PROGRAM-UTVECKLING

EDITORER

- Några exempel: `vi, vim, emacs`
  - `vi` är populär men inte den enklaste att lära
  - `emacs` är kraftfull (går att programmera)
  - `vi` och `emacs` finns även i GUI-varianter som kan vara enklare att använda i början:
    - `gvim`
    - `xemacs`
- Det går också att skriva koden på en pc och över filerna till den Unix/Linux maskin som programmet ska kompileras/köra på.
- Kan behöva undvika de integrerade miljöerna på denna kurs.

PREPROCESSORN

- Med den kan man styra vilken kod som ska kompileras, definiera macron etc.
- Används traditionellt till att ersätta värden
-
```
#define TRUE 1
#define FALSE 0
#define SNUTT 238
void main ( void ) {
  if( SNUTT == TRUE )
    return SNUTT;
}
```

- Om vi sedan kör detta genom preprocessorn så får vi
-
```
void main ( void ){
  if ( 238 == 1 )
    return 238 ;
}
```

DEFINIERA MACRON

- Macron med parametrar fungerar på ungefär samma sätt.
-
```
#define MAX(x,y) ((x)<(y)?(y):(x))
void main ( void ) {
  int fun= MAX(46, 23);
}
```

- Efter att ha kört cpp (preprocessorn) så får man
-
```
void main ( void ) {
  int fun =(( 46 )<( 23 )?( 23 ):( 46 )) ;
}
```

PROBLEM MED MACRON

- Vad händer i detta fall ?
- `#define MUL(x,y) x * y`

```
void main ( void ) {
   int x;
   x = MUL(2+3, 5);
   …
}
```

- `x = 2 + 3 * 5;  /* 17 och ej 25`

PROBLEM MED MACRON

- Sätter in parenteser runt argumenten
- `#define MUL(x,y) (x) * (y)`

```
void main ( void ) {
   int x;
   x = MUL(2+3, 5);
   …
}
```

- `x = (2 + 3) * (5);  /*`

PROBLEM MED MACRON

- Vad händer i detta fall ?
- `#define MAX(x,y) (x)<(y)?(y):(x)`

```
void main ( void ) {
   int i = 1;
   int j = 1;
   int fun:
   fun = MAX(1,2) * 3;
}
```

- `fun = (1) < (2) ? (2) : (1) * 3;`
  `/* blir 2, ej 6`

PROBLEM MED MACRON

- Sätter parentes kring hela uttrycket
- `#define MAX(x,y) ((x)<(y)?(y):(x))`

```
void main ( void ) {
   int i = 1;
   int j = 1;
   int fun:
   fun = MAX(1,2) * 3;
}
```

- `fun = ((1) < (2) ? (2) : (1)) * 3;`
  `/* blir`

PROBLEM MED MACRON

- Vad händer i detta fall ?
- `#define MAX(x,y) ((x)<(y)?(y):(x))`

```
void main ( void ) {
  int i = 1;
  int j = 1;
  int fun:
  fun = MAX(i++, j++);
}
```

- `fun = ((i++) < (j++) ? (j++) : (i`

ALTERNATIV KOMPILERING

- `#ifdef`
- `#ifndef`
- `#define`

KOMPILERING

- Separatkompilering
  - Headerfiler, kodfiler och objektfiler
- Flaggor
  - `c` skapa objektfil
  - `o` *`filename`* namn på programmet
  - `S` ger assemblerkod
  - `E` köra bara preprocessorn
  - `Wall` slå på de flesta varningarna
  - `pedantic` kolla väldigt noga
  - `ansi` följ ANSI C standarden
  - `o` optimera koden
  - `l` länka in bibliotek (t.ex. `-lm` för mattebiblioteket)
  - `D` *`name`* definiera *name* som ett macro med värde 1

LÄNKNING

- Ser till att objektfilerna kombineras till ett program.
- Statisk länkning
  - Rutiner från libraries kompileras in i den körbara filen. Filen blir stor, men behöver inte andra filer för att kunna köras. Bra utifall man gör ett katastrof-program, som ska fungera på ett skadat system där viktiga filer kanske saknas.
- Dynamisk länkning
  - Delar på koden mellan olika program (processer)
  - Enbart referenser till rutiner i libraries hamnar i den körbara filen. Vid uppstart länkas programmet ihop i minnet. Filen blir mindre, men beroende av andra filer. Om flera program använder samma library kan det räcka med en kopia i minnet som alla program delar på

MAKE

- Används för att underlätta kompileringen
- Specificerar beroenden mellan filerna och hur de ska kompileras
- Körs med kommandot `make`
- `-f` *`filnamn`*
  - om annat än standardnamn på filen (`Makefile` eller `makefile`)
- `-n`
  - skriv ut vilka kommandon som skulle ha körts, men kör ej

MAKE – GRUNDFORMAT

```
# Kommentar

VAR=värde

mål: lista av beroenden
<tab>    kommando1 för att uppnå taget
[<tab>   kommando2]
```

- Använd \ för att bryta långa rader

MAKE

- Kommentarer
  - Börjar med #
  - Pågår till slutet av raden
- Variabler (kallas för macron)
  - Tilldelning
    `var=value`
  - Användning:
    `$(var)`
  - Kan vara skalärer eller listor (”arrayer”)

MAKE – REGLER

- *Målet* (target)
- *Beroenden*
  lista med saker som måste vara uppfyllt innan regeln kan utföras
- *Actions*
  lista med actions (kommandon) som utförs så snart som beroendena uppfylls

MAKE – MÅL

- Kan vara olika saker
  - Bara ett namn
  - Ett filnamn
  - En variabel
- Kan finnas flera mål på samma rad om de har samma beroenden
- Målet följs av
  - Ett kolon ":"
  - Sen en lista av beroenden, separerade med blank
- Defaultmålet är "`all:`" eller det första målet i filen

MAKE – BEROENDEN

- Kan vara (tomt innebär "*utför alltid dessa actions*")
  - Filnamn
  - Andra mål
  - Variabler
- Kolla om målet är up-to-date (jämför filers datum, finns ej => gammal)
  - Om något av beroendena är nyare än målet – utför actions
  - Om beroendet är ett annat mål – "rekursivt" ta detta nya mål
- Om man vill utföra actions utan att editera en beroendefil kan man använda kommandot `touch` *`fil`* som ändrar datumet på filen

MAKE – ACTIONS

- En följd av "actions" som behövs för att kunna nå fram till målet
- Kan var tom
- Vanligen ett kommando
- Varje action **måste** föregås av ett **tab**-tecken!

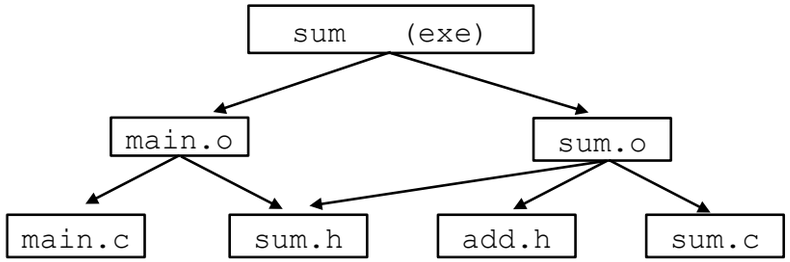
MAKE

- Några fördefinierade macron

```
eval.o : eval.c eval.h
  gcc -c eval.c
```

  - `$@` - namnet på målet (eval.o)
  - `$<` - namnet på det första beroendet (`eval.c`)
  - `$^` - namnen på alla beroenden (`eval.c eval.o`)
  - `$?` – namnen på alla beroenden som är nyare än målet

MAKE – EXEMPEL




```
sum: main.o sum.o
    gcc -o sum main.o sum.o

main.o: main.c sum.h
    gcc -c main.c

sum.o: sum.c sum.h add.h
    gcc -c sum.c
```

MAKE – EXEMPEL 2

```
CC=gcc
CFLAGS=-Wall -pedantic -ansi
LDFLAGS=-L. -lm -lminarutiner

all:     server client

server:  server.o help.o
    $(CC) -o server server.o help.o $(LDFLAGS)

client:  client.o help.o
    $(CC) -o client client.o help.o $(LDFLAGS)

server.o client.o help.o:  help.h
    $(CC) -c $*.c $(CFLAGS)

clean:
```

MAKE – SAMMA SOM FÖREGÅENDE

```
CC=gcc
CFLAGS=-Wall -pedantic -ansi
LDFLAGS=-L. -lm -lminarutiner

all:     server client

server:  help.o

client:  help.o

server.o client.o help.o:  help.h

clean:
    rm -f server client *.o core
```

HEADERFILE

- **EJ** C-kod!
- Exempel på innehåll:
  - Macron
  - Typedefs
  - Funktionsprototyper
  - Extern-deklarationer av globala variabler
  - Includes

fil.c

```
int global = 0;
```

fil.h

```
extern int global;
```

main.c

```
#include "fil.h"
```

FELSÖKNING/DEBUGGNING

- `gdb`
  - Man kan använda debuggern för att kolla på vad som händer när man exekverar programmet eller för att reda ut varför ett program kraschade.
- `ddd`
  - Grafiskt användargränssnitt till `gdb`
- `valgrind` (installerat på linux)
  - Kollar minneshanteringen och hur systemanrop/signaler hanteras.
- `purify`
  - Annat vanligt program för att kontrollera minnesläckor (ej installerat på inst)
- `printf` och `fprintf`
  - Primitivt men fungerar i de flesta lägen

ANVÄNDBARA KOMMANDON

- `cflow`
  - Genererar en C-flödesgraf (från källkoden)
- `strace`
  - Kör ett kommando och skriver ut vilka systemanrop som anropas, mm

ENVIRONMENT

- `char *getenv(const char *name)`
  - funktion för att läsa av ett värde som finns i en environment variable
- `int putenv(const char *str)`
  - tar en sträng på formen ”name=value” och lägger till den till omgivningen
- ```
  int setenv(const char *name,
             const char *value,
             int rewrite)
  ```
- `unsetenv(const char *name)`
  - tar bort en definition av `name`

MINNESHANTERING

- Som ni känner till så kan man få mera minne till variabler etc vid behov. I C gör man det genom att anropa någon funktion i `alloc`-familjen
- `malloc`
  - Den kanske vanligaste är `malloc` som ger dig en pekare till ett minnesutrymme av en viss storlek. Innehållet i minnes-utrymmet är odefinierat när du få tillgång till det.
- `calloc`
  - `calloc` ger dig möjlighet att får utrymme för X antal objekt av storleken Y. Det utrymme man då får har "nollats”
  ```

MINNESHANTERING

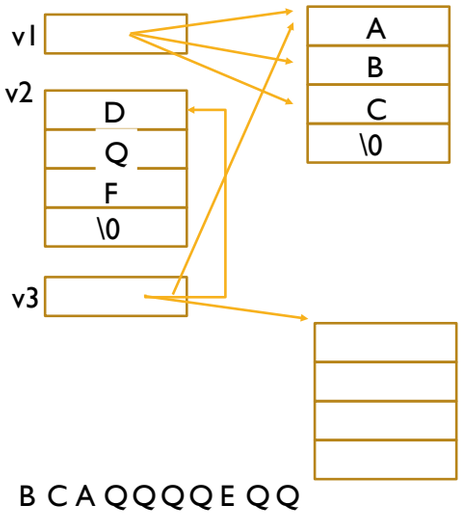
- `realloc`
  - Om redan har ett utrymme för data men behöver ändra storleken kan du använda dig av `realloc`. Notera att det existerade datat ev flyttas om, så använd dig av den nya pekaren. Innehållet i det eventuella nya utrymmet är odefinierat
- `alloca`
  - `alloca` plockar utrymme från stacken i stället för heapen. Det innebär att man inte behöver göra `free` när rutinen är klar men man kan bara använda minnet i funktionen eller i rutiner som anropas av den.
- `free`
  - När man är klar med ett minnessegment så ska man lämna tillbaka det genom att anropa `free`.

PEKARE – ARRAYER

- Vad är p+1 om p är en pekare? (pekararitmetik)
  - Adressen p plus storleken på det p pekar på
  - Varierar alltså beroende på typ
- Arrayer
  - Är egentligen bara en address
  - Indexering är pekararitmetik plus dereferens
  - `v[i]` är samma sak som `*(v + i)`
    - dvs adressen till det i:te elementen räknat från startadressen
  - Kan därför inte enkelt hålla koll på så att man inte refererar utanför arrayen

```
char *v1 = "ABC";
char v2[] = "DEF";
char *v3;

v3 = malloc(4);
v3 = v1;
v3 = v2;
v3[1] = 'Q';
v1 = v1 + 2;
v1--;
putchar(v1[0]);
putchar(v1[1]);
putchar(v1[-1]);
putchar(v2[1]);
putchar(v3[1]);
putchar(*(v2+1));
putchar(*(v3+1));
Putchar(*v2+1);
putchar(*(1+v3));
putchar(1[v3]);
```

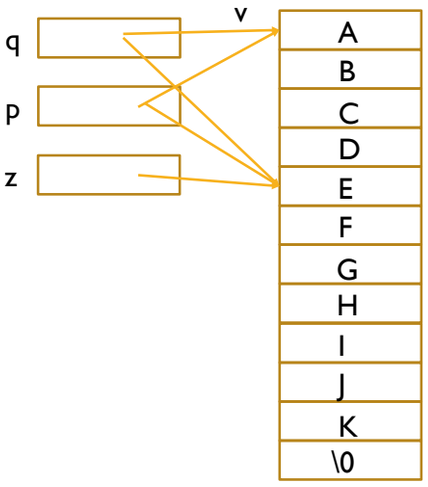


B C A Q Q Q Q E Q Q

```
char v[] = "ABCDEFGHIJK";
char *z;
int *q;
void *p;

q = p = v;
q = q + 1;
z = p = q;

putchar(z[0]);
```




```
char *q;
```

```
double *q;
```

E B I

POPEN/PCLOSE

- Det som vi gjorde tidigare (skapa en pipe till en annan process och sedan läsa eller skriva till den) är så pass vanligt att man har ett speciellt anrop för att göra just detta: `popen` och `pclose`.
- ```
  FILE *popen(const char *command,
              const char *mode);
  ```
- ```
  int pclose(FILE *stream);
  ```