Introduktion till C-programmering

F1 och F2

C-programmering

Målsättning

Introducera imperativ programmering med C

Litteratur

- Kernighan, Ritchie: *The C Programming Language*, Second edition,

På nätet finns också t.ex
`http://computer.howstuffworks.com/c.htm`
`http://www.acm.uiuc.edu/webmonkeys/book/c_guide/`

Vad är C?

- Imperativt programmeringsspråk
  - Satser ("kommandon") som utförs i *sekvens*
  - Data ("variabler") som manipuleras
  - Funktioner med sidoeffekter
  - Ofta iterationer ("loopar")
- Maskinnära men maskinoberoende (nåja . . .)
  - minnesadresser, adressaritmetik
  - bitmanipulering
  - . . .
- Litet, primitivt: ingen grafik, fönster, gui, internet . . .
- Likheter med språk som Pascal och Fortran

Historik

- Utformades ursprungligen av Dennis Ritchie 1972 för implementering operativsystem och andra systemprogram
- Unix skrevs tidigt om i C
- ANSI-standard från 1989 (”ANSI-C”)
- Reviderad standard från 1999 (”C99”)
- Inspirerat språk som C++ och Java

Jämförelse med ML

- C är imperativt, ML är funktionellt
- Språken har ganska olika syntax
- C är statiskt typat: alla variabler har en typ
- C är svagt typat: vissa typomvandlingar görs automatiskt och okontrollerade brutala typomvandlingar tillåts
- C har ingen list-typ
- I C kan man arbeta med minnesadresser (pekare)
- Minneshanteringen i C måste ofta göras explicit
- C kompileras till maskinkod medan ML ofta interpreteras (på någon nivå)
- Det görs vanligen ingen runtime-kontroll när C-program exekverar (vild adressering, arraygränser, odefinierade variabelvärden . . .)

När, till vad och varför används C

- För att vara nära maskinvaran (delar av OS, drivrutiner, . . .)
- Inbyggda system (t ex tvättmaskiner, bränslestyrning, . . .)
- Tids- och/eller minneskritiska tillämpningar
- När inga andra språk finns eller går att använda (litet språk - finns för alla processorer, kan köras på ”nakna” system)
- Programmering av parallelldatorer (Typiskt med C, C++ eller Fortran). (OpenMP finns bara för dessa språk)

Dessutom är det en grund för förståelse av i första hand C++

Exempel

Ett program som skriver ut texten *Hello, world*

```
/* hello.c
   Ett första, klassiskt exempel på ett C-program
*/

#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

Exempel: iteration, variabler

```c
/* squares.c
   Program som skriver en tabell över kvadrater av talen 1, 2, ... 10
*/

#include <stdio.h>

int main() {
    int i = 1;
    while (i<=10) {
        printf("%d \t %d\n", i, i*i);
        i = i + 1;
    }
    return 0;
}
```

Hur kör man ett C-program?

1. Källkoden *kompileras* till *objektkod*
2. Objektkoden *länkas* med biblioteksfunktioner till ett exekverbart program
3. Programmet körs

Detta kan antingen göras

- i en *integrerad utvecklingsmiljö* (IDE) (ex: *Eclipse*, *Xcode*, . . .) eller
- ”manuellt” dvs med kommandon i ett kommandofönster (”command tool”, ”terminal”)
  Kommandot för kompilering (inklusive länking) är typiskt `gcc` eller `cc`

Kompilering och länkning

```
bellatrix$ ls
squares.c
bellatrix$ cat squares.c
/* squares.c
   Program som skriver en tabell över kvadrater av talen 1, 2, ... 10
*/
#include <stdio.h>

int main() {
    int i = 1;
    while ( i<=10 ) {
        printf( "%d \t %d \n", i, i*i);
        i = i + 1;
    }
    return 0;
}
bellatrix$ gcc squares.c
bellatrix$ ls
a.out* squares.c
```

Exekvering

```
bellatrix$ a.out
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
bellatrix$
```

Anm: Du kan behöva skriva `./a.out` (beror på hur din s. k. ”path” är satt).

Kompilatorn och andra hjälpmedel

Några optioner till `gcc`:

`-Wall`	Varna alltid. Använd alltid!
`-g`	Om man vill använda en debugger (även `-ggdb`)
`-std=c99`	För att använda C99-standard
`-c`	Endast kompilering. Ger objektfiler (typ `.o`)
`-o` *filnamn*	Namn på outputfil
`-lm`	Till länkaren: ta med matematik-funktioner (vissa OS)

Andra verktyg:

`gdb`	debugger
`lint`	”statisk” avlusning
`make`	hålla reda på vad och hur filer skall kompileras
`man`-kommandot	dokumentation av standardfunktioner

Exempel: flera funktioner

```
/* factorial.c
   Program som tabulerar fakultetsfunktionen
*/

#include <stdio.h>

int factorial(int n) {
     int result = 1;
     while (n > 0) {
          result = result * n;
          n = n - 1;
     }
     return result;
}

int main() {
    int i = 0;
    while (i<=15) {
        printf("%2d %12d \n", i, factorial(i));
        i = i + 1;
    }
    return 0;
}
```

Exempel: rekursion, villkorssats

```
/* factorialRec.c
   Program som tabulerar fakultetsfunktionen
   Rekursiv version.
*/

#include <stdio.h>

int factorial(int n) {
    if (n<=0) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
    /* return (n<=0) ? 1 : n*factorial(n-1); */
}

int main() {
    int i = 0;
    while (i<=15) {
        printf("%2d %12d \n", i, factorial(i));
        i = i + 1;
    }
    return 0;
}
```

Exempel: Läsa och skriva tecken

```c
/* cat1.c
   Kopierar standard input till standard output */

#include <stdio.h>

int main() {
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
    return 0;
}
```

Exempel: Läsa och skriva tecken (version 2)

```c
/* cat2.c
   Kopierar input till output. Version 2 */

#include <stdio.h>

int main() {
    int c;
    while ((c = getchar()) != EOF) putchar(c);
    return 0;
}
```

Exempel: Ett program som räknar tecken och rader

```
#include <stdio.h>

int main() {
   int c;
   int nChars = 0, nLines = 0;

   c = getchar()
   while (c != EOF) {
      ++nChars;
      if (c == '\n') {
         ++nLines;
      }
      c = getchar();
   }
   printf("Tecken: %d \nRader: %d \n", nChars, nLines);
   return 0;
}
```

Sammanfattning

- Ett C-program består av en eller flera *funktioner* som lagras på en (eller flera) *filer*
- Kommentarer omges med `/*` och `*/`. Från och med C99 kan `//` användas för kommentar till radslut (som C++ och Java)
- `#include <stdio.h>` för att kunna använda biblioteksfunktioner för in- och utmatning (`printf`, `getchar`, `putchar`, `EOF`)
- En C-funktion har en *returtyp*, ett *namn*, en *parameterlista* och en *funktionskropp*
- Funktionskroppen kan innehålla variabeldeklarationer och satser.
- Semikolon (`;`) används för att avsluta deklarationer och satser.
- Filerna kan innehålla deklarationer utanför funktionerna.

Sammanfattning forts

- Varje fil måste *kompileras* innan programmet kan exekveras
- Exekveringen börjar i funktionen med namnet `main`
- Variabler måste *deklareras* med *typ* och *namn*
- Variabler tilldelas värden (av rätt typ) med *tilldelningssatser*
- Satser i en funktion utförs i tur och ordning
- Satser för *selektion*: **`if`** (samt **`switch`**)
- Satser *repetition*: **`while`** (samt **`for`** och **`do-while`**).

if-satsen

Två varianter:

```
if ( u ) { s }
```

och

```
if ( u ) { s1 } else { s2 }
```

där

- *u* är ett uttryck av *godtycklig* typ och
- s, s_1, s_2 är vardera *en* sats

Uttrycket *u* tolkas som *false* om det är 0, 0.0, eller `NULL` (pekare) annars som *true*.

Inom klammerparenteserna { och } kan man ha ett godtyckligt antal satser. Om man bara har en sats kan klammerparenteserna utelämnas.

for-satsen

Iterationssats med syntaxen:

for (u_1 ; u_2 ; u_3) { s }

Funktionen kan uttryckas med **while**-satsen:

u_1 ;
while (u_2) {
    s ;
    u_3 ;
}

Vanligt "ideom" för att upprepa något ett visst antal gånger. T ex:

```
for (i = 1; i<=10; i++) {
    s;
}
```

Operatorer (ej fullständig)

binära aritmetiska	:	`+ - * /`
unära aritmetiska	:	`+ - ++ --`
tilldelning	:	`= += -= *= /=`
relation	:	`== < > <= >= !=`
logiska	:	`&& || !`

Datatyper

- **`void`**
- skalära typer
  - aritmetiska typer
    - heltalstyper **`char`**, **`short int`**, **`long int`**, **`long long int`** (ev. **`unsigned`**)
    - flyttalstyper **`float`**, **`double`**, **`long double`**
  - pekare
- arrayer (egentligen ingen egen typ utan hjälpsyntax för pekare)
- sammansatta typer **`struct`**

Obs: ingen typ för logiska värden (`true`, `false`)!

Heltalstyperna

Vanliga storlekar och talområden (beroende av plattform och kompilator):

`long int`	4	-2^{31} till $2^{31}-1$
`short int`	2	-2^{15} till $2^{15}-1$
`unsigned long int`	4	0 till $2^{32}-1$
`unsigned short int`	2	0 till $2^{16}-1$
`signed char`	1	-128 till 127
`unsigned char`	1	0 till 255

Typen `int` är normalt liktydigt med `long` eller `short int` (implementationsberoende).

Typen `char` är vanligen liktydigt med `unsigned char`.

Konstanter

Heltalskonstanter

decimal form `3, 8, -255`

oktal form `003, 010, -0377`

hexadecimal form `0x03, 0x08, -0xFF`

Flyttalskonstanter

Skrivs med decimalpunkt och/eller exponent

```
-1.5, .26, 100., 0.57721566
1e10, 0.5e2, 1e-10
```

Teckentypen char

Typen **char** är en *heltalstyp* och teckenkonstanter är ett sätt att skriva små heltal.

Man kan således *räkna* direkt med tecken:

```
char toUpper(char c) {
  /* Om c är en gemen så returneras motsvarande VERSAL */
  if (c>='a' && c <='z')
     return c + 'A' - 'a';
  else
     return c;
}
```

Biblioteket `ctype.h`

Detta bibliotek innehåller representationsoberoende funktioner för att klassificera tecken:

```
#include <ctype.h>

int isalpha(int c);
int isdigit(int c);
int isalnum(int c);
int isspace(int c);
int isupper(int c);
int islower(int c);
int isprint(int c);
int iscntrl(int c);
int tolower(int c);
int toupper(int c);
```

Exempel: Skriv ut en ASCII-tabell

```
/* Program som skriver en tabell över teckenkoder */

int main() {
  char c;
  for (c = ' '; c<127; c++)
     printf( "%d \t %c \n", c, c );
  return 0;
}
```

(I ASCII-kod är blanktecknet det första tryckbara och ~ (126) det sista tryckbara)

Alla variabler måste typdeklareras

Exempel:

```
int i, j, k;
float x, y;
char c;
short int p, q;
unsigned short int r;
unsigned char ch;

int start = 0, stop = 10;
char c=getchar();
```

Deklarationerna måste ligga först i ett "block" som definieras av ett { }-par (C99-standarden tillåter deklarationer även bland de utförande satserna).

Typkonverteringar

- Vid operander med olika typer sker automatiska konverteringar från ”trängre” typer till ”vidare” type
- Tilldelningar från ”vidare” typer till ”trängre” typer kan ge varningar men är tillåtna
- Explicita typkonverteringar med *casts*:

  (*typ*) *uttryck*

  Exempel:

```
y = power((double) i, n);
```

Formaterad utmatning

`printf(` *formatsträng* `,` *värde* `,` *värde* `, ... )`

Några formatspecifikationer:

`%d`	**`int`**	
`%ld`	**`long int`**	
`%u`	**`unsigned int`**	
`%lu`	**`unsigned long int`**	
`%o`	**`int`**	oktal form
`%x`	**`int`**	hexadecimal form
`%c`	**`char`**	
`%f`	**`double`**	decimalform
`%e`	**`double`**	exponentform
`%g`	**`double`**	decimal- eller exponentform
`%s`	teckensträng	

Formatterad utmatning forts

Specifikationerna kan förses med attribut för att specificera, fältbredd, antal decimaler, justering mm:

```
/* formatExample.c - Demonstrerar användning av formatkoder */
#include <stdio.h>
int main() {
  float x;
  for (x=0; x<=10.; x++) {
    printf(" %2.0f %8.4f %12.4e\n", x, sin(x), exp(x));
  }
}
/* Output:
  0 0.0000 1.0000e+00
  1 0.8415 2.7183e+00
  2 0.9093 7.3891e+00
  3 0.1411 2.0086e+01
  4 -0.7568 5.4598e+01
  5 -0.9589 1.4841e+02
  6 -0.2794 4.0343e+02
  7 0.6570 1.0966e+03
  8 0.9894 2.9810e+03
  9 0.4121 8.1031e+03
 10 -0.5440 2.2026e+04
*/
```

Inläsning av tal

Funktionen `scanf` för formaterad inläsning.

Exempel:

```
int main() {

   float x;
   int n;

   printf("Ge x och n: ");
   scanf("%f %d", &x, &n );
   printf("%f upphöjt till %d är %f", x, n, power(x,n));
   return 0;
}
```

Några fakta om funktioner

- Alla funktioner är på samma nivå dvs funktionsdefinitioner kan inte innehålla lokala funktioner
- En funktion har noll eller flera parametrar av godtycklig typ
- Vid anrop måste parametrarna överensstämma i ordning, antal och typ Viss automatisk typkonvertering kan dock ske.
- Parameteröverföringen sker enligt ”call by value”
- En funktion kan vara av typen `void` som anger att den inte returnerar något värde

- Funktioner kan returnera skalära värden och poster (`struct`) men ej arrayer
- Funktioner returnerar ett värde av typen `int` om inget annat sägs
- Lokala variabler dör vid return (om ej `static`)
- För att en funktion skall kunna anropas måste den vara ”känd”. Känd kan den bli genom en *definition* eller en *deklaration*:
  typ namn(*parameterlista*);
  typ namn(`void`);

Exempel: primtalskontroll

Uppgift: *Skriv ett program som läser en sekvens av positiva heltal och för varje tal avgör om det är ett primtal eller ej.*

```
int isPrime(int n) {
  int answer = n-1; // Hanterar n == 1
  int i;

  for (i = 2; i<n; i++) {
    if (n % i == 0) {
      answer = 0;
    }
  }
  return answer;
}
```

- Ny operator: %

Bättre:

```
int isPrime(int n) {
  /* Precondition: n is an integer >= 0
   * Returns: 1 if n is a prime number else 0
   */

  int answer = n-1; // Hanterar n == 1
  int i;

  for (i = 2; i <= sqrt(n) && answer; i++) {
    if (n % i == 0) {
      answer = 0;
    }
  }
  return answer;
}
```

- Ny operator: `&&`
- Funktionen `sqrt`
- Minns att 0, 0.0, och NULL tolkas som *false*, allt annat som *true*

Programmet:

```
/* checkPrimes.c
   Läser en sekvens av tal och avgör vilka som är primtal.
   Avbryter när talet 0 läses.
*/

#include <stdio.h>
#include <math.h>

int isPrime(int n) { ... } // Som ovan

int main() {
  int number=1; // Tal att kolla
  while (number!=0) {
    printf("Tal att kolla: ");
    scanf("%d", &number);
    if (number!=0) {
      if (isPrime(number)) {
        printf("%d är ett primtal\n", number);
      } else {
        printf("%d är ej ett primtal\n", number);
      }
    }
  }
  return 0;
}
```

Körning:

```
kursa$ gcc -o checkPrimes checkPrimes.c
Undefined symbol sqrt first referenced in file /var/tmp/ccZwIZDu.o
ld: fatal: Symbol referencing errors. No output written to checkPrimes
collect2: ld returned 1 exit status
kursa$ gcc -o checkPrimes -lm checkPrimes.c
kursa$ checkPrimes
Tal att kolla: 2
2 är ett primtal
Tal att kolla: 4
4 är ej ett primtal
Tal att kolla: 12
12 är ej ett primtal
Tal att kolla: 13
13 är ett primtal
Tal att kolla: 4731
4731 är ej ett primtal
Tal att kolla: -4
-4 är ett primtal
Tal att kolla: 0
kursa$
```

Ytterligare en villkorssats: `switch`

```
int daysInMonth(int month) {
    int days;
    switch(month) {
        case 11: // november
        case 4: // april
        case 6: // juni
        case 9: // och september
            days = 30; // 30 dar har
            break;
        case 2;
            days = 28; // februari 28 allen
            break;
        default:
            days = 31; // alla övriga 31
   }
   return days;
}
```

Glöm inte **`break`** (när den skall vara med)!

En iterationssats till: **do — while**

```
int readint() {
  /* Läser och returnerar ett heltal från standardinput
     Upprepar läsning tills ett legalt heltal kommer
  */
    int tal;
    int err;
    do {
        err = scanf("%d", &tal);
        if (err != 1) {
             printf("Not an integer! Try again: ");
             while (getchar() != '\n') // Skippa resten av raden
                 ;
        }
    } while (err!=1);
    return tal;
}
```

Anm 1: Skrivs snyggare med en **while**-sats!

Anm 2: Ej säker! Varför inte?

forts...

Testprogram som läser och summerar tal tills 0 lästs:

```
int main() {
    int sum = 0;
    int tal;
    printf("Tal att summera (avbryt med 0): ");
    do {
        sum +=tal;
        tal = readint();
    } while (tal!=0);
    printf("Summan är %d\n", sum);
}
```

Anm: Också lite udda – bygger på att det är OK att lägga till sista.

Övningar till F1 och F2

1. Skriv ett program som läser tecken från standard input och räknar antalet meningar. Meningar avslutas med punkt, utropstecken eller frågetecken.
2. Skriv ett program som kopierar standard input till standard output. Vid kopieringen skall all text omgiven av < och > inklusive dessa tecken utelämnas.
3. Skriv ett program som läser tecken från standard input och räknar antalet ord. Med *ord* menas en obruten följd av bokstäver.
4. Modifiera programmet ovan så att det också skriver ut hur många bokstäver det är i det längsta ordet.
5. Skriv en C-funktion för att beräkna den harmoniska summan

$$1 + 1/2 + 1/3 + 1/4 ... + 1/n$$

   Vilka parametrar och och vilken returtyp bör funktionen ha?
6. Skriv ett C-program som tabulerar ovanstående summa för $n = 1, 2, \ldots, 10$

Övningar till F1 och F2 (forts)

7. Skriv ett C-program räknar ut skriver hur många tal som behövs för att summan

$$1 + 1/2 + 1/3 + 1/4... + 1/n$$

   skall överstiga 10.

8. Skriv en funktion `int isEqual(char c1, char c2)` som returnerar 1 om tecknen `c1` och `c2` är lika annars 0. Om tecknen är bokstäver skall de betraktas som lika oavsett om de är versaler eller gemena (dvs 'a' och 'A' skall betraktas som lika).
9. Skriv ett program som läser en rad från standard input och skriver ut raden översatt till rövarspråket. Översättningen tillgår så att om `x` är en konsonant så ersätts den med `xox` medan vokaler lämnas oförändrade. Exempel: Texten "Don't panic" blir "Dodonon'tot popanonicoc".

Övningar till F1 och F2 (forts)

10. Skriv en funktion som läser en rad av godtycklig längd från standard input och skriver ut raden baklänges så att det sist inlästa tecknet skrivs först. Funktionen behöver inte använda någon array eller lista.
11. Följden

    $$0, 1, 1, 2, 3, 5, 8, 13 \ldots$$

    kallas Fibonaccital.

    a) Skriv en funktions `int fib(int n)` som beräknar och skriver ut det n:te Fibonaccitalet
    b) Skriv ett program som läser in ett tal n samt beräknar och skriver de n första Fibonaccitalen.
    c) Skriv ett program som läser in ett tal m samt beräknar hur många av Fibonaccitalen som är mindre än eller lika med m.
12. Skriv en rekursiv funktion `void printb(int x, int b)` som skriver ut `x` i basen `b`. Förutsätt för enkelhetens skull, att $b <= 10$.

Arrayer

F4

Arrayer

En array är en samling bestående flera *element*.

En arrayvariabel deklareras med

typ arraynamn[*storlek*]

- Elementen numreras 0, 1, 2 ... *storlek*-1
- Storleken måste vara en *konstant* (behövs ej i C99-standarden)
- Alla element är av samma typ
- Indexoperatorn [] används för att adressera enskilda element
- Arrayer allokeras i sammanhängande (konsekutivt) minne

Exempel:

```
/* Läs 100 positiva flyttal och skriv ut dem normerade */

int main() {
  int i;
  float x[100], max=0;

  for (i=0; i<=99; i++) {
      scanf("%f", &x[i]);
      if (x[i] > max)
         max = x[i];
  }

  for (i=0; i<=99; i++)
      printf("%f", x[i]/max);
  return 0;
}
```

Arrayer som inparametrar

Exempel: Beräkna skalärprodukten $\sum_{i=1}^{n} u_i v_i$

```
#include <stdio.h>
#include <math.h>

double scalprod(double u[], double v[], int n) {
  double result = 0;
  int i;
  for (i= 0; i<n; i++)
    result += u[i]*v[i];
  return result;
}

int main() {
  double a[] = {3.0, 4.0};
  printf("Normen blir: %6.2f\n", sqrt(scalprod(a, a, 2)));
}
```

Arrayer som utparametrar

```
void sort(int v[], int n) { // enkel urvalssortering
  int i;
  for (i=0; i<n-1; i++) {
    int min = i;
    int j;
    int temp = v[i];
    for (j=i+1; j<n; j++) {
      if (v[j]<v[min])
        min = j;
    }
    v[i] = v[min];
    v[min] = temp;
  }
}

int main() {
  int i, a[] = {3, 1, 2, 5, 8, 4};
  sort(a,6);
  for(i=0; i<6; i++)
    printf("%3d", a[i]);
  printf("\n");
}
```

Teckensträngar

- Ingen `String`-typ! Lagras i arrayer av `char`
- Sist lagras **`NULL`**-tecknet, `\0`, (konvention)
- Kan initieras i vid deklaration
- Kan *inte* flyttas med tilldelning eller jämföras med relationsoperatorerna
- `string.h` innehåller strängfunktioner
- Formatkoden `%s` till `printf` och `scanf`

Exempel på strängfunktioner

```
int strlen(char s[]) {
    int n=0;
    while (s[n] != '\0') {
       n++;
    }
    return n;
}
```

Alternativ kod

```
int strlen(char s[]) {
    int n=0;
    while (s[n++])
      ;
    return n-1;
}
```

Bra eller dåligt?

Biblioteket `string.h`

Ett bibliotek med funktioner för hantering av strängar. Några av dessa är:

```
int strlen (char s[]);
void strcpy (char s1[], char s2[]);
void strncpy(char s1[], char s2[], int n);
void strcat (char s1[], char s2[]);
void strncat(char s1[], char s2[], int n);
int strcmp (char s1[], char s2[]);
```

(Lite förenkling av parametertyper och returvärden...)

Exempel: funktionen `readWord`

Vag specifikation:

Skriv en funktion `readWord` som lokaliserar och läser in nästa *ord* och, på något sätt, förmedlar det till anroparen.

Med *ord* avses en obruten följd av (engelska) bokstäver.

Frågor:

1. Hur skall ordet returneras?
2. Vad skall funktionen göra om den inte hittar något ord?

Övningar

1. Skriv en funktion `double enorm(double x[], int n)` som beräknar och returnerar den euklidiska vektornormen för vektorn x med n element. Vektornormen av vektorn $x_1, x_2, \ldots x_n$ definieras som $\sqrt{\sum_{i=1}^{n} {x_i}^2}$
2. Skriv funktionerna `int isVowel(char c)` och `int isConsonant(char c)` som returnerar 1 om argumetet `c` är en *vokal* respektive en *konsonant* annars 0. Skriv också ett huvuprogram som läser tecken från standard input och räknar antalet vokaler respektive konsonanter.
3. Implementera funktionen `readWord` enligt diskussionen ovan. Skriv också ett testprogram till funktionen.
4. Skriv ett program som läser standardinput och skriver ut det längsta ordet.
5. Skriv ett program som läser standard input och räknar (engelska) bokstävernas förekomstfrekvens. Ledning: Använd en heltalsarray med 26 platser. Använd plats 0 för att räkna *a*, plats 1 för *b* etc.

6. Standardfunktionen `strcpy` kontrollerar inte att utrymmet för mottagande område är tillräckligt stort. Skriv en egen funktion `stringCopy` som gör detta. Vilka parametrar behövs? Vad skall funktionen göra om det mottagande området är för litet?

7. En array innehåller heltal sorterade i storleksordning. Arrayen har en viss deklarerad storlek men det är inte säkert att hela utnyttjas. Man måste således också hålla reda på hur många tal som verkligen är lagrade. Skriv en funktion som lägger in ett nytt tal i arrayen så att sorteringsordningen bibehålls. Vilka parametrar är lämpliga? Hur skall man hålla reda på storleken respektive antal lagrade element?

   Skriv också en funktion för sökning efter ett givet värde.

8. Skriv en funktion `int match(char m[], char s[])` som returnerar index för första förekomst av strängen `m` i strängen `s`. Om strängen inte finns skall `-1` returneras.

9. Skriv en funktion `int nMatch(char m[], char s[])` som räknar och returnerar *antalet förekomster* av strängen `m` i strängen `s`.

Pekare

F5

Pekare

- Varje variabel har en *adress* som avser den plats i datorns minne där dess värde är lagrat
- En variabels adress ges av den unära adressoperatorn `&`
- Adresser hanteras kan lagras i variabler ("pekarvariabel")
- Den unära "avrefereringsoperatorn" `*` används dels vid deklaration av pekarvariabler och dels för att "komma åt" det som pekaren refererar
- Det går att utföra aritmetik med pekare
- Pekarkonstant `NULL` (som alltid är 0)

Pekare som parametrar

```
void swap(int *x, int *y) {
   int temp = *x;
   *x = *y;
   *y = temp;
}
```

Anrop:

```
   int a, b;
   a = 1;
   b = 2;
   printf("a: %d b: %d", a, b);

   swap(&a, &b);

   printf("a: %d b: %d", a, b);
```

Exempel:

```
int x = 1, y = 2;
int a[] = {4, 3, 2, 1};
int *ip, *jp;

ip = &x; // ip pekar till x
y = *ip; // som y = x
*ip = 3; // som x = 3
jp = ip; // jp pekar också till x
*jp = *jp + 5; // som x = x + 5
(*ip)++; // som x++
jp = NULL; // nullreferens. Samma som jp = 0
*jp = 2; // illegalt!

ip = &a[0]; // pekar till första elementet i a
jp = ip + 1; // pekar till andra elementet i a
*jp = 0; // som a[1] = 0
*(ip+3) = *(ip+1) + 2; // som a[3] = a[1] + 2
ip = a; // som ip = &a[0]
```

Arrayer som parametrar - igen

```
int strlen(char *s) {
    int n=0;
    while (*s != '\0') {
      s++;
      n++;
    }
    return n;
}
```

Alternativ:

```
int strlen(char *s) {
    char *p = s;
    while (*p != '\0')
       p++;
    return p - s;
}
```

OBS 1: Anropas på samma sätt som tidigare!

OBS 2: Talar INTE om hur stor arrayen är!

Fyra versioner av strängkopiering

```
void strcpy(char *s, char *t) {
    int i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}

void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++;
        t++;
   }
}

void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0')
        ;
}

void strcpy(char *s, char *t) {
     while (*s++ = *t++)
       ;
}
```

Arrayer av pekare

```
int main(int argc, char *argv[]) {

    int i;

    for (i = 0; i < argc; i++)
      printf("%s%s",
             argv[i],
             (i < argc-1) ? "_" : "");
    printf("\n");
    return 0;
}
```

(Notera villkorsuttrycket u_1 ? u_2 : u_3)

Om programmet sparas på en fil med namnet *eka* så kan en testkörning se ut som

```
$ ./eka My God, it is full of stars!
eka_My_God,_it_is_full_of_stars!
$
```

Pekare som funktionsvärde

```
char *findBlank(char *s) {
  /* Letar första blanktecken i en sträng
     Pre: s - första tecken i en '\0'-terminerad sträng
     Returns: adressen till första blanka tecken eller NULL om
              strängen inte innehåller blanka.
  */
  while (*s != ' ' && *s != '\0') s++;
  if (*s == '\0' ) return NULL;
  else return s;
}
```

Vad får `p` för värde efter nedanstående kod?

```
char a[] = {'a', 'b', 'c'};
char *p = findBlank(a);
```

Varningar

Pekarfel är både vanliga och ofta svåra att upptäckta eller spåra!

Några vanliga sätt att hitta pekarfel:

- Försök att följa NULL-pekaren
- Är någon pekares värde någonsin odefinerat?
- Använder man någonsin en pekare vars utpekade utrymme som är återlämnat?
- Aliasproblematiken: finns flera pekare som (oavsiktligt) refererar samma objekt? Används de konsekvent?

Pekarkonstanter som är läsbara i en debugger, t.ex. `0xDEADBEEF` m.fl.

Definiera egna typer: `typedef`

Exempel

```
typedef float Real;
typedef unsigned char Digit;
typedef char String[100];
typedef int * IntPointer;

Real x;
String name, adress;
IntPointer ip, jp;
```

- **`typedef`** definierar en egen datatyp
- Syntaxen är exakt som en variabeldeklaration
- Ger möjlighet att lätt byta datatyp i ett helt program/modul
- Använd `typedef`!

Sammansatta datatyper: `struct`

Exempel

```
struct Point {
  double x, y;
}; // Observera semikolonet!
typedef struct Point Point;

typedef struct Triangle { // Ovanstående i ett enda steg
    Point a, b, c;
} Triangle;

Point p1 = {0, 0};
Point p2;
Triangle t;
p2.x = 1.;
p2.y = 0.;
t.a = p1;
t.b = p2;
t.c.x = p1.x + 2;
t.c.y = t.a.x + 1;
```

Dynamiska variabler: `malloc` och `free`

Exempel

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

int i, n, *buffer;

printf("Hur många tal vill du kunna behandla:");
scanf("%d", &n);

buffer = (int *) malloc(sizeof(int)*n);

for (i = 0; i < n; i++) {
    buffer[i] = i*i; ...
}
free(buffer);
```

Observera

- Operatorn `sizeof` som returnerar storleken i bytes av en godtycklig typ
- `malloc` returnerar **`NULL`** om det inte fanns tillräckligt med minne
- Typkastningen `(int *)` — `malloc` returnerar typen **`void`** `*`
- Minne återlämnas med `free` — ingen automatisk minneshantering
- Arraynotationen
- Alla möjligheter att göra fel ...

Övningar

1. Skriv en funktion som löser andragradsekvationen

$$ax^2 + bx + c = 0$$

Lösningen ges av

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Koefficienterna a, b och c skall skickas till funktionen. Skriv tre olika varianter för att returnera lösningen:

   a) via två parametrar,
   b) som en post (funktionsvärde) och
   c) som en pekare till en post.

Skriv också ett huvudprogram som testar funktionen med en följd av inlästa koefficienter.

2. Följande post är avsedd att lagra ett varierande antal heltal

```
struct container {
  int *buffer; /* Pekare till lagringsarea */
  int size; /* Allokerad storlek */
  int number; /* Aktuellt antal */
};

typedef struct container container;
```

   Skriv en funktion `int store(int data, container *c)` som lagrar `data` på första lediga plats och returnerar dess index. Om behållaren är full (dvs om `size==number`) så skall en ny, dubbelt så stor buffer anskaffas och innehållet i den gamla flyttas över till den nya.

   Skriv också en funktion som returnerar talet som är lagrat på en viss plats. Hur bör parametrarna till denna funktion se ut?

3. Samma problem som ovan *men* med lagring av textsträngar i stället för heltal.

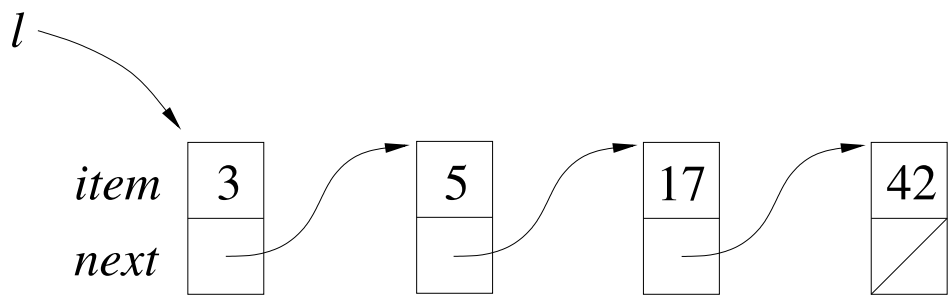
Länkade strukturer

F7

Länkade (rekursivt definierade) strukturer

Byggs med hjälp av poster (**`struct`**) som innehåller pekare till poster av samma typ.

Exempel: Länkad lista med heltal




```
typedef struct listElem {
    int item;
    struct listElem *next;
} listElem, *Link;
```

Heltalslista

```
Link cons(int item, Link next) {
  /* Skapar en listnod
   * Parametrar:
   * it - godtyckligt heltal
   * nx - pekare till en listnod eller NULL
   * Returnerar
   * Pekare till nya noden eller NULL om allokeringen misslyckades
   */
    Link link = (Link) malloc(sizeof(listElem) );
    if (link==NULL) {
        return NULL;
    } else {
        link->item = item;
        link->next = next;
        return link;
    }
}
```

Heltalslista forts

```
Link addLast(int item, Link link) {
  /*
   * Lägger till en ny nod sist en lista
   * Parametrar
   * item - godtyckligt heltal
   * link - pekare till en listnod eller NULL
   * Returnerar
   * Pekare till första noden i den modifierade listan
   *
   */
    if (link==NULL) {
        return cons(item, NULL);
    } else {
        link->next = addLast(item, link->next);
    } return link;
  }
}
```

Heltalslista forts

```
void print(Link link) {
  /* Skriver listan på standard output
   * Parametrar
   * link - pekare till första noden i listan eller NULL
   */
    printf("[");
    while (link) {
        printf("%d", link->item);
        if (link->next) printf(", ");
        link = link->next;
    }
    printf("]");
}
```

Heltalslista forts

```
int main() {
    link l = NULL;
    l = cons(5, l);
    print(l); putchar('\n');
    l = cons(3, l);
    print(l); putchar('\n');
    l = addLast(17, l);
    l = addLast(42, l);
    print(l); putchar('\n');
    return 0;
}
```

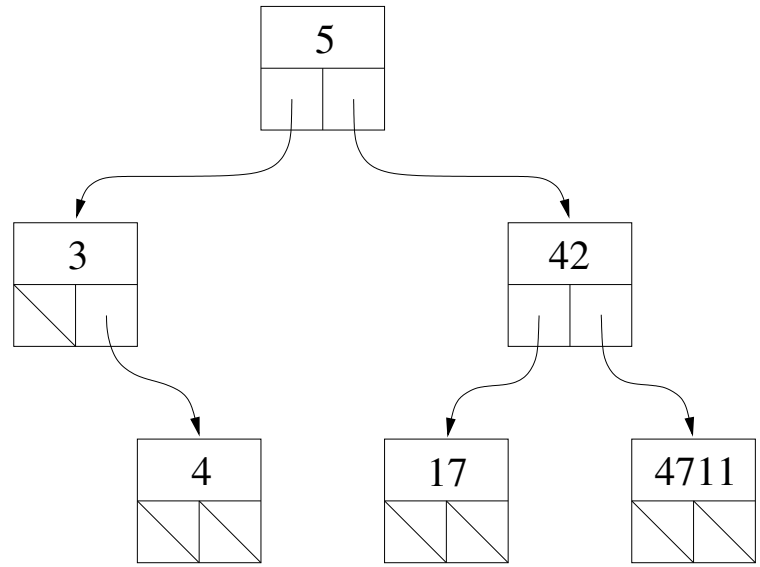
Körning:

```
trilian$ ./a.out
[5]
[3, 5]
[3, 5, 17, 42]
trilian$
```

Övningar på länkade listor

1. Skriv en funktion `int length(Link link)` som returnerar längden av listan vars första nod pekas ut av `link`. Skriv både en iterativ och en rekursiv variant!
2. Skriv en funktion `int search(int item, Link link)` som returnerar 1 om `item` finns i listan som börjar vid `link`, annars 0. Skriv både en iterativ och en rekursiv variant!
3. Skriv en funktion `insert` som, givet en pekare till första noden i en lista där elementen är sorterade i storleksordning, lägger in ett nytt element i listan så att sorteringen bibehålls. Gör först en variant som gör en ny lista och sedan en variant som modifierar den befintliga listan.
4. Skriv en funktion `void addLast(int itm, link *lastPointer)` som lägger in ett nytt element med `item` sist i listan som pekas ut av `*lastPointer`.
5. Skriv en funktion som tar bort alla element med ett givet innehåll.
6. Skriv en funktion `Link merge(Link l1, Link l2)` som, givet att `l2` och `l2` pekar till första noden i var sin sorterade lista, skapar och returnerar en ny sorterad lista bestående av elementen från både `l2` och `l2`.

Länkade strukturer: Binära träd




```
typedef struct node {
    int key;
    struct node *left;
    struct node *right;
} node, *Link;
```

```
int cons(int key) {
   Link result = (Link) malloc(sizeof(node));
   result->key = key;
   result->left = NULL;
   result->right = NULL;
   return result;
}
```

Exempel: Binärt sökträd

```
link add(int key, link t) {
   if (t==NULL) return cons(key);
   else if (key == t->key) return t;
   else if (key < t->key) t->left = add(key, t->left);
   else t->right = add(key, t->right);
   return t;
}

int main() {
   link t=NULL;
   t = add(3, t); t = add( 1, t); t = add(17, t); t = add( 5, t); t = add(42, t);
   print(t);
   printf("\n");
   return 0;
}
```

Exempel: Binärt sökträd (forts)

```
void print(link t) {
   if (t) {
      print(t->left);
      printf("%2d ", t->key);
      print(t->right);
   }
}
```

```
Körning:
marvin$ ./bst
1 3 5 17 42
marvin$
```

Övningar på binära träd

1. Deklarera en post för att representera noder i ett binärt träd och skriv en konstruktor (cons-funktion) för sådana poster.
2. Skriv funktioner som beräknar antalet noder respektive höjden i ett sådant träd. Funktionerna skall få en pekare till rotnoden som parameter.
3. Skriv en funktion avgör om två träd är isomorfa (har samma struktur).
4. Skriv en funktion som skriver ut ett träd i preorder med en nod per rad och en indentering som visar strukturen (dvs proportionell mot nodens djup)

Mindre funktionella kodexempel

Nu följer ett antal kodexempel med syfte att illustrera typiska idiom i imperativ programmering.

”Funktionella” exempel innehåller typiskt rekursion och oföränderligt data.

Imperativ programmering innehåller typiskt loopar och uppdatering av föränderligt data.

Imperativ append

Insättning i slutet av länkad lista. Notera funktionens avsaknad av returvärde och dess interativa definition. Funktionen är här uppdelad i två och följer signaturerna i inlupp 1.

```
static inline void __append(List *list, int element) {
  List *cursor = list;
  while (*cursor) {
    cursor = &((*cursor)->next);
  }
  *cursor = mkListNode(element);
}

List append(List list, int element) {
  __append(&list, element);
  return list;
}
```

En rimligare implementation skulle ha en pekare till sista elementet för $O(1)$-insertering.

Rekursiv insertering i binärträd

Funktionell kod, men med betydligt färre tilldelningar är tidigare motsvarande exempel tack vare "dubbelpekare".

```
static void __insert(Tree *t, int element) {
  if (*t) {
    Tree *branch = (element < (*t)->element) ? &((*t)->left) : &((*t)->right);
    __insert(branch, element);
  } else {
    *t = mkTNode(element);
  }
}

Tree insert(Tree t, int element) {
  __insert(&t, element);
  return t;
}
```

Koden är relativt elegant. Variabeln `branch` gör att endast ett rekursivt anrop görs i hela funktionen. Det blir då enkelt att se hur den kan skrivas om på imperativt format. (Nästa bild, fundera först själv!)

Imperativ binärträdsinsertering

```
static inline void __insert(Tree *t, int element) {
  while (*t) {
    t = (element < (*t)->element) ? &((*t)->left) : &((*t)->right);
  }
  *t = mkTreeNode(element);
}

Tree insert(Tree t, int element) {
  __insert(&t, element);
  return t;
}
```

Rekursiv traversering av specifik sökväg

Denna funktion förenklar skrivandet av testkod för binärträd utan att förlita sig på detaljer om dess implementation.

```
int getElementForPath(Tree tree, char* path) {
  if (tree == NULL) return -1;
  switch (*path++) { // **
  case 'L':
  case 'l':
    return getElementForPath(tree->left, path);
  case 'R':
  case 'r':
    return getElementForPath(tree->right, path);
  case '\0':
    return tree->element;
  default:
    // handle errors
    break;
  }
  return -2;
}
```

Rad ** är litet för "fancy". Varför? Bättre sätt att skriva?

Imperativ motsvarighet

```
int getElementForPath(Tree t, char *path) {
  while (t && *path) {
    switch (*path++) {
    case 'L':
    case 'l':
      t = t->left;
      continue;
    case 'R':
    case 'r':
      t = t->right;
      continue;
    default:
      return -2;
    }
  }
  return t ? t->element : -1;
}
```

Enklare eller svårare att läsa?

Rekursiv sökning

Denna funktion söker upp ett element i ett binärträd och returnerar en C-sträng med sökvägen till elementet från roten.

```
static inline int __search(Tree t, int element, char *path) {
  if (t == NULL) return 1;
  if (element < t->element) {
    *path++ = 'L';
    return __search(t->left, element, path);
  }
  if (element > t->element) {
    *path++ = 'R';
    return __search(t->right, element, path);
  }
  *path = '\0';
  return 0;
}

char *search(Tree t, int element) {
  char *path = (char*) malloc(depth(t));
  int error = __search(t, element, path);
  return (error) ? NULL : path;
}
```

Övning: skriv om funktionen på imperativt vis. (Exempel på nästa sida.)

Imperativ variant av föregående funktion

```
static inline int __search(Tree t, int element, char* path) {
  while (t) {
    if (element == t->element) {
      *path = '\0';
      return 0;
    } else if (element < t->element) {
      *path++ = 'L';
      t = t->left;
    } else {
      *path++ = 'R';
      t = t->right;
    }
  }
  return 1;
}

char *search(Tree t, int element) {
  char *path = (char*) malloc(depth(t));
  int error = __search(t, element, path);
  return (error) ? NULL : path;
}
```

Manuell Minneshantering

F8

Minnesmodellen

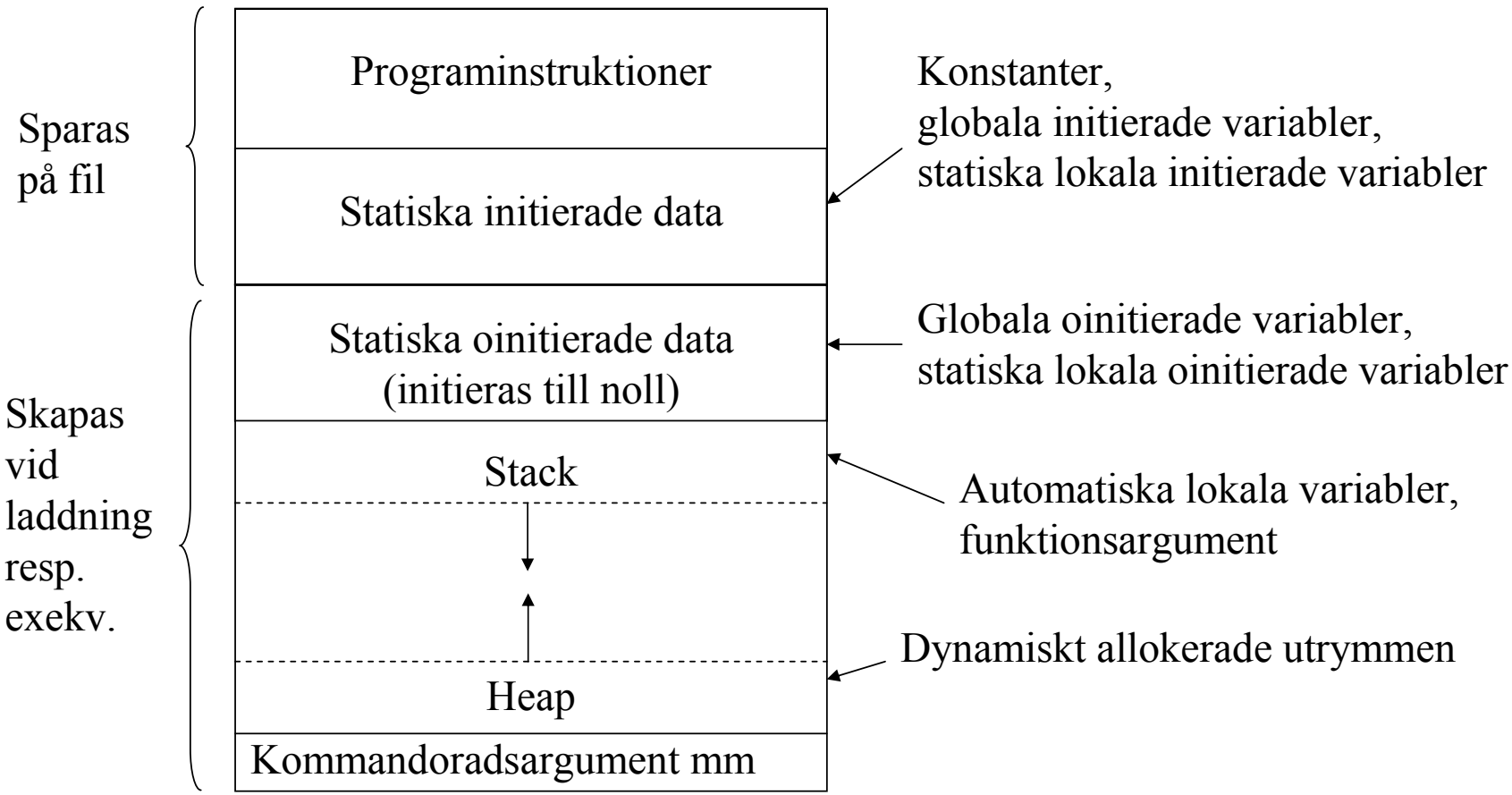


Bild: Jozef Swiatycki

Fibonacci: heltal på stacken

```
unsigned int fib3(int n) {
  if (n==0) {
    return 0;
  } else if (n==1) {
    return 1;
  } else {
    int result = fib3(n-1) + fib3(n-2);
    return result;
  }
}
```

Fibonacci: heltal på heapen, återanvända utrymmen

```
unsigned int fib2(int *n) {
  if (*n==0) {
    free(n);
    return 0;
  } else if (*n==1) {
    free(n);
    return 1;
  } else {
    int *b = malloc(sizeof(int));
    *b = *n-2;
    *n = *n-1;
    int result = fib2(n) + fib2(b);
    return result;
  }
}
```

Fibonacci: heltal på heapen, ”functional-style”

```
unsigned int fib1(int *n) {
  if (*n==0) {
    free(n);
    return 0;
  } else if (*n==1) {
    free(n);
    return 1;
  } else {
    int *a = malloc(sizeof(int));
    *a = *n-1;
    int *b = malloc(sizeof(int));
    *b = *n-2;
    free(n);
    int result = fib1(a) + fib1(b);
    return result;
  }
}
```

Tider för körningar med fib1–3

```
zoo-2:Temp tobias$ time ./timing 1 30
832040

real 0m0.290s
user 0m0.287s
sys 0m0.001s
zoo-2:Temp tobias$ time ./timing 2 30
832040

real 0m0.162s
user 0m0.160s
sys 0m0.002s
zoo-2:Temp tobias$ time ./timing 3 30
832040

real 0m0.018s
user 0m0.015s
sys 0m0.001s
zoo-2:Temp tobias$
```

Malloc

Ingen magi; `malloc` har en pekare till starten av allt ledigt minne. Vid anrop av `malloc(n)` markeras *n* bytes som använda, och pekaren till starten av allt ledigt minne flyttas fram.

Avallokering (`free`) komplicerar: (a) det lediga minnet bli fragmenterat, och (b) hur mycket minne skall `free` återlämna?

Lösning (a): en lista med pekare till block av ledigt minne samt deras storlek. (Den s.k. ”free-listan”.)

Lösning (b): en lista med pekare till allokerade block samt deras storlek.

Free-listan

Hur den är sorterad får stora konsekvenser för prestanda och fragmentering. Några exempel:

Minnesaddress Free-listan är sorterad på stigande minnesadress. Det gör det enkelt att slå samman två intilliggande fria utrymmen, och kan ofta ge referenslokalitet.

Ökande storlek Free-listan är sorterad på de fria blockens storlek i stigande ordning. Resultatet av split är ofta bara ett användbart stort utrymme.

Minskande storlek Free-listan är sorterad på de fria blockens storlek i fallande ordning. Kortare söktid än föregående. Resultatet av split är ofta två användbart stora utrymmen. Resulterar ofta i fragmentering.

Implementera en egen malloc

Behöver man *väldigt* sällan göra. Det är dock en intressant övning. Vissa specialiserade egna implementationer kan ge högre prestanda än vanliga `malloc`.

Se utdelad kod för denna föreläsning:

1. `smalloc.h` och `smalloc.c` – en väldigt naiv minnesallokerare som endast har en enda lista av blandade allokerade och fria block.
2. `rmalloc.h` och `rmalloc.c` – minnesallokeraren ovan utökad med stöd för referensräkning.
3. `imalloc.h` och `imalloc.c` – en minnesallokerare som är specialbyggd för att snabbt kunna allokera `int`:ar.

Läs gärna denna kod! (Den är säkert även full av fel, guldstjärnor till de som hittar!)

Körning av fib1–3 med `imalloc`

```
zoo-2:Temp tobias$ time ./timing 1 30
832040

real 0m0.226s
user 0m0.224s // tidigare 0.287, 22% fortare
sys 0m0.002s
zoo-2:Temp tobias$ time ./timing 2 30
832040

real 0m0.116s
user 0m0.114s // tidigare 0.160, 29% fortare
sys 0m0.002s
zoo-2:Temp tobias$ time ./timing 3 30
832040

real 0m0.018s
user 0m0.015s // oförändrad
sys 0m0.002s
zoo-2:Temp tobias$
```

Referensräkning

- En vanlig form av minneshanteringsmetod där varje allokerat block har en räknare som motsvarar antalet pekare som pekar på blocket.
- När en ny pekare skapas inkrementerar man referensräknaren. När en pekare tas bort dekrementerar man räknaren.
- När räknaren når 0 finns inte längre någon pekare till blocket, som då kan städas bort.
- Problem med cykliska strukturer – kan leda till minnesläckage. (Speciellt i system där referensräknarna manipuleras automatiskt av exekveringsmiljön.)
- När bör man (inte) manipulera referensräknaren?
- Multitrådade system?

Exempel

```
#include <rmalloc.h>

init(1024 * 16); // Använd 16K minne

int *temp = (int*) rmalloc(sizeof(int));
printf("%d\n", refcount(temp)); // 1

*temp = 7;

retain(temp);
printf("%d\n", refcount(temp)); // 2

release(temp);
printf("%d\n", refcount(temp)); // 1

release(temp); // Nu tas temp bort
printf("%d\n", refcount(temp)); // ?
```

Övningar

1. Skriv om `smalloc.c` i enlighet med first fit–ökande storlek ovan (två listor, en free-lista och en used-lista).
   Testkör programmet och se om det påverkar hastigheten. Nu kan du ganska enkelt ändra sorteringsordningen och exprimentera med hastigheten där.
2. `imalloc.c` är inte helt färdigskriven. Om det utrymme som allokeras av `init` tar slut borde en ny `chunk` allokeras som pekas ut av `H->next`. Skriv färdigt `imalloc.c` så att dess minnesanvändning växer dynamiskt.
3. Fundera på vad som är en vettig strategi för att lämna tillbaka minne med den nya `imalloc.c`, och om man kan vilja avallokera en hel `chunk` och i så fall när.
4. Skriv om `imalloc.c` så att den inte är specifik för `int`:ar, utan fungerar med element av en godtycklig (fix) storlek.
   Man skall inte behöva kompilera om `imalloc.c` för att byta storlek på elementet.

I/O

F10

Exempel: Läsa och skriva tecken

Exempel: Kopiera filen "in.txt" till "ut.txt"

```
FILE *infil = fopen("in.txt", "r"); // r för läsning
File *utfil = fopen("ut.txt", "w"); // w för skrivning
int c;
while ((c=getc(infil)!=EOF)
   putc(c, utfil);
fclose(infil);
fclose(utfil);
```

Anm: `getchar()` och `putchar(c)` är liktydigt med `getc(stdin)` respektive `putc(c, stdout)`.

Filhantering

`stdio.h` deklarerar en datatyp `FILE` samt funktionerna `fopen`, `fclose`, `fprintf`, `fscan`, `feof` m. fl.

Exempel

```
FILE *inf = fopen("indata.txt", "r");
FILE *utf = fopen("utdata.txt", "w");
int tal;
if (inf==NULL) {
   printf("Gick ej att öppna\n");
    return 1;
}

while (fscanf(inf, "%d", &tal) == 1) {
    if (tal > 0) {
        fprintf(utf, "%d\n", tal);
    }
}

fclose(inf);
fclose(utf);
```

Observera att följande kod inte fungerar helt korrekt – sista talet kommer att skrivas ut två gånger

```
while (!feof(inf)) {
    fscanf(inf, "%d", &tal);
    if (tal > 0) {
        fprintf(utf, "%d\n", tal);
    }
}
```

Exempel: inläsning av ord

```
#include <stdio.h>

int readWord(char *s, FILE *f);
  /* Läser ett ord (obruten sekvens av engelska bokstäver)
     Pre : s - en area tillräckligt stor för att rymma det längsta ordet
           f - en öppen indataström
     Post: s innehåller nästa ord
     Returvärde: radnumret för ordet
  */

// Testprogram
int main() {
    char buffer[100];
    int nr;
    while ((nr = readWord(buffer, stdin)) != -1) {
        printf("%2d %s\n", nr, buffer);
    }
}
```

```
int readWord(char *s, FILE *f) {
    static lineNumber = 1;
    int c;

    while (!isalpha(c=getc(f))) { // Leta upp nästa ord.
        if (c==EOF) // Hantera EOF
            return -1;
        else if(c=='\n') // och radbyten
            lineNumber++;
    }

    while (isalpha(c)) { // Samla bokstäverna
        *(s++) = c;
        c = getc(f);
    }
    *s = '\0'; // Terminera strängen

    ungetc(c,f); // Lägg tillbaka icke-bokstaven
    return lineNumber;
}
```

Observationer och frågor

- *Deklaration* respektive *definition* av `readWord`. Varför?
- Användning av `stdin` i `main`
- `static int lineNumber = 1`
- `ungetc(int c, FILE *f)`
- Vad skall `readWord` göra om ord längre än 99 tecken?

Övningar

1. Skriv en funktion `int nextChar(FILE *f)` som läser förbi "white space" (blank, tab och radbyte) och returnerar det tecknet som följer. Tecknet skall läggas tillbaka i strömmen så att nästa läsning börjar vid det tecknet. Vid filslut skall `EOF` returneras.
2. Skriv ett program för att lista filer med angivande av radnummer. Programmet skall anropas med angivande av filnamn. Exempel:

   ```
   marvin$ list uppgift1.c
   ... output ...
   marvin$
   ```

   Om filen inte hittas så skall ett felmeddelande skrivas.

Preprocessorn

F11

Preprocessorn

Preprocessorn utför vissa textsubstitutioner som ett försteg till kompilatorn.

Filinkludering (Kan vara nästlade.)

`#include` "*filnamn*"
`#include` <*filnamn*>

Makrofacilitet

`#define` *namn utbytestext*

Exempel:

```
#define Euler 0.57721566
#define TRUE 1
#define FALSE 0
#define begin {
#define end }
#define debug 1
```

Obs: Inget semikolon på slutet av raden!

Makron med parametrar

Makron kan ha parametrar, som substitueras in *före evaluering*

```
#define sqr(x) (x)*(x)
#define max(x,y) ((x) > (y) ? (x) : (y))
```

Obs: Parenteserna viktiga!

Hitta felen!

```
#define sqrA(x) (x)*(x)
#define sqrB(x) (x)*(x);
#define sqrC(x) x*x
int a = 9;
int b = sqrA(++a);
int c = sqrB(a)+10;
int b = sqrC(a-10);
```

#-operatorn; att göra om en token till en sträng

Ibland kan man vilja göra en en token till en sträng, t.ex. för att skriva ut den.

```
#define PRINT(t) printf(#t " är %d", t)
```

`PRINT(foo)` expanderar nu till:

```
printf("foo" " är %d", foo)
```

Observera att `"b" "c"` i C är detsamma som `"bc"`, d.v.s. strängliteraler utan mellanliggande operatorer konkateneras.

Man man använda `PRINT` ovan för t.ex. debugging:

```
PRINT(x + y); ==> printf("x + y är %d", x+y);
```

#-operatorn; exempel

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[] = join(x, y); // equivalent to char p[] = "x ## y";
```

Sista raden ovan expanderar, stegvis till:

```
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

##-operatorn konkatenerar en token med ett uttryck:

```
#define DO(a,b,c,d) case a: \
                    case b: c ## order(d); break

switch(c) {
  DO('i', 'I', in, c); ==> case 'i': case 'I': inorder(c); break;

}
```

Makron på flera rader

```
#define SWAP(a, b) { \
                     a ^= b; \
                     b ^= a; \
                     a ^= b; \
                   }
```

(Notera användningen av bitoperatorn för XOR.)

Vanliga mönster

```
#define SWAP(a, b) a ^= b; b ^= a; a ^= b;

int x = 10;
int y = 5;

// Fungerar bra
SWAP(x, y);

// Fungerar mindre bra
if(x < 0)
    SWAP(x, y);
```

Vanliga mönster

```
#define SWAP(a, b) { a ^= b; b ^= a; a ^= b; }

int x = 10;
int y = 5;
int z = 4;

// Kompilerar
if(x < 0)
    SWAP(x, y);

// Kompilerar inte!
if(x < 0)
    SWAP(x, y);
else
    SWAP(x, z);
```

Vanliga mönster

```
#define SWAP(a, b) do { a ^= b; b ^= a; a ^= b; } while ( 0 )

int x = 10;
int y = 5;
int z = 4;

// Fungerar detta?
if(x < 0)
    SWAP(x, y);
else
    SWAP(x, z);
```

Proprocessor fortsättning

Villkorlig inkludering

```
#if debug == 1
    printf("Nu är vi här\n");
#endif
```

Mycket använt för deklarationsfiler (s.k. *headerfiler*):

```
#ifndef _stack_
#define _stack_

  ...

#endif
```

Kompileringsenheter och deklarationsfiler

- Program av någon storlek delas normalt upp på flera källkodsfiler
- Kompilatorn behandlar en källkodsfil i taget
- Ingen information om om funktioner, datatyper etc. förs över automatiskt mellan filer utan allt som skall användas i en fil måste *deklareras* där (däremot behöver de inte *definieras* där
- Begrepp (funktioner, variabler, typer) definierade på översta nivån i en fil är globala
- Nyckelordet **`static`** kan användas för att förbjuda extern access (dvs access från andra filer) (jmf. nyckelordet **`extern`**)
- Deklarationer samlas ofta på särskilda "headerfiler" (`.h`-filer) som inkluderas av de filer som vill referera dessa saker.
- Kompilatorflaggor `-Dnamn`, `-Dnamn=definition` och `-Unamn` för att definiera/avdefiniera namn vid kompilering.
- Pröva gärna att köra preprocessorn (`cpp`) separat

Exempel: Sorterade listor av heltal på flera filer

```
/* list.h - Deklarationsfil för listor av heltal */
#ifndef __list__
#define __list__

typedef struct listElem {
    int item;
    struct listElem *next;
} listElem;

typedef listElem *Link;

/* Lägger in en nytt tal i en sorterad lista
   Pre: item - godtyckligt int
        link - pekare till första noden i en sorterad lista
   Post: Listan är fortfarande sorterad
   Return: Pekare till första noden i den modifierade strukturen
*/
Link insert(int item, Link link);
Link mkLink(int item, Link next);
void print(Link link);
int length(Link link);
#endif
```

```c
/* list.c - Implementationsfil för heltalslistor */

#include <stdio.h>
#include <stdlib.h>
#include "list.h"

static inline Link *find(Link *link, int item) {
  while (*link && (*link)->item < item) link = &(*link)->next;
  return link;
}

Link insert(int item, Link link) {
  Link *place = link ? find(&link, item) : &link;
  *place = mkLink(item, *place);
  return link;
}

Link mkLink(int item, Link next) {
  Link link = (Link) malloc(sizeof(listElem));
  link->item = item;
  link->next = next;
  return link;
}
```

```
void print(link link) {
    printf("[");
    while (link) {
        printf("%d", link->item);
        if (link->next) printf(", ");
        link = link->next;
    }
    printf("]");
}
```

```c
/* listTest.c - Testprogram för heltalslistor
 */

#include <stdio.h>
#include <stlib.h>
#include "list.h"

int main() {
    int i;
    link list=NULL;

    for (i = 0; i<10; ++i) {
        list = insert(rand()%10, list);
    }
    print(list);
    printf("\n");

    return 0;
}
```

C-exempel: Stack

F13

Viktigt exempel: Stack med godtyckligt data

Antag att vi vill ha en generell *stack*.

En stack är en mekanism som man kan lagra i och hämta från enligt principen ”sist in, först ut” (eng. LIFO).

Man skall alltså kunna

- Skapa en stack
- Lägga värden av ospecificerad typ på stacken (*push*)
- Ta bort senast lagrade värde (”översta”) värdet från stacken (*pop*
- Ta reda på om stacken är tom
- Titta på senast lagrade (ej borttagna) värde på stacken

Användaren skall inte behöva veta hur den är implementerad.

Vill ha en typ `Stack` som vi använder i deklarationer.

Representation av stacken

- Kravet på generell typ $\Longrightarrow$ ospecificerad pekare (`void` $*$)
- Array eller lista? Array enkel men fix storlek.
- Hur skall stacken representeras utåt?
  1. En pekare till första elementet eller
  2. en särskild stackpost eller
  3. en pekare till en särskild stackpost?

Stack (fortsättning)

Vi väljer alternativ 3, en pekare till första elementet i en länkad lista.

```
Stack newStack(); // Konstruktor
void push(void *item, Stack stack); // Lagrar på stack
void *pop(Stack s); // Tar bort från stack
void *top(Stack s); // Översta värdet
```

Med användning

```
Stack s = newStack();
push("hej", s);
push("hopp", s);
char *t = (char*) top(s);
char *v = pop(s);
```

Deklarationsfil `stack.h`

```
#ifndef __stack__
#define __stack__

typedef struct stackElem {
    void *item;
    struct stackElem *next;
} stackElem, *Link;

typedef struct stack {
  Link top;
} stack, *Stack;

Stack newStack(); // Returnar en ny, tom, stack
void push(void *item, Stack stack); // Lagrar ett nytt element överst
void *top(Stack s); // Returnerar översta
void *pop(Stack s); // Returnerar och tar bort översta
int isEmpty(Stack s); // Returnar 1 om tom stack, annars 0

#endif
```

Implementationsfil `stack.c`

```
/**
 * stack.c
 * Implementationsfil för stack med ospecificerad typ (void *)
 */

#include <assert.h>
#include <stdlib.h>
#include "stack.h"

Stack newStack() {
  Stack s = (Stack) malloc(sizeof(stack));
  s->top = NULL;
  return s;
}
```

Implementationsfil `stack.c` forts

```
int isEmpty(Stack stack) {
  assert(stack);
  return stack->top==NULL;
}

void* top(Stack stack) {
  assert(stack && stack->top);
  return stack->top->top->item;
}

void push(void *item, Stack stack) {
  ... Övning!
}

void * pop(Stack stack) {
   ... Övning!
}
```

Testprogram för stack-mekanismen

```
#include <stdio.h>
#include "stack.h"
#define topVal(st) (*(int *)top(st))
int main() {
    Stack s = newStack();
    int t1 = 1, t2 = 2, t3 = 3;
    double d = 4711.0;

    printf("Integers:\n");
    push(&t1,s); printf("Top: %d\n", topVal(s));
    push(&t2,s); printf("Top: %d\n", topVal(s));
    pop(s); printf("Top: %d\n", topVal(s));
    push(&t3,s); printf("Top: %d\n", topVal(s));
    t3 = 333; printf("Top: %d\n", topVal(s));

    printf("\nA double:\n"); push(&d,s);
    printf("Top: %lf\n", *(double*) top(s));
    printf("\nStrings:\n");
    push("Don't", s); push("panic!", s); printf("%s ", (char*) pop(s));
    printf("\n\nPop until empty:\n");
    while(!isEmpty(s)) printf("%d ", *(int *)pop(s));
    printf("\nBye!\n");
    return 0;
}
```

Testprogram för stack-mekanismen (körning)

Körning:

```
$ gcc -Wall stack.c stackTest.c
$ ./a.out

Integers:
Top: 1
Top: 2
Top: 1
Top: 3
Top: 333

A double:
Top: 4711.000000

Strings:

panic!

Pop until empty:
661548868 0 333 1

Bye!
```

Användning av stack: evaluera uttryck i RPN

```
/* rpn.c - Räknare för uttryck i omvänd polsk notation.

   Demonstrerar hur stack-mekanismen kan användas.

   OBS: Saknar vettig felhantering
*/

#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

/** Eget interface till stackmekanismen för att få en stack med double */

static Stack operands; // Global "privat" stack

double topVal() { // Topp-värdet som double
    return *(double *) top(operands);
}
```

Användning av stack: evaluera uttryck i RPN forts

```
double popVal() { // Poppar och returnerar toppvärdet
    double *topptr = (double*) top(operands);
    double topVal = *topptr
    operands = pop(operands);
    free(topptr);
    return topVal;
}

void pushVal(double val) { // Pushar double
    double *heapVal;
    heapVal = (double*) malloc(sizeof(double));
    *heapVal = val;
    operands = push(heapVal, operands);
}

/* slut på stackinterface */
```

Användning av stack: evaluera uttryck i RPN forts

```
/** IO-interface */

int next() {
/* Returnerar nästa icke-blanka tecken UTAN att 'konsumera' det dvs
   nästa inputoperation kommer att börja med detta tecken
*/

    int c;
    while ((c=getchar()) == ' ')
        ;
    ungetc(c, stdin); // "Lägg tillbaka"
    return c;
}

/* slut på IO-interface */
```

Användning av stack: evaluera uttryck i RPN forts

```
void perform(int operand) {
    /* Utför operation 'operand' på den globala stacken */
    double y = popVal(); // Obs: ordningen
    double x = popVal();
    if (operand == '+')
        pushVal(x+y);
    else if (operand == '-')
        pushVal(x-y);
    else if (operand == '*')
        pushVal(x*y);
    else if (operand == '/')
        pushVal(x/y);
    else; // Fel ...
}
```

Användning av stack: evaluera uttryck i RPN forts

```
int main() {
    int c;

    operands = newStack(); // Initiera stacken
    while (1) { // oändlig loop
        c = next();
        if (isdigit(c)) { // Om tal så läs in och lagra det
            double x;
            scanf("%lf", &x);
            pushVal(x);
        } else { // annars kommando eller operator
            getchar(); // Läs förbi
            if ( c=='\n' ) {
                printf("%f\n", topVal());
            } else if ( c=='q' ) {
                return 0;
            } else {
                perform(c);
            }
        }
    }
    return 0;
}
```

Exempel: Evaluering av uttryck i infix notation

Aritmetiska uttryck i vanlig, infix, notation kan (bl.a.) evalueras med den s.k. järnvägsalgoritmen. Den är baserad på två stackar – en för operatorer och en för operander.

Algoritmskiss:

```
Om tal
    Pusha talet på operandstacken.
annars om slut på uttryck
    Poppa operatorstacken och utför varje operation tills operatorstacken tom.
    Skriv ut det enda kvarvarande värdet på stacken.
annars om vänsterparentes
    Lägg den på stacken.
annars om högerparentes
    Poppa och utför operationerna tills vänsterparentes hittas
annars om operator
    Poppa operatorstacken och utför operationerna tills operatorn på
    stacktoppen har lägre prioritet än den aktuella operatorn.
    Pusha operatorn på stacken.
annars fel
```

Exempel: Evaluering av uttryck i infix-notation

```
/* calc.c - Kalkylator för aritmetiska uttryck i infix-(vanlig)-notation.

   Använder två stackar: en med flyttal för operander och en med
   heltal (tecken) för operatorer.

   OBS: Ingen vettig felhantering. Förutsätter att varje rad i
   indata innehåller ett syntaktiskt korrekt uttryck.

*/

#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

/** Eget interface till resultatstacken */
static Stack operands; // Global "privat" resultatstack

double topVal() {...} // Som i RPN-exemplet
double popVal() {...} // Som i RPN-exemplet
void pushVal(double val) {...} // Som i RPN-exemplet

/** IO-interface */
int next() {...} // Som i RPN-exemplet
```

Exempel: Evaluering av uttryck i infix-notation forts

```
/** Räknaren */

void perform(int operator) {...} // Som i RPN-exemplet

int priority(int operator) {
  /* Returnerar operatorns proritet
     Pre : operator - teckenkod för en giltig operator
     Retunerar: operatorns prioritet (när den ligger på stacken)
  */
  switch (operator) {
  case '(':
    return 0;
  case '+':
  case '-':
    return 1;
  case '*':
  case '/':
    return 2;
  }
}
```

Exempel: Evaluering av uttryck i infix notation forts

```
int main() {
  /* Parsar och evaluerar aritmetiska uttryck med den s.k. järnvägsalgoritmen */

    operands = newStack(); // Initiera operandstacken
    Stack operators = newStack(); // Initiera operatorstacken

    while (1) {
        int oper;
        int c = next();
        if (isdigit(c)) { // Om tal så lagra
            double x;
            scanf("%lf", &x);
            pushVal(x);
        } else { // annars
            getchar(); // Läs förbi
            if ( c=='\n' ) { // slut på uttryck?
                while (!isEmpty(operators)) {
                    perform((int)top(operators));
                    operators = pop(operators);
                }
                printf("%f\n", topVal());
                popVal();
```

Exempel: Evaluering av uttryck i infix notation forts

```
            } else if (c == 'q') { // eller quit?
                return 0;
            } else if (c == '(') { // eller '('
                operators = push((void*) c,operators); // Oops! Pekare?
            } else if (c== ')') { // eller ')'
                while ((oper = (int) top(operators)) != '(') { // Oops!
                    perform(oper);
                    operators = pop(operators);
                }
                operators = pop(operators);
            } else { // eller operator
                while (!isEmpty(operators) &&
                       priority(c) <= priority(oper = (int) top(operators))) { //Oops!
                    perform(oper);
                    operators = pop(operators);
                }
                operators = push((void*) c, operators); //Oops!
            }
        }
    }
    return 0;
}
```

Problem med den använda stackdesignen

Kan man skriva en funktion som skriver ut stackens innehåll?

Kan man skriva en funktion som letar efter en nod med viss egenskap?

Vad är problemet?

- Stackmekanismen vet inte vad det är för typ på lagrade data så den vet inte hur elementen t ex skall skrivas ut (formatkoder . . .) eller hur de skall jämföras om man letar efter visst innehåll.
- Utifrån kommer man bara åt det översta värdet. Den interna strukturen är avsiktligt dold.

Iteratorer

F14

Iteratorer

Eftersom man kan skicka *funktionspekare* som parameter så *går* det att skriva en funktion i stackmekanismen som traverserar stacken och utför en specificerad operation på alla lagrade värden.

Vi skall dock välja att implementera ett annat begrepp: en *iterator*.

En iterator är en mekanism som gör det möjligt att på ett kontrollerat sätt besöka alla lagrade element i en struktur (lista, träd, array . . .).

Finns inbyggd i språk som C++ och Java men i C får vi göra den själva.

Vi väljer en enkel variant där man bara kan gå åt ena hållet och där man bara kan titta på värdena, inte ändra dem.

Iteratorer (forts)

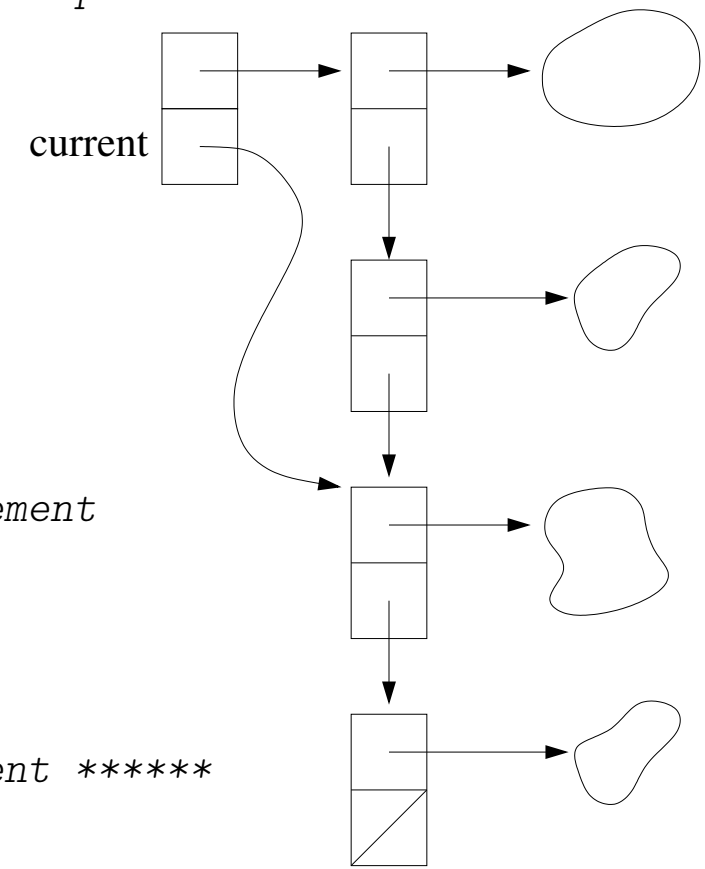
Vi skall förutom att hålla reda på stacktoppen också hålla reda på ett *aktuellt element*

Det medför att vi inte längre kan representera stacken utåt med en pekare till stacktoppen utan vi måste skapa en post som innehåller både pekare till toppen och pekare till aktuellt element.

Utåt låter vi stacken vara representerat med en pekare till en sådan post.

Deklarationsfil för stack med iterator

```
/* stack.h - Deklarationsfil för stack med iterator
   En generell stackmekanism som lagrar pekare till ospecificerad typ
   (void*). Stacken representeras av en post med en pekare till
   översta elementet och, för iteratorändamål, en pekare till
   aktuellt element. Själva stacken består
   av en länkad lista.
*/

#ifndef __stack__
#define __stack__

typedef struct stackElem { // Listelementen
    void *item; // Ospecificerad pekartyp
    struct stackElem *next; // Nästföljande element
} stackElem, *link;

typedef struct Stack {
    link top; // Pekare till översta elementet
    link current; // Pekare till aktuellt element ******
} *Stack;
```



Deklarationsfil för stack med iterator forts

```
Stack newStack(); // Konstruktor
int push(void *it, Stack s); // Lagrar ett element på stacken
void *pop(Stack s); // Tar bort och returnerar översta elementet
int isEmpty(Stack s); // 1 om stacken tom, annars 0
void *top(Stack s); // Returnerar översta elementet på stacken
void iterInit(Stack s); // Initierar iteratorn till översta elementet
int iterMore(Stack s); // 1 om det finns aktuellt element, annars 0
void *iterNext(Stack s); // Returnerar aktuellt element och stegar fram

#endif
```

Implementationsfil för stack med iterator

```
/* stack.c - Imlementationsfil för stack med iterator */

#include <assert.h>
#include <stdlib.h>
#include "stack.h"

Stack newStack() {
  /* Skapar en tom stack */
    Stack s = (Stack) malloc(sizeof(struct Stack));
    s->top = NULL;
    s->current = NULL;
    return s;
}
```

Implementationsfil för stack med iterator forts

```
int push(void *item, Stack stack) {
  /* Lagrar ett element överst på stacken.
     Pre : item - En godtycklig pekare.
                : stack - Pekare till en initierad stack.
     Post : Det som it pekar på ligger överst på stacken.
     Returnerar: 1 om operationen lyckades, annars false
  */

     Övning!

}
```

Implementationsfil för stack med iterator forts

```
void *pop(Stack stack) {
  /* Tar bort och returnerar översta elementet på stacken.
     Pre : stack - Pekare till en initierad stack.
     Post : Om stacken inte var tom är det översta elementet borttaget.
     Returnerar: NULL om stacken var tom, annars det värde som var överst.
  */
  assert(stack);
  if (stack->top==NULL)
    return NULL;
  else {
    link temp = stack->top;
    stack->top = stack->top->next;
    void *result = temp->item;
    free(temp);
    return result;
  }
}
```

Implementationsfil för stack med iterator forts

```
int isEmpty(Stack stack) {
  /* Undersöker om stacken är tom.
     Pre : stack - Pekare till initierad stack.
     Post : Stacken oförändrad.
     Returnerar: 1 om stacken tom, annars 0.
  */
  assert(stack);
  return stack->top==NULL;
}

void *top(Stack stack) {
  /* Tittar på översta elementet i stacken.
     Pre : stack - Pekare till initierad stack.
     Post : Stacken oförändrad.
     Returnerar: Översta värdet på stacken.
  */
  assert(stack);
  return stack->top->item;
}
```

Implementationsfil för stack med iterator forts

```
void iterInit(Stack stack) {
  /* Initiera iteratorn .
     Pre : stack - Pekare till initierad stack.
     Post : Stackens innehåll oförändrad.
                Översta element satt som aktuellt element .
  */
    assert(stack);
    stack->current = stack->top;
}

int iterMore(Stack stack) {
  /* Undersöka om alla elementen genomgångna.
     Pre : stack - Pekare till initierad stack.
     Post : Stackens innehåll oförändrad.
     Returnerar: 1 om det finns ett aktuellt element, annars 0.
  */
    assert(stack);
    return stack->current!=NULL;
}
```

Implementationsfil för stack med iterator forts

```
void *iterNext(Stack stack) {
  /* Stegar fram iteratorn till nästa element.
     Pre : stack - Pekare till initierad stack.
     Post : Stackens innehåll oförändrad.
                Om det finns ett aktuellt element kommer dess efterföljare
                vara det nya aktuella elementet.
     Returnerar: Om det vid inträdet finns ett aktuellt element kommer
                dess värde returneras, annars NULL.
  */
  link temp = stack->current;
  if (stack->current) {
    stack->current = stack->current->next;
    return temp->item;
  } else {
    return NULL;
  }
}
```

Testprogram av stack med iterator

```
/* stackTest.c - Test av stack med iterator */

#include <stdio.h>
#include "stack.h"

int main() {
    Stack s = newStack();

    // Pusha tre heltal (som vore de pekare...) och poppa dem
    push((void*) 1, s);
    push((void*) 2, s);
    push((void*) 3, s);

    printf("Poppade tal: ");
    while (!isEmpty(s)) {
        printf("%d ", (int)pop(s));
    }
    printf("\n");
```

Testprogram av stack med iterator forts

```
    // Pusha några (pekare till) strängar
    push("fisken", s);
    push("för", s);
    push("tack", s);

    // Iterara genom stacken m h a iteratorn
    iterInit(s);
    printf("Itererat: ");
    while (iterMore(s)) {
        printf("%s ", iterNext(s));
    }

    // Pusha ytterligare en sträng och poppa sedan stacken
    push("Ajöss och", s);
    printf("\nPoppat : ");
    while (!isEmpty(s)) {
        printf("%s ", pop(s));
    }
    printf("\n*Slut på test*\n");
}
```

Utskrifter från testprogram

Utskrift från testkörning:

```
bellatrix$ gcc -o stackTest stack.c stackTest.c
bellatrix$ stackTest
Poppade tal: 3  2  1
Itererat: tack för fisken
Poppat  : Ajöss och tack för fisken
*Slut på test*
bellatrix$
bellatrix$
```

Vad går och vad går inte att göra med iteratorn?

- Kan avläsa innehållet på stacken
- Kan ändra på innehållet i det som stackelementen pekar på
- Kan *inte* ta byta ut element
- Kan *inte* ta bort element
- Kan *inte* stega bakåt

Om man vill byta ut element?

```
void **iterCurrent(Stack s) {
  assert(s&&s->current);
  return &(s->current->item);
}
```

Test:

```
  push("soir!\n", s);
  push("Bon ", s);
  iterInit(s);
  while(iterMore(s)) {
    printf("%s", (char *)iterNext(s));
  }
  iterInit(s);
  void **p = iterCurrent(s);
  *p = "Guten ";
  iterNext(s);
  *iterCurrent(s) = "Abend!\n";
  iterInit(s);
  while(iterMore(s)) {
    printf("%s", (char *)iterNext(s));
  }
```

```
  push((void *)17L, s);
  iterInit(s);
  printf("\n%ld\n", (long int)top(s));
  *iterCurrent(s) = (void *)42L;
  printf("%ld\n", (long int)top(s));
```

.

Output:

```
Bon soir!
Guten Abend!

17
42
```

Andra iteratoroperationer

? Kan man göra `removeCurrent(s)`?

Ja, låt `current` peka på pekaren till aktuellt element!
Övning! De som klarar det har förstått pekare!

? Kan man göra en `iterPrev(s)` som stegar åt andra hållet?

Ja. Det kräver en dubbellänkad struktur.

? Kan man ha flera iteratorer för samma struktur (som i t.ex. Java)?

Ja, men då får man ha en separat iteratorpost med pekare dels till stacknoden och dels till (pekaren till?) aktuellt element. Måste dessutom ha någon metod att förhindra access till borttagna element.

Funktionspekare

F16

Pekare till funktioner

```
/* simpleList.h - listor med heltal */
typedef struct listNode {
  int item;
  struct listNode *next;
} listNode, *Link;

void apply(Link l, void (*f)(int *)) {
  while (l) {
    f(&(l->item));
    l = l->next;
  }
}

void add1(int *it) { *it = *it + 1; }

void pr(int *it) { printf(" %d ", *it); }

int main() {
  // ... stuff
  apply(l, pr);
  printf("\n");
  apply(l, add1);
  apply(l, pr);
  return 0;
}
```

Output:

```
[0, 0, 2, 3, 6, 6, 8, 9]
 1  1  3  4  7  7  9  10
```

Pekare till funktioner (forts, med `typedef`)

```
typedef void (*int_to_void)(int *);

typedef struct listNode {
  int item;
  struct listNode *next;
} listNode, *Link;

void apply(Link l, int_to_void f) {
  while (l) {
    f(&(l->item));
    l = l->next;
  }
}

void add1(int *it) { *it = *it + 1; }

void pr(int *it) { printf(" %d ", *it); }

int main() {
  ...
```

Output:

```
[0, 0, 2, 3, 6, 6, 8, 9]
 1  1  3  4  7  7  9  10
```

Exempel: En generell `treeMap`

En *map* är (i detta sammanhang) en avbildning från en mängd med *nycklar* till en mängd med *värden*.

Exempel:

- Personnamn – telefonnummer
- Registeringsnummer för fordon – ägaruppgift
- Personnummer – folkbokföringsuppgifter
- Symboltabell i C-kompilator – uppgift om datatyp, adress mm

Representeras typiskt som ett *binärt sökträd* eller en *hashtabell*

Skall visa hur man kan implementera en sådan som ett binärt sökträd.

Exemplifierar bl.a. *funktionspekare*, en mer avancerad *iterator* och *stackanvändning*.

Generell `treeMap` forts

- En nod i treemap:en skall innehålla en nyckel parat med ett värde
- Trädet skall sorteras med avseende på nycklarna
- Kravet på generalitet innebär att både nycklar och värden måste ha typen `void *`
- Hur jämföra nycklar av typen `void *` inuti trädet för att bibehålla sortering vid insertering?
- Det skall finnas operationer för att

lagra nyckel – värdepar (`put`),
hämta nyckel – värdeparvärdet som hör till en viss nyckel (`get`),
fråga om en viss nyckel finns (`containsKey`)

Dessutom skall det finnas en iterator för *in-order*-traversering

Generell `treeMap` forts

Ett första utkast till gränssnitt:

```
Tree makeTree();
void put(Tree t, void *key, void *value);
void *get(Tree t, void *key);
int containKey(Tree t, void *key);

void iterInit(Tree t);
int iterMore(Tree t);
void *currentKey(Tree t);
void *currentValue(Tree t);
void iterNext(Tree t);
```

Ännu olöst problem: `put`, `get` och `containsKey` kräver att nycklarna kan jämföras storleksmässigt.

Generell `treeMap` forts

Möjliga lösningar: Förse dessa med en funktionsparameter som avgör ordningen mellan två godtyckliga nycklar.

Exempel:

```
void put(Tree t, void *key, void *value, int (*compare)(void *, void *));
```

där funktionen `compare` returnerar ett negativt värde om första argumentet är mindre än det andra, noll om de är lika och ett positivt värde om det första är större än det andra (jämför `strcmp`)

Krångligt och felbenäget att behöva skicka in en sådan pekare hela tiden. En mer elegant lösning vore att "upplysa trädet" om hur nycklar skall jämföras när det skapas!

Generell `treeMap` forts

Lösning: Lagra en pekare till jämförelsefunktionen i trädposten.

Vi får då följande deklarationer:

```
typedef struct treeNode { // Noderna i trädet
    void *key;
    void *value;
    struct treeNode *left;
    struct treeNode *right;
} treeNode, *TreeNode;

typedef int (*comparator)(void *, void *); // Jämförelsefunktionens interface

typedef struct { // Trädpost
    TreeNode root;
    comparator cmp; // Pekare till jämföresefunktionen
    Stack current; // Används av iteratorn. Förklaras senare.
} treeRecord, *Tree;
```

Generell `treeMap` forts

Resten av deklarationsfilen:

```
/* Tree interface functions */
Tree makeTree(comparator cmp); // Skicka med jämförelsefunktionen
void put(Tree t, void *key, void *value);
void *get(Tree t, void *key);
int containKey(Tree t, void *key);

/* Iterator functions */
void iterInit(Tree t);
int iterMore(Tree t);
void *currentKey(Tree t);
void *currentValue(Tree t);
void iterNext(Tree t);

/* Internal functions */
static TreeNode putPrivate(TreeNode, void *, void *, comparator);
static TreeNode search(TreeNode, void *, comparator);
```

Implementationsfil `treeMap.c`

```
static TreeNode makeTreeNode(void *key, void *value) {
   TreeNode result = (TreeNode) malloc(sizeof(treeNode));
   result->key = key;
   result->value = value;
   result->left = NULL;
   result->right = NULL;
   return result;
}

Tree makeTree(comparator cmp) {
   Tree result = (Tree) malloc(sizeof(treeRecord));
   result->root = NULL;
   result->cmp = cmp;
   result->current = newStack(); // Förklaras senare
   return result;
}
```

Implementationsfil `treeMap.c`

Intern funktion för att lokalisera trädnod med viss nyckel. Används av `get` och `containsKey`:

```
static TreeNode search(TreeNode r, void *key, comparator cmp) {
  if (r==NULL)
    return NULL;
  else if (cmp(key, r->key)==0)
    return r;
  else if (cmp(key, r->key)<0)
    return search(r->left, key, cmp);
  else
    return search(r->right, key, cmp);
}
```

Iterator för In-order-traversering

Mer komplicerad än en listiterator – måste hålla reda på vägen tillbaka till roten.

Lösning: använd en stack och push:a varje passerad nod.

- Se till att det aktuella elementet alltid ligger överst på stacken.
- Initiera genom att gå ner till vänster och pusha alla noder på vägen
- Vid stegning: Poppa stacken. Om den poppade noden har högerbarn så pusha det och hela dess vänstergren

Implementation av iteratorn

```
void iterInit(Tree t) {
   drain(t->current); // Töm stacken (ny funktion)
   TreeNode tn = t->root;

   while(tn) {
      push(tn,t->current);
      tn = tn->left;
   }
}

int iterMore(Tree t) {
   return !isEmpty(t->current);
}
```

Implementation av iteratorn

```
void *currentKey(Tree t) {
  assert(!isEmpty(t->current));
  return ((TreeNode)top(t->current))->key;
}

void *currentValue(Tree t) {
  assert (!isEmpty(t->current));
  return ((TreeNode)top(t->current))->value;
}

void iterNext(Tree t) {
  // Övning!
}
```

Testkod

```
Tree t = makeTree((comparator) strcmp);

put(t, "j", "jjj");
put(t, "a", "aaa");
put(t, "k", "kkk");
put(t, "l", "lll");

for (iterInit(t); iterMore(t); iterNext(t)) {
  printf("%10s = %s\n",
         (char*) currentKey(t),
         (char*) currentValue(t));
}

put(t, "a", "bbb");

printf("Nya värdet på a: %s\n", (char *)get(t, "a"));
```

Output:

```
Marvin$ ./a.out
         a = aaa
         j = jjj
         k = kkk
         l = lll
Nya värdet på a: bbb
```

Övningar

1. Implementera funktion `int size(Tree t)` som returnerar antalet noder i trädet.
2. Implementera funktionen `void drain(Stack s)` som tömmer stacken.
3. Implementera funktionen `int containsKey(Tree t, void *key)` som letar efter nod med angiven nyckel och returnerar 1 om den finns, annars 0.
4. Implementera funktionen `void *get(Tree t, void *key)` som returnerar värdet som hör till angiven nyckel eller `NULL` om nyckeln inte finns.
5. Implementera funktionen `void put(Tree t, void *key, void *value)` som lagrar angivet par av nyckel och värde. Om nyckeln redan finns så skall värdet bytas ut.
6. Implementera funktionen `void iteNext(Tree t)` som stegar fram iteratorn

Konstanter

F17

Symboliska konstanter

Tre sätt

- med preprocessormacron
- med **`const`**-deklaration
- med **`enum`**-deklaration (endast heltalskonstanter)

Exempel:

```
#define SIZE 100
const int ANSWER = 42;
const double PI = 3.141592654;
enum {IT, DV, NV}; // IT=0, DV=1, NV=2
```

Mer om uppräkningsbara typen **enum**

Man kan använda **enum** som typdeklaration:

```
enum program {IT=1, DV=2, NV=3}; // Kan ge egna värden
enum program p;

for (p = IT; p <= NV; ++p) {
   printf("%d ", p);
}
```

Deklarationerna ovan är således liktydigt med

```
const int IT = 1, DV = 2, NV = 3;
int program;
```

Använd alltid typedef för bättre abstraktion:

```
typedef enum {IT, DV, NV} program;
program p;
```

`const` och pekare

```
char *unsafe(char *s) {
  // Returnerar pekare till det andra tecknet i s
  printf("%s \n", s);
  s++; // OK
  printf("%s \n", s);
  *s = '*'; // Ej säkert OK
  printf("%s \n", s);
  return s;
}

int main() {
  char s[] = "Don't panic!";
  char *u;
  u = unsafe(s);
  printf("Första anrop: %s \n", u);
  u = unsafe("Don't panic!");
  printf("Andra anrop : %s \n", u);
}
```

Output:

```
Don't panic
on't panic
*n't panic
Första anrop: *n't panic
Don't panic!
on't panic!
Segmentation fault (core dumped)
```

`const` och pekare forts

```
char *safer(const char *s) { // <<
  printf("%s \n", s);
  s++;
  printf("%s \n", s);
  // *s = '*'; // << Skulle ge kompileringsfel
  return s; // << Ger varning
}

int main() {
  char *sIn = "Don't panic";
  char tIn[] = "Don't panic";
  char *sUt;
  char *tUt;
  sUt = safer(sIn);
  tUt = safer(tIn);
  printf("sUt: %s \n", sUt);
  printf("tUt: %s \n", tUt);
  *sUt = '*';
  *tUt = '*';
}
```

Output:

```
$ gcc const2.c
const2.c: In function ‘safer’:
const2.c:12: warning: return discards qualifiers
from pointer target type
$ ./a.out
Don't panic
on't panic
Don't panic
on't panic
sUt: on't panic
tUt: on't panic
Segmentation fault (core dumped)
```

`const` och pekare forts

```
const char *safer(const char *s) { // <<<
  printf("%s \n", s);
  s++;
  printf("%s \n", s);
  // *s = '*';
  return s;
}

int main() {
  char *sIn = "Don't panic";
  char tIn[] = "Don't panic";
  char *sUt;
  char *tUt;
  sUt = safer(sIn); // << Ger varning
  tUt = safer(tIn); // << Ger varning
  printf("sUt: %s \n", sUt);
  printf("tUt: %s \n", tUt);
  *sUt = '*';
  *tUt = '*';
}
```

Output:

```
$ gcc const3.c
const3.c: In function ‘main’:
const3.c:19: warning: assignment discards
qualifiers from pointer target type
const3.c:20: warning: assignment discards
qualifiers from pointer target type
$ ./a.out
Don't panic
on't panic
Don't panic
on't panic
sUt: on't panic
tUt: on't panic
Segmentation fault (core dumped)
```

`const` och pekare forts

```
/* Safest */

const char *safest(const char *s) {
  printf("%s \n", s);
  s++;
  printf("%s \n", s);
  return s;
}
```

```
int main() {
  char *sIn = "Don't panic";
  const char *sUt; // <<<
  sUt = safest(sIn);
  printf("sUt: %s \n", sUt);
  //*sUt = '*'; Kompileringsfel!
}
```

- **`const`** framför en pekardeklaration anger att det utpekade inte får ändras (inte att pekaren inte får ändras)
- Använd **`const`** för att garantera att utpekade strukturer inte oavsiktligt ändras.
- Tag varningar på allvar!

(Man kan deklarera konstanta pekare. Exempel:
"`int * const p = q`" om `q` är en pekare eller ett arraynamn)

Återbesök av treeMap

```
void print(Tree t) {
  printf("Contents:\n");
  for (iterInit(t); iterMore(t); iterNext(t)) {
    printf("%10s = %s\n", (char *)currentKey(t), (char *)currentValue(t));
  }
}

int main() {
  char a[]="a", b[]="b", c[]="c", d[]="d";
  Tree t = makeTree((comparator)strcmp);
  put(t, b, "bb");
  put(t, a, "aa");
  put(t, d, "dd");
  put(t, c, "cc");
  print(t);
  iterInit(t);
  char *k = currentKey(t);
  strcpy(k,"urk!");
  print(t);
  return 0;
}
```

Inte så bra!

Output:

```
Marvin ./a.out
Contents:
         a = aa
         b = bb
         c = cc
         d = dd
Contents:
      urk! = aa
         b = bb
         c = cc
         d = dd
```

Återbesök av `treeMap` forts

Kan man förhindra att man utifrån saboterar trädet?

Nej men man kan åtminstone se till att det inte sker opåtalat:

```
typedef
struct treeNode {
  const void * key; //<<<<<
  void *value;
  struct treeNode *left;
  struct treeNode *right;
} treeNode, *TreeNode;
```

```
Marvin$ gcc -Wall -g stack.c treeMap.c
treeMap.c: In function <putPrivate>:
treeMap.c:33: warning: passing argument 2 of 'cmp' discards qualifiers from
                pointer target type
       ... (några till likadana)
treeMap.c: In function 'currentKey':
treeMap.c:93: warning: return discards qualifiers from pointer target type
```

Återbesök av `treeMap` forts

I `treeMap.h`:

```
typedef int (*comparator)(const void *, const void *);

const void *currentKey(Tree t);
```

och i `treeMap.c`

```
const void *currentKey(Tree t) {
```

Kompileringsförsök:

```
Marvin$ gcc -Wall -g stack.c treeMap.c
treeMap.c: In function 'main':
treeMap.c:128: warning: initialization discards qualifiers from pointer target type
Marvin$
```

vilket är raden:

```
char *k = currentKey(t);
```

Återbesök av `treeMap` forts

Ändring i `main`:

```
const char *k = currentKey(t);
```

Kompileringsförsök:

```
Marvin$ gcc -Wall -g stack.c treeMap.c
treeMap.c: In function <main>:
treeMap.c:129: warning: passing argument 1 of '__builtin___strcpy_chk' discards
qualifiers from pointer target type
treeMap.c:129: warning: passing argument 1 of '__inline_strcpy_chk' discards
qualifiers from pointer target type
Marvin$
```

Igen: Tag varningar på allvar!

Deklarationsfil för stack med iterator

```
/* stack.h - Deklarationsfil för stack med iterator
   En generell stackmekanism som lagrar pekare till ospecificerad typ (void*).
   Stacken representeras av en post med en pekare till översta elementet
   och, för iteratorändamål, en pekare till aktuellt element.
   Själva stacken består av en länkad lista.
*/

#ifndef __stack__
#define __stack__

typedef struct stackElem { // Listelementen
    void *item; // Ospecificerad pekartyp
    struct stackElem *next; // Nästföljande element
} stackElem, *link;

typedef struct Stack {
    link top; // Pekare till översta elementet
    link current; // Pekare till aktuellt element
} *Stack;
```

Deklarationsfil för stack med iterator forts

```
Stack newStack(); // Konstruktor
int push(void *it, Stack s); // Lagrar ett element på stacken
void *pop(Stack s); // Tar bort och returnerar översta elementet
int isEmpty(Stack s); // 1 om stacken tom, annars 0
void *top(Stack s); // Returnerar översta elementet på stacken
void iterInit(Stack s); // Initierar iteratorn till översta elementet
int iterMore(Stack s); // 1 om det finns aktuellt element, annars 0
void *iterNext(Stack s); // Returnerar aktuellt element och stegar fram

#endif
```

Eventuella ändringar i deklarationsfilen

- Om man vill förhindra att innehållet ändras:

```
const void *iterNext(Stack s);
```

- Om man vill kunna byta ut hela element i stacken

```
void **iterNext(Stack s);
```

(dock inte så troligt i en stackmekanism)

Övningar

- Se `const`-uppgiften i exempeltentan

Bitmanipulering, variabla argumentlistor, etc.

F18

Några saker som vi inte talat om

- *Bit-operatorer*:

  `~ & | ^ << >>`

- *Bitfält*: **`struct`** med angivande av antalet bitar för varje komponent (endast `int`)
- **`union`**: som **`struct`** men komponenterna delar på samma minnesutrymme
- *Flerdimensionella arrayer*:

```
int matrix[2][3] = {{0, 1, 2}{10, 11, 12}}
matrix[1][2] = 4;
```

- Ett antal *standardfunktioner*: tid, datum, cputid, felsignalering, variabelt antal parametrar, systemanrop. . .

Vi skall titta på några av dessa – i övrigt, se kursboken!

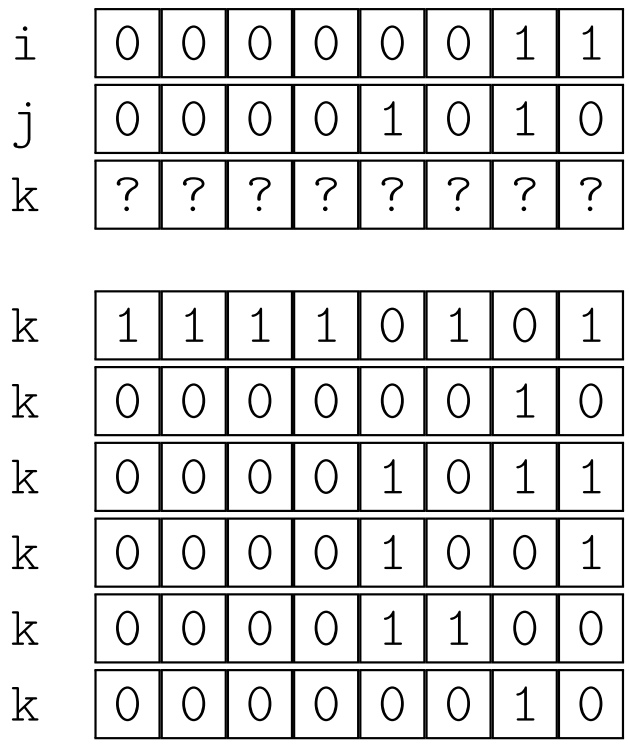
Bitmanipulering

~	unärt, bitvist icke
&	bitvist och
|	bitvist eller
^	bitvist exklusivt eller
<<	vänsterskift
>>	högerskift

Bitmanipulering, exempel

Operanderna för bitvisa operatorer är bitvektorer. Om man skapar en bitvektor från ett heltal bör detta vara **`unsigned`** för att undvika signed-bit:en.

```
unsigned char i = 3;
unsigned char j = 10;
unsigned char k;

k =~ j;

k = i & j;

k = i | j;

k = i ^ j;

k = i << 2;

k = j >> 2;
```



Bitflaggor

Ett vanligt användande av bitmanipulering är bitflaggor. Ibland har man många alternativ av typen ja/nej (av/på, sant/falskt, etc). och man vill inte slösa bort en massa minne genom att koda dem i t.ex. en **`char`**.

Sätt bit nr 15 (bitar brukar numreras från höger, högraste biten har nr 0):

```
flags |= 1UL << 15;
```

Släck bit nr 15:

```
flags &= ~(1UL << 15);
```

Kolla om bit nr 15 är satt:

```
if (flags & 1UL << 15) ...
```

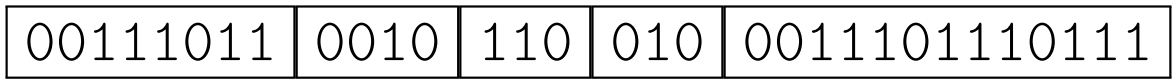
Släck gruppen om fyra bitar nr 9–12:

```
flags &= ~(0xFUL << 9);
```

```
#define SWITCHED_ON(flags,flag) (flags & 1U << flag)
#define USES_PADDING(flags) SWITCHED_ON(flags,7)
```

Bitfält

Det är möjligt att specificera på bitnivå hur stort ett fält i en struct skall vara. Läsning och skrivning av dessa fält avser sedan bara just denna fältstorlek.




```
typedef struct {
  unsigned int opcode : 8;
  unsigned int reg : 4;
  unsigned int indx : 4;
  unsigned int mode : 3;
  unsigned int addr : 13;
} instruction;

enum Opcode { NOOP, LOAD, STORE, ADD, JUMP, ... };
enum Mode { DIRECT, INDIRECT, IMMEDIATE, INDEXED, ... };

instruction i1 = { LOAD, 3, 0, DIRECT, 0xF77 };
instruction i2;
i2.opcode = ADD;
i2.reg = 2;
...
```

OBS!

Bitfält är helt implementationsoberoende.

Lagringsordning i minnesceller, huruvida ett bitfält får korsa en ordgräns, etc. är inte standardiserat.

Kod med bitfält är därför svårligen portabel.

Unioner: struktar med variabelt innehåll

Det här är JVMTI-kod som jag skrev med min magisterstudent för ett par veckor sedan, och som råkar innehålla både bitfält och unioner.

```
struct threadAccessInfo {
    jlong objectID;
    bool isThreadLocal : 1;
    bool isFinalizerShared : 1;
    bool isGCShared : 1;
    union {
        jlong threadID;
        list *accesses;
    };
};
```

Om strukten avser data som är trådlokalt så finns ett fält `threadID` av typen `jlong`. Annars finns ett fält `accesses` som är en pekare till en `list`. (Hur stor är strukten?)

```
  if (threadAccessInfoPtr->isThreadLocal) { threadAccessInfoPtr->threadID = ...
```

Hur fungerar `printf`?

```
printf("Hello, world");

printf("%d, %d\n", a, b);

printf("%d, %d, %d\n", a, b, c);

printf("%d, %d, %d, %d, %d\n", a, b, c, d, e);
```

Hur många argument tar `printf` egentligen (1, 3, 4, 6, något annat)?
Vad har argumenten för typ?

Variabla argumentlistor

Svaret på den första frågan från föregående sida: `printf` tar ett *variabelt antal argument*.

Det är ingenting magiskt med detta, och det är mycket enkelt att skapa en egen sådan funktion. Ponera denna funktion som skapar ett binärt träd och populerar det med 0 till flera värden:

```
Tree mkTree(Comparator cmpfunc, int argc, ...) { ... }
```

Som nu kan användas:

```
Tree t0 = mkTree(strcmp, 2, "Foo", "Bar);
Tree t1 = mkTree(strcmp, 0);
Tree t2 = mkTree(strcmp, 1, "Foo");
```

Minst ett argument måste vara namngivet. `...` måste stå sist i argumentlistan (varför?).

Vi ”måste” också inkludera `stdarg.h` som innehåller hjälpfunktioner för variabla argumentlistor.

Makron i `stdarg.h`

Standardbiblioteket `stdarg.h` definierar följande makron:

- `va_list` är typ som avser en pekare till en argumentlista
- `va_start(argptr, parameter)` initierar variabeln `argptr` till att peka på det argument som följer efter parametern `parameter`
- `va_arg(argptr, typ)` returnerar nästa argument som ett värde av typen `typ` och flyttar `argptr` till att pekar ut nästa argument
- `va_end(argptr)` skall alltid anropas när argumentlistan är genomgången! (dess beteende är implementationsberoende)

Känns ovanstående beteende igen från tidigare inslag på kursen?

Exempel 1

Implementationen av `mkTree` som tar emot ett variabelt antal element och stuvar in dem i det skapade trädet.

```
Tree mkTree(Comparator cmpfunc, int argc, ...) {
  Tree result = (Tree) malloc(sizeof(struct _tree));
  assert(result);

  result->root = NULL;
  result->cmpfunc = cmpfunc;

  va_list elemPtr;
  va_start(elemPtr, argc);
  while(argc--) {
    TreeElem elem = va_arg(elemPtr, TreeElem);
    insert(&(result->root), elem);
  }
  va_end(elemPtr);

  return result;
}
```

Exempel 2

Implementationen av funktionen `pathCompare` som tar emot en stig `(L|R)*` i ett binärträd samt ett antal element, vandrar stigen och jämför trädet med de inskickade elementen. element och stuvar in dem i det skapade trädet.

```
int pathCompare(Tree tree, char *path, ...) {
  Node n = tree->root;
  int result = 0;
  va_list elemPtr;
  va_start(elemPtr, path);
  do {
    TreeElem elem = va_arg(elemPtr, TreeElem);
    if (result = tree->cmpfunk(n->element, elem)) break;
    n = (*path == 'L') ? n->left : n->right;
  } while (*path++);
  va_end(elemPtr);
  return !result;
}
```

Hantering av otypat minne med `string.h`

- `void *memcpy(void *dest, const void* source, size_t n)` motsvarar `strcpy`; kopierar `n` bytes från `source` till `dest`, returnerar `dest`. Garanteras inte fungera om minnesareorna överlappar varandra.
- `void *memmove(void *dest,const void* source, size_t n)` som `memcpy` men fungerar även när minnesareorna överlappar.
- `int memcmp(const void *m1, const void *m2, size_t n)` motsvarar `strcmp`; jämför innehållet i areorna som pekas ut av `m1` respektive `m2` byte för byte tills antingen `n` bytes har gåtts igenom eller två olika bytes har påträffats, returnerar skillnaden mellan byten i `m1` och byten i `m2` eller 0 om alla *n* bytes var lika.
- `void *memchr(const void *m, int c, size_t n)` motsvarar `strchr`; returnerar pekare till den första påträffade förekomsten av `c` (omvandlad till **`unsigned char`**) i arean som pekas ut av `m` eller `NULL` om `c` inte finns bland de första `n` bytes.
- `void *memset(void *m, int c, size_t n)` kopierar in `c` (omvandlad till **`unsigned char`**) i de första `n` bytes i arean `m`

Exempel: xor-kryptering

```
// Exemplifierar
// Hantering av argument,
// IO mot filer samt
// bitoperationer

#include <stdio.h>
#include <string.h>

#define xor(a,b) (a)^(b)

/* eller, för att exemplifiera de andra bitoperatorerna:

   #define xor(a,b) (a)&~(b)|~(a)&(b)

*/

int main(int argc, char *argv[]) {
   char infileName[256];
   char outfileName[256];
   unsigned char key[256];
   unsigned char cut;
   int c, i;
```

xor-kryptering forts

```
FILE
   *infile = stdin, // default input
   *outfile = stdout; // default output

if (argc<2 || argc>4) {
    printf("xor: Wrong number of arguments\n");
    printf("Usage: xor key [file] [file]\n" );
    return 1;
}

strcpy(key, argv[1]);
if (argc>=3) { // input file given
    strcpy(infileName,argv[2]);
    infile = fopen(infileName, "r");
    if ( infile==NULL ) {
        printf("Could not open: %s\n", infileName);
        return 2;
    }
}

if (argc==4) { // output file given
    strcpy(outfileName,argv[3]);
    outfile = fopen(outfileName, "w");
}
```

xor-kryptering forts

```
   i = 0;
   while ( (c=getc(infile))!=EOF ) {
       cut = xor((char)c, key[i++]);
       putc(cut, outfile);
       if ( i==strlen(key) )
         i = 0;
   }
   return 0;
}

/* Exempel
kursa$ xor apa xor.c|xor hej|xor apa|xor hej>temp
kursa$ diff xor.c temp
kursa$
kursa$
*/
```

Övning: Skriv kod som byter innehåll på variablerna `x` och `y` *utan* att använda någon extra variabel.

Extramaterial

Exempel: Generell, grundläggande lista

- Skapa listor med vad som helst i
- Lägga in först
- Lägga in så att en sortering bibehålls
- Söka visst element
- Iterera över listan
- Byta ut element
- Ta bort element med visst innehåll

Dessutom:
Vill kunna ändra i den interna strukturen och lägga till funktioner utan att existerande applikationer behöver ändras.

Generell, grundläggande lista, forts

Innehållet ”vad som helst” indikerar `void *`

Frågor:

- Vad menas med att söka efter ett visst element?
- Vad menas med att ”en sortering bibehålls”. Sorterat på vad?

Endast användaren av list-mekanismen vet svaret — alltså måste användaren ge den informationen.

En *jämförelsefunktion*:

```
int compare(void *x, void *y)
```

som returnerar något negativt om `x<y`, 0 om `x=y` och något positivt om `x>y`.

Generell, grundläggande lista: `list.h`

```
/* list.h - En generell, grundläggande lista */
#ifndef __list__
#define __list__

typedef struct listElem {
    void *item;
    struct listElem *next;
} listElem, *link;

typedef int (*comparator)(void *, void *);

typedef struct listRecord {
    comparator cmp;
    link first;
    link current;
} listRecord, *List;
```

Generell, grundläggande lista: `list.h`

```
List newList(comparator cmp); // Skapa ett nytt listobjekt
void putFirst(List l, void *item); // Lagra ett element
void putInOrder(List l, void *item); // Lagra ett element sorterat
void *removeItem(List l, void *item); // Tag bort första förekomst av item
int member(List l, void *item); // Se om visst element finns
int size(List l); // Antal lagrade par
void clear(List l); // Tömmer listan
void itReset(List l); // Initiera iteratorn
int itMore(List l); // Genomitererad?
void *current(List l); // Aktuellt element
void **currentReference(List l); // Pekare till pekaren till aktuellt element
void *itNext(List l); // Aktuellt element samt stega fram
#endif
```

Generell, grundläggande lista: `list.c`

```c
/* list.c Implementationsfil för en generell listmekanism */
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "list.h"

/* Internal functions */

static link cons(void *item, link next) {
  link l = (link) malloc(sizeof(listElem));
  l->item = item;
  l->next = next;
  return l;
}
```

Generell, grundläggande lista: `list.c`

```
static link search(link l, void *item, comparator cmp) {
  if (l==NULL || cmp(l->item, item)==0) {
    return l;
  } else {
    return search(l->next, item, cmp);
  }
}

static link insert(link l, void *item, comparator cmp) {
  if (l==NULL || cmp(item, l->item)<0) {
    return cons(item, l);
  } else {
    l->next = insert(l->next, item, cmp);
    return l;
  }
}
```

Generell, grundläggande lista: `list.c`

```
/* User interface */

/**
 * Skapar en lista
 * Parametrar:
 * cmp - Funktionspekare till en jämförelsefunktion
 */
List newList(comparator cmp) {
  List s = (List) malloc(sizeof(*s));
  s->cmp = cmp;
  s->first = NULL;
  s->current = NULL;
  return s;
}
```

Generell, grundläggande lista: `list.c`

```
/**
 * Letar efter element med visst innehåll
 * Pre: s - initierad lista
 * Post: Om item finns kommer detta vara aktuellt element.
 * Om ej så är aktuellt element oförändrat
 * Return: 1 om finns, annars 0
 */
int member(List s, void *item) {
  link l = search(s->first, item, s->cmp);
  if (l!=NULL) {
    s->current = l;
    return 1;
  } else {
    return 0;
  }
}
```

Generell, grundläggande lista: `list.c`

```
/**
 * Lagrar ett nytt element först i listan
 * Pre: s är en initierad lista
 * Post: item är första element i listan
 * Sidoeffekt: Nya elementet är aktuellt element
 */
void putFirst(List s, void *item) {
  s->first = s->current = cons(item, s->first);
}
```

Generell, grundläggande lista: `list.c`

```
/**
 * Lagrar ett nytt element i före första förekomst av element som
 * är större (enligt listans comparator) än det nya elementet.
 * Pre: s är en initierad lista
 * Post: item är första element i listan
 * Sidoeffekt: Nya elementet är aktuellt element
 */
void putInOrder(List s, void *item) {
  s->first =s->current = insert(s->first, item, s->cmp);
}
```

Generell, grundläggande lista: `list.c`

```
/**
 * Tömmer listan
 * Pre: s en initierad lista
 * Post: s är en initierad men tom lista
 */
void clear(List s) {
  link l = s->first;
  while (l) {
    link r = l;
    l = l->next;
    free(r);
  }
  s->first = NULL;
  s->current = NULL;
}
```

Generell, grundläggande lista: `list.c`

```
/**
 * Sätter listans första element som aktuellt element
 */
void itReset(List s) {
  s->current = s->first;
}

/**
 * Returnerar 1 om det finns ett aktuellt element, annars 0
 */
int itMore(List s) {
  return s->current != NULL;
}

/**
 * Returnerar det aktuella elementet
 */
void *current(List s) {
  assert(s->current);
  return (s->current->item);
}
```

Generell, grundläggande lista: `list.c`

```
/**
 * Returnerar det aktuella elementet och stegar fram iteratorn
 */
void *itNext(List s) {
  if (s->current == NULL) {
    return NULL;
  } else {
    void *ret = s->current->item;
    s->current = s->current->next;
    return ret;
  }
}
```

Generell, grundläggande lista: Övningar

- Implementera `void *removeItem(List l, void *item)`
- Implementera `void **currentReference(List l)` som skall göra det möjligt att byta ut det aktuella elementet i listan.
- Implementera en `void *removeCurrent(List l)` som tar bort det aktuella elementet ur listan. Vilken komplexitet (i "ordo-mening") har operationen?

Generell, grundläggande lista: `listTest.c`

```c
/* listTest.c */

#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include "list.h"

/**
 * readWord - läser in nästa ord
 * Side effects: Mumsar i sig en bit av strömmen fp.
 * Konsumerar dynamiskt minne
 * Returns: Pekare till en dynamiskt allokerad sträng innehållande
 * nästa ord (obruten sekvens bokstäver) med gemena
 * eller NULL vid EOF
 */
```

Generell, grundläggande lista: `listTest.c`

```
char *readWord(FILE *fp) {
  char buffer[256];
  char *res;
  int c;
  int i = 0;
  while (!isalpha(c=getc(fp))) { // Hitta första bokstav
    if (c==EOF)
      return NULL;
  }
  do { // Samla bokstäver
    buffer[i++] = tolower(c);
  } while (isalpha(c=getc(fp)));
  buffer[i++] ='\0';
                                 // Skaffa utrymme och kopiera dit
  res = (char *)calloc(i,sizeof(char));
  strcpy(res, buffer);
  return res;
}
```

Generell, grundläggande lista: `listTest.c`

```
int compare(void *x, void *y) {
  // För att slippa warning
  return strcmp((char *)x, (char *)y);
}

void print(List s) {
  itReset(s);
  putchar('<');
  while (itMore(s)) {
    printf("\"%s\" ", (char *)itNext(s));
  }
  printf(">\n");
}
```

Generell, grundläggande lista: `listTest.c`

```
int main() { // Litet test
  char *word;
  List s = newList(compare);

  printf("Ord: ");
  while ((word = readWord(stdin))) {
    putInOrder(s, word);
  }
  print(s);

  itReset(s);
  printf("First: %s\n", (char *)(current(s)));
  strcpy(current(s) ,"Detta är livsfarligt! Varför?");
  print(s);

  return 0;
}
```

Tillämpning: Ordstatistik

```
/* wordStatistics.c - Räknar ordfrekvenser i standard input */

#include diverse headerfiler
#include "list.h"

/* Poster som skall lagras i listan */
typedef struct word {
  char *theWord;
  int count;
} word, *wordPointer;


/* Konstruktor för ordposter */
wordPointer makeWord(char *theWord) {
  wordPointer wp = (wordPointer) malloc(sizeof(word));
  wp->theWord = theWord;
  wp->count = 1;
  return wp;
}
```

Tillämpning: Ordstatistik

```
/* Skriver en ordpost på standard output */
void printWord(wordPointer wp) {
  printf("%-20s %d\n", wp->theWord, wp->count);
}

/* Jämför ordposter med avseende på ordet */
int compare(void *x, void *y) {
  wordPointer wpx = (wordPointer) x;
  wordPointer wpy = (wordPointer) y;
  return strcmp(wpx->theWord, wpy->theWord);
}

/* Jämför ordposter med avseende på frekvens */
int compareCount(void *x, void *y) {
  wordPointer wpx = (wordPointer) x;
  wordPointer wpy = (wordPointer) y;
  return wpy->count - wpx->count; // Fallande ordning
}
```

Tillämpning: Ordstatistik

```
/**
 * readWord - läser in nästa ord
 * Side effects: Mumsar i sig en bit av strömmen fp.
 * Konsumerar dynamiskt minne
 * Returns: Pekare till en dynamiskt allokerad sträng innehållande
 * nästa ord (obruten sekvens bokstäver) med gemena
 * eller NULL vid EOF
 */
char *readWord(FILE *fp) {
 // som tidigare
}
```

Tillämpning: Ordstatistik

```
/* Skiver en ordlista */
void print(List s) {
  itReset(s);
  while (itMore(s)) {
    printWord((wordPointer)itNext(s));
  }
}
```

Tillämpning: Ordstatistik - `main`

```
int main() {
  char *aWord;
  wordPointer wp;
  List s;
  List f;
  setlocale(LC_ALL, "sv");

  // Läs in och räkna
  s = newList(compare);
  while ((aWord = readWord(stdin))) {
    wp = makeWord(aWord);
    if (member(s, wp)) {
      (((wordPointer) current(s))->count)++;
      free(wp);
    }
    else {
      putInOrder(s, wp);
    }
  }
```

Tillämpning: Ordstatistik

```
  // Skriv i bokstavsordning
  printf("Ordfrekvenser i bokstavsordning\n");
  print(s);

  // Sortera i frekvensordning och skriv
  f = newList(compareCount);
  itReset(s);
  while (itMore(s)) {
    putInOrder(f, itNext(s));
  }
  printf("\nOrdfrekvenser i frekvensordning\n");
  print(f);

  return 0;
}
```