Workshop 1 - Enkla övningar i C, h-filer, flera filer

Denna workshop kommer introducera er till de filer som ni kommer att använda i laborationerna samt gå igenom lite kring hur man kan lägga upp ett lite större program i C.

För endel (förhoppningsvis alla? :-)), så kommer detta vara repetition, men är till för att de som är lite mer osäker eller inte hållit på med C på ett tag ska komma igång snabbare.

Förslag till lösning på de frågor som ställs under övningen är markerade i rött!

Dessa övningar förutsätter mer eller mindre att ni arbetar på den linuxmiljö som skolan använder sig av, så antingen att man göra detta via t.ex. ssh till itchy/scratchy eller på en miljö hemma som påminner om denna där Gnu C Compiler är installerat med sökvägar så den kan köras från kataloger där filerna ligger. Se andra guider för hur man installerar eller kommer åt dessa miljöer.

De kommandon som denna workshop ber er testa ska alltså köras från någon form av terminal eller kommandoprompt, t.ex. ”gcc” och andra milöer kan visa filerna och katalogers innehåll på olika sätt.

Övning 1

A Ladda hem kodbasen ni ska använda i laborationerna. (version 1.0.8.2.)
A1. Packa upp den.
"unzip datastructures-v1.0.8.2.zip" under Linux, annars med någon annat, t.ex. winzip eller i utforskaren direkt.
A2. Nu får ni en katalog som heter "datastructures-v1.0.8.2/". Gå in i den katalogen.
”cd datastructures-v1.0.8.2/" om det är en underkatalog där du står.
A3. Lista filer och kataloger där och verifiera att ni har katalogerna "include/", "lib/" och "src/" samt några filer.
Detta görs enklast med ”ls -la” i linux eller ”dir” i dos/windowsmiljö, men annars går det bra att kolla med någon filhanterare också.

B - Gå in i katalogen "include/"
”cd include” fungerar i linux och i windows kommandopromptar.
B1 - Lista filerna som finns där och se om ni förstår vad vilka datatyper som de motsvarar.
Detta görs enklast med ”ls -la” i linux eller ”dir” i dos/windowsmiljö, men annars går det bra att kolla med någon filhanterare också.
B2 - Öppna nu t.ex. "dlist.h"
B2.1 - Vad är detta för datatyp den representerar?
Svar:
dlist.h deklarerar funktioner för den abstrakta datatypen Directed List, dvs enkellänkad lista.

B2.2 - I toppen av filen finns det två #include-satser, vad är det för skillnad på dessa två?
Svar:
#include <XXX.h> ger en fil som ligger bland ”standard” bibliotekens filer
#include ”YYY.h” försöker inkludera en fil som ligger i samma katalog eller har en sökväg relativt till den fil man nu försöker kompilera

B2.3 - I denna fil ser man många funktioner deklarerade men ingen kod, varför inte det?
Svar:
Header-filer som denna används för att DEKLARERA för en kompilator att det FINNS funktioner med de namnen, vilka parametrar de tar och vilken typ av data de returnerar. Själva koden lägger man på ett annat ställe och kompileras separat.

B2.4 - Det finns även några rader med "typedef struct dlist dlist;" och "typedef struct cell *dlist_pos;",
Borde inte dessa ha något mer då man vill att de ska definera strukturer med någon typ av data?
Svar:
Fördelen med att bara DEKLARERA att det ska finnas en STRUCT dlist som vi kallar dlist och en som pekare till en STRUCT cell, utan att säga vad de ska innehålla gör att vi i vår .c-fil sedan kan bestämma precis vad vi behöver ha i dessa UTAN att vi behöver ändra detta INTERFACE som nu vår h-fil deklarerar.

C - Gå tillbaka en katalog och gå nu in i katalogen "src/". I denna kataloger finns en mängd underkataloger.
C1 - Gå in i katalogen "dlist/".
C2 - Öppna "dlist.c" och kolla på denna.
Öppna den i antingen en texteditor eller din utvecklingsmiljö.
C2.1 - Jämför den med h-filen du kollade på innan.
C2.1A - Varför har man här en #include "dlist.h"?
Svar:
Vår Header-fil dlist.h DEKLARERAR vilka funktioner som ska finnas, vilka parametrar de tar och vilket data de ska returnera. Att inkludera den i början av vår C-fil så dels så kommer vi få fel om vi sedan DEFINERAR funktionerna på ett annat sätt men framförallt så kan vi ANROPA ALLA funktioner redan från början av vår fil så vi behöver inte se till att de defineras i rätt ordning om de anropar varandra.

C2.1B - Nu har vi inga typedef, men vi har istället struct-satser som nu innehåller saker, varför då?
Svar:
Som vi beskrev tidigare, så deklarar H-filen att vi HAR en strukt men inte vad den innehåller, men nu är det dags att faktiskt stoppa i saker i dem. Typedef som var med i h-filen har bara skapat ett ”alias” vi kan använda när vi behöver åsyfta denna struktur.

C2.1C - I andra filen var det bara massa funktioner deklarerade, nu innehåller det kod. Finns det någon funktion här som inte fanns med i h-filen? Går det och i så fall, varför kan och ibland bör man ha det?
Svar:
Tanken med .h-filen är att man kan deklarera de funktioner man UTÅT vill kunna använda sig av, dock kan de funktioner som man har för det vara i behov av egna hjälpfunktioner för att både återanvända kod eller underlätta läsbarheten på något sätt.

C3 - Öppna och kolla på "dlist_mwe1.c".
C3.1A - Vad är detta för fil?
Svar:
Ett Minimum working example- är ett så kort program som möjligt för att visa hur man ska använda i detta fall den Directed List som vi har, ofta att man deklarera initierar, använder och sedan avallokerar det som behövs.

C3.1B - Varför är det bra att ha med sådana här "Minimum Working Examples" när ni bygger egna funktionsbibliotek eller datastrukturer?
Svar:
Det låter någon annan snabbt komma igång och använda den kod eller funktioner/datatyper som ni skapat, genom detta exempel och inte behöva läsa massa dokumentation.

C3.2 - Nu vill vi kompilera den första av dessa. Testa "gcc dlist_mwe1.c"
C3.3 - Detta gick inte så bra, vad gick fel och hur åtgärdar vi detta?
"dlist_mwe1.c:5:10: fatal error: dlist.h: No such file or directory"
Svar:
Då vi inte angivit VAR exakt kompilatorn ska hitta denna .h-fil så göra den inte det och vi får felmeddelanden om detta.

C3.4 - Man löser detta med ange parametern -I som ska innehålla sökvägen till de H-filer som du själv skapat.
"gcc dlist_mwe1.c -I ../../include/"
C3.5 - Vad gick fel denna gång? Man får följande utskrift:

```
/usr/bin/ld: /tmp/ccCmHHNG.o: in function `main':
dlist_mwe1.c:(.text+0xbd): undefined reference to `dlist_empty'
/usr/bin/ld: dlist_mwe1.c:(.text+0xcd): undefined reference to `dlist_first'
/usr/bin/ld: dlist_mwe1.c:(.text+0x102): undefined reference to `dlist_insert'
etc. etc.
```

C3.6 - Är det kompilatorn eller är det något annat som ger dessa fel?
Svar:
Tekniskt sett är det LÄNKAREN som ger detta fel. Kompilatorn översätter koden till en objektkod (.o-filer) som innehåller de körbara instruktionerna men har bara symboliska länkar till funktioner och vart data ska lagras. Länkaren lägger ihop all körbar kod och fyller i dessa symboliska länkar med de korrekta minnesadresserna för funktioner och data. Nu anropar man med kommandot ”gcc” även länkaren då dessa är ihopsatta i denna C-kompilator men ibland är det helt separata program.

Denna ger nu fel eftersom man hänvisar till endel funktioner för att hantera dessa listor som den har blivit lovade att de ska finnas och hur de ska anropas, men när den sedan försöker göra en körbar kod av detta så hittar den inte igen de faktiska funktionerna, dvs. den saknar den .o-fil som innehåller dem.

D - Gå nu tillbaka en katalog och gå nu in i katalogen "lib/"
D1 - I denna katalog finns bara en fil "Makefile"
D1.1 - Vad är detta för fil? Öppna den och kolla på den.
Makefile är en fil som man kan låta innehålla information om hur olika filer är beroende av varandra och på sådant sätt sätta upp regler för att kunna separat-kompilera delarna i ett större projekt. Detta gör man för att man inte ska behöver kompilera om allt som man inte ändrat, men om något är det så måste även delarna som är beroende av dessa göras om.
D2 - Denna fil körs med kommandot "make", gör detta.
”make” är ett kommando som finns att köra via kommandoprompten under Linux, många mer avancerade utvecklingsmiljöer håller reda på hur filer ändrats i ett projekt automatiskt och kompilerar på så sätt om de delar som behövs. Det är dock bra om ni förstår och kan göra detta själva.
D3 - Kolla i katalogen, du har nu fått en ny fil. Vad är detta för fil?
Just denna Makefile skapar istället ett färdigt bibliotek med de funktioner som man behöver för att använda de datatyper som finns implementerade i detta ”datastructures”-paket. Detta är för att man inte ska behöva ha separat .o-filer för alla datatyper, men fungerar på samma sätt.

E - Gå tillbaka till din "../src/dlist/" katalog och kör make igen.
E1 - Kör "./dlist_mwe1" och "./dlist_mwe2" och se till att du förstår de steg du nu har gjort gör.

Övning 2

A1. Gå nu tillbaka några kataloger så du är i katalogen du packade upp "datastructures-v1.0.8.2/" i.
Typ ”cd ../../..” eller liknande.
A2. Skapa nu en egen katalog "my_doa_code/" och gå in i den.
”mkdir my_doa_code” och sedan ”cd my_doa_code”
A3. Nu vill vi att vi ska använda den "abstrakta" datatypen enkellänkad lista i ett eget program.
A4. Till detta ska vi använda den kodbas som vi nyss laddade hem och kompilerade.

A5. Skapa en C-fil som ser ut något liknande denna:

```
#include <stdio.h>
int main(int argc, char **argv)
{
   int tal[100];
   int i;
   i = 0;
   do {
      scanf("%d", &tal[i]);
      i++;
   }
   while ((i < 100) && (tal[i - 1] >= 0));
   return 0;
}
```

Detta program läser in maximalt 100 heltal eller avbryter när man anger ett negativt tal.

B - Detta är ganska "stelt" program på olika sätt. Antingen så har du nu allokerat plats för 100 tal där man kanske bara använder några få stycken av dessa eller annars så kanske du skulle behöva FLER platser än de 100 du angivit. Så vad vi hade velat är att den istället DYNAMISKT ska kunna lägga till så många tal till en LISTA du behöver tills programmet avbryts.

Lösningen är såklart att du använder den "abstrakta datatypen lista" du i övning 1 kollat närmare på!

B1 - Modifiera ditt program ovan så att den börjar med att inkludera h-filen för dlist.h som du använt nyss.
Svar:
Lägg till #include "dlist.h" som rad 2.

Men för att den ska hitta igen denna fil så måste man kompilera med kommandot:
gcc -o myfirstlist_with_dlist myfirstlist_with_dlist.c -I ../datastructures-v1.0.8.2/include/

DETTA är det INTERFACE du ska följa.
Den deklarerar alla funktioner som du behöver för att använda listan. Dessa följer de axiom som har gåtts igenom på föreläsningen.

För att kunna kompilera och länka ihop med de funktionerna som finns där så bör vi även kompilera dlist.c som ligger under src/ i datastructures-katalogen:
”gcc -c ../datastructures-v1.0.8.2/src/dlist/dlist.c -I ../datastructures-v1.0.8.2/include/”

Detta resulterar i en .o-fil, alltså en objektfil som sedan ska länkas in när vi kompilerar vårt program.

B2 - Istället för att skapa en int tal[100], skapa en tom lista du ska använda för att lagra talen.

```
dlist *myList = dlist_empty(free);
```

Här är free egentligen en pekare till funktionen free() som listan förväntas anropa för dess data när ett element tas bort. Detta går att byta ut mot en egen funktion också.

Modifiera sedan inläsningen så att den läser in ett tal i taget och lägger in dessa i denna lista.

```
int tal;
scanf(”%d”,&tal);
```

När vi ändrat de två raderna så ska vi lägga till en ny rad som lägger in talet som finns i ”tal” till vår lista. Dock måste vi först kopiera detta till en ny minnesposition så man inte hänvisar till samma minne allt eftersom man lägger till nya element i listan.

```
int *ptrInt = (int*)malloc(sizeof(int));
*ptrInt = tal;
```

Sedan lägger vi till pekaren till talet till vår lista:

```
dlist_insert(myList, ptrInt, dlist_first(myList));
```

B3. Lägg till kod som går igenom och skriver ut de tal som är inlästa EFTER du har läst in klart.

Detta kan se ut något som detta:

```
for(dlist_pos dpos = dlist_first(myList); !dlist_is_end(myList, dpos); dpos = dlist_next(myList, dpos))
{
	int tmpTal;
	tmpTal = *(int *)dlist_inspect(myList, dpos);
	printf("%d\n", tmpTal);
}
```

B4 - Gå igenom listan IGEN och summera ihop talen du nyss skrev ut.

```
int summa = 0;
for(dlist_pos dpos = dlist_first(myList); !dlist_is_end(myList, dpos); dpos = dlist_next(myList, dpos))
{
	int tmpTal;
	tmpTal = *(int *)dlist_inspect(myList, dpos);
	summa = summa + tmpTal;
}
printf("%d\n", summa);
```

B5 - Gå igenom listan en TREDJE gång och denna gång ska du TA BORT alla JÄMNA tal du har i listan.
OBS. Ta bort från den BEFINTLIGA listan, inte skapa en ny med de talen du ville ha kvar.

```
for(dlist_pos dpos = dlist_first(myList); !dlist_is_end(myList, dpos); dpos = dlist_next(myList,
dpos))
{
	int tmpTal;
	int *ptrTal;
	ptrTal = (int *)dlist_inspect(myList, dpos);
	tmpTal = *ptrTal;
	if (tmpTal % 2 == 0)
	{
		dpos = dlist_remove(myList, dpos);
	}
}
```

B6 - Skriv ut och summera även dessa.

```
int uddaSumma = 0;
for(dlist_pos dpos = dlist_first(myList); !dlist_is_end(myList, dpos); dpos = dlist_next(myList,
dpos))
{
	// Läser talet på den positionen med dlist_inspect och skriver ut det
	int tmpTal;
	tmpTal = *(int *)dlist_inspect(myList, dpos);
	printf("%d\n", tmpTal);
	uddaSumma += tmpTal;
}
printf("%d\n", uddaSumma);
```

B7 - Loopa igenom och även ta bort de element som är kvar.

```
for(dlist_pos dpos = dlist_first(myList); !dlist_is_empty(myList) && !dlist_is_end(myList, dpos);
dpos = dlist_next($
{
	int *ptrTal;
	ptrTal = (int *)dlist_inspect(myList, dpos);
	dpos = dlist_remove(myList, dpos);
}
```

B8 - Avallokera listan den nu förhoppningsvis tomma listan så ALLT minne har återlämnats till systemet.

```
dlist_kill(myList);
```

C

Du har nu förhoppningsvis använt interfacet och dess implementation för strukturen "dlist", dvs en enkel-länkad (directed) lista. Om du tycker att det vore bra att ha en som är länkad åt bägge hållen så finns det en lista du kan använda som är dubbellänkad istället.

C1 - Hur mycket av koden måste du ändra för att den ska använda interfacet "list.h" istället?

Svar:

Eftersom den abstrakta datatypen dubbellänkad lista ”list” har till stor del samma interface som den enkel-länkade listan så kan du egentligen bara byta ut alla referenser där du använt dlist → list. Det

som skilljer dessa åt är att den dubbellänkade listan har funktionalitet att även gå baklänges igenom listan men för att gå igenom den gör man på samma sätt.

C2 - Hur gör du för att kompilera det nya programmet med denna?

Svar:
Eftersom vi inte längre använder dlist, så är det inte längre dlist.c ’s funktioner vi använder. Därför behöver vi nu kompilera list.c så vi får en list.o som innehåller de nya funktionerna. Detta gör vi på samma sätt som tidigare.