Preliminär föreläsningsöversikt

1. C jämfört med SML. Grundläggande C: skalära datatyper, variabler, uttryck, tilldelningar, satser, funktioner, programstruktur. `gcc`–kompilatorn, `make`.
2. Fortsättning. Arrayer, strängar, pekare, loopar, standardbibliotek. Preprocessorn. Bit–operatorer. `static`. Intro inlämningsuppgift 1.
3. Poster, dynamiska datastrukturer och minneshantering. Listor och träd. Pekararitmetik.
4. Fortsättning. Genomgång inlämningsuppgift 1. Intro inlämningsuppgift 2.
5. Unionstyper och fallanalys. Intro inlämningsuppgift 3. "Överkurs" (allt det som har hoppats över) i mån av tid.

C versus Standard ML

- C har *satser*, konstruktioner som enbart utför sidoeffekter.
- I C betecknar variabler tilldelningsbara minnesceller.
- C är semantiskt "nära" hårdvaran.
- C har ej nästlade funktioner, funktioner-med-omgivningar, automatisk minneshantering, eller inbyggd mönstermatchning.
- What you see is what you get. Inga gömda kostnader.
- C är förvånansvärt portabelt och ett de facto standardpråk för operativsystem och andra systemapplikationer.
- C är ej förlåtande: logiska fel ger upphov till runtimefel.

Skalära datatyper

`int`: Den "normala" heltalstypen för aktuell maskin. Vanligen 32 bitar i 2-komplementsform, men det garanteras ej av språket.

`unsigned int`: Som `int` fast utan tecken.

Dessutom finns `long`, `short`, och `char` som betecknar heltalstyper av olika storlek. `char` är speciell eftersom det är den minsta adresserbara datatypen ("byte"), och alla objekt anses bestå av en sekvens av bytes. Tecken i strängar är av typen `char`.

`double`: flyttal, vanligen i standard IEEE 64-bitars "double precision"

`float`: flyttal med lägre precision, vanligen 32 bitar

En boolesk typ finns inte: istället används `int`.

Enkla variabeldeklarationer

En enkel variabeldeklaration skrivs som ett typnamn följt av ett eller flera variabelnamn, och avslutad med semikolon.

```
char the_first_char;
unsigned int i, j27;
double d;
```

Ett variabelnamn börjar med bokstav eller `_` och följs av en obruten sekvens bokstäver, `_` eller siffror. Små och stora bokstäver är olika: `foo` ≠ `Foo`.

En variabel i C är bunden till en minnescell, och inte ett specifikt värde som i SML. Värdet hämtas automatiskt i uttryck.

Vanliga aritmetiska uttryck

```
int i, j;
double x, y;
i + j, i - j, i * j, i % j
x + y, x - y, x * y
x + j, x - j, x * j
-i, -x
(i + j) * x
```

I varje binär operation konverteras (vid runtime) den mindre operanden till samma typ som den större operanden. Detta gäller både heltal av olika storlek (`long + char`) och blandning av heltal och flyttal.

Relationsuttryck

```
int i, j;

i < j, i <= j, i >= j, i > j
i == j, i != j
!i
```

Resultatet är 1 för sant och 0 för falskt. Resultattypen är `int`, oavsett vad argumenten är.

Används vanligen i styrande satser (`if` etc).

`!i` (“not i”) är samma sak som `i == 0`.

Även flyttal kan jämföras, men eftersom flyttal inte är exakta måste man vara försiktig.

Tilldelningar

Uttrycket `var = exp` har samma värde som uttrycket `exp`, men tilldelar variablen `var` detta värde som en sidoeffekt.

`var += exp` är samma sak som `var = var + exp` bortsett från att variabeln bara evalueras en gång. (`-=`, `*=`, och några till)

`var++` har samma värde som `var`, men variabeln ökas med 1 som sidoeffekt. `var--` minskar istället variabeln med 1.

`++var` och `--var` utför +1 eller -1 innan variabelns värde blir uttryckets värde.

Dessa konstruktioner är idiom i C, så ni måste känna till dem.

Enkla satser

expression ;

Ett uttryck föjt av ett semikolon är en sats. Uttrycket evalueras enbart för ev. sidoeffekters skull.

Ex: `i = 27; func(x, 1);`

;

Den tomma satsen skrivs som ett ensamt semikolon.

Villkorliga satser – if

```
if ( expr )
    stmt1
stmt2
```

Om uttrycket är sant (inte 0) exekveras `stmt1`, varefter programmet fortsätter vid `stmt2`. Annars fortsätter programmet direkt vid `stmt2`.

```
if ( expr )
    stmt1
else
    stmt2
stmt3
```

Om det finns en s.k. `else`–gren exekveras den istället om styruttrycket `expr` är falskt (0).

Om `sats2` i sin tur också är en `if`–sats skivs den vanligen på samma rad som `else`–ordet utan extra indentering (s.k. *fork–style* `if`:s).

```
if( expr1 )          |   if( expr1 )
    stmt1            |       stmt1
else if( expr2 )     |   else
    stmt2            |       if( expr2 )
else if( expr3 )     |           stmt2
    stmt3            |       else
else                 |           if( expr3 )
    stmt4            |               stmt3
                     |           else
                     |               stmt4
```


Block

```
...
{   /* c = (a * b) squared */
    int temp;
    temp = a * b;
    c = temp * temp;
}
...
```

Ett block är en sammansatt sats som består av deklarationer av lokala variabler (optionellt) och en sekvens av satser. Det ska *inte* finnas något semikolon efter det avslutande `}`.

Funktionsdefinitioner

$\text{type}_{\text{result}}\ \text{name}_{\text{func}}(\text{type}_{\text{arg1}}\ \text{name}_{\text{arg1}}, \ldots, \text{type}_{\text{argN}}\ \text{name}_{\text{argN}})$

{
    local variables
    statements
}

En funktionsdefinition består av ett funktionshuvud och en funktionskropp.

Huvudet deklarerar funktionens resultattyp, namn, antal formella parametrar, samt varje formell parameters namn och typ.

Funktionskroppen är ett *block*. En funktion returnerar sitt värde med satsen `return expr;`

```
double add(double x, double y)
{
    return x + y;
}

int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```


Funktionsanrop

```
x = 5 + func(exp1, ..., expn)
```

C använder *call-by-value* precis som SML: uttrycken i anropet (de *aktuella* parametrarna) evalueras före anropet.

När en funktion anropas skapas nya variabler för de *formella* parametrarna, och de aktuella parametrarnas värden kopieras till dessa variabler. Funktionen kan tilldela dessa variabler utan att påverka variabler hos den som gjorde anropet.


```
int stupid(int z)  | int foo(int x)
{                  | {
    int x;         |     int y;
    z = z + 1;     |     y = x;
    x = z + 3;     |     stupid(y);
    return x;      |     /* y == x is still true */
}                  | }
```

Synlighetsregler för variabelnamn är precis som i SML: namnen på formella parametrar och lokala variabler är privata och påverkar inte andra funktioner.

Svansanrop? – undvik!

```
int sum(int from, int to)
{    return sum_with_accum(from, to, 0);
}
int sum_with_accum(int from, int to, int accum)
{
    if (from >= to)
        return accum;
    else
        return sum_with_accum(from+1, to, accum+from);
}
```

C optimerar inte bort svansanrop, så anropsstacken kommer att växa snabbt och ta slut ⇒ runtimefel. Använd C:s loop–satser istället.

Repetition – `while`–loop

```
while (expr)
    stmt1
```

Uttrycket evalueras. Om det är sant exekveras *kroppen* (`stmt1`), varefter kontrollen går tillbaka till *huvudet* och processen upprepas. Först när uttrycket blir falskt går kontrollen till nästa sats.

```
int sum(int from, int to)
{   int s = 0;
    while (from < to) {
        s += from; from += 1;
    }
    return s;
}
```


`void` – ingen resultattyp, inga parametrar

```
void some_procedure(...) { ... }
```

Definierar en funktion utan returvärde ("procedur"). Funktionen returnerar med `return;` eller genom att "ramla ut" ur sin funktionskropp. Anrop till funktionen måste vara egna satser.

```
int some_function(void) { ... }
```

Definierar en funktion utan parametrar. `void` får *inte* utelämnas.

Globala variabler

Variabler kan definieras även utanför funktioner:

```
int ugly;

void foo(void) { ugly = 1; bar(); }

void bar(void) { /* use ugly */ }
```

En global variabel skapas då programmet startar och finns kvar tills programmet avslutas.

Användning av globala variabler bör i allmänhet undvikas.

Programstruktur

Ett C–program består av:

- En huvudfunktion `int main(void)` där programmet startar.
- Ett antal andra egendefinerade funktioner.
- Ett antal egendefinierade globala variabler.
- Ett antal egendefinierade typer (senare föreläsning).
- Ett antal typer, variabler, och funktioner som länkats med från olika bibliotek. (I/O, matterutiner, grafik, etc.)

Den enda begränsningen är att deklarationen av en variabel eller funktion måste komma före användningen av densamma.

Separatkompilering

I praktiken behöver man dela upp de flesta C–program i mindre delar, ex.vis. därför att de olika delarna löser olika deluppgifter vilket ger bättre överblick och möjlighet att återanvända kod i andra applikationer.

Man skapar då flera källkodsfiler (`.c`–filer) och lägger ex.vis. variabler och funktioner för en komponent i en fil, och variabler och funktioner för en annan komponent i en annan fil.

Ex: `database.c`, `network.c`, `utilities.c`

.h–filer, extern–deklarationer

Före uppdelning:

```
double compute_it(void)
{
    ...
    return ...;
}

int main(void)
{
    double x;
    x = compute_it();
    /* use x */
}
```


Efter uppdelning:

```
/* main.c */           | /* compute.c */
int main(void)         | double compute_it(void)
{                      | {
    double x;          |     ...
    x = compute_it();  |     return ...;
    /* use x */        | }
}                      |
```

Problem! När `main.c` kompileras är inte definitionen av `compute_it()` synlig, vilket leder till varningar, kompileringsfel, eller t.om. runtimefel.

Lösning: Deklarationer i `.h`–fil:

```
/* compute.h */
extern double compute_it(void);

/* main.c */           | /* compute.c */
#include "compute.h"   | #include "compute.h"
int main(void)         | double compute_it(void)
{                      | {
    double x;          |     ...
    x = compute_it();  |     return ...;
    /* use x */        | }
}                      |
```

`compute.h` är en *interface*–fil som deklarerar allt som ska vara synligt för `compute.c`:s klienter. Man kan även lägga variabel- och typdeklarationer i `.h`–filen.

Kompilering – gcc

`gcc -c fil.c`: kompilera bara till en objektfil (`.o`–fil), länka ej ihop ett huvudprogram

`gcc -g -c fil.c`: lägg till extra debuginformation

`gcc -O -c fil.c`: förbättra koden så att den exekverar snabbare

`gcc -I/path/to/dir ...`: sök även efter `.h`–filer i katalogen `/path/to/dir`

`gcc -o prog prog.c`: kompilera, länka och namnge den exekverbara filen `prog`

`gcc -o prog fil1.o fil2.o`: länka och bygg exekverbart program från separatkompilerade `.o`–filer

Automatiskt kompilering – make

Utveckling av program med separatkompilering skapar problem:

- Det enklaste är att alltid kompilera om samtliga källfiler, men det tar onödig tid om man testar en begränsad förändring.
- Att manuellt hålla reda på vilka filer som behöver kompileras om kan vara svårt och ge upphov till fel. Säg att en deklaration i en `.h`–fil ändrats: då måste samtliga filer som använder den `.h`–filen kompileras om.
- `make` är ett verktyg som låter oss beskriva dels hur enstaka filer och program ska byggas, men också vilka filer som beror på vilka andra filer (typiskt beror `.o`–filer på `.h`–filer). `make` kollar dessa beroenden mha filernas tidstämplar och applicerar sedan reglerna för att bygga om de delar som måste byggas om.

Manuell kompilering, men `make` kollar beroenden:

```
myprog: fil1.o fil2.o
        gcc -o myprog fil1.o fil2.o

fil1.o: fil1.c fil2.h
        gcc -O -g -c fil1.c

fil2.o: fil2.c fil2.h
        gcc -O -g -c fil2.c

clean:
        rm -f fil1.o fil2.o myprog core a.out
```


Användning av inbyggda regler för kompilering av C–filer:

```
CC=gcc
CFLAGS=-O -g

myprog: fil1.o fil2.o
        $(CC) -o myprog fil1.o fil2.o

fil1.o: fil2.h
fil2.o: fil2.h

clean:
        rm -f fil1.o fil2.o myprog core a.out
```

Demo

Emacs + skalfönster. Makefile. Kompilering. Spårning av exekvering i grafisk debugger (ddd).