Föreläsningsöversikt

- arrayer
- `for`–satsen för loopar
- pekare, relationen mellan arrayer och pekare
- strängar
- programparametrar
- formatterad utmatning och inmatning
- symboliska konstanter
- något om inlämningsuppgift 1

Arrayer

Arrayer är homogena sekvenser av någon specifik elementtyp. Antalet element kan vara fix eller variabel. Med en array kan man alltså gruppera samman och manipulera ett antal element av samma typ.

Deklaration

```
int a[100];
```

deklarerar att `a` är en array med plats för 100 heltal.

Arrayelement indexeras *alltid* från index 0 upp till *storleken* $-$ 1: i det här fallet 0 till 99.

Indexering

Arrayer indexeras med `arr[exp]`, där `exp` måste vara ett heltalsuttryck som evaluerar till ett legalt index, dvs minst 0 och högst *storleken* $-$ 1.

`arr[exp]` är en variabel: dess värde hämtas automatiskt i uttryck, och den kan tilldelas med `=`.

Begränsningar

Hela arrayer kan ej tilldelas – elementvis tilldelning är nödvändig.

Arrayparametrar till funktioner fungerar inte som andra parametrar – återkommer till detta strax.

Flerdimensionella arrayer

En array kan ha en annan arraytyp som elementtyp:

```
int    mat[2][3] = { {1,0,0}, {0,9,9} };
/*     mat         -- deklarera mat som
 *        [2]      -- array med 2 element av
 *           [3]   -- array med 3 element av
 * int             -- int
 */

mat[1][2] = 3;
```

Representation i minnet

Elementen ligger efter varandra, med [0] på första adressen, [1] omedelbart efter, osv.

En flerdimensionell array ser ut som en vektor:

```
-------------------------------------------------------
| [0][0] | [0][1] | [0][2] | [1][0] | [1][1] | [1][2] |
-------------------------------------------------------
\_________________________/\_________________________/
          mat[0]                      mat[1]
```


Repetition – for

```
for(init; cond; step)
    stmt
```

Evaluerar först `init` för ev. sidoeffekters skull.
Evaluerar därefter `cond`: om det blev sant exekveras kroppen, därefter evalueras `step` för ev. sidoeffekters skull, och kontrollen går tillbaka för att evaluera och testa `cond`.

Både `init` och `step` kan utelämnas: i så fall sker ingenting vid initialiseringen resp. stegningen.

Även villkoret `cond` kan utelämnas: då blir det en "oändlig" loop som endast kan avslutas med kontrollbrytande satser som `return` eller `break`.


```
int vec[6], mat[2][3], i, j, sum;

sum = 0;
for(i = 0; i < 6; ++i)
    sum += vec[i];

for(i = 0; i < 2; ++i)
    for(j = 0; j < 3; ++j)
        mat[i][j] = vec[i*3+j];

for(;;) { /* loop with non-trivial structure */
   work_part_1;
   if (termination_condition)
       break; /* or return */
   work_part_2;
}
```


Pekare

En pekare är en *indirekt* referens som kan referera variabler av en viss typ eller ha ett speciellt "ingen referens"–värde.

Deklaration

```
int *ptr;
```

deklarerar att `ptr` är en pekare som kan peka på variabler av typen `int`, eller anta värdet "ingen referens".

Typen `T*` är "pekare till T". (En typ kan beskrivas som en deklaration där variabelnamnet har suddats ut.)

Operatorerna & och *

Operatorn `&` tar en variabel av typ `T` och ger som resultat en pekare till variabeln, av typ `T*`. `&` kallas för *adressoperatorn*.

Operatorn `*` tar en pekare av typ `T*` och ger som resultat variabeln, av typ `T`, som pekaren refererar till. Pekaren får inte ha "ingen referens"–värdet. `*` kallas för *indirektionsoperatorn*.

```
int i, a[10];
int *ptr;
i = 2;
ptr = &i;        /* *ptr == 2 */
*ptr = 3;        /* i == 3 */
ptr = &a[5];
*ptr = 27;       /* a[5] == 27 */
```


NULL–pekaren

"ingen referens"–värdet skrivs som `NULL` eller `0` i källkoden. Här är `0` *inte* ett heltalsvärde, utan speciell syntax som tolkas av kompilatorn.

Den symboliska konstanten `NULL` fås genom att göra `#include <stddef.h>` i början av C–filen.

```
ptr = 0;
ptr = NULL;
if (ptr != 0) ...
if (ptr != NULL) ...
if (ptr) ...
```


Pekarindexering och pekararitmetik

```
     -----------------
arr: | x | y | z |...|
     -----------------
       |
      ptr
```

En pekare indexeras som en array: `ptr[i]`. Detta är samma sak som `*(ptr + i)`. Add addera ett heltal `i` till en pekare kallas för *pekararitmetik*: pekaren flyttas i steg i den underliggande arrayen. (En skalär variabel ses som en array med ett element.)

```
ptr = &arr[0]; ptr[i] == *(ptr + i) == arr[i]
ptr = &arr[2]; ptr[i] == *(ptr + i) == arr[2+i]
ptr[-2] == arr[0]
```


Arrayer i funktionsanrop

I uttryck konverteras ett arraynamn automatiskt till en pekare till arrayens första element.

En array som skickas som parameter till en funktion beter sig annorlunda än andra typer av parametrar, eftersom arraynamnet ersätts med en pekare till arrayen. Om den anropade funktionen gör en tilldelning via pekaren modifieras arrayen!

En funktionsparameter kan deklareras som `T name[]`, med eller utan storlek, men den konverteras automatiskt till `T *name`. (Detta gäller endast funktionsparametrar.)

Varför? Implementationstekniska skäl. Kopiering av arrayer tar tid, men fungerar även inte i vissa fall (flera steg av funktionsanrop).

```
void proc1(int a[])     /* same as: int *a */
{
    a[5] = 99;          /* *(a + 5) = 99; */
}

void proc2(void)
{
    int arr[10], *ptr;
    ptr = arr;          /* same as: ptr = &arr[0] */
    arr[5] = 0;
    proc1(a);           /* proc1(&a[0]); */
    /* arr[5] == 99 !! */
}
```


Strängar

Strängar representerar text av olika slag. En sträng är en array av `char`, exempel: `char name[10]`.

Strängar är ofta av variabel längd. Konventionen i C är att en extra noll-byte placeras efter strängen för att markera dess slut.

Ex: strängen `"DATOR"` representeras (i ASCII) som:

```
-------------------------
| 68| 65| 84| 79| 82|  0|
-------------------------
 [0] [1] [2] [3] [4] [5]
```


Tecken

En strängkonstant består av en sekvens tecken. Vanliga tecken skrivs som de är, men specialtecken skrivs med \–notation:

`\n` är radslutstecknet
`\t` är tabuleringstecknet
`\\` är `\`
`\"` är `"`
`\007` är heltalet 7 (BEL)

En teckenkonstant skrivs som ett tecken (vanligt eller \–notation) mellan ett par `'`–tecken. Exempel: `'A'`, `'\n'`. En teckenkonstant är ett heltal med motsvarande teckenkod som värde.

Biblioteksfunktioner för strängar

Biblioteket `<string.h>` definierar funktioner för att manipulera strängar. Även andra bibliotek har strängrelaterade funktioner.

`strlen(str)` beräknar längden av strängen

`strcpy(to, from)` kopierar strängen `from` till strängen `to`

`strcmp(str1, str2)` jämför strängarna och returnerar 0 om de är lika; annars ett negativt (positivt) tal om den första är mindre (större) än den andra

`atoi(str)` (`<stdlib.h>`) konverterar strängen till ett heltal (`"123"` blir `123`)

Strängjämförelser

Fungerar `string1 == string2` ?

Detta kollar endast om strängarna är identiska, dvs samma pekare.

```
char *str1, *str2;

if (str1 == str2)
    printf("they are the same string\n");
else if (strcmp(str1, str2) == 0)
    printf("they look the same\n");
else
    printf("they are different\n");
```


Strängar och pekarare

```
int my_strlen1(char str[])
{
    int len = 0;
    while (str[len] != '\0')
        ++len;
    return len;
}
int my_strlen2(char *str)
{
    int len = 0;
    while (*str++ != '\0')
        ++len;
    return len;
}
```


Programparametrar till `main()`

Man kan ange parametrar när man startar ett program. Dessa parametrar skickas till `main` som en array av strängar:

```
int main(int argc, char *argv[])
{
    int i;
    printf("there are %d parameters\n", argc);
    for(i = 0; i < argc; ++i)
        printf("parameter %d is %s\n", i, argv[i]);
    return 0;
}
```

Programnamnet skickas automatiskt i `argv[0]`. Parametrarna på kommandoraden hamnar i `argv[1]` och uppåt.

Sekvensiell text–I/O med `<stdio.h>`

`#include <stdio.h>` ger tillgång till filoperationer från C:s standardbibliotek – nästan alla program behöver dessa.

En fil representeras som en pekare till ett abstrakt filobjekt, `FILE*`.

`<stdio.h>` har operationer för att öppna och stänga filer, skriva och läsa enskilda bytes eller större block.

Dessutom finns operationer för *formatterade* utskrifter och inmatning av datavärden.

C har tre fördefinierade standardfiler: `stdin` (standard input), `stdout` (standard output) och `stderr` (standard error). Dessa är normalt sekvensiella textfiler kopplade användarens tangentbord och bildskärm.

Formatterad utmatning: `printf`

```
int printf(char *fmt, arg1, ...);
```

`fmt` är en *formatsträng*. Funktionen kopierar tecken från formatsträngen till `stdout`.

Tecknet `%` inleder en *konverteringsspecifikation*. Nästa argument efter formatsträngen läses, konverteras till en sträng, och kopieras till `stdout`.


```
printf("Her name is %s and she is %d years old\n",
       "Anna", 27);
printf("%d looks like %#x in hex\n", 1024, 1024);
printf("'%c' == 65\n", 65);
```

genererar utskriften:

```
Her name is Anna and she is 27 years old
1024 looks like 0x400 in hex
'A' == 65
```

`%d` konverterar heltal till decimal form, `%x` till hexadecimal form.

`%s` skriver ut en sträng, `%c` skriver ut ett heltal som ett tecken.

(Finns mycket mer. `man -s 3c printf` eller läs dokumentation.)

Formatterad inmatning: `scanf`

```
int scanf(char *fmt, ptr1, ...)
```

Formatsträngen består av en sekvens av *formatdirektiv*:

Ett blankt tecken matchar en sekvens av blanka tecken i `stdin`.

Ett icke–blankt tecken (utom `%`) matchar samma tecken.

`%` inleder en *konverteringsspecifikation*. Indata matchas, konverteras till ett värde, och skrivs dit nästa pekarargument pekar.

`scanf()` matchar `stdin` mot direktiven tills formatsträngen tar slut eller en matchning misslyckades. Den returnerar antalet lyckade *konverteringar*.


```
#include <stdio.h>

int main(void)
{
   int i;
   double d;
   printf("Skriv ett heltal och ett flyttal\n");
   scanf("%d %lf", &i, &d);
   printf("Du skrev: %d %f\n", i, d);
   return 0;
}
```

Även om `printf`:s och `scanf`:s formatsträngar liknar varandra så finns det väsentliga skillnader.

Symboliska konstanter

Typproblem: hur ge symboliska namn på heltalskonstanter?

Alternativ 1: en ny uppräknad typ

```
enum day { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
enum { MAX_ELEMENTS = 100 };
```

Alternativ 2: konstanter i preprocessorn:

```
#define MAX_ELEMENTS 100
#define Mon 0
#define Tue 1
...
```

I praktiken är `#define` mycket vanligare än `enum`.