Föreläsningsöversikt

- Poster
- Dynamiska datastrukturer, `malloc()`
- Rekursiva datastrukturer mha poster
- Exempel: listor, hashtabeller.

Sammansatta data utan poster

Exempel 1: Ett antal (x, y) punkter.

```
#define MAX_POINTS 100
double x[MAX_POINTS], y[MAX_POINTS];
unsigned int nrpoints;

void move(unsigned int pointnr, double dx, double dy)
{
    x[pointnr] += dx;
    y[pointnr] += dy;
}
```


Sammansatta data utan poster

Exempel 2: Databas med bilmodeller.

brand	type	modelyear	weight	doors
FORD	sedan	1972	1250	4
SAAB	combi	1968	1070	5
FORD	minibus	1968	1840	6

```
#define MAX_CARS 1000
char brand[MAX_CARS][10];
enum type type[MAX_CARS];
int modelyear[MAX_CARS];
int weight[MAX_CARS];
int doors[MAX_CARS];
```


Sammansatta data utan poster

Problem:

- Indirekt namngivning mha arrayindex. Svag typsäkerhet.
- Omöjligt att se ett “objekt” som en egen entitet eftersom vi alltid måste referera det via den mängd det lagras i (= arrayerna).

Poster

En *post* är en fix sekvens av namngivna *fält* av möjligen olika typer.

Deklaration av posttyp

```
struct postnamn {
     deklarationer
     ...
};
```

deklarerar typen `struct` *postnamn* som en ny posttyp. De inre deklarationerna bestämmer fältens typer, namn och ordning.

Deklaration av postvariabel

```
struct postnamn varnamn;
```

deklarerar variabeln *varnamn* till att vara av typen `struct` *postnamn*.

```
struct complex { double re; double im; };

struct complex x, y;
```


Access av postvariabel

```
varnamn.fältnamn = uttryck;
pekare->fältnamn = uttryck;
varnamn1 = varnamn2;
```

Uttrycket *varnamn*.*fältnamn* är en variabel. Dess typ är den typ som *fältnamn* har i den posttyp som *varnamn* är deklarerad till.
Uttrycket *pekare*->*fältnamn* är samma sak som (**pekare*).*fältnamn*.

```
struct complex x, y, *p;
double d;
x.re = d + 3.14;
d = y.im;
p = &y; p->im = 3.14;
x = y;
```


Exempel på posttyper

```
struct point {
    double x;
    double y;
};

struct point move(struct point p, double dx, double dy)
{
    struct point newp;
    newp.x = p.x + dx;
    newp.y = p.y + dy;
    return newp;
}
```

```
void move(struct point *p, double dx, double dy)
{
    p->x += dx;
    p->y += dy;
}
```

Det är vanligare att poster skickas via pekare, och att förändringar görs med sidoeffekter.


```
struct car {
    char brand[10];
    enum type type;
    int modelyear;
    int weight;
    int doors;
};
```



```
void example(void)
{
    struct car mycar;
    strcpy(mycar.brand, "OPEL");
    mycar.type = sedan;
    mycar.modelyear = 1990;
    mycar.weight = 975;
    mycar.doors = 4;
    print_car(mycar);
}
void example(void)
{
    struct car mycar = { "OPEL", sedan, 1990, 975, 4 };
    print_car(mycar);
}
```


Statiska versus Dynamiska datastrukturer

Statisk datastruktur: Antalet element är bestämt i förväg, och kan ej ändras när programmet körs. Ofta lagras data i överdimensionerade arrayer, vilket slösar med minne i normala fall, men ger programfel då fler element behövs än vad arrayerna har plats för.

Dynamisk datastruktur: Antalet element kan anpassas under körning. Data placeras i dynamiskt allokerat minne, och programvariablerna är *pekare* till detta minne.

Exempel: dynamisk array

```
#include <stdlib.h>
int *array = NULL;
int nrelements = 0;
void grow_the_array(void)
{   int *old_array = array, i, new_nrelements;
    if (nrelements) new_nrelements = nrelements * 2;
    else new_nrelements = 10; /* whatever */
    array = malloc(new_nrelements*sizeof(int));
    for(i = 0; i < nrelements; ++i)
        array[i] = old_array[i];
    free(old_array);
    nrelements = new_nrelements;
}
```


- operatorn `sizeof(`*typnamn*`)` talar om hur många bytes ett element av den typen behöver; multiplicerat med antal element fås antalet bytes för en array med den storleken
- funktionen `malloc(`*nrbytes*`)` allokerar *nrbytes* dynamiskt minne, och returnerar en "generisk" pekare till detta minnesblock; pekaren konverterar man sedan till den typ som minnet ska användas som
- `free(`*pekare*`)` frigör ett minnesblock som `malloc()` har allokerat; efter detta anrop får programmet *INTE* använda detta minnesblock

Rekursiva datastrukturer

Med en *rekursiv* datastruktur menas att strukturen refererar till objekt av samma typ som strukturen själv.

Evivalent: en rekursiv datatyp är en typ som innehåller delar av samma typ som sig själv.

Två typiska exempel:

- listor: varje listelement innehåller (refererar till) ett annat listelement av samma typ
- träd: varje trädnod refererar till ett antal andra trädnoder (listor är specialfall av träd)

Rekursiva datastrukturer med poster

Exempel: listor av heltal. Det finns en tom lista, och varje icke-tom lista består av ett heltal och en annan lista.

Ansats:

```
struct list {
    int i;
    struct list next;
};
```

Fungerar inte!

- C tillåter inte direkt självreferens, då värdena av den typen skulle bli oändligt stora
- Hur beskriver vi den tomma listan?

Rekursiva data med poster och pekare

Korrekt formulering av list–datatypen:

```
struct list {
    int i;
    struct list *next;
};
```

C tillåter att man använder pekare till posttyper innan dessa är fullt definierade.

I detta fall kommer ett listelement att bestå av två fält i minnet: ett fält med heltalsvärdet, och ett fält som *pekar* på resten av listan.

Den tomma listan representeras normalt med `NULL`–pekaren.

Listor: beräkna längd

Naiv ML–liknande lösning:

```
int length(struct list *list)
{
    if (list == NULL)
        return 0;
    else
        return 1 + length(list->next);
}
```

Obra, ty rekursionsdjupet beror på listans längd. Stacken kan ta slut ⇒ runtimefel.

Listor: beräkna längd

ML–liknande lösning med svansanrop:

```
int length2(struct list *list, int len)
{
    if (list == NULL)
        return len;
    else
        return length(list->next, len+1);
}
int length(struct list *list) { return length2(list, 0); }
```

Skulle fungerat bra i ML, men C (liksom alla andra procedurella språk) hanterar inte svansanrop ⇒ stacken kan ta slut.

Listor: beräkna längd

```
int length(struct list *list)
{
    int len = 0;
    while (list != NULL) {
        len = len + 1;
        list = list->next;
    }
    return len;
}
```

Typisk och korrekt lösning med hjälp av loop–sats.

Listor: skapa listnod

```
#include <stdlib.h>
struct list *mk_list(int i, struct list *next)
{
    struct list *tmp = malloc(sizeof(struct list));
    if (tmp == NULL) {
        fprintf(stderr, "OUT OF MEMORY\n");
        exit(1);
    }
    tmp->i = i;
    tmp->next = next;
    return tmp,
}
```

Typisk stil på hjälpfunktion för skapande av dynamiskt data.

Listor: insättning i början

```
struct list *head;

void prepend(int i)
{
    head = mk_list(i, head);
}
```


Listor: insättning i slutet

```
void append(int i)
{
    if (head == NULL) {
        head = mk_list(i, NULL);
    } else {
        struct list *last = head;
        while (last->next != NULL)
            last = last->next;
        last->next = mk_list(i, NULL);
    }
}
```


Listor: insättning i ordnad lista

```
void insert(int i)
{
    struct list *prev = NULL, *here = head;
    while (here != NULL && here->i < i) {
        prev = here;
        here = here->next;
    }
    if (prev == NULL)
        head = mk_list(i, here);
    else
        prev->next = mk_list(i, here);
}
```

Kan användas för ex.vis. sortering.

Kortsluten evaluering av booleska uttryck

Operatorn `&&` beräknar logiskt AND av två booleska uttryck, men det andra uttrycket evalueras endast om det första var sant.

Operatorn `||` beräknar logiskt OR av två booleska uttryck, men det andra uttrycket evalueras endast om det första var falskt.

Listor: borttagning av visst element

```
void delete(int i)
{
    struct list *prev = NULL, *here = head;
    while (here != NULL && here->i != i) {
        prev = here;
        here = here->next;
    }
    if (here != NULL)
        if (prev == NULL)
            head = here->next;
        else
            prev->next = here->next;
}
```


Hashtabeller

Problem: implementera en datastruktur som låter oss lagra specifika nyckelvärden (ex.vis. heltal eller strängar), testa om ett nyckelvärde finns i tabellen eller inte, och eventuellt associera ett annat värde med nyckelvärdet.

Detta kan användas för att implementera mängder och avbildningar $x \mapsto y$.

Observation: en vanlig lista “fungerar”, men blir alldeles för långsam då antalet element är stort.

Hashtabeller

Idén är att dela upp mängden element i mindre delmängder, och ha ett sätt att snabbt identifiera vilken delmängd ett element skall eller borde tillhöra. Därefter fortsätter sökningen i delmängden, som kan ha en enklare struktur, ex.vis. som en lista.

En hashtabell består av:

- En *hashfunktion* H som beräknar ett *hashvärde* (ett heltal) utifrån nyckelvärdet.
- En array A av storlek N vars element är delmängder av element.

En nyckel K lagras i $A[H(K) \bmod N]$.

Hashtabeller

```
struct list {
    int key;
    struct list *next;
};

int list_is_member(struct list *list, int key) { ... }
struct list *list_insert(struct list *list, int key) { ... }
struct list *list_delete(struct list *list, int key) { ... }

#define HASH_TAB_SIZE 512
struct list *hashtab[HASH_TAB_SIZE];

unsigned int hash(int key) { return key; }
```


Hashtabeller

```
int hash_is_member(int key)
{   int index = hash(key) % HASH_TAB_SIZE;
    return is_member_of_list(hashtab[index], key);
}
void hash_insert(int key)
{   int index = hash(key) % HASH_TAB_SIZE;
    hashtab[index] = list_insert(hashtab[index], key);
}
void hash_delete(int key)
{   int index = hash(key) % HASH_TAB_SIZE;
    hashtab[index] = list_delete(hashtab[index], key);
}
```


Hashtabeller

- I verkligheten låter man arrayen vara dynamisk: när antalet element blir stort i förhållande till arrayens storlek låter man arrayen växa och flyttar om elementen till sina nya positioner. Detta för att hålla storleken nere på dellistorna.
- Design av hashfunktioner är en hel vetenskap i sig. Målet är att kombinera slumpmässighet (hashvärdena ska vara utspridda oavsett om nycklarna liknar varandra eller inte) med effektivitet (hashfunktionen får inte vara för dyr att beräkna).