasser innehåller layouten för superklassen i omodifierad form. Notera att offset/index för variabler/funktionspekare är samma i båda klasserna för samma variabel. Detta hjälper till att implementera dynamisk bindning.

Metodanrop (meddelandesändning)

Meddelandesändning implementeras med dynamisk bindning i funktionen `call`. Man skickar in mottagaren, namnet på meddelandet i form av en *selektor* (se nedan), antalet argument, samt argumenten:

```
call(s, set, 2, 0, 'A');
```

En selector för funktionen ”set” skapas så här (variabeln nedan används i anropet ovan):

```
selector set = mkSelector("set");
```

En viktig del av designen av selektorerna är att två anrop till `mkSelector("set")` returnerar identiska objekt, d.v.s. två objekt a och b för vilka $a == b$ gäller.

Uppgiften

1. Implementera stöd för överlagring på argumentantal, d.v.s. ett objekt o skall kunna ha metoderna `foo(int)` och `foo(int,int)`. Vilken som körs som resultat av att meddelandet `foo` skickas till o bestäms av antalet argument till meddelandet (1 eller 2).

   Ledning: Antingen kan selektorerna utökas som t.ex. `foo/1` och `foo/2` för ovanstående, eller så används `argc`-argumentet till `call`. Notera att de båda metoderna `foo` ovan skall implementeras som två olika funktioner och att C inte stöder överlagring på argumentantal.

2. Vtables är konstanta och bör därför delas mellan samtliga instanser av en och samma klass. Använd **`static`**-variabler för att implementera detta så att t.ex. `stringVtable` bara skapar en vtable vid första anropet, och vid efterföljande anrop (givet att argumenten är samma förstås), så returneras samma vtable som skapades första gången.

3. Modularisera koden på lämpligt sätt och ge de resulterande delarna lämpliga gränssnitt.

4. Det nuvarande sättet att skapa vtables kräver att man manuellt bygger vtables som uppfyller kraven på prefixing. Implementera stöd för att detta skall ske automatiskt. D.v.s., om man vill skapa en ny vtable för strängar och sträng är en subklass till sekvens skall man bara behöva ge de nya metoderna för strängar – sekvensens data bör kopieras in automagiskt.

   På samma sätt borde man inte behöva ange index manuellt.

5. Implementera stöd för superanrop.

   Om klassen `Point3d` i ett tidigare exempel också hade haft en metod `setX` hade denna *override:at* den i `Point2d`. Om vi skriver `Point2d::setX` respektive `Point3d::setX` för dessa båda metoder så kan vi konstatera att `Point2d::setX` inte längre går att anropa på en 3d-punkt eftersom den göms av `Point3d::setX`.

   Ett superanrop är ett anrop till en metod (aka meddelandesändning) till en superklass version av en metod. Följaktligen kan man tänka sig denna idiotiska men illustrerande kod:

```
class Point3d extends Point2d {
  ... som ovan ...
  void setX(int x) {
    super.setX(x+1);
  }
  ... som ovan ...
}
```

   Ett anrop `p.setX(7)` på en 3d-punkt är alltså implementerad som i termer av ett anrop `p.setX(8)` med 2d-punktens definition av `setX`.

   Ledning: Metodanrop via `call` använder klassen för det aktuella värdet, alltså även om man är inuti en metod som är implementerad i en 2d-punkt och skriver `setX` så får man 3d-punktens `setX`. Detta är ett resultat av dynamisk bindning – alltså vilken metod som körs som resultat av ett meddelande räknas ut under körning.

   Superanrop kan bindas statiskt, d.v.s. det går alltid att avgöra vilken metod som **`super`**`.setX(...)` avser. En hybrid är att inte binda statiskt, men att åtminstone statiskt avgöra vilken vtable man bör starta med (den för 2-punkt i Java-kodexemplet ovan).

6. Implementera stöd för överlagring på typer.

   Överlagring på argumentantal ger oss möjligheten att ha två eller fler metoder med samma namn så länge som de tar olika många argument (aka har olika aritet).

   Överlagring på typer ger oss möjligheten att ha två eller fler metoder med samma aritet, så länge som de tar emot olika typer.

   Nu måste `call` också ta i beräkning argumentens typ för att matcha mot existerande funktioner. Detta kan lämpligen kodas in i selektorn.

7. **SVÅR!** Utöka överlagringen på typer med stöd för dynamisk bindning.

   Denna *frivilliga* utökning av föregående feature kräver att `call` tittar på argumentens dynamiska typ, vilket kräver att det finns en representation av typer under körning som kan jämföras mot selektorerna. T.ex. om det finns två selektorer `foo/A` och `foo/B` där `A` och `B` är objekttyper, måste det vara möjligt att titta på ett objekt och avgöra om det är av typ `A` etc. (Mer eller mindre en implementation av **`instanceof`** i Java.)

   En ytterlig svårighet med denna uppgift är att typer inte alltid matchar exakt, utan har en subtypsrelation. T.ex. kanske man anropar `foo` med ett argument av typen `D` och vi måste först räkna ut att `D` ärver `C` som i sin tur ärver `A` innan vi kan räkna ut att vi borde leta efter `foo/A`.

   Det blir lämpligt att skriva en funktion för att ta fram klassen för ett objekt och för att se om en klass är en subklass av en annan. Man kan med fördel också bygga en hjälpfunktion som givet namnet på en klass ger en referens till klassen (d.v.s. dess vtable). På så sätt blir det enkelt att gå från ett klassnamn i en signatur till en pekare till en vtable så att man kan kolla om ett objekts klass (en annan vtable) är en subklass av den första...