Föreläsning 13

Pekare, vektorer, moduler och strängar

Moduler i C

Anta att vi vill definiera två funktioner för summa och fakultet, och lägga dem i en modul “intmath”.

Då skapar man följande filer:

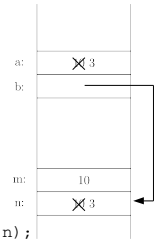
intmath.h

intmath.c

Simulering av call-by-reference med pekare

```
#include <stdio.h>
void change (int a , int *b) {
  a = 3;
  *b = 3;
}

int main () {
  int m = 10;
  int n = 10;
  change (m, &n);
  printf (“m=%d och n=%d\n”, m, n);
  return 0;
}
```



intmath.h

```
int fact (int n);

int sum (int n);
```

Vi kan göra något liknande i Python

Vi efterliknar en variabels minnesplats med vektorelement:

```
def change (a, b):
  a = 3
  b[0] = 3

m = 10
n = [10]
change (m, n)

print (m, n) #kommer att skriva ut 10 [3]
```

intmath.c

```
#include “intmath.h”

int fact (int n) {
  if (n == 0)
    return 1;
  else
    return n * fact (n - 1);
}

int sum (int n) {
  if (n == 0)
    return 0;
  else
    return n + sum (n - 1);
}
```

intprg.c

- Huvudprogram:

```
#include <stdio.h>
#include "intmath.h"

int main (){
  int n = 0;
  scanf("%d", &n);
  printf("%d\n", fact (n));
  printf("%d\n", sum (n));
  return 0;
}
```

- Programmet kompileras med följande kommando:

```
gcc -o intprg intprg.c intmath.c
```

Vektorer

- Deklaration
  `<elementtyp> <namn> [<antal element>];`
- Exempel:

```
int vektor[6];
int v1[5] = {10, 22, 31, 44, 15};
int v2[] = {10, 22, 31, 44, 15};
int w[10] = {0};
```

- Åtkomst:
  - En vektor med n element har index från 0 till n - 1.
  - Exempel:

```
vektor[5] = 10;
int n = vektor[0];
```

Räckvidd och livslängd

- Räckvidden (scope) för en variabel är den *del av programmet* där man kan referera (komma åt) variabeln.
  - global: hela programmet
  - modulglobal: en fil (kompileringsenhet)
  - lokal: i en funktion eller ett block
- Livslängden (extent) är den *tid under programmets exekvering* då en variabel har ett (meningsfullt) värde.
  - statisk: hela exekveringstiden
  - automatisk: minne för variabeln skapas (allokeras) då vi går in i funktion eller block och tas bort (avallokeras) då vi går ut.
  - dynamisk (gäller datastrukturer): Vi allokerar/avallokerar

Likheter mellan vektorer och pekare

- Stora vektorregeln:
  En vektors namn ger en pekare till vektorns första element.
- Indexregeln:
  *Om p är en pekare, gäller att p[i] och *(p + i) är ekvivalenta.*
- Exempel:

```
int v[5];
int *p;
p = v;
*(v + 2) = 10;
p[3] = 2;
```

Exempel

```
int m = 0; /* m är global och statisk /
static int n = 0 ; /* modulglobal och statisk /
extern int p; /* definierad i någon annan fil */

int foo (){
  int i = 0; /* i är lokal och automatisk */
  static int j = 0; /* j är lokal och statisk */
  return i;
}
```

Olikheter mellan vektorer och pekare

- Ett vektornamn är en **konstant**, en pekare är en **variabel**

```
int v[5];
int *p;
p = v; /* OK /
v = p; /* Fel - varför? */
```

- En vektor får minne reserverat för elementen.

```
int v[5], *p;
v[0] = 10; /* OK */
p[0] = 10; /* Fel - varför? */
p = &v[1]; /* OK /
p[0]= 10;  /* OK nu - varför? */
```

Dynamisk minnesallokering

```
#include <stdio.h>

int main (){
  int max = 10;
  int n;
  int vek[max];
  printf ("Hur många tal ska matas in?");
  scanf ("%d",&n);
  if (n > max)
    printf ("Max 10 tal!");
  ...
  return 0;
}
```

Nytt försök

```
#include <stdio.h>
#include <stdlib.h>
int main (){
  int n;
  int *vek;
  printf ("Hur många tal ska matas in?%d\n");
  scanf("%d",&n);
  vek = malloc (n * sizeof (int));
  if (vek == NULL) {
    printf ("Slut på minne!”);
    exit(1);
  }
  ...
  free (vek);
  return 0;
}
```

Dynamisk minnesallokering

- `<pekare> = malloc (<antal bytes>)`

  ```
  #include <stdlib.h>
  void *malloc (size_t size);
  ```

- `malloc` returnerar en generisk pekare (`void *`)
  - `void *` kan peka till vilken typ som helst.
  - Om `malloc` misslyckas, returneras den speciella pekaren `NULL` (definierad i `stdlib.h`)
- Mängden minne som ska allokeras anges som antal bytes (ett tal med den speciella typen `size_t`).

Vanligt misstag

```
#include <stdio.h>

int *getnumbers (){
  int v[10]; // automatisk variabel
  int i;
  for (i=0; i < 10; i++)
    v[i] = i;
  return v;
}
```

Tänkbara lösningar:

- Låt v vara en pekare och allokera minne dynamiskt (malloc)
- Passa in vektorn som argument (deklareras som pekare)

Dynamisk minnesallokering

- `<nBytes> = sizeof (<typ/uttryck>)`
  - Operatorn `sizeof` beräknar storleken hos sin operand
  - Operanden är en typ eller ett uttryck vars kontrolleras
- `free (<pekare>)`
  - Minne som allokeras av `malloc` ska återlämnas med `free`:

    ```
    void free (void *ptr);
    ```

Strängar

- En sträng i C är en vektor av char.
  - Efter det sista tecknet som ingår i strängen lägger man tecknet nul, ‘\0’.
- Strängkonstanter
  - En strängkonstant omringas av citationstecken. De får inte modifieras.

    ```
    char str[] = “hej”;
    ```

- Ovanstående är detsamma som:

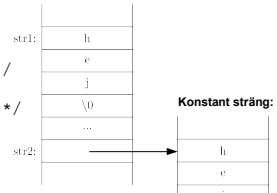
    ```
    char str[] = {‘h’, ’e’, ‘j’, ’\0’};
    ```

Skillnad mellan pekare och vektor

```
char str1[] = “hej”;
char *str2 = “hej”;

str1[1]=‘a’; /* OK */

str2[1]=‘a’; /* FEL */
```



I/O med strängar

```
#include <stdio.h>

int main () {
  char name[20];
  printf ("Namn: “);
  scanf ("%s", name); /* OBS ingen & före name */
  printf ("Wow %s, vilket fint namn!\n”, name);
  return 0;
}
```

Strängar som parameter/argument

```
#include <stdio.h>

int strlen (char str[]) {
  int i = 0;
  while (str[i] != '\0')
    ++i;
  return i;
}

int main () {
  char name[20];
  printf ("Namn: ");
  scanf ("%s", name);
  printf ("Ditt namn har %d tecken\n", strlen(name));
  return 0;
}
```