
Systemprogrammering i C

Föreläsning 4

Tobias Gedell

gedell@cs.chalmers.se

Idag

- ▷ Mer om variabler
- ▷ switch
- ▷ hopp
- ▷ Typomvandling
- ▷ union
- ▷ Mer om filer
- ▷ Funktionspekare / qsort
- ▷ Övningstillfällen

Systemprogrammering i C, för. 4 – p. 1/30

Mer om variabler

scope - synlighet för variabler, var de är kända.

En lokal variabel har ett scope från sin deklarasjonspunkt till slutet av det block i vilken variabeln är deklarerad.

En global variabel har ett scope från sin deklarasjonspunkt till slutet av filen.

Systemprogrammering i C, för. 4 – p. 3/30

Systemprogrammering i C, för. 4 – p. 2/30

Mer om variabler - exempel

```
/*  
 scope.c  
*/  
  
int add(int a, int b)  
{  
    int v;           // | scope för a, b  
                    // | | scope för v  
                    // | |  
    v = a + b;      // | |  
                    // | | a, b här är inte samma  
    return v;       // | | a, b som i main!  
}  
  
int main(void)  
{  
    int x, a, b;     // | scope för x, a, b  
                    // |  
    x = a = 1;      // |  
    b = add(x, a);  // |  
                    // |  
    return 0;       // |  
}
```

Systemprogrammering i C, för. 4 – p. 4/30

Mer om variabler - exempel

```
/*  
 scope2.c  
*/  
  
#include <stdio.h>  
  
void f(int a, int b)  
{  
    // | scope för a, b  
    // |  
    for(int i = a; i < b; i++)  
    {  
        // | | scope för i  
        // | |  
        int a = i * i;  
        // | | | scope för a  
        // | | | döljer gamla  
        printf("%d^2 = %d\n", i, a);  
        // | | | a (shadowing)  
    }  
    // |  
    // |  
}
```

Systemprogrammering i C, för. 4 – p. 5/30

Variablers livslängd

Det finns två sorters livslängd för variabler:

- ▷ Automatisk - Variabeln lever inom ett visst block. Vid varje invokering av blocket skapas variabeln för att sedan kastas bort när kontrollflödet lämnar blocket. Exempel på automatiska variabler är `x`, `y`:

```
int f(int x)  
{  
    int y;  
    y = x * 2;  
    return y;  
}
```

Automatiska variabler är oftast allokerade på stacken.

- ▷ Statisk - Variabeln lever från det att den skapats tills programmet terminerar.

Systemprogrammering i C, för. 4 – p. 7/30

Definition och deklaration

En *deklaration* associerar en typ med ett namn.

En *definition* associerar en typ med ett namn samt reserverar minne stort nog att för att lagra värden av typen.

En definition är alltså även en deklaration.

Systemprogrammering i C, för. 4 – p. 8/30

Lagringsklasser

I C finns det fyra lagringsklasser som kan anges vid deklarationer och definitioner: `auto`, `static`, `extern`, `register`.

- ▷ `auto` - Kan enbart användas för lokala variabler. Anger att variabeln ska ha automatisk livslängd. En `auto`-deklaration är även en definition. Om man inte anger någon lagringsklass för en lokal variabel antas den vara `auto`.

`int a;` är alltså ekvivalent med `auto int a;`

Ofta ser man aldrig `auto` eftersom det är "default".

Systemprogrammering i C, för. 4 – p. 8/30

Lagringsklasser - forts

- ▷ `static` - För lokala variabler anger det statisk livslängd. En lokal `static`-variabel kommer ihåg sitt värde mellan funktionsanropen.

Ett exempel:

```
int counter(void)
{
    static int n = 0;
    return n++;
}
```

Om en global variabel ges lagringsklassen `static` kommer den bara kunna användas i samma fil. Detsamma gäller även för funktioner.

Systemprogrammering i C, för. 4 – p. 9/30

Lagringsklasser - forts

- ▷ `register` - Tipsar kompilatorn om att en variabel bör placeras i ett register för att snabba upp åtkomsten.

En bra kompilator som `gcc` gör dock antagligen ett bättre jobb än du! (Om ni vill kan ni kolla det genom att köra "`gcc -O2 -S`")

En `register`-deklaration är även en definition och har automatisk livslängd.

Det är inte tillåtet att ta adressen (med operatoren `&`) till en `register`-deklarerad variabel. *Varför?*

Sammanfattningsvis: Du behöver aldrig använda `register`!

Systemprogrammering i C, för. 4 – p. 11/30

Lagringsklasser - forts

- ▷ `extern` - Är *endast* en deklaration. Anger att en variabel eller funktion finns definierad någon annanstans (i samma fil eller i en annan fil).

En variabel eller funktion kan deklarerars flera gånger i olika filer men ska definieras *exakt en gång* i någon fil.

`extern`-variabler har statisk livslängd.

Exempel:

```
/* var.c */
int x = 0;

/* foo.c */
extern int x;

int f(int y)
{
    return x + y;
}
```

Systemprogrammering i C, för. 4 – p. 10/30

const

```
const int c = 10;
```

- ▷ Typmodifieraren `const` används för att beteckna att ett värde är konstant.
- ▷ En konstant måste initieras!
- ▷ Kompilatorn klagar om den upptäcker försök att förändra konstanter.
- ▷ En variabel deklarerad som `const` brukar läggas i minne som är *read-only*. Ett försök att tilldela en sådan variabel kan då resultera i t.ex. ett *Segmentation fault* (programmet kraschar).

Systemprogrammering i C, för. 4 – p. 12/30

const - forts

- ▷ Läs typen “inifrån och ut” för att förstå vad som är konstant:

```
const int x = 5;      x är en int som är konstant.
const int *p;        p pekar till en int som är konstant.
                    Alltså kan p bara peka på konstanter.

int y;
int * const cp = &y; cp är en konstant pekare till en int.
                    Dvs cp pekar alltid på y.
```

- ▷ När `const` finns med i typen för en funktionsparameter, säger den något om funktionens beteende:

```
size_t strlen(const char *s);
```

Funktionen `strlen` kommer inte att förändra den sträng som pekas ut av `s`. I funktionens kropp betraktas `s` som en pekare till konstanta chars.

goto

I C kan man hoppa till en utvald rad genom att använda `goto`. Ett exempel:

```
/*
 * goto.c
 */

#include <stdio.h>

int main(void)
{
    printf("a\n");
    goto lbl1;

    printf("b\n");

lbl1:
    printf("c\n");

    return 0;
}
```

switch

Med en `switch`-sats kan multipla val göras lite enklare:

```
switch(x)
{
    case 1:
        printf("Ett!\n");
        break;

    case 2:
    case 3:
        printf("Två eller Tre!\n");
        break;

    default:
        printf("Något annat!\n");
}
```

goto - forts

`goto lbl;` Hoppa till labeln `lbl:` i samma funktion.

`goto` kan ibland vara praktiskt men bör generellt undvikas då de lätt leder till “spagetti-kod” (se övn. 4.31).

Ett “berättigat” `goto` kan tex direkt hoppa ut ur en djup nästlad loop och/eller en mängd `if`-satser, när ett fel upptäckts. En form av exception.

Hoppsatser

- ▷ `break;` - Hoppar ut ur närmsta omslutande `for`-, `while`-, `do-loop` eller `switch`-sats.
- ▷ `continue;` - Hoppar till nästa varv i närmast omslutande `for`-, `while`- eller `do-loop`.
- ▷ `return expr;` - Returnerar (direkt) från den omslutande funktionen med värdet av `expr`.

I C kan man inte namnge loopar som man kan i Java:

```
lbl: for(int i = 0; i < 10; i++)
    for(int j = 0; j < i; j++)
    {
        if(...) break lbl;
    }
```

Systemprogrammering i C, för. 4 – p. 17/30

Type cast

Ofta när man hanterar pekare behöver man göra explicita type casts. Ett exempel är `malloc` som returnerar ett värde av typ `void*`. Den returnerar den typen eftersom den inte vet vilka slags värden du tänkt spara i minnet.

För att kunna använda pekaren så måste du casta om den till `tex char*` eller `int*`.

```
int *p;
p = (int*)malloc(sizeof(int) * 100);
```

Systemprogrammering i C, för. 4 – p. 19/30

Typomvandling

För många operatörer sker automatisk typomvandling (*widening* eller *integral promotion*) när de två operanderna inte har samma typ. Den ena eller båda av operanderna omvandlas till en "bredare" typ enligt (ungefär):

```
char -> short -> int/unsigned -> float -> double
```

så att de båda får samma typ. Denna omvandling sker innan operatören tar effekt. Exakt vilka omvandlingar som sker är komplicerat, se boken sid 131-133 för en mer ingående förklaring.

Ibland, men dock inte så ofta, kan det vara nödvändigt att explicit omvandla typer med en sk *type cast*: `(float)2`.

För funktioner fungerar det på ungefär samma sätt. Om en funktion vill ta in en `float` men får en `int` så kommer typkonvertering att ske.

Systemprogrammering i C, för. 4 – p. 18/30

Type cast

```
int a = 12345;
char *p = &a;
```

warning: initialization from incompatible pointer type

```
int a = 12345;
char *p = (char*)&a;
```

```
for(int i = 0; i < sizeof(int); i++)
    printf("%x ", p[i]);
```

Systemprogrammering i C, för. 4 – p. 20/30

union

En unionstyp i C deklaras på samma sätt som en `struct`:

```
union person
{
    char  name[20];
    float weight;
    int   age;
};
```

- ▷ Till skillnad från en `struct`, som innehåller alla komponenterna, så lagrar en union endast *en* av sina komponenter åt gången.
- ▷ Det finns inget sätt att fråga en union vilken komponent den innehåller!
- ▷ unioner manipuleras på samma sätt som `struct`ar: (. och -> för selektering).

Systemprogrammering i C, för. 4 – p. 21/30

Mer om filer - forts

- ▷ `int fflush(FILE *stream);`
Tömmer bufferten genom att skriva ut all buffrad data.
- ▷ `void setbuf(FILE *stream, char *buf);`
`int setvbuf(FILE *stream, char *buf, int type, size_t size);`

Kontrollerar buffringen för `stream`. Värdet på `type` anger hur buffring kommer att utföras:

```
_IOFBF  Full buffring (upp till size)
_IOLBF  Radbuffring
_IONBF  Ingen buffring
```

- ▷ `int fgetpos(FILE *stream, fpos_t *pos);`
Lagrar filpositionen för `stream` i `pos`.
- ▷ `int fsetpos(FILE *stream, const fpos_t *pos);`
Sätter filpositionen för `stream` till den som lagrats i `pos`.

Systemprogrammering i C, för. 4 – p. 23/30

Mer om filer

Lite kort:

- ▷ **Abstrakt typ:** `FILE *`.
- ▷ **Hantera filer:** `fopen()`, `fclose()`, `freopen()`.
- ▷ **Teckenbaserad I/O:**
`getchar()`, `fgetc()`, `putchar()`, `fputc()`, `fgets()`, `fputs()`.
- ▷ **Formatterad I/O:**
`printf()`, `fprintf()`, `scanf()`, `fscanf()` [, `sprintf()`, `sscanf()`].
- ▷ **Direct I/O:** `fread()`, `fwrite()`.
- ▷ **End-of-file/felindikator:** `feof()`, `ferror()`, `clearerr()`.

Systemprogrammering i C, för. 4 – p. 22/30

Mer om filer - forts

- ▷ `long ftell(FILE *stream);`
Returnerar filpositionen i `stream`.
- ▷ `void rewind(FILE *stream);`
`int fseek(FILE *stream, long offset, int whence);`
Sätter filpositionen i `stream` beroende på `whence`:
`SEEK_SET` Sätt till offset
`SEEK_CUR` Sätt till offset från nuvarande pos.
`SEEK_END` Sätt till offset från slut-pos.
Anropet `rewind(stream)` är ekvivalent med `(void)fseek(stream, 0L, SEEK_SET)`.
- ▷ `FILE *tmpfile(void);`
Skapa en temporär fil.
- ▷ `char *tmpnam(char *s);`
Returnerar ett filnamn som kan användas som temporär fil. Om `s` inte är `NULL` fylls filnamnet i där.

Systemprogrammering i C, för. 4 – p. 24/30

Funktionspekare

I C är det möjligt att peka till funktioner, lagra funktionspekare i variabler och datastrukturer samt skicka funktionspekare som argument och även returnera dem från funktioner.

- ▷ Givet en funktionsdefinition:

```
int foo(int n, char *str) { ... }
```

är funktionsnamnet (`foo`) en pekare till funktionen.

- ▷ I C sker funktionsanrop med postfixoperatoren (`()`). I uttrycket: `foo(3, "hej")` får alltså operatoren (`()`) 3 argument: `foo`, `3` och `"hej"`.

Systemprogrammering i C, för. 4 – p. 25/30

Funktionspekare - exempel

```
/*
   qsort.c
*/

/*- void qsort(void *base, size_t nmemb, size_t size,
// -         int(*compar)(void *, void *));

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUMBERS 20

int compare_int(const void *v1, const void *v2)
{
    int *i1 = (int*)v1;
    int *i2 = (int*)v2;

    if(*i1 < *i2) return -1;
    if(*i1 == *i2) return 0;
    return 1;
}
```

Systemprogrammering i C, för. 4 – p. 27/30

Funktionspekare - forts

- ▷ Vi kan deklarera och tilldela variabler av funktionstyp:

```
int (*f)(int, char *);
f = foo;
```

- ▷ Anrop av den funktion `f` pekar på sker med `f(3, "hej")`.

- ▷ Funktioner kan returnera pekare till funktioner:

```
int (*fie(int n)) (int, char *) { ... }
```

`fie` är en funktion av ett `int`-argument, som returnerar pekare till en funktion av (`int`, `char *`) som returnerar `int`.

Returvärdet från `fie` är ett vanligt funktionspekarvärde:

```
f = fie(2);           det kan lagras
fie(5)(7, "hopp!");  eller anropas direkt.
```

- ▷ Funktionspekare kan användas för att dynamiskt ändra ett programs beteende. Detta används mycket tex inom implementering av grafiska användargränssnitt.

Systemprogrammering i C, för. 4 – p. 26/30

Funktionspekare - exempel

```
int main(void)
{
    int arr[NUMBERS];

    srand(time(NULL));

    for(int i = 0; i < NUMBERS; i++)
        arr[i] = rand() % NUMBERS;

    printf("The array before sorting:\n");
    for(int i = 0; i < NUMBERS; i++) printf("%d ", arr[i]);
    printf("\n");

    qsort(arr, NUMBERS, sizeof(int), compare_int);

    printf("The array after sorting:\n");
    for(int i = 0; i < NUMBERS; i++) printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Systemprogrammering i C, för. 4 – p. 28/30

Övningstillfällen

Jag har lagt upp en enkät om vilka övningstillfällen ni helst vill ha handledning på.

Logga in och svara på enkäten så fort som möjligt!

Läs-/övningsanvisningar

Läs kapitel:

- ▷ 4: 14, 16
- ▷ 5: 9 - 13
- ▷ 6: 16 - 18
- ▷ 8: 8 - 14, (15)
- ▷ 9: 7
- ▷ 10: 1 - 8, (9)
- ▷ 11: 1 - 7
- ▷ A: 12, 14

Förslag på lämpliga övningar är:

- ▷ 10: 5, 9, 14, 15, 34, (20)
- ▷ 11: 4, 12, 13, 23, 24, (7, 14, 15)